Creating Packages

Objectives

After completing this lesson, you should be able to do the following:

- Describe packages and list their possible **components**
- Create a package to group together related **variables, cursors, constants, exceptions, procedures, and functions**
- Designate a package construct as either public or **private**
- Invoke a package construct
- Describe a use for a bodiless package

Packages Overview

Packages bundle related PL/SQL types, items, and subprograms into one conta iner. For example, a Human Resources package can contain hiring and firing procedures, commission and bonus functions, and tax exemption variables.

A pa ckage usually has a specification and a body, stored separately in the database. The specification is the int erface to your applications. It declares the types, variables, consta nts, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMAs, which are directives to the compiler. The body fully defines cursors and subprograms, and so implements the specification.

The package itself cannot be called, paramet erized, or nested. S till, the format of a package is similar to t hat of a subprogram. Once written and compiled, t he cont ents can be shared by ma ny applications. When you call a packa ged PL/SQL construct for the first t ime, the whole packa ge is loaded into memory. Thus, later calls to constructs in the same package require no disk input/output (I/O).

Package Development

You create a package in two pa rts: first the package specification, and then the package body. Public package constructs are those that are declared in the package specification and defined in the package body. Private package constructs are those that are defined solely within the package body.

Scope of the Construct Description Placement within the Package

Public Can be referenced from any Oracle server environment Declared w ithin the package specification and may be defined within the package body

Private Can be referenced only by other constructs which are part of the same package Declared and defined within the package body

Note: The Oracle server stores the specification and body of a package separately in the da tabase. This ena bles you to change the definition of a program construct in the package body without causing the Oracle server to invalidate other schema objects that call or reference the progra m construct.

Visibility of the Construct Description

Local A variable defined within a subprogram that is not visible to external users.

Private (local to the package) variable: You can define variables in a package body. These variables can be accessed only by other objects in the same package. They are not visible to any subprograms or objects outside of the package.

Global A variable or subprogram that can be referenced (and changed) outside the package and is visible to external users. Global package items must be declared in the package specification.

Example

```
CREATE OR REPLACE PACKAGE comm_package IS
g_comm NUMBER := 0.10; --initialized to 0.10
PROCEDURE reset_comm
(p_comm IN NUMBER);
END comm_package;
/
```

Creating the Package Body

To create packages, define all public and private constructs within the package body.

- Specify the REPLACE option when the package body already exists.
- The order in which subprograms are defined within the package body is important: you must declare a variable before another variable or subprogram can refer to it, and you must declare or define private subprograms before calling them from other subprograms. It is quite common in the package body to see all private variables and subprograms defined first and the public subprograms defined last.

You can define a private procedure or function to modularize and clarify the code of public procedures and functions.

Note: When you are coding the package body, the definition of the private function has to be above the definition of the public procedur e.

Only subprograms and cursors decla rations without body in a package specification have an underlying implementation in the package body. So if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. However, the body can still be used t o initialize items declared in the package specification.

comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
 FUNCTION validate_comm (p_comm IN NUMBER)
RETURN BOOLEAN
IS
v_max_comm NUMBER;
BEGIN
SELECT MAX(commission_pct)
INTO v_max_comm
FROM employees;
IF p_comm > v_max_comm THEN RETURN(FALSE);
ELSE RETURN(TRUE);
END IF;
END validate_comm;
...
```

Define a function to validate the commission. The commission may not be greater than the highest commission among all existing employees.

```
PROCEDURE reset_comm (p_comm IN NUMBER)
IS
BEGIN
IF validate_comm(p_comm)
THEN g_comm:=p_comm; --reset global variable
ELSE
```

```
RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
END IF;
END reset_comm;
END comm_package;
/
```

Define a procedure that enables you to reset and validate the prevailing commission.

Invoking Package Constructs

Example 1: Invoke a function from a procedure within the same package.

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
...
PROCEDURE reset_comm
(p_comm IN NUMBER)
IS
BEGIN
IF validate_comm(p_comm)
THEN g_comm := p_comm;
ELSE
RAISE_APPLICATION_ERROR
(-20210, 'Invalid commission');
END IF;
END reset_comm;
END comm_package;
```

After the package is stored in the database, you can invoke a package construct within the package or from outside the package, depending on whether the construct is private or public. When you invoke a package procedure or function from within the same package, you do not need to qualify its name.

Example 2: Invoke a package procedure from iSQL*Plus.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

```
EXECUTE scott.comm_package.reset_comm(0.15)
```

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```

When you invoke a package procedure or function from outside the package, you must qualify its name with the name of the package.

Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE global_consts IS
mile_2_kilo CONSTANT NUMBER := 1.6093;
kilo_2_mile CONSTANT NUMBER := 0.6214;
yard_2_meter CONSTANT NUMBER := 0.9144;
meter_2_yard CONSTANT NUMBER := 1.0936;
END global_consts;
/
EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = '||20*
global_consts.mile_2_kilo||' km')
```

You can dec lare public (global) variables that exist for the duration of the user session. You can create a package specification that does not need a package body. As discussed earlier , if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary.

In the example , a package specification containing several conversion rates is defined.

All the global identifiers are declared as constants. A package body is not required to support this package specification because implementation details are not required for any of the constructs of the package specification.

Referencing a Public Variable from a Stand-Alone Procedure

Example:

```
CREATE OR REPLACE PROCEDURE meter_to_yard
(p_meter IN NUMBER, p_yard OUT NUMBER)
IS
BEGIN
p_yard := p_meter * global_consts.meter_2_yard;
END meter_to_yard;
/
VARIABLE yard NUMBER
EXECUTE meter_to_yard (1, :yard)
PRINT yard
```

Example

Use the METER_TO_YARD procedure to convert meters to yards, using the conversion rate packaged in GLOBAL_CONSTS.
When you reference a variable, cursor, constant, or exc eption from outside the packa ge, you must qualify its name with the name of the package.

Guidelines for Writing Packages

Keep your packages as general as possible so that they can be reused in future applications. Also, avoid writing packages that duplicate features provided by the Oracle server. Package specifications reflect the design of your application, so define them before defining the package bodies.The package specification should contain onl y those constructs that must be visible to users of the package. That way other developers cannot misuse the package by basing code on irrelevant details. Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions. For example, declare a variable called UMBER_EMPLOYED as a private variable, if each call to a procedure that uses the variable needs to be maintained. When declared as a global variable in the package specification, the value of that global variable gets initia liz ed in a session the first time a construct from the package is invoked.

Changes to the package body do not require recompilation of dependent constructs, whereas changes to the package specification require recompilation of every stored subprogram that references the package. To reduce the need for recompiling when code is changed, place as few constructs as possible in a package specification.

Advantages of Using Packages

Modul arity

You encapsulate logically related programming structures in a named module. Each package is easy to understand, and the interface between packages is simple, clear, and well defined.

Easier Application Design

All you need initially is the interface information in the package specification. You can code and compile a specification wit hout its body. Then stored subprograms that reference the package can compile as well. You need not define t he package body fully until you are ready to complet e the application.

Hiding Information

You can decide which constructs are public (visible and accessible) or private (hidden a

nd inaccessible). Only the declarations in the package specification are visible and accessible to applications. The package body hides the definition of the private constructs so that only the package is affected (not your application or any calling programs) if the definition changes. This enables you to change the implementation without having to recompile calling programs. Also, by hiding implementation details
from users, you protect the integrity of the package.

Added Functionality

P ackaged public variables and cursors persist for t he duration of a session. Thus, they can be shared by all subprograms that execute in the environment. They also enable you to maintain data across transactions without havi ng to store it in the database. Private constructs also persist for the durat ion of the session, but can only be accessed within the package.

Better Performance

When you call a packaged subprogram the first time, the entire package is loaded into memory. This way, later calls to related subprograms in the package require no further disk I/O. Packaged subprogra ms also stop cascading dependencies and so avoid unnecessary compilation.

Overloading

With packages you can overload procedures and functions, whi ch means you can create multiple subprogra ms with the same name in the same package, ea ch taking parameters of different number or datatype.

More Package Concepts

Objectives

After completing this lesson, you should be able to do the following:

- Write packages that use the overloading feature
- Describe errors with mutually referential **subprograms**
- Initialize variables with a one-time-only procedure
- Identify persistent states

Overloading

This feature enables you to define different subprograms with the sa me name. You can distinguish the subprograms both by name and by parameters. Sometimes the processing in two subprograms is the same, but t he paramet ers passed to them varies. In t hat ca se it is logical t o give them the same name. PL/SQL determines which subprogra m is called by checking its formal parameters. Only local or packaged subprograms can be overloaded. Stand-alone subprograms cannot be overloaded.

Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same family (NUMBER and DECIMAL belong to the same family)
- Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same fa mily (VARCHAR and STRING are PL/SQL subtypes of VARCHAR2)
- Two functions that differ only in return type, even if the types are in different families

Note: The above restrictions apply if the names of the para meters are also the same. If you use different names for the para meters, then you can invoke the subprogra ms by using na med notation for the parameters.

Resolving Calls

The compiler tries to find a declaration that ma tches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the na me matches the name of the called subprogram. For like-named subprograms at the same level of scope, the compiler needs a n exact match in number, order, and data type between the actual a nd formal parameters.

over_pack.sql

```
CREATE OR REPLACE PACKAGE over_pack
IS
PROCEDURE add_dept
(p_deptno IN departments.department_id%TYPE,
p_name IN departments.department_name%TYPE
DEFAULT 'unknown',
p_loc IN departments.location_id%TYPE DEFAULT 0);
PROCEDURE add_dept
(p_name IN departments.department_name%TYPE
DEFAULT 'unknown',
p_loc IN departments.location_id%TYPE DEFAULT 0);
END over_pack;
/
```

The package contains ADD_DEPT as the name of two overloaded procedures. The first definition takes three pa rameters to be able to insert a new department to the department table. The second definition takes only two parameters, because the department ID is populated through a sequence.

over_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY over_pack IS
PROCEDURE add_dept
(p_deptno IN departments.department_id%TYPE,
p_name IN departments.department_name%TYPE DEFAULT 'unknown',
p_loc IN departments.location_id%TYPE DEFAULT 0)
IS
BEGIN
INSERT INTO departments (department_id, department_name, location_id)
VALUES (p_deptno, p_name, p_loc);
END add_dept;
PROCEDURE add_dept
(p_name IN departments.department_name%TYPE DEFAULT 'unknown', p_loc
IN departments.location_id%TYPE DEFAULT 0)
IS
BEGIN
INSERT INTO departments (department_id, department_name, location_id)
VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
END over_pack;
/
```

If you call ADD_DEPT with an explicitly provided department ID, PL/SQL uses the first version of the procedure. If you call ADD_DEPT with no department ID, PL/SQL uses the second version. Most built-in functions are overloaded. For example, the function TO_CHAR in the package STANDARD has four different declarations. The function can t ake either the DATE or the NUMBER data type and convert it to the character data type. The format into which the date or number has to be converted can also be specified in the function call.

If you redeclare a built-in subprogram in another PL/SQL program, your local declaration overrides the standard or built-in subprogram. To be able to access the built-in

subprogram, you need to qualify it with its package name. F or example, if you redeclare the TO_CHAR function, to access the built-in function
you refer it as: STANDARD.TO_CHAR.

If you redeclare a built-in subprogram as a stand-alone subprogram, to be able to access your subprogram you need to qualify it with your schema name, for example, SCOTT.TO_CHAR.

Using Forward Declarations

PL/SQL does not allow forward references. You must declare an identifi er before using it. Therefore, a subprogram must be declared before calling it.

PL/SQL enables for a special subprogram declaration called a forward declaration. It consists of the subprogram specification terminated by a semicolon. You can use forward dec larations to do the following:

- Define subprograms in logical or alphabetical order
- Define mutually recursive subprograms
- Group subprograms in a packa ge

Mutually recursive programs are programs that call each ot her directly or indirectly.

Note: If you receive a compilation error that CALC_RATING is undefined, it is only a problem if CALC_RATING is a priva te packaged procedure. If CALC_RATING is declared in the package specification, the reference to the public procedure is resolved by the compiler.

- The formal parameter list must appear in both the forward declaration a nd the subprogram body.
- The subprogram body can appear anywhere after the forward declaration, but both must appear in thesameprogramunit.

For ward Declarations and Packages

Forward declarations typically let you group related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to the applications. In this way, packages enable you to hide implementation details.

```
CREATE OR REPLACE PACKAGE taxes
IS
 tax NUMBER;
... -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes
IS
... -- declare all private variables
... -- define public/private procedures/functions
BEGIN
SELECT rate_value
INTO tax
FROM tax_rates
WHERE rate_name = 'TAX';
END taxes;
/
```

Define an Automatic, One-Time-Only Procedure

A one-time-only procedure is executed only once, when the package is first invoked within the user session. The current value for TAX is set t o he value in the TAX_RATES table the first time the TAXES package is referenced.

Note: Initialize public or private variables with an automatic, one-time-only procedure when the derivation is too complex to embed within the variable

declaration. In this case, do not initialize the variable in the declaration, because the value is reset by the one-time-only procedure. The keyword END is not used at the end of a one-time-only procedure.

Controlling Side Effects

Side effects are changes to database tables or public packaged variables (those declared in a package specification). Side effects could delay the execution of a query, yield order-dependent (therefore indeterminate) results, or require that the package state variables be maintained across user sessions. Various side effects are not allowed when a function is called from a SQL query or DML statement. Therefore, the following restrictions apply to stored functions called from SQL expressions:

- A function called from a query or DML statement can not end the current transaction, create or roll back to a savepoint, or alter the system or session
- A function called from a query statement or from a parallelized DML statement can not execute a DML statement or otherwise modify the database
- A function called from a DML statement can not read or modify the particular table being modified by that DML statement

Calling Package Functions

You call PL/SQL functions the same way that you call built-in SQL functions.

```
CREATE OR REPLACE PACKAGE comm_package IS
g_comm NUMBER := 10; --initialized to 10
PROCEDURE reset_comm (p_comm IN NUMBER);
END comm_package;
/
CREATE OR REPLACE PACKAGE BODY comm_package IS
FUNCTION validate_comm (p_comm IN NUMBER)
RETURN BOOLEAN
IS v_max_comm NUMBER;
BEGIN
... -- validates commission to be less than maximum commissioninthetable
END validate_comm;
PROCEDURE reset_comm (p_comm IN NUMBER)
IS BEGIN
... -- calls validate_comm with specified value
END reset_comm;
END comm_package;
/
```

Persistent State of Package Variables

This sample package illustrates the persistent state of package variables. The VALIDATE_COMM function validates commission to be no more than maximum currently earned. The RESET_COMM procedure invokes the VALIDATE_COMM function. If you try to reset the commission to be higher than the maximum, the exception RAISE_APPLICATION_ERROR is raised.

In the VALIDATE_COMM function, the maximum salary from the EMPLOYEES table is selected into the variable V_MAXSAL. Once the variable is assigned a value, the value persists in the session until it is modified again.

Controlling the Persistent State of a Package Variable

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

1. Initialize the variable within its declaration or within an automatic, one-time-only procedure.
2. Change the value of the variable by means of pa ckage procedures.
3. The value of the variable is released when the user disconnects.

Example:

```
CREATE OR REPLACE PACKAGE pack_cur
IS
CURSOR c1 IS SELECT employee_id
FROM employees
ORDER BY employee_id DESC;
PROCEDURE proc1_3rows;
PROCEDURE proc4_6rows;
END pack_cur;
/
```

Controlling the Persistent State of a Package Cursor Example

Use the following steps to control a public cursor:

1. Declare the public (global) cursor in the package specification.
2. Open the cursor and fetch successive rows from the cursor, using one (public) packaged procedure, PROC1_3ROWS.
3. Continue to fetch successive rows from the cursor, and then close the cursor by using another (public) packaged procedure, PROC4_6ROWS.

Controlling the Persistent State of a Package Cursor

```
CREATE OR REPLACE PACKAGE BODY pack_cur IS
v_empno NUMBER;
PROCEDURE proc1_3rows IS
BEGIN
OPEN c1;
LOOP
FETCH c1 INTO v_empno;
DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
EXIT WHEN c1%ROWCOUNT >= 3;
END LOOP;
END proc1_3rows;
PROCEDURE proc4_6rows IS
BEGIN
LOOP
FETCH c1 INTO v_empno;
DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
EXIT WHEN c1%ROWCOUNT >= 6;
END LOOP;
CLOSE c1;
END proc4_6rows;
END pack_cur;
/
```

PL/SQL Tables and Records in Packages

```
CREATE OR REPLACE PACKAGE emp_package IS
TYPE emp_table_type IS TABLE OF employees%ROWTYPE
INDEX BY BINARY_INTEGER;
PROCEDURE read_emp_table
(p_emp_table OUT emp_table_type);
```

```
END emp_package;
/
CREATE OR REPLACE PACKAGE BODY emp_package IS
PROCEDURE read_emp_table
(p_emp_table OUT emp_table_type) IS
i BINARY_INTEGER := 0;
BEGIN
FOR emp_record IN (SELECT * FROM employees)
LOOP
p_emp_table(i) := emp_record;
i:= i+1;
END LOOP;
END read_emp_table;
END emp_package;
/
```

Passing Tables of Records to Procedures or Functions Inside a Package

Oracle Supplied Packages

Objectives

After completing this lesson, you should be able to do the following:

- Write dynamic SQL statements using DBMS_SQL **and EXECUTE IMMEDIATE**
- Describe the use and application of some Oracle **server-supplied packages:**
  - DBMS_DDL
  - DBMS_JOB
  - DBMS_OUTPUT
  - UTL_FILE
  - UTL_HTTP and UTL_TCP

Using Native Dyna mic SQL (Dynamic SQL)

You can write PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program but rather are stored in character strings that are input to, or built by, the program. That is, the SQL statements can be created dynamically at run t ime by using variables. For example, you use dyna mic SQL to create a procedure that operates on a table whose name is not known until run time, or to write and execute a data definition language (DDL) statement (such as CREATE TABLE), a data control statement (such as GRANT), or a session control stat ement (such as ALTER SESSION). In PL/SQL, such statements cannot be executed statica lly.

In Oracle 8i, you can use DBMS_SQL or native dyna mic SQL. The EXECUTE IMMEDIATE stat ement can perform dynamic single-row queries. Also, this is used for functionality such a s objects and collections, which are not supported by DBMS_SQL. If the statement is a mult irow SELECT statement, you use OPEN-FOR, FETCH,andCLOSE statement s.

Steps to Process SQL Statements

All SQL statements have to go through various stages. Some st ages may be skipped.

Parse

Every SQL st atement must be parsed. Parsing the statement includes checking the stat ement's syntax and validating the statement, ensuring that all references to objects are correct, and ensuring that the relevant privileges to those objects exist.

Bind

After parsing, the Oracle server knows the mea ning of the Oracle statement but still may not have enough information to execute the statement. The Oracle server may need values for any bind variable in the statement. The process of obtaining these values is called binding variables.

Execute

At this point, the Oracle server has all necessary information and resources, and the statement is executed.

Fetch

In the fetch stage, rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched. You can fetch queries, but not the DML statements.

Using the DBMS_SQL Package

Using DBMS_SQL, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. DBMS_SQL can issue data definition language statements in PL/SQL. For example, you can choose to issue a DROP TABLE statement from within a stored procedure. The operations provided by this packa ge a re performed under the current user, not under the package owner SYS. Therefore, if the caller is an anonymous PL/SQL block, the operations are performed
according t o the privileges of the current user; if the caller is a stored procedure, the operations are performed according to the owner of the stored proc edure.

Using this package to execute DDL statements can result in a deadlock. The most likely reason for this is that the package is being used to drop a procedure that you are still using.

Components of the DBMS_SQL Package

The DBMS_SQL package uses dynamic SQL to access the database.

Function or Procedure Descripti on

OPEN_CURSOR Opens a new cursor and assigns a cursor ID number

PARSE Parses the DDL or DML statement: that is, checks the statement's syntax and associates it with the opened cursor (DDL statements are immediately
executed when parsed)

BIND_VARIABLE Binds the given value to the variable identified by its name in the parsed statement in the given cursor

EXECUTE Executes the SQL statement and returns the number of rows processed

FETCH_ROWS Retrieves a row for the specified cursor (for multiple rows, call in a loop)

CLOSE_CURSOR Closes the specified cursor

```
CREATE OR REPLACE PROCEDURE delete_all_rows (p_tab_name IN
VARCHAR2, p_rows_del OUT NUMBE R)
IS
cursor_name INTEGER;
BEGIN
cursor_name := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(cursor_name, 'DELETE FROM '||p _tab_nam e,
DBMS_SQL.NATIVE );
p_rows_del := DBMS_SQL.EXECUTE (cursor_name);
DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Use dynamic SQL to delete rows

```
VARIABLE dele ted NUMB ER
```

```
EXECUTE delete_all_rows('employees', :deleted)
PRINT deleted
```

Example of a DBMS_SQL Package

The table name is passed into the DELETE_ALL_ROWS procedure by using an IN parameter. The procedure uses dynamic SQL to delete rows from the specified table. The number of rows that are deleted as a result of the successful execution of the dynamic SQL are passed to the calling environment through an OUT parameter.

Using the EXECUTE IMMEDIATE Statement

Syntax Definition

Parameter	**Description**
dynamic_string	A string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator)
define_variable	A variable that stores the selected column value
record	A user -defined or %ROWTYPE record that stores a selected row
bind_argument	An expression whose value is passed to the dynamic SQL statement or PL/SQL block

You can use the INTO clause for a single-row query, but you must use OPEN-FOR, FETCH,and CLOSE for a multirow query.

In the EXECUTE IMMEDIATE statement :

•TheINTO clause specifies the variables or record into which column values are retrieved. It is used only for single-row queries. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.

•TheRETURNING INTO clause specifies the variables into which column values are returned. It is used only for DML statements that have a RETURNING clause (without a BULK COLLECT clause). For each value returned by the DML statement, there must be a corresponding, type-compatible variable in the RETURNING INTO clause.

•TheUSING clause holds all bind arguments. The default parameter mode is IN.For DML statements that have a RETURNING clause, you can place OUT arguments in the RETURNING INTO clause without specifying the parameter mode, which, by definition, is OUT. If you use both the USING clause and the RETURNING INTO clause, the USING clause can contain only IN arguments.

At run time, bind arguments replace corresponding placeholders in the dynamic string. Thus, every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE,andNULL).

Dynamic SQL supports all the SQL data types. For example, define variables and bind arguments can be collections, LOBs , instances of an object type, and REFs. As a rule, dynamic SQL does not support PL/SQL-specific types. For example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause. You can execute a dynamic SQL statement repeatedly, using new values for the bind arguments.

However, you incur some overhead because EXECUTE IMMEDIATE reparses the dynamic string before every execution.

Dynamic SQL Using EXECUTE IMMEDIATE

```
CREATE PROCEDURE del_rows
(p_table_name IN VARCHAR2,
p_rows_deld OUT NUMBER)
IS
BEGIN
EXECUTE IMMEDIATE 'delete from '||p_table_name;
p_rows_deld := SQL%ROWCOUNT;
```

```
END;
/
VARIABLE deleted NUMBER
EXECUTE del_rows('test_employees',:deleted)
PRINT deleted
```

This is the same dynamic SQL as seen with DBMS_SQL,usingtheOracle8i statement EXECUTE IMMEDIATE.TheEXECUTE IMMEDIATE stat ement prepares (parses) and immediately execut es t he dynamic SQL statement.

Using the DBMS_DDL package

This package provides access to some SQL DDL statements, which you can use in PL/SQL programs. DBMS_DDL is not allowed in triggers, in procedur es called from Forms Builder, or in remote sessions. This package runs with the privileges of calling user, rather than the package owner SYS.

Practical Uses

• You can recompile your modified PL/SQL program units by using DBMS_DDL.ALTER_COMPILE. The object type must be either procedure, function, package,
package body, or trigger.

• You can analyze a single object, using DBMS_DDL.ANALYZE_OBJECT. (Thereisawayof analyzing more than one object a t a time, using DBMS_UTILITY.) The object type should be TABLE, CLUSTER,orINDEX. The method must be COMPUTE, ESTIMATE,orDELETE.

• This package gives developers access to ALTER and ANALYZE SQL statements through PL/SQL environments.

Scheduling Jobs by Using DBMS_JOB

The package DBMS_JOB is used to schedule PL/SQL programs to run. Using DBMS_JOB, you can submit PL/SQL progra ms for execution, execute PL/SQL programs on a schedule, identify when PL/SQL programs should run, remove PL/SQL programs from the schedule, and suspend P L/ SQL progra ms from running.

It can be used to schedule batch jobs during nonpeak hours or to run maintenance programs during
times of low usage.

DBMS_JOB Subprograms

Subprogram	**Description**
SUBMIT	Submits a job to the job queue
REMOVE	Removes a specified job from the job queue
CHANGE	Alters a specified job that has already been submitted to the job queue (you can alter the job description, the tim e at which the job will be run, or the interval between executions of the job)
WHAT	Alters the job description for a specified job
NEXT_DATE	Alters the next executio n time fo r a specified job
INTERVAL	Alters the interval between executions for a specified job
BROKEN	Disables job execution (if a job is m arked as broken, the Oracle server does not attempt to execute it)
RUN	Forces a specified job to run

DBMS_JOB.SUBMIT Parameters

The DBMS_JOB.SUBMIT procedure adds a new job to the job queue. It accepts five parameters and returns the number of a job submitted through the OUT parameter JOB.

The descript ions of the parameters are listed below.

Parameter Mode Description

JOB OUT	Unique identifier of the job
WHAT IN PL/SQL	code to execute as a job
NEXT_DATE IN	Next execution date of the job
INTERVAL IN	Date functio n to compute the next execution date of a job
NO_PARSE IN	Boolean flag that ind icates whether to parse the job at job su bmission (the default is false)

Note: An exception is raised if the interval does not evalua te to a time in the future.

```
VARIABLE jobno NUMBER
BEGIN
DBMS_JOB.SUBMIT ( job => :jobno, what => 'OVER_PACK.ADD_DEPT
(''EDUCATION'',2710);', next_date => TRUNC(SYSDATE + 1),
interval => 'TRUNC(SYSDATE + 1)' );
COMMIT;
END;
/
PRINT jobno
```

The block of c ode in the slide submits the ADD_DEPT procedur e of the OVER_PACK package to the job queue. The job number is returned through the JOB parameter. The WHAT parameter must be enclosed in single quotation marks and must include a semicolon a t the end of the text string. This job is submitted to run every day at midnight.

Note: In the example, the parameters are passed using na med notation. The t ransactions in the submitted job are not commit ted until either COMMIT is issued, or DBMS_JOB.RUN is execut ed to run t he job. COMMIT in the slide commits the transaction.

Changing Jobs After Being Submitted

The CHANGE, INTERVAL, NEXT_DATE,andWHAT procedures enable you to modify job characteristics after a job is submitted to the queue. Each of t hese procedures takes the JOB parameter as an IN parameter indicating which job is to be changed.

Example

The following code changes j ob number 1 to execute on the following day at 6:00 a.m. and every four hours after that.

```
BEGIN
DBMS_JOB.CHANGE(1, NULL, TRUNC(SYSDATE+1)+6/24, ’SYSDATE+4/24');
END;
/
```

Note: Each of these procedures can be executed on jobs owned by the username to which the session is connected. If the parameter what, next_date,orinterval is NULL, then the last values assigned to those parameters are used.

Running, Removing, and Breaking Jobs

The DBMS_JOB.RUN procedure executes a job immediately. P ass the job number that you want to run immediately to the procedure. EXECUTE DBMS_JOB.RUN(1)

The DBMS_JOB.REMOVE procedure removes a submitted job from the job queue. Pass the job number that you want to remove from the queue to the procedure.

EXECUTE DBMS_JOB.REMOVE(1)

The DBMS_JOB.BROKEN marks a job as broken or not broken. Jobs are not broken by default. You can change a job to the broken status. A broken job will not run. There are three parameters for this procedure. The JOB parameter identifies the job to be marked as broken or not broken. The BROKEN parameter is a Boolean parameter. Set t his

parameter t o FALSE t o indicate that a job is not broken, and set it to TRUE to indicate that it is broken. The NEXT_DATE parameter identifies the next execution date of the job.
EXECUTE DBMS_JOB.BROKEN(1, TRUE)

Viewing Information on Submitted Jobs
The DBA_JOBS and DBA_JOBS_RUNNING dictionary views display information about jobs in the queue and jobs that have run. To be able to view t he dictionary information, users should be granted the SELECT privilege on SYS.DBA_JOBS.
The query shown in the slide displays the job number, the user who submitted the job, the scheduled date for t he job to run, the time for the job to run, and the PL/SQL block execut ed as a job. Use t he USER_JOBS data dictionary view to display information about jobs in the queue for you. This view has the same structure as the DBA_JOBS view.

Using the DBMS_OUTPUT Package

The DBMS_OUTPUT package outputs values and messages from any PL/SQL block.

Function or Procedure	**Description**
PUT	Appends text from the procedure to the current line of the line output buffer
NEW_LINE	Places an end_of_line marker in the output buffer
PUT_LINE	Combines the action of PUT and NEW_LINE
GET_LINE	Ret rieves the current line from the output buffer int o the procedure
GET_LINES	Retrieves an array of lines from the output buffer into the procedure
ENABLE/DISABLE	Enables or disables calls to the DBMS_OUTPUT procedures

Practical Uses

- You can output intermediary results to the window for debugging purposes.
- This package enables developers to closely follow the execution of a function or procedure by sending messages and values to the output buffer.

Inte racting with Operating System Files

Two Oracle-supplied packa ges are provided. You can use them to access operating system files. With the Oracle-supplied UTL_FILE package, you can read from and write to operating system files. This package is available with database version 7.3 and later and the PL/SQL version 2.3 and later. With the Oracle-supplied package DBMS_LOB, you can read from binary files on the operating system. This package is available from the database version 8.0 and later.

The UTL_FILE Package

The UTL_FILE package provides text file I/O from within PL/SQL. Client-side security implementation uses normal operating system file permission checking. Server-side security is i mplemented through restrictions on the directori es that can be accessed. In the init.ora file, the initialization para met er UTL_FILE_DIR is set to the accessible directories desired. UTL_FILE_DIR = directory_name For example, the following initializa tion setting indicates that the directory /usr/ngreenbe/my_app is accessible to the fopen function, assuming that t he directory is a ccessible to the database server processes. This parameter setting is case-sensitive on case- sensitive operating systems.

UTL_FILE_DIR = /user/ngreenbe/my_app
The directory should be on the same machine as the database server. Using the following setting t urns off database permissions and makes all directories that are accessible to t he database server processes also accessible to the UTL_FILE packa ge.
UTL_FILE_DIR = *
Using the procedures and functions in the packa ge, you can open files, get text from files, put text into files, and close files. There are seven exceptions decla red in the package to account for possible errors raised during execution.

File Processing Using the UTL_FILE Package

Before using the UTL_FILE package to read from or writ e to a text file, you mus t first check whether thetextfileis openbyusingtheIS_OPEN function. If the file is not open, you open the file with the FOPEN function. You then either rea d the file or write to the file until processing is done. At the end of file processing, use the FCLOSE procedure to close the file.

The UTL_FILE Package: Procedures and Functions

Function or Procedure	**Description**
FOPEN	A func tion that opens a file for input or output and returns a file handle used in subsequent I/O operations
IS_OPEN	A func tion that returns a Boolean value wheneve r a file handle refers to an open file
GET_LINE	A proc edure that reads a line of te xt from the opened file and places the text in the output buffer parameter (the maximum size of an input record is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN)
PUT, PUT_LINE	A proc edure that writes a text string stored in the buffer parameter to the opened file (no line terminator is appended by put;usenew_line to terminate the line, or use PUT_LINE to write a complete line with a terminator)
PUTF	A formatted put procedure with two format specifie rs: %s and \n (use %s to substitute a value into the output string. \n is a new line character)
NEW_LINE	Procedure that terminates a line in an output file
FFLUSH	Procedure that writes all data buffered in memory to a file
FCLOSE	Procedure that closes an opened file
FCLOSE_ALL	Procedure that closes all opened file handles for the session

Note: The maximum size of an input rec ord is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN.

Exceptions to the UTL_FILE Package

The UTL_FILE package declares seven exceptions that are raised to indicate an error condition in the operating system file processi ng.

Exception Name	**Description**
INVALID_PATH	The file location or filename was invalid.
INVALID_MODE	The OPEN_MODE parameter in FOPEN was invalid.
INVALID_FILEHANDLE	The file handle was invalid.
INVALID_OPERATION	The file could not be opened or operated on as requested.

READ_ERROR An operating system error occurred during the read operation.

WRITE_ERROR An operating system error occurred during the write operation.

INTERNAL_ERROR An unspecified error occurred in PL/SQL.

Note: These exceptions must be prefaced with the package na me.

UTL_FILE procedures can also raise predefined PL/SQL exceptions s uch as NO_DATA_FOUND or VALUE_ERROR.

FOPEN Function Parameters

Syntax Definitions

Where location Is the operating-system-specific string that specifies the directory or area in which to open the file

filename Is the name of the file, including the extension, without any pathing information

open_mode Is string that specifies how the file is to be opened; Supported values are: ‘r’ read text (use GET_LINE) ‘w’ write text (PUT, PUT_LINE,
NEW_LINE, PUTF, FFLUSH) ‘a’ append text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

The return value is the file handle that is passed to all subsequent routines that operate on the file.

IS_OPEN Function

The function IS_OPEN tests a file handle to see if it identifies an opened file. It returns a Boolean value indicating whet her t he file has been opened but not yet closed.

sal_status.sql

```
CREATE OR REPLACE PROCEDURE sal_status
(p_filedir IN VARCHAR2, p_filename IN VARCHAR2)
IS
v_filehandle UTL_FILE.FILE_TYPE;
CURSOR emp_info IS
SELECT last_name, salary, department_id FROM employees
ORDER BY department_id;
v_newdeptno employees.department_id%TYPE;
v_olddeptno employees.department_id%TYPE := 0;
BEGIN
v_filehandle := UTL_FILE.FOPEN (p_filedir, p_filename,'w');
UTL_FILE.PUTF (v_filehandle,'SALARY REPORT: GENERATED ON %s\n', SYSDATE);
UTL_FILE.NEW_LINE (v_filehandle);
FOR v_emp_rec IN emp_info LOOP
v_newdeptno := v_emp_rec.department_id;
...
...
IF v_newdeptno <> v_olddeptno THEN
UTL_FILE.PUTF (v_filehandle, 'DEPARTMENT: %s\n', v_emp_rec.department_id);
END IF;
UTL_FILE.PUTF (v_filehandle,' EMPLOYEE: %s earns: %s\n', v_emp_rec.last_name, v_emp_rec.salary); v_olddeptno := v_newdeptno;
END LOOP;
UTL_FILE.PUT_LINE (v_filehandle, '*** END OF REPORT ***');
UTL_FILE.FCLOSE (v_filehandle);
```

```
EXCEPTION
WHEN UTL_FILE.INVALID_FILEHANDLE THEN
RAISE_APPLICATION_ERROR (-20001, 'Invalid File.');
WHEN UTL_FILE.WRITE_ERROR THEN
RAISE_APPLICATION_ERROR (-20002, 'Unable to write to
file');
END sal_status;
/
```

Example

The SAL_STATUS procedure creates a report of employees for each department and their salaries. This information is sent to a text file by using the UTL_FILE procedures and functions. The variable v_filehandle uses a type defined in the UTL_FILE package. This package defined type is a record with a field called ID of the BINARY_INTEGER datatype. TYPE file_type IS RECORD (id BINARY_INTEGER); The contents of file_type are private to the UTL_FILE package. Users of the package should not reference or change components of this record. The names of the text fil e and the location for t he text file are provided as parameters to the program.

Note: The file location shown in the above example is defined as va lue of UTL_FILE_DIR in the init.ora file as follows: UTL_FILE_DIR = C:\UTL_FILE.

When reading a complete file in a loop, you need to exit the loop using the NO_DATA_FOUND exception. UTL_FILE output is sent synchronously. DBMS_OUTPUT procedures do not produce output until the procedure is completed.

The UTL_HTTP Package

UTL_HTTP is a package that allows you to make HTTP requests directly from the database. The UTL_HTTP package makes hypertext transfer protocol (HTTP) callouts from PL/SQL and SQL. You can use it to access data on the Int ernet or to call Oracle Web Server Cartridges. By coupling UTL_HTTP with the DBMS_JOBS package, you can easily schedule reoccurring requests be made from your data base server out to the Web.

This package contains two entry point functions: REQUEST and REQUEST_PIECES. Both functions take a string universal resource locator (URL) a s a parameter, contact the site, and return the HTML data obtained from the site. The REQUEST function returns up to the first 2000 bytes of data retrieved from the given URL. The REQUEST_PIECES function returns a PL/SQL table of 2000-byte pieces of the data retrieved from the given URL.

If the HTTP call fails, for a reason such as that the URL is not properly specified in the HTTP syntax then the REQUEST_FAILED exception is ra ised. If initialization of the HTTP-callout subsyst em fails, for a reason such as a lack of available memory, then the INIT_FAILED exception is raised.

If there is no response from the specified URL, then a formatted HTML error message may be returned.

If REQUEST or REQUEST_PIECES fails by returning either an exception or an error message, then verify the URL with a browser, to verify network availability from your machine. If you are behind a firewall, then you need to specify proxy as a parameter, in addition to the URL.

```
SELECT UTL_HTTP.REQUEST('http://www.oracle.com', 'edu-
proxy.us.oracle.com') FROM DUAL;
```

Using the UTL_HTTP Package

The SELECT statement and the output in the slide show how to use the REQUEST function of the UTL_HTTP package to retrieve contents from the URL www.oracle.com. The second parameter to the function indicates the proxy because the client being tested is behind a firewall. The ret ri eved output is in HTML format.

You can use the function in a PL/SQL block as shown below. The function retrieves up to

100 pieces of data, each of a maximum 2000 bytes from the URL. The number of pieces and the total length of the data retrieved are printed.

```
DECLARE
x UTL_HTTP.HTML_PIECES;
BEGIN
x := UTL_HTTP.REQUEST_PIECES('http://www.oracle.com/',100, 'edu-proxy.us.oracle.com');
DBMS_OUTPUT.PUT_LINE(x.COUNT || ' pieces were retrieved.');
DBMS_OUTPUT.PUT_LINE('with total length ');
IFx.COUNT<1THENDBMS_OUTPUT.PUT_LINE('0');
ELSE DBMS_OUTPUT.PUT_LINE((2000*(x.COUNT - 1))+LENGTH(x(x.COUNT)));
END IF;
END;
/
```

Using the UTL_TCP Package

The UTL_TCP package enables PL/SQL applications to communicate with external TCP/IP-based servers using TCP/IP. Because many Internet application protocols are based on TCP/IP, this package is useful to PL/SQL applications that use Internet protocols.

The package contains functions such as:

OPEN_CONNECTION: This function opens a TCP/IP connection with the specified remote and local host and port details. The remote host is the host providing the service. The remote port is the port number on which the service is listening for connections. The local host and port numbers represent those of the host providing the service. The function returns a connection of PL/SQL record type.

CLOSE_CONNECTION: This procedure closes an open TCP/IP connection. It takes the connection details of a previously opened connection as parameter. The procedure CLOSE_ALL_CONNECTIONS closes all open connections.

READ_BINARY()/TEXT()/LINE(): This function receives binary, text, or text line data from a serviceonanopenconnection.

WRITE_BINARY()/TEXT()/LINE(): This function transmits binary, text, or text line message to a serviceonanopenconnection.

Exceptions are raised when buffer size for the input is too small, when generic network error occurs, when no more data is available to read from the connection, or when bad arguments are passed in a function call.

Using Oracle-Supplied Packages

Package	Description
DBMS_ALERT	Provides notification of database events
DBMS_APPLICATION_INFO	Allows application tools and application developers to inform the database of the high level of actions they are currently performing
DBMS_DESCRIBE	Returns a description of the arguments for a stored procedure
DBMS_LOCK	Requests, converts, and releases userlocks, which are managed by the RDBMS lock management services
DBMS_SESSION	Provides access to SQL session information
DBMS_SHARED_POOL	Keeps objects in shared memory
DBMS_TRANSACTION	Controls logical transactions and improves the performance of short, nondistributed transactions
DBMS_UTILITY	Analyzes objects in a particular schema, checks whether the server is running in parallel mode, and returns the time

Oracle-Supplied Packages

The following list summarizes and provides a brief description of the packages supplied with Oracle9i.

Built-in Name **Description**

CALENDAR Provides calendar maintenance functions

DBMS_AQ Provides message queuing as part of the Oracle server; is used to add a message (of a predefined object type) onto a queue or dequeue a message

DBMS_AQADM Is used to perform administrative functions on a queue or queue table for messages of a predefined object type

DBMS_DDL Is used to embed the equivalent of the SQL commands ALTER, COMPILE,andANALYZE within your PL/SQL programs

DBMS_DEBUG A PL/SQL API to the PL/SQL debugger layer, Probe, in theOracleserver

DBSM_DEFER
DBMS_DEFER_QUERY
DBMS_DEFER_SYS

Is used to build and administer deferred remote procedure calls (use of this feature requires the Replication Option)

DBMS_DISTRIBRUTED_
TRUST_ADMIN Is used to maintain the Trusted Servers list, which is used in conjunction with the list at the central authority to determine whether a privileged database link from a particular server can be accepted

DBMS_HS Is used to administer heterogeneous services by registering or dropping distributed external procedures, remote libraries, and non-Oracle systems (you use dbms_hs to create or drop some initialization variables for non-Oracle systems)

DBMS_HS_EXTPROC Enables heterogeneous services to establish security for distributed external procedures

DBMS_HS_PASSTHROUGH Enables heterogeneous services to send pass-through SQL statements to non-Oracle systems

DBMS_IOT Is used to schedule administrative procedures that you want performed at periodic intervals; is also the interface for the job queue

DBMS_LOB Provides general purpose routines for operations on Oracle large objects (LOBs) data types: BLOB, CLOB (read only) and BFILES (read-only)

DBMS_LOGMNR Provides functions to initialize and run the log reader

DBMS_LOGMNR_D Queries the dictionary tables of the current database, and creates a text based file containing their contents

DBMS_OFFLINE_OG Provides public APIs for offline instantiation of master groups

DBMS_OFFLINE_SNAPSHOT Provides public APIs for offline instantiation of snapshots

DBMS_OLAP Provides procedures for summaries, dimensions, and query rewrites

DBMS_ORACLE_TRACE_AGENT Provides client callable interfaces to the Oracle TRACE instrumentation within the Oracle7 server

DBMS_ORACLE_TRACE_USER Provides public access to the Oracle7 release server Oracle TRACE instrumentation for the calling user

DBMS_OUTPUT Accumulates information in a buffer so that it can be retrieved out later

DBMS_PCLXUTIL Provides intrapartition parallelism for creating partition- wise local ind exes

DBMS_PIPE Provides a DBMS pipe service that enables messages to be sent between sessions

DBMS_PROFILER Provid es a Probe Profiler API to profile existing PL/SQL applications and identify performance bottlenecks

DBMS_RANDOM Provides a built-in random number generator

DBMS_RECTIFIER_DIFF Provides APIs used to detect and resolve data inconsistencies between two replicated sites

DBMS_REFRESH Is used to create groups of snapshots that can be refreshed together to a transactionally consistent point in time; requires the Distributed option

DBMS_REPAIR Provides data corruption repair procedures

DBMS_REPCAT Provides routines to administer and update the replication catalog and environment; requires the Replication option

DBMS_REPCAT_ADMIN Is used to create users with the privileges needed by the symmetric replication facility; requires the Replication option

DBMS_REPCAT_INSTATIATE Instantiates d eployment templates; requires the Replication op tion

DBMS_REPCAT_RGT Controls the maintenance and definition of refresh group templates; req uires the R eplication option

DBMS_REPUTIL Provides routines to generate shadow tables, triggers, and packages for table replication

DBMS_RESOURCE_MANAGER Maintains plans, consumer groups, and plan directives; it also provides semantics so that you may group together changes to the plan schema

DBMS_RESOURCE_MANAGER_PRIVS Maintains privileges associated with resource consumer groups

DBMS_RLS Provides row-level security administrative interface

DBMS_ROWID Is used to get information about ROWIDs, including the data block number, the object number, and other components

DBMS_SESSION Enables programmatic use of the SQL ALTER SESSION statement as well as ot her session-level commands

DBMS_SHARED_POOL Is used to keep objects in shared memory, so that they are not aged out with the normal LRU mechanism

DBMS_SNAPSHOT Is used to refresh one or more snapshots that are not pa rt of the same refresh group and purge logs; use of this feature requires the Distributed option

DBMS_SPACE Provides segment space information not available through standard views

DBMS_SPACE_ADMIN Provides tablespace and segment space administration not available through standard SQL

DSMS_SQL Is used to write stored procedure and anonymous PL/SQL blocks using dynamic SQL; also used to parse any DML or DDL statement

DBMS_STANDARD Provides language facilities that help your application interact with the Oracle server

DBMS_STATS Provides a mechanism for users to view and modify optimizer statistics gathered for databa se objects

DBMS_TRACE Provides routines to start and stop PL/SQL tracing

DBMS_TRANSACTION Provides procedures for a programmatic interface to transaction management

DBMS_TTS ` Checks whether if the transportable set is self-contained

DBMS_UTILITY Provides functionality for managing procedures, reporting errors, and other information

DEBUG_EXTPROC Is used to debug external procedures on platforms with debuggers that can a ttac h to a running process

OUTLN_PKG Provides the interface for procedures and functions associated with management of stored outlines

PLITBLM Handles index-table operations

SDO_ADMIN Provides functions implementing spatial index creation and maintenance for spatial objects

SDO_GEOM Provides functions implementing geometric operations on spatial objects

SDO_MIGRATE Provides functions for migrating spatial data from release 7.3.3 and 7.3.4 to 8.1.x

SDO_TUNE Provides functions for selecting parameters that determine the behavior of the spatial indexing scheme used in the Spa tial Cartridge

STANDARD Declares types, exceptions, and subprograms that are available automa tically to every PL/SQL program

TIMESERIES Provides functions that perform operations, such as extraction, retrieval, arithmetic, and aggregation, on time series data

TIMESCALE Provides scale-up and scale-down functions

TSTOOLS Provides administrative tools procedures

UTL_COLL Enables PL/SQL programs to use collection locators to query and update

UTL_FILE Enables your PL/SQL programs to read and write operating system (OS) text files and provides a restricted version of standard OS stream file I/O

UTL_HTTP Enables HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges

UTL_PG Provides functions for converting COBOL numeric data into Oracle numbers and Oracle numbers into COBOL numeric data

UTL_RAW Provides SQL functions for RAW data types that concatenate, obtain substring, and so on, to and from RAW data types

UTL_REF Enables a PL/SQL program to access an object by providing a reference to the object

VIR_PKG Provides analytical and conversion functions for visual information retrieval

Manipulating Large Objects

Objectives

After completing this lesson, you should be able to do the following:

- Compare and contrast LONG and large object (LOB) **data types**
- Create and maintain LOB data types
- Differentiate between internal and external LOBs
- Use the DBMS_LOB PL/SQL package
- Describe the use of temporary LOBs

Overview

A LOB is a data type that is used to store large, unstructured data such as text, graphic images, video clippings, and so on. Structured data such as a customer record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger. Also, multimedia data may reside on operating system (OS)
files, which may need to be accessed from a database.
There are four large object data types :

- BLOB represents a binary large object, such as a video clip.
- CLOB represents a character large obj ect.
- NCLOB represents a multibyte cha racter large object.
- BFILE represents a binary file stored in an operating system binary file outside the database. The BFILE column or a ttribute stores a file locator that points t o the external file.
- LOBs are characterized in two ways, according to their interpretation by the Oracle server (binary or character) and their storage aspects. LOBs can be stored internally (inside the dat abase) or in host files.

There are two categories of LOBs:

- Internal LOBs(CLOB, NCLOB, BLOB) are stored in the database.
- External files (BFILE) are stored outside the database.

The Oracle9i Server performs implicit conversion between CLOB and VARCHAR2 data types. The other implicit conversions between LOBs are not possible. For example, if the user creates a table T with a CLOB column a nd a tableS with a BLOB column, the data is not directly transferable between these two columns.
BFILEs can be accessed only in read-only mode from an Oracle server.

LONG and LOB Data Types

LONG and LONG RAW dat a types wer e previ ously used for unstructured data, such as binary images, documents, or geographical information. These data types are superseded by the LOB data types. Oracle 9i provides a LONG-to-LOB API to migrate from LONG columns t o LOB columns.It is beneficial to discuss LOB functionality in comparison to the older types. In the bulleted list below,
LONGs refers to LONG and LONG RAW,andLOBs refers to all LOB data types:

- A table can have multiple LOB columns and object type attributes. A table can have only one LONG column.
- The maximum si ze of LONGs is 2 gigabyt es; LOBs can be up to 4 gigabytes.
- LOBs return the locator; LONGs return the data.
- LOBs store a locator in t he table and the data in a different segment, unl ess the data is less than 4,000 bytes; LONGs store all data in the same data block. In addition, LOBs allow data to be st ored in a separate segment and tablespace, or in a host file.
- LOBs can be object type attributes; LONGs cannot.

- LOBs support random piecewise access to the da ta through a file-like interfa ce; LONGsare restricted to sequent ial piecewise access.

The TO_LOB functioncanbeusedtocovertLONG and LONG RAW values in a column to LOB values. You use this in the SELECT list of a subquery in an INSERT stat ement.

Components of a LOB

There are two distinct parts of a LOB:

- LOB value: The data that constitutes the real object being stored.
- LOB locator: A pointer to the location of the LOB value stored in the data base.

Regardless of where the value of the LOB is stored, a locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value.

A LOB column does not contain the dat a; it conta ins the locator of the LOB value. When a user creates an internal LOB, the value is stored in the LOB segment and a locat or to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table. External LOBs store the data out side the database, so only a locat or to the LOB valueisstoredinthetable.

To access and manipulate LOBs without SQL DML, you must create a LOB locator. Programmatic int erfaces operate on the LOB values, using these locat ors in a manner si milar to operating syst em file handles.

Features of Internal LOBs

The int ernal LOB is stored inside the Oracle server. A BLOB, NCLOB,orCLOB can be one of the following:

- An attribute of a user-defined type
- A column in a table
- A bind or host varia ble
- A PL/SQL variable, parameter, or result

Int ernal LOBs can take advantage of Oracle feat ures such as:

- Concurrency mecha nisms
- Redo logging and recovery mechanisms
- Transactions with commit or rollbacks

The BLOB data type is int erpreted by the Oracle server as a bitstrea m, similar to the LONG RAW data type.

The CLOB data type is interpreted as a single-byte character stream.

The NCLOB data type is interpreted as a multiple-byte character stream, based on the byte length of the database national character set.

How to Manage LOBs

Use the following method to manage an internal LOB:

1. Create and populate the table containing the LOB data type.
2. D eclare and initialize the LOB locator in the program.
3. Use SELECT FOR UPDATE to lock the row containing the LOB into the LOB locator.
4. Manipulate the LOB with DBMS_LOB package procedures, OCI calls, Oracle Objects for OLE, Oracle precompilers, or JDBC using the LOB locator as a reference to the LOB value. You can also manage LOBs through SQL.
5. Use the COMMIT command to make any changes permanent.

What Are BFILEs?

BFILEs are external large objects (LOBs) stored in operating system files outside of the database tablespaces. The Oracle SQL data type to support these large objects is BFILE.TheBFILE da ta type stores a locator to the physical file. A BFILE can be in GIF, JPEG, MPEG, MPEG2, t ext, or other formats. The External LOBs may be located on hard disks, CDROMs, photo CDs, or any such devi ce, but a single LO B ca nnot extend from one device to another. The BFILE data t ype is available so that database users can access the external file system. The Oracle9i server provides for:

- Definition of BFILE objects
- Association of BFILE objects to corresponding external files
- SecurityforBFILEs

The rest of the operat ions required to use BFILEs are possible through the DBMS_LOB packa ge a nd the Oracle Call Interface. BFILEs are read-only, so they do not participate in transactions. Any support for integrity and durability must be provided by the operating system. The user must create the file and place it in the appropriate
direct ory, giving the Oracle process privileges to read the fi le. When the LOB is deleted, the Oracl e server does not delete the file. The administration of the actual files and the OS directory structures to house the files is the responsibility of the database administrator (DBA), system administrator, or user. The maximum size of an external large object is operating system dependent but cannot exceed four gigabyt es.

Note: BFILEs are available in the Oracle8 database and in later releases.

Securing BFILEs

Unaut henticated access to files on a server presents a security risk. The Oracl e9i Server can act as a
security mechanism t o shiel d the operating system from unsecured access while removing the need t o
manage additional user accounts on an enterprise computer system.

File Location and Access Privileges

The file must reside on the machine where the da tabase exists. A time-out to read a nonexistent BFILE is based on the operating system value. You can read a BFILE in the same way as you read an internal LOB. However, there could be restrictions related to the file itself, such as:

- Access permissions
- File system space li mits
- Non-Oracle ma nipulations of files
- OS maximum file size

The Oracle9i RDBMS does not provide transactional support on BFILEs. Any support for integrity and durabilit y must be provided by t he underl ying file system and the OS. Oracle backup and recovery methods support only the LOB locators, not the physical BFILEs.

A New Database Object: DIRECTORY

A DIRECTORY is a nonschema database object that provides for administration of access and usage of BFILEsinanOracle9i Server.

A DIRECTORY specifies an alias for a directory on the file system of the server under which a BFILE is located. By granting suita ble privileges for these items to users, you can provide secur e access to files in the corresponding directories on a user-by-user basis (certain directories can be ma de read-only, inaccessible, and so on).

Further, these directory aliases ca n be used while referring to files (open, close, read, and so on) in PL/SQL and OCI. This provides application abstraction from hard-coded path names, and gives flexibility in portably managing file locations.

The DIRECTORY object is owned by SYS and created by the DBA (or a user with CREATE ANY DIRECTORY privil ege). Directory objects have object privileges, unlike any other nonschema object. Privileges to the DIRECTORY object can be granted and revoked. Logical path names are not supported. The permissions for the actual directory are operating system dependent. They may differ from those defined for the DIRECTORY object and could change after the creation of the DIRECTORY object.

Guidelines for Creating Directory Objects

To associate an operating system file to a BFILE, you should first create a DIRECTORY object tha t is an alias for the full pathname to the operating system file.

Create DIRECTORY objects by using the following guidelines:

• Directories should point to paths that do not contain database files, because ta mpering with these files could corrupt the database. Currently, only the READ privilege ca n be given for a DIRECTORY object.

• The syst em privileges CREATE ANY DIRECTORY and DROP ANY DIRECTORY should be used carefully and not granted to users indiscriminately.

• DIRECTORY objects are not schema objects; all are owned by SYS.

• Create the dir ectory paths with a ppropriate permissions on the OS prior to creating the DIRECTORY object. Oracle does not create the OS path.

If you migrate the database to a different operating system, you ma y need to change the path value of the DIRECTORY object.

The DIRECTORY object information that you creat e by using the CREATE DIRECTORY command is stored in the data dictionary views DBA_DIRECTORIES and ALL_DIRECTORIES.

How to Manage BFILEs

Use the following method to manage the BFILE and DIRECTORY objects:

1. Create the OS directory (as an Oracle user) and set permissions s o that the Oracle server can read the contents of the OS directory. Load files into the the OS directory.
2. Create a table containing the BFILE data type in the Oracle server.
3. Create the DIRECTORY object.
4. Grant the READ privilege to it.
5. Insert rows into the table using the BFILENAME function and associate the OS files with the corresponding row and column intersection.
6. Declare and initialize the LOB locator in a program.
7. Select the row and column containing the BFILE into the LOB locator.
8. Read the BFILE withanOCIoraDBMS_LOB function, using the locator as a reference to the file.

PreparingtoUseBFILEs

In order to use a BFILE within an Oracle table, you need to have a table with a column of BFILE type. For the Oracle server to access an external file, the server needs to know the location of the file on the operating system. The DIRECTORY object provides the means to specify the location of the BFILEs. Use the
CREATE DIRECTORY command to specify the pointer to the location where your BFILEs are st ored. You need the CREATE ANY DIRECTORY privilege.

Syntax Definition: CREATE DIRECTORY dir_name AS os_path;

dir_name is the name of the directory database object

Where:

os_path is the location of the BFILEs

The following commands set up a pointer to BFILEs in the system directory / $HOME/LOG_FILES and give users the privilege to read the BFILEs from the directory.

CREATE OR REPLACE DIRECTORY log_files AS '/$HOME/LOG_FILES';

```
GRANT READ ON DIRECTORY log_files TO PUBLIC;
```

In a session, the number of BFILEs that can be opened in one session is limit ed by the parameter SESSION_MAX_OPEN_FILES. This paramet er is set in the init.ora file. Its default value is 10.

The BFILENAME Function

BFILENAME is a built-in function that initializes a BFILE column to point to an external file. Use the BFILENAME function as part of an INSERT statement to initialize a BFILE column by associating it with a physical file in the server file system. You can use the UPDATE statement to change t he reference target of the BFILE.ABFILE can be initialized to NULL and upda ted la ter by using the BFILENAME function.

Syntax Definitions

Where: directory_alias is the name of the DIRECTORY database object

filename is the name of the BFILE to be read

Example

```
UPDATE employees
SET emp_video = BFILENAME('LOG_FILES', 'King.avi') WHERE employee_id = 100;
```

Once physical fil es are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILEs, and so they cannot be updated or deleted through BFILEs.

Loading BFILEs

```
CREATE OR REPLACE PROCEDURE load_emp_bfile (p_file_loc IN VARCHAR2) IS v_file BFILE; v_filename VARCHAR2(16);
CURSOR emp_cursor IS
SELECT first_name FROM employees WHERE department_id = 60 FOR UPDATE;
BEGIN
FOR emp_record IN emp_cursor LOOP
v_filename := emp_record.first_name || '.bmp';
v_file := BFILENAME(p_file_loc, v_filename);
DBMS_LOB.FILEOPEN(v_file);
UPDATE employees SET emp_video = v_file WHERE CURRENT OF emp_cursor;
DBMS_OUTPUT.PUT_LINE('LOADED FILE: '||v_filename
|| ' SIZE: ' || DBMS_LOB.GETLENGTH(v_file));
DBMS_LOB.FILECLOSE(v_file);
END LOOP;
END load_emp_bfile;
/
```

Load a BFILE pointer to an image of each employee into the EMPLOYEES table by using the DBMS_LOB package. The images are .bmp files stored in the / home/LOG_FILES directory.

Using DBMS_LOB.FILEEXISTS

This function finds out whether a given BFILE locator points to a file that actua lly exists on the server's file system. This is the specification for the function:

Syntax Definitions

```
FUNCTION DBMS_LOB.FILEEXISTS (file_loc IN BFILE)
RETURN INTEGER;
```

Where: file_loc is name of the BFILE locator

RETURN INTEGER returns 0 if the physical file does not exist returns 1 if the physical file exists

If the FILE_LOC parameter contains an invalid value, one of three exceptions may be raised.

Exception Name **Description**

NOEXIST_DIRECTORY The directory does not exist.
NOPRIV_DIRECTORY You do not have privileges for the directory.
INVALID_DIRECTORY The directory w as invalidated after the file w as opened.

Migrating from LONG to LOB

Oracle9i Server supports the LONG-to-LOB migration using API.

Data migration: Where existing tables that contain LONG columns need to be moved to use LOB columns. This can be done using the ALTER TABLE command. In Oracle8i,anoperatornamedTO_LOB hadtobe used to copy a LONG to a LOB. You can use the syntax shown to:

•ModifyaLONG column to a CLOB or an NCLOB column
•ModifyaLONG RAW column to a BLOB column

The constraints of the LONG column (NULL and NOT-NULL are the only allowed constraints) are maintained for the new LOB columns. The default value specified for the LONG column is also copied to the new LOB column.

For example, if you ha d a ta ble with the following definition:

CREATE TABLE Long_tab (id NUMBER, long_col LONG);

you can change the LONG_COL column i n table LONG_TAB to the CLOB data type a s foll ows:

ALTER TABLE Long_tab MODIFY (long_col CLOB);

Application Migration: Where the existing LONG applications change for using LOBs. You can use SQL and PL/SQL to access LONGsandLOBs. This API is provided for both OCI and P L/S QL.

With the new LONG-to-LOB API int roduced in Oracle9i,datafromCLOB and BLOB columns can be referenced by regular SQL and PL/SQL statements.

Implicit assignment and parameter passing: The LONG-to-LOB migra tion API supports assigning a CLOB (BLOB)variabletoaLONG (LONG RAW)oraVARCHAR2(RAW) variable, and vice versa.

Explic it conversion functions: In PL/SQL, the following two new explicit conversion functions have been added in Oracle9i to convert other data types to CLOB and BLOB as part of LONG-to-LOB migration:

• TO_CLOB() converts LONG, VARCHAR2,andCHAR to CLOB
• TO_BLOB() converts LONG RAW and RAW to BLOB

TO_CHAR() is ena bled to convert a CLOB to a CHAR type.

Function and procedure parameter passing: This allows all the user-defined procedures and functions to use CLOBsandBLOBs as actual parameters where VARCHAR2, LONG, RAW,andLONG RAW are formal parameters, and vice versa.

Accessing in SQL and PL/SQL built-in functions and operators: A CLOB canbepassedtoSQLand PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2.OrtheVARCHAR2 variablecanbepassedintoDBMS_LOB APIs acting like a LOB locat or.

The DBMS_LOB Package

In rel eases prior to Oracle9i, you need to use the DBMS_LOB package for retrieving data from LOBs. To create the DBMS_LOB package, the dbmslob.sql and prvtlob.plb scripts must be executed as SYS.Thecatproc.sql script executes the scripts. Then users can be granted appropriate privileges to use the package.

The package does not support any concurrency control mechanism for BFILE operations. The user is responsible for locking the row cont aining the dest ination internal LOB before calling any subprograms that involve writing to the LOB value. These DBMS_LOB routines do not implicitly lock the row containing the LOB.

Two constants are used in the specification of procedures in this package: LOBMAXSIZE and FILE_READONLY. These constants are used in the procedures and funct ions of DBMS_LOB;for exa mple, you can use them to achieve the maximum possible level of purity so that they can be used in SQL expressions.

Using the DBMS_LOB Routines

Functions and procedures in this package can be broadly classified i nto two types: mutat ors or observers.
Mutators can modify LOB values, whereas observers can only read LOB values.
•Mutators:APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL,and FILEOPEN
• Observers: COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN
APPEND Append the contents of the source LOB to the destination LOB
COPY Copy all or part of the source LOB to the destination LOB
ERASE Erase all or part of a LOB
LOADFROMFILE Load BFILE data into an internal LOB
TRIM Trim the LOB value to a specified shorter length
WRITE WritedatatotheLOB from a specified offset
GETLENGTH Get the length of the LOB value
INSTR Return the matchi ng position of the nth occurrence of the pattern in the LOB
READ Read data from t he LOB starting at the specified offset
SUBSTR Return part of the LOB value starting at the specified offset
FILECLOSE Close the file
FILECLOSEALL Close all previously opened files
FILEEXISTS Check if the file exists on the server
FILEGETNAME Get the directory alias and file name
FILEISOPEN Check if the fil e was opened using the input BFILE locators
FILEOPEN Open a file

Using the DBMS_LOB Routines

All functions in the DBMS_LOB packa ge return NULL if any input parameters are NULL . All mutator procedures in the DBMS_LOB package raise an exception if the destination LOB /BFILE is input as NULL. Only positive, absolute offs ets are a llowed. They repres ent the number of bytes or characters from the beginning of LOB data from which to start the operation. Negative offsets and ranges observed in SQL string functions and opera tors are not allowed. Corresponding exceptions are raised upon violation. The default value for an offset is 1, which indicates the first byte or chara cter in the LOB value. S imilarly, only natural number values are allowed for the amount (BUFSIZ) parameter. Negative values are not allowed.

DBMS_LOB.READ

Call the READ procedure to read and return piecewise a specified AMOUNT of data from a given LOB, starting from OFFSET. An exception is raised when no more data remains to be read from the source LOB. The value returned in AMOUNT will be less than the one specified, if the end of t he LOB is reached before the specified number of byt es or characters could be read. In the case of CLOBs, the character set of data in BUFFER isthesameas thatintheLOB.
PL/SQL allows a maximum length of 32767 for RAW and VARCHAR2 parameters.Makesurethe allocated system resources are adequate to support these buffer sizes for the given number of user sessions. Otherwise, the Oracle server raises the appropriate memory exceptions.
Note: BLOB and BFILE return RAW; the others return VARCHAR2.
DBMS_LOB.WRITE

Call the WRITE procedure to write piecewise a specified AMOUNT of data into a given LOB,fromthe user-specified BUFFER, starting from an absolute OFFSET from the beginning of the LOB value.Make sure (especially with multibyte characters) that the amount in bytes corresponds to the amount of buffer data. WRITE has no means of checking whether they match, and will writ e AMOUNT bytes of the buffer contents into the LOB.

Adding LOB Columns to a Table

LOB columns are defined by way of SQL data definition language (DDL), as in the ALTER TABLE statement . The cont ents of a LOB column is stored in t he LOB segment , whereas the column in the table contains only a reference to that specific storage area, called the LOB locator. In P L/SQL you can define a variable of type LOB, which contains only the value of the LOB locator.

Populating LOB Columns

You can insert a value directly into a LOB column by using host variables in SQL or in P L/S QL, 3GL-embedded SQL, or OCI.

You can use the special functions EMPTY_BLOB and EMPTY_CLOB in INSERT or UPDATE statement s of SQL DML to initialize a NULL or non-NULL internal LOB to empty. These are available a s special functions in Oracle SQL DML, and are not part of the DBMS_LOB package.

Before you can start writing data to a n internal LOB using OCI or t he DBMS_LOB package, the LOB column must be made nonnull, that is, it must contain a locator that points to an empty or populated LOB value. You can init ia lize a BLOB column's value to empty by using the funct ion EMPTY_BLOB in the VALUES clause of an INSERT statement. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB.

The result of using the function EMPTY_CLOB() or EMPTY_BLOB() means tha t the LOB is initialized, but not populated with data. To populate the LOB column, you can use an update stat ement. You can use an INSERT statement to insert a new row and populate the LOB column at the same time.When you create a LOB instance, the Oracle server creates and places a locator to the out-of-line LOB value in the LOB column of a particular row in the table. SQL, OCI, and ot her programmatic interfaces operate on LOBs through these locators.

The EMPTY_B/CLOB() functioncanbeusedasaDEFAULT column constraint, as in the example below. This initializes the LOB columns wit h locators.

```
CREATE TABLE emp_hiredata
(employee_id NUMBER(6),
first_name VARCHAR2(20),
last_name VARCHAR2(25),
resume CLOB DEFAULT EMPTY_CLOB(),
picture BLOB DEFAULT EMPTY_BLOB());
```

Updating LOB by Using SQL

You can update a LOB column by setting it to another LOB value, to NULL, or by using the empty functi on appropriate for the LOB data type (EMPTY_CLOB() or EMPTY_BLOB()). You can update the LOB using a bind va riable in embedded SQL, the value of which may be NULL,empty,orpopulated. When you set one LOB equal to another, a new copy of the LOB value is created. These actions do not require a SELECT FOR UPDATE statement. You must lock the row prior to the update only when updating a piece of the LOB.

```
DECLARE
lobloc CLOB; -- serves as the LOB locator
text VARCHAR2(32767):='Resigned: 5 August 2000';
amount NUMBER ; -- amount to be written
offset INTEGER; -- where to start writing
BEGIN
SELECT resume INTO lobloc FROM employees
WHERE employee_id = 405 FOR UPDATE;
offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
amount := length(text);
DBMS_LOB.WRITE (lobloc, amount, offset, text );
text := ' Resigned: 30 September 2000';
SELECT resume INTO lobloc FROM employees
WHERE employee_id = 170 FOR UPDATE;
amount := length(text);
DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
COMMIT;
END;
```

Updating LOB by Using DBMS_LOB in PL/SQL

In the example , the LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text you want to add. The SELECT FOR UPDATE statement locks the row and returns the LOB locator for the RESUME LOB column. Finally, the PL/SQL package procedure WRITE is called to write the text into the LOB value at the specified offset. WRITEAPPEND appends to the existing LOB value.

This shows how to fetch a CLOB column in releases before Oracle9i. In those releases, it was not possible to fetch a CLOB column directly into a character column. The column value needed to be bound to a LOB locator,whichisaccessedbytheDBMS_LOB package. An example later shows that you can directly fetch a CLOB column by binding it to a character variable.

Note: In versions prior to Oracle9i, Oracle did not allow LOBsintheWHERE clause of UPDATE and SELECT. Now SQL functions of LOBs are allowed in predicates of WHERE. An example is shown later

```
SELEC T employee_id, last_name , resume -- C LOB
FROM employees
WHERE employee_id IN (405, 170);
```

Selecting CLOB Va lues by Using SQL

It is possible to see the data in a CLOB column by using a SELECT statement. It is not possibl e to see the data in a BLOB or BFILE column by using a SELECT statement in iSQL*Plus. You have to use a tool that can display binary informati on for a BLOB, a s well as the releva nt software for a BFILE; for example, you can use Oracle Forms.

DBMS_LOB.SUBSTR

Use DBMS_LOB.SUBSTR to display part of a LOB. It is similar in functionality to t he SQL function SUBSTR.

DBMS_LOB.INSTR

Use DBMS_LOB.INSTR to search for informati on withi n the LOB. This function returns the numerical posit ion of the information.

Note: Starting with Ora cle9i, you can also use SQL functions SUBSTR and INSTR to perform the operat ions shown in the sli de.

```
DECLARE
text VARCHA R2(4 001);
```

```
BEGIN
SELECT resum e IN TO text FROM employe es
WHERE employ ee_i d = 170;
DBMS_OUTPUT. PUT_ LINE('text is: '|| text);
END;
/
```

Selecting CLOB Values in PL/SQL

This shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2 in Oracle9i. The value of the column RESUME, when selected into a VARCHAR2 variable TEXT,is implicitly converted. In prior releases, to access a CLOB column, first you must to retrieve the CLOB column value into a CLOB variable and specify the amount and offset size. Then you use the DBMS_LOB packagetoreadthe selected value. The code using DBMS_LOB is as follows:

```
DECLARE
rlob clob;
text VARCHAR2(4001);
amtnumber:=4001;
offset number := 1;
BEGIN
SELECT resume INTO rlob FROM employees
WHERE employee_id = 170;
DBMS_LOB.READ(rlob, amt, offset, text);
DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

Removing LOBs

A LOB inst ance can be delet ed (destroyed) using appropriate SQL DML statements. The SQL statement DELETE deletesarowanditsassociatedinternalLOB value. To preserve the row and destroy only the reference to the LOB, you must update the row, by replacing the LOB column value with NULL or an empty string, or by using the EMPTY_B/CLOB() function.

Note: Replacing a column value with NULL and using EMPTY_B/CLOB are not the same. Using NULL sets the value to null, using EMPTY_B/CLOB ensures there is nothing in the column. A LOB is destroyed when the row conta ining the LOB column is deleted when the table is dropped or truncated, or implicitly when all the LOB data is updated. You must explicitly remove the file associated with a BFILE using operating system commands. To erase part of an interna l LOB, you can use DBMS_LOB.ERASE.

Temporary LOBs

Temporary LOBs provide an interface to support the creation a nd deletion of LOBs that act like l ocal variables. Temporary LOBscanbeBLOBs, CLOBs, or NCLOBs.

Features of temporary LOBs:

- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBs are faster than persistent LOBs because they do not generate any redo or rollback information.
- Temporary LOBs lookup is localized to each user’s own session; only the user who creates a temporary LOB can access it, and all temporary LOBs are deleted at the end of the session in which they were created.
- You can creat e a temporary LOB using DBMS_LOB.CREATETEMPORARY.

Temporary LOBs are useful when you want to perform some transforma tional operation on a LOB,for example, changing an image type from GIF to JPEG. A temporary LOB is empty when created and does not support the EMPTY_B/CLOB functions.
Use the DBMS_LOB package to use and manipulate temporary LOBs.

```
CREATE OR REPLACE PR OCEDURE IsTempLOBOpen
(p_lob_loc IN OUT BLOB, p_retval OUT INTEGER)
IS
BEGIN
-- create a tem por ary LOB
DBMS_LOB.CREATE TEM PORARY (p_lob_loc, TRUE);
-- see if the L OB is open: returns 1 if open
p_retval := DBM S_L OB.ISOPEN (p_lob_loc);
DBMS_OUTPUT.PUT _LI NE ('The file returned a value
....' || p_retval);
-- free the tem por ary LOB
DBMS_LOB.FREETE MPO RARY (p_lob_loc);
END;
/
```

Creating a Temporary LOB

The example shows a user-defined PL/SQL procedur e, IsTempLOBOpen,thatcreatesa temporary LOB. This procedure accepts a LOB locator as input, creates a temporary LOB, opens it , and tests whether the LOB is open.
The IsTempLOBOpen procedure uses the procedures and functions from the DBMS_LOB package as follows:
•TheCREATETEMPORARY procedure is used to create the t emporary LOB.
•TheISOPEN function is used to test whether a LOB is open: this funct ion returns the value 1 if the LOB is open.
•TheFREETEMPORARY procedure is used to free the temporary LOB; memory increases incrementall y as the number of temporary LOBs grows, and you can reuse temporary LOB space in your session by explicitly freeing temporary LOBs.

Creating Database Triggers

Objectives

After completing this lesson, you should be able to do the following:

- Describe different types of triggers
- Describe database triggers and their use
- Create database triggers
- Describe database trigger firing rules
- Remove database triggers

Types of Triggers

Application triggers execute implicitly whenever a particula r data manipulation language (DML) event occurs within an application. Database triggers execute implicitly when a data event such as DML on a table (an INSERT, UPDATE,orDELETE triggering statement), an INSTEAD OF trigger on a view, or data definition langua ge (DDL) statements such as CREATE and ALTER are issued, no matter which user is connect ed or which applicat ion is used. Database triggers also execute implicit ly when some user actions or database system actions occur, for example, when a user logs on, or the DBA shut downs the database.

Note: Database triggers can be defined on tables and on views. If a DML operation is issued on a view, the INSTEAD OF trigger defines what actions take place. If these actions include DML operations on tables, then any t riggers on the base tables are fired. Database triggers can be syst em t riggers on a database or a schema. With a database, triggers fire for each event for all users; with a schema, triggers fire for each event for that specific user.

Guidelines for Designing Triggers

• Use triggers to guara ntee that when a specific operation is performed, related actions are performed.

• Use database triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or application issues the statement.

• Do not defi ne triggers to duplicat e or replace the functionalit y already built into the Oracle database. For example do not define triggers to implement integrity rules that can be done by using declarative const raints. An easy way to remember the design order for a business rule is t o:

- Use built-in constraints in the Oracle server such as, primary key, foreign key and so on

- Develop a data base trigger or develop an application such as a servlet or Enterprise JavaBean (EJB) on your middle ti er

- Use a present ation int erface such as Oracle Forms, dynamic HTML, Java ServerPages (JSP) and so on, if you cannot develop your business rule as mentioned above, which might be a presentation rule.

• The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications. Only use triggers when necessary, and beware of recursive and cascading effects.

• If the logic for the trigger is very lengthy, create stored proc edures with the logic and invoke them in the trigger body.

• Note that database triggers fire for every user each time the event occurs on which the trigger is created.

Example of a Database Trigger

The database trigger CHECK_SAL checks salary values whenever any application tries to insert a row int o the EMPLOYEES table. Values that are out of range according to the job category can be reject ed, or can be a llowed and recorded in an audit table.

Database Trigger

Before coding the trigger body, decide on the values of the components of the trigger: the trigger timing, the triggering event, a nd the trigger type.

Part	**Description**	**Possible Values**
Trigger timing	When the trigger fires in relation to the triggering event	BEFORE AFTER INSTEAD OF
triggering event	Which data manipulation operation on the table or view causes the trigger to fire	INSERT UPDATE DELETE
Trigger type	How many times the trigger body executes	Statement Row
Trigger body	What action the trigger performs	Complete PL/SQL block

If multiple triggers are defined for a table, be aware that the order in which multiple triggers of the same type fire is arbitrary. To ensure that triggers of the same type are fired in a particular order, consolidate the triggers into one trigger that calls separate proc edures in the desired order.

BEFORE Triggers

This type of trigger is frequently used in the following situations:

- To determine whether that triggering statement should be allowed to complete. (This situation enables you to eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing a triggering INSERT or UPDATE statement.
- To initialize global variables or flags, and to validate complex business rules.

AFTER Triggers

This type of trigger is frequently used in the following situations:

- To complete the triggering statement before executing the triggering action.
- To perform different actions on the same triggering statement if a BEFORE trigger is already present.

INSTEAD OF Triggers

This type of trigger is used to provide a transparent way of modifying views that cannot be modified dir ectly through SQL DML statements because the view is not inherently modifiable.

You can write INSERT, UPDATE,andDELETE statements against the view. The INSTEAD OF trigger works invisibly in the background performing the action coded in the trigger body directly on the underlying tables.

The Triggering Event

The triggering event or statement can be an INSERT, UPDATE,orDELETE stat ement on a table.

- When the triggering event is an UPDATE statement, you ca n include a column list to identify which c olumns must be c hanged to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement, because they always affect ent ire rows.
- The triggering event can contain one, two, or all three of these DML operations.

. . . INSERT or UPDATE or DELETE
. . . INSERT or UPDATE OF job_id . . .

Statement Triggers and Row Triggers

You can specify that the trigger will be executed once for every row affected by the triggering stat ement (such as a multiple row UPDATE) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event , even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself: for example, a trigger that performs a complex security check on the current user.

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of rows that are affected or on data provided by the triggering event itself.

Trigger Body

The trigger action defines what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors,exceptions, and so on. You can also call a PL/ SQL procedure or a Java procedure. Additionally, row triggers use correlation names t o access t he old and new column values of the row beingprocessedbythetrigger.

Note: The size of a trigger cannot be more than 32K.

Creating Row or Statement Triggers

Create a statement trigger or a row trigger based on t he requirement that t he trigger must fire once for each row affected by the triggering statement, or just once for the t riggering statement, regardless of the number of rows affected. When the triggeri ng da ta manipulation statement affects a single row, both t he statement trigger and the row trigger fire exactly once.

When the triggering data manipulation statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Syntax for Creating a Statement Trigger

trigger name Is the name of the trigger

timing Indicates the time when the trigger fires in relation to the triggering event:

BEFORE

AFTER

event Identifies the data manipulation operation that causes the trigger to fire:

INSERT

UPDATE [OF column]

DELETE

table/view_name Indicates the table associated with the trigger

trigger body Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN,ending

with END, or a call to a procedure

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique wit h respect to other schema objects, such as tables, views, and procedures. Using column names along with the UPDATE clause in the trigger improves performance, because the trigger fires only when that particular column is updated and thus avoids unintended firing when any other column is updated.

Example:

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR (TO_CHAR
(SYSDATE,'HH24:MI') NOT BETWEEN '08:00' AND '18:00')
THEN RAISE_APPLICATION_ERROR (-20500,'You may insert into
EMPLOYEES table only during business hours.');
END IF;
END;
/
```

Creating DML Statement Triggers

You can create a BEFORE statement trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

For example, create a trigger to restrict inserts into the EMPLOYEES table to certain business hours, Monday through Friday. If a user attempts to insert a row into the EMPLOYEES table on Saturday, the user sees the message, the trigger fails, a nd the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user andcausesthePL/SQLblocktofail. When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
IF (TO_CHAR (SYSDATE,'DY') IN ('SAT','SUN')) OR (TO_CHAR (SYSDATE,
'HH24') NOT BETWEEN '08' AND '18')
THEN
IF DELETING THEN
RAISE_APPLICATION_ERROR (-20502,'You may delete from EMPLOYEES
table only during business hours.');
ELSIF INSERTING THEN
RAISE_APPLICATION_ERROR (-20500,'You may insert into EMPLOYEES table
only during business hours.');
ELSIF UPDATING ('SALARY') THEN RAISE_APPLICATION_ERROR (-20503,
'You may update SALARY only during business hours.');
ELSE
RAISE_APPLICATION_ERROR (-20504,'You may update EMPLOYEES table
only during normal hours.');
END IF;
END IF;
END;
```

Combining Triggering Events

You can combine severa l triggering events into one by taking advantage of the special conditional predicat es INSERTING, UPDATING,andDELETING within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the EMPLOYEES table t o certain business hours, Monday through Friday.

Syntax for Creating a Row Trigger

trigger_name Is the name of the trigger

timing Indicates the time when the trigger fires in relation to the triggering event:

BEFORE

AFTER

INSTEAD OF

event Identifies the data manipulation operation that causes the trigger to fire:

INSERT

UPDATE [OF column]

DELETE

table_name Indicates the table associated with the t rigger

REFERENCING Specifies correlation names for the old and new values of the current row (The default values are OLD and NEW)

FOR EACH ROW Designates that the trigger is a row trigger

WHEN Specifies the trigger restriction; (This conditional predicate must be enclosed in parenthesis and is evaluat ed for each row to determine whether
or not the trigger body is executed.)

trigger body Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, ending with END, or a call to a
procedure

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
```

```
IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
AND :NEW.salary > 15000
THEN
RAISE_APPLICATION_ERROR (-20202,'Employee cannot earn this amount');
END IF;
END;
/
```

Creating a Row Trigger

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

Create a trigger to allow only certain employees to be able to earn a salary of more than 15,000.

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
INSERT INTO audit_emp_table (user_name, timestamp, id, old_last_name,
new_last_name, old_title,
new_title, old_salary, new_salary)
VALUES (USER, SYSDATE, :OLD.employee_id, :OLD.last_name, :
NEW.last_name, :OLD.job_id, :NEW.job_id, :OLD.salary, :NEW.salary );
END;
/
```

Using OLD and NEW Qua lifiers

Within a ROW trigger, reference the value of a column before and after the data change by prefixi ng it with the OLD and NEW qualifier.

Data Operation	**Old Valu e**	**New Value**
INSERT	NULL	Inserted val ue
UPDATE	Value before update	Value after update
DELETE	Valuebeforedelete	NULL

- TheOLD and NEW qualifiers are available only in ROW triggers.
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced i n the WHEN restricting condition.

Note: Row triggers can decrease the performance if you do a lot of updates on larger tables.

Restricting a Row Trigger

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
IF INSERTING
THEN :NEW.commission_pct := 0;
ELSIF :OLD.commission_pct IS NULL
THEN :NEW.commission_pct := 0;
ELSE
:NEW.commission_pct := :OLD.commission_pct + 0.05;
END IF;
```

END;
/

Example

To restrict t he trigger action to those rows that satisfy a cert ain condition, provi de a WHEN clause. Create a trigger on the EMPLOYEES t able to calculat e an employee's commissi on when a row is added to the EMPLOYEES table, or when an employee's salary is modified.

The NEW qualifier cannot be prefixed with a colon in the WHEN cla use because the WHEN clause is outside t he PL/SQL blocks.

INSTEAD OF Triggers

Use INSTEAD OF triggers t o modify data in whic h the DML statement has been issued against an inherently nonupdatable view. These triggers are called INSTEAD OF triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. This trigger is used to perform an INSERT, UPDATE,orDELETE operation directly on the underlying tables. You can write INSERT, UPDATE,orDELETE statements against a view, and the INSTEAD OF
trigger works invisibly in the background to make the right actions take place.

Why Use INSTEAD OF Triggers?

A view cannot be modified by normal DML statements if the view query contains set operators, group funct ions, clauses such as GROUP BY, CONNECT BY, START,theDISTINCT operator, or joins. For example, if a view consists of more than one table, an insert to the view ma y entail an insertion into one ta ble and an update to another. So, you write an INSTEAD OF trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatea ble and has INSTEAD OF triggers, the triggers take precedence.

INSTEAD OF triggers are row triggers.

The CHECK option for views is not enforc ed when insert ions or updates to the view are performed by using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

Syntax for Creating an INSTEAD OF Trigger

trigger_name Is the name of the trigger.
INSTEAD OF Indicates that the trigger belongs to a view
event Identifies the data manipulation operation that causes the trigger to fire:
INSERT
UPDATE [OF column]
DELETE
view_name Indicates the view associated with trigger
REFERENCING Specifies correlation names for t he old and new values of the current row (The defaults are OLD and NEW)
FOR EACH ROW Designates the trigger to be a row trigger;
INSTEAD OF triggers can only be row triggers: if this is omitted, the trigger is still defined as a row trigger.
trigger body Is the trigger body that defines the action pe rformed by the trigger, beginning with either DECLARE or BEGIN, and ending
with END or a call to a procedure

Note: INSTEAD OF triggers can be written only for views. BEFORE and AFTER options are not vali d.

Example:

The following example creates two new tables, NEW_EMPS and NEW_DEPTS, based on the EMPLOYEES and DEPARTMENTS tables respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables. The example also creates an INSTEAD OF trigger, NEW_EMP_DEPT. When a row is insertedintotheEMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, based on the data in the INSERT statement. Similarly, when a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

```
CREATE TABLE new_emps AS
SELECT employee_id, last_name, salary, department_id, email, job_id, hire_date FROM
employees;
CREATE TABLE new_depts AS
SELECT d.department_id, d.department_name, d.location_id, sum(e.salary) tot_dept_sal
FROM employees e, departments d
WHERE e.department_id = d.department_id
GROUP BY d.department_id, d.department_name, d.location_id;
CREATE VIEW emp_details AS
SELECT e.employee_id, e.last_name, e.salary, e.department_id, e.email, e.job_id,
d.department_name, d.location_id
FROM employees e, departments d
WHERE e.department_id = d.department_id;
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
IF INSERTING THEN
INSERT INTO new_emps VALUES (:NEW.employee_id, :NEW.last_name, :
NEW.salary, :NEW.department_id, :NEW.email, :New.job_id, SYSDATE);
UPDATE new_depts
SET tot_dept_sal = tot_dept_sal + :NEW.salary WHERE department_id = :
NEW.department_id;
ELSIF DELETING THEN
DELETE FROM new_emps WHERE employee_id = :OLD.employee_id;
UPDATE new_depts
SET tot_dept_sal = tot_dept_sal - :OLD.salary
WHERE department_id = :OLD.department_id;
ELSIF UPDATING ('salary')
THEN
UPDATE new_emps
SET salary = :NEW.salary WHERE employee_id = :OLD.employee_id;
UPDATE new_depts
SET tot_dept_sal = tot_dept_sal + (:NEW.salary - :OLD.salary) WHERE department_id
= :OLD.department_id;
ELSIF UPDATING ('department_id')
THEN
UPDATE new_emps
SET department_id = :NEW.department_id WHERE employee_id = :OLD.employee_id;
UPDATE new_depts SET tot_dept_sal = tot_dept_sal - :OLD.salary WHERE
department_id = :OLD.department_id;
UPDATE new_depts
SET tot_dept_sal = tot_dept_sal + :NEW.salary WHERE department_id = :
```

```
NEW.department_id;
END IF;
END;
/
```

Differentiating Between Database Triggers and Stored Procedures

Triggers	**Procedures**
Defined with CREATE TRIGGER	**Defined with CREATE PROCEDURE**
Data dictionary contains source code in USER_TRIGGERS	**Data dictionary co ntains source co de in USER_SOURCE**
Implicitly invoked	**Explicitly invoked**
COMMIT, SAVEPOINT,and ROLLBACK are not allowed	**COMMIT, SAVEPOINT,andROLLBACK are allowed**

Triggers are fully compiled when the CREATE TRIGGER comma nd is issued and the P code is stored in the data dictionary. If errors occur during the compilation of a trigger, the trigger is still created.

Trigger Modes: Enabled or Disabled

- When a trigger is first created, it is enabled automatically.
- The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, mana ges the dependencies, and provides a t wo-phase commit process if a trigger updates remote tables in a distributed database.
- Disable a specific trigger by using the ALTER TRIGGER syntax, or disable all triggers on a table by using the ALTER TABLE syntax.
- Disable a trigger to improve performance or to avoid data integrity checks when loading massive amounts of data by using utilities such as SQL*Loader. You may also wa nt to disable the trigger when it references a database object that is currently unavailable, owing to a failed network connection, disk crash, offline data file, or offline tablespace.

Compile a Trigger

- UsetheALTER TRIGGER command to explicitly rec ompile a trigger that is invalid.
- When you issue an ALTER TRIGGER statement with the COMPILE option, the trigger recompiles, regardless of whet her it is valid or inva lid.

Removing Triggers

When a trigger is no longer required, you can use a SQL sta tement in iSQL*Plus to drop it.

Testing Triggers

- Ensure that t he trigger works properly by t est ing a number of cases separately.
- Take advantage of the DBMS_OUTPUT procedures to debug triggers. You can also use the Procedure Builder debugging tool to debug triggers.

Trigger Execution Model

A single DML stat ement can pot entially fir e up to four types of triggers: BEFORE and AFTER stat ement and row triggers. A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. Triggers can also cause other triggers to fire (ca scading triggers). All actions and checks done as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, all actions performed because of the
original SQL statement are roll ed back. This includes actions performed by firing

triggers. This guarantees that int egrit y constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may under go changes by other users' transact ions. In all cases, a read-consist ent image is guaranteed for modified values t he trigger needs to read (query) or write (update).

A Sample Demonstration

The following pages of PL/SQL subprograms are an example of the interaction of triggers, packaged procedures, functions, and global variables.

The sequence of events:

1. Issue an INSERT, UPDATE,orDELETE command that can manipulate one or many rows.
2. AUDIT_EMP_TRIG,theAFTER ROW trigger, calls the packaged procedure t o increment the global variables in the package VAR_PACK. Because this is a row trigger, the trigger fires once for each row that you updated.
3. When the statement has finished, AUDIT_EMP_TAB,theAFTER STATEMENT trigger, calls the procedure AUDIT_EMP.
4. This procedure assigns the values of the global variables into local variables using the packaged functions, updates the AUDIT_TABLE, and then resets the global variables.

After Row and After Statement Triggers

```
CREATE OR REPLACE TRIGGER audit_emp_trig
AFTER UPDATE or INSERT or DELETE on EMPLOYEES
FOR EACH ROW
BEGIN
IF DELETING THEN var_pack.set_g_del(1);
ELSIF INSERTING THEN var_pack.set_g_ins(1);
ELSIF UPDATING ('SALARY')
THEN var_pack.set_g_up_sal(1);
ELSE var_pack.set_g_upd(1);
END IF;
END audit_emp_trig;
/
CREATE OR REPLACE TRIGGER audit_emp_tab
AFTER UPDATE or INSERT or DELETE on employees
BEGIN
audit_emp;
END audit_emp_tab;
/
```

AFTER Row and AFTER Statement Triggers

The trigger AUDIT_EMP_TRIG is a row trigger that fires after every row manipulated. This trigger invokes the package procedures depending on the type of DML performed. For example, if the DML updates salary of an employee, then the trigger invokes the procedure SET_G_UP_SAL. This package procedure inturn invokes the function G_UP_SAL. This function increments the package variable GV_UP_SAL that keeps account of the number of rows being changed due to update of the salary. The trigger AUDIT_EMP_TAB will fire after the statement has finished. This trigger invokes the procedure AUDIT_EMP, which is on the following pages. The AUDIT_EMP procedure upda tes t he AUDIT_TABLE table.AnentryismadeintotheAUDIT_TABLE table with the information such as the user who performed the DML, the table on which DML is performed, a nd the tota l number of such data manipulations performed so far on the table (indicated by the value of the corresponding column in the AUDIT_TABLE table). At the end, the AUDIT_EMP procedure reset s the package variables to 0.

Demonstration: VAR_PACK Package Specification

var_pack.sql

```
CREATE OR REPLACE PACKAGE var_pack
IS
-- these functions are used to return the
-- values of package variables
FUNCTION g_del RETURN NUMBER;
FUNCTION g_ins RETURN NUMBER;
FUNCTION g_upd RETURN NUMBER;
FUNCTION g_up_sal RETURN NUMBER;
-- these procedures are used to modify the
-- values of the package variables
PROCEDURE set_g_del (p_val IN NUMBER);
PROCEDURE set_g_ins (p_val IN NUMBER);
PROCEDURE set_g_upd (p_val IN NUMBER);
PROCEDURE set_g_up_sal (p_val IN NUMBER);
END var_pack;
/
```

Demonstration: VAR_PACK Package Body

var_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY var_pack IS
gv_del NUMBER := 0; gv_ins NUMBER := 0;
gv_upd NUMBER := 0; gv_up_sal NUMBER := 0;
FUNCTION g_del RETURN NUMBER IS
BEGIN
RETURN gv_del;
END;
FUNCTION g_ins RETURN NUMBER IS
BEGIN
RETURN gv_ins;
END;
FUNCTION g_upd RETURN NUMBER IS
BEGIN
RETURN gv_upd;
END;
FUNCTION g_up_sal RETURN NUMBER IS
BEGIN
RETURN gv_up_sal;
END;
PROCEDURE set_g_del (p_val IN NUMBER) IS
BEGIN
IF p_val = 0 THEN
gv_del := p_val;
ELSE gv_del := gv_del +1;
END IF;
END set_g_del;
PROCEDURE set_g_ins (p_val IN NUMBER) IS
BEGIN
IF p_val = 0 THEN
```

```
gv_ins := p_val;
ELSE gv_ins := gv_ins +1;
END IF;
END set_g_ins;
PROCEDURE set_g_upd (p_val IN NUMBER) IS
BEGIN
IF p_val = 0 THEN
gv_upd := p_val;
ELSE gv_upd := gv_upd +1;
END IF;
END set_g_upd;
PROCEDURE set_g_up_sal (p_val IN NUMBER) IS
BEGIN
IF p_val = 0 THEN
gv_up_sal := p_val;
ELSE gv_up_sal := gv_up_sal +1;
END IF;
END set_g_up_sal;
END var_pack;
/
```

Demonstration: Using the AUDIT_EMP Procedure

```
CREATE OR REPLACE PROCEDURE audit_emp IS
v_del NUMBER := var_pack.g_del;
v_ins NUMBER := var_pack.g_ins;
v_upd NUMBER := var_pack.g_upd;
v_up_sal NUMBER := var_pack.g_up_sal;
BEGIN
IF v_del + v_ins + v_upd != 0 THEN
UPDATE audit_table SET
del = del + v_del, ins = ins + v_ins,
upd = upd + v_upd
WHERE user_name=USER AND tablename='EMPLOYEES' AND column_name
IS NULL;
END IF;
IF v_up_sal != 0 THEN
UPDATE audit_table SET upd = upd + v_up_sal
WHERE user_name=USER AND tablename='EMPLOYEES' AND column_name
= 'SALARY';
END IF;
-- resetting global variables in package VAR_PACK
var_pack.set_g_del (0); var_pack.set_g_ins (0);
var_pack.set_g_upd (0); var_pack.set_g_up_sal (0);
END audit_emp;
```

Updating the AUDIT_TABLE with the AUDIT_EMP Procedure

The AUDIT_EMP procedure upda tes the AUDIT_TABLE and calls the functions in the package VAR_PACK that reset the package variables, ready for the next DML statement.

More Trigger Concepts

Objectives

After completing this lesson, you should be able to do the following:

- Create additional database triggers
- Explain the rules governing triggers
- Implement triggers

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema l evel. For example, a database shutdown trigger is defined at the database level. Triggers on data definition langua ge (DDL) statements, or a user logging on or off, ca n also be defined at either the database level or schema level.

Triggers on DML statements are defined on a specific table or a view. A trigger defined at the da tabase level fires for all users, and a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shut down or startup
- A specific or any error that occurs

Create Trigger Syntax

DDL_Event **Possible Values**

CREATE Causes the Oracle server to fire the trigger whenever a CREATE statement adds a new database object to the dictionary

ALTER Causes the Oracle server to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary

DROP Causes the Oracle server to fire the trigger whenever a DROP statement removes a database object in the data dictionary

The trigger body represents a complete P L/S QL block.

You can create triggers for these events on DATABASE or SCHEMA. You also specify BEFORE or AFTER for the timing of the trigger.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespa ce, trigger, type, view, or user.

Creating Triggers on System Events

CREATE [OR REPLACE] TRIGGER trigger_name
timing [database_event1 [OR database_event2 OR ...]] ON {DATABASE|SCHEMA}
trigger_body

Database_event **Possibl e Values**

AFTER SERVERERROR Causes the Oracle server to fire the trigger whenever a server error message is logged

AFTER LOGON Causes the Oracle server to fire the trigger whenever a user logs on to the database

BEFORE LOGOFF Causes the Oracle server to fire the trigger whenever a user logs off the database

AFTER STARTUP Causes the Oracle server to fire the trigger whenever the database is opened

BEFORE SHUTDOWN Causes the Oracle server to fire the trigger whenever the database is shut down

You can create triggers for these events on DATABASE or SCHEMA except SHUTDOWN and STARTUP, which apply only to the DATABASE.

LOGON and LOGOFF Trigger Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
INSERT INTO log_trig_table(user_id, log_date, action) VALUES (USER,
SYSDATE, 'Logging on');
END;
/
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
INSERT INTO log_trig_table(user_id, log_date, action) VALUES (USER,
SYSDATE, 'Logging off');
END;
/
```

You can create this trigger to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you a re logged on. When you specify ON SCHEMA,the trigger fires for the specific user. If you specify ON DATABASE, the t rigger fires for all users.

CALL Statements

```
timing event1 [OR event2 OR event3]
ON table_name [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution
/
CREATE [OR REPLACE] TRIGGER trigger_name
```

A CALL statement enables you to call a stored procedure, rather than coding the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.The call can reference the trigger attributes :NEW and :OLD a s parameters as in the following example:

```
CREATE TRIGGER salary_check
BEFORE UPDATE OF salary, job_id ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')
CALL check_sal(:NEW.job_id, :NEW.salary)
/
```

Note: There is no semicolon at the end of the CALL stat ement. In the example above, the trigger calls a procedure check_sal. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Reading Data from a Mutating Table

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of ON DELETE CASCADE.

Mutating Table

A mutating table is a table that is currentl y being modified by an UPDATE, DELETE,orINSERT statement, or a table tha t might need to be updated by the effects of a dec larative DELETE CASCADE referential integrity action. A table is not considered mutating for STATEMENT triggers. The triggered table itself is a mut ating table, as well as any table referencing it with the FOREIGN KEY constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
BEFORE INSERT OR UPDATE OF salary, job_id ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
v_minsalary employees.salary%TYPE;
v_maxsalary employees.salary%TYPE;
BEGIN
SELECT MIN(salary), MAX(salary) INTO v_minsalary, v_maxsalary
FROM employees
WHERE job_id = :NEW.job_id;
IF :NEW.salary < v_minsalary OR :NEW.salary > v_maxsalary THEN
RAISE_APPLICATION_ERROR(-20505,'Out of range');
END IF;
END;
/
```

Mutating Table: Example

The CHECK_SALARY trigger in the example, attempts to guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's sa lary or job ID is changed, the employee's salary falls within the established salary range for the employee's job. When an employee record is updated, the CHECK_SALARY trigger is fired for each row that is updat ed. The trigger code queries the same table that is being updated. Hence, it is said that the EMPLOYEES
table is mutating table.

```
UPDATE emp loyees
SET salar y = 3400
WHERE las t_name = 'Stiles' ;
```

If you restrict the salary within a range between the minimum existing value and the maximum existing value you get a run-time error. The EMPLOYEES table is mutating, or in a state of change; therefore, the t rigger cannot read from it. Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Implementing Triggers

D evelop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

Feature	Enh ancement
Security	The Oracle server allows table access to users or roles. Triggers allow table access according to data va lues.
Auditing	The Orac le server tracks data operations on

tables. Triggers track values for data operations on tables.

Data integrity The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.

Referential integrity The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.

Table replication The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.

Derived data The Oracle server computes derived data values manually. Triggers compute derived data values automatically.

Event logging The Oracle server logs events explicitly. Triggers log events transparently.

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON employees TO clerk;
 -- database role
GRANT clerk TO scott;
```

Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data manipulation, and data definition privileges.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
v_dummy VARCHAR2(1);
BEGIN
IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN'))
THEN RAISE_APPLICATION_ERROR (-20506,'You may only change data during normal business hours.');
END IF;
SELECT COUNT(*) INTO v_dummy FROM holiday WHERE holiday_date = TRUNC (SYSDATE) ;
IF v_dummy > 0 THEN RAISE_APPLICATION_ERROR(-20507, 'You may not change data on a holiday.');
END IF;
END ;
/
```

Controlling Security With a Database Trigger

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data manipulation privileges only.

```
AUDIT INSERT, UPDATE, DELETE
ON departments
BY ACCESS
WHENEVER SUCCESSFUL;
```

The Oracle server stores the audit information in a data dictionary table or operating system file.

Auditing Data Operations

You can audit data operations within the Oracle server. Database auditing is used to monitor and gather data about specific database activities. The DBA can gather statistics about which tables are being updated, how many I/Os are performed, how many concurrent users connect at peak time, and so on.

- Audit users, statements, or objects.
- Audit data retrieval, data manipulation, and data definition statements.
- Write the audit trail to a centralized audit table.
- Generate audit records once per session or once per access attempt.
- Capture successful attempts, unsuccessful attempts, or both.
- Enable and disable dynamically.

Executing SQL through PL/SQL program units may generate several audit records because the program units may refer to other database objects.

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
IF (audit_emp_package.g_reason IS NULL) THEN
RAISE_APPLICATION_ERROR (-20059, 'Specify a reason for the data operation
through the procedure SET_REASON
of the AUDIT_EMP_PACKAGE before proceeding.');
ELSE
INSERT INTO audit_emp_table (user_name, timestamp, id, old_last_name,
new_last_name, old_title, new_title,
old_salary, new_salary, comments)
VALUES (USER, SYSDATE, :OLD.employee_id, :OLD.last_name, :
NEW.last_name, :OLD.job_id, :NEW.job_id, :OLD.salary,
:NEW.salary, audit_emp_package.g_reason);
END IF;
END;
CREATE OR REPLACE TRIGGER cleanup_audit_emp
AFTER INSERT OR UPDATE OR DELETE ON employees
BEGIN
audit_emp_package.g_reason := NULL;
END;
```

Audit Data Values

Audit actual data values with triggers.

You can:

- Audit data manipulation statements only
- Write the audit trail to a user-defined audit table
- Generate audit records once for the statement or once for each row
- Capture successful attempts only
- Enable and disable dynamically

Using the Oracle server, you can perform database auditing. Database auditing cannot record changes to specific column values. If the changes to the table columns need to be tracked and column values need to be stored for each change, use application auditing. Application auditing can be done either through stored
procedures or database triggers, as shown in the example

```
ALTER TABLE employees ADD
CONSTRAINT ck_salary CHECK ( salary >= 500);
```

Enforcing Data Integrity within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key. Use these rules to:

- Provide constant default values
- Enforce static constraints
- Enable and disable dynamically

Example

The code sample ensures that the salary is at least $500.

```
CREATE OR REPLACE TRIGGER check_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.salary < OLD.salary)
BEGIN
RAISE_APPLICATION_ERROR (-20508, 'Do not decrease salary.') ;
END;
/
```

Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample ensures that the salary is never decreased.

```
ALTER TABLE employees
ADD CONSTRAINT emp_deptno_fk
FOREIGN KEY (department_id)
REFERENCES departments(department_id)
ON DELETE CASCADE;
```

Enforcing Referential Integrity within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

- Restrict updates and deletes.
- Cascade deletes.
- Enable and disable dynamically.

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
AFTER UPDATE OF department_id ON departments
FOR EACH ROW
BEGIN
UPDATE employees
SET employees.department_id=:NEW.department_id
WHERE employees.department_id=:OLD.department_id;
UPDATE job_history
SET department_id=:NEW.department_id
WHERE department_id=:OLD.department_id;
END;
```

```
/
```

Prote cting Referentia l Integrity with a Trigger

D evelop triggers to implement referential integrity rules that are not supported by declarative constraint s.

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential int egrity in a dist ributed system.
- Enable and disable dynamically.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table. F or a complete referential integrity solution using triggers, a single trigger is not enough.

```
CREATE SNAPSHOT em p_copy AS
SELECT * FROM employees@ny;
```

Creating a Snapshot

A snapshot is a local copy of a table data that originat es from one or more remot e master tables. An application can query the data in a read-only table snapshot, but cannot insert, update, or delete rows in the snapshot. To keep a snapshot's data current with the data of its master, the Oracle server must periodically refresh the snapshot. When this statement is used in SQL, replication is performed implicitly by the Ora cle server by using internal triggers. This has better performance over using user-defined PL/SQL triggers for replication.

Copying Tables with Server Snapshots

Copy a table wit h a snapshot.

- Copy t ables asynchronously, at user-defined intervals.
- Base snapshots on multiple master tables.
- Read from snapshots only.
- Improve the performance of data ma nipulation on the master table, particularly if the network fails. Alt ernatively, you can replicat e tables using triggers.

Example

In San Francisco, create a snapshot of the remot e EMPLOYEES table in New York.

```
CREATE OR REPLACE TRIGGER emp_replica
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN /*Only proceed if user initiates a data operation, NOT through the
cascading trigger.*/
IF INSERTING THEN
IF :NEW.flag IS NULL THEN
INSERT INTO employees@sf
VALUES(:new.employee_id, :new.last_name,..., 'B');
:NEW.flag := 'A';
END IF;
ELSE /* Updating. */
IF :NEW.flag = :OLD.flag THEN
UPDATE employees@sf
SET ename = :NEW.last_name, ...,
flag = :NEW.flag
WHERE employee_id = :NEW.employee_id;
```

```
END IF;
IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
ELSE :NEW.flag := 'A';
END IF;
END IF;
END;
```

Replicating a Table with a Trigger

Replicate a table with a trigger.

- Copy tables synchronously, in real time.
- Base replicas on a single master table.
- Read from replicas, as well as write to them.
- Impair the performance of dat a manipulation on the master table, particularly if t he network fails.

Mainta in copies of tables automa tically with snapshots, particularly on remote nodes.

Example

In New York, replicate the local EMPLOYEES table to San Francisco.

```
UPDATE departments
SET t otal_sal=(SELECT SUM(salary) FROM emplo yees WHERE empl oyees.department_ id =
d epartments.departm ent_id);
```

Computing Derived Data within the Server

Compute derived values in a batch job.

- Comput e derived c olumn va lues asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

Keep the salary total for each department within a special TOTAL_SALARY column of the DEPARTMENTS table.

```
CREATE OR REPLACE PROCEDURE increment_salary
(p_id IN departments.department_id%TYPE,
p_salary IN departments.total_sal%TYPE)
IS
BEGIN
UPDATE departments
SET total_sal = NVL (total_sal, 0)+ p_salary WHERE department_id = p_id;
END increment_salary;
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE ON employees
FOR EACH ROW
BEGIN
IF DELETING THEN
increment_salary(:OLD.department_id,(-1*:OLD.salary));
ELSIF UPDATING THEN
increment_salary(:NEW.department_id,(:NEW.salary-:OLD.salary));
ELSE increment_salary(:NEW.department_id,:NEW.salary);--INSERT
END IF;
END;
```

Computing Derived Data Values with a Trigger

Compute derived values with a trigger.

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a singl e pass to the da tabase.

Example

K eep a running total of the salary for each department within the special TOTAL_SALARY column of the DEPARTMENTS table.

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
ON inventories FOR EACH ROW
DECLARE
v_descrip product_descriptions.product_description%TYPE;
v_msg_text VARCHAR2(2000);
stat_send number(1);
BEGIN
IF :NEW.quantity_on_hand <= :NEW.reorder_point THEN
SELECT product_description INTO v_descrip FROM product_descriptions
WHERE product_id = :NEW.product_id;
v_msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)|| ...'Yours,' ||CHR
(10) ||user || '.'|| CHR(10)|| CHR(10);
ELSIF
:OLD.quantity_on_hand < :NEW.quantity_on_hand THEN NULL;
ELSE
v_msg_text := 'Product #'||... CHR(10);
END IF;
DBMS_PIPE.PACK_MESSAGE(v_msg_text);
stat_send := DBMS_PIPE.SEND_MESSAGE('INV_PIPE');
END;
```

Logging Events with a Trigger

Within the server, you can log events by querying data and performing operations manually. This sends a message using a pipe when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package DBMS_PIPE to send the message.

Logging Events within the Server

- Query data explicitly to determine whether an operation is necessary.
- In a second step, perform the opera tion, such as sending a message.

Using Triggers to Log Events

- Perform operations implicitly, such as firing off an automatic electronic memo.
- Modify data and perform its dependent opera tion in a single step.
- Log events automa tically as data is changi ng.

Logging Events Transparently

In the trigger code:

- CHR(10) is a carriage return
- Reorder_point is not nul l
- Another transaction can receive and read the message in the pipe

Example

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
```

```
BEFORE UPDATE OF amount_in_stock, reorder_point
ON inventory FOR EACH ROW
DECLARE
v_descrip product.descrip%TYPE;
v_msg_text VARCHAR2(2000);
stat_send number(1);
BEGIN
IF :NEW.amount_in_stock <= :NEW.reorder_point THEN
SELECT descrip INTO v_descrip
FROM PRODUCT WHERE prodid = :NEW.product_id;
v_msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)|| 'It has come to my
personal attention that, due to recent'
||CHR(10)||'transactions, our inventory for product # '|| TO_CHAR(:NEW.product_id)||'--
'||v_descrip || ' -- has fallen below acceptable levels.' || CHR(10) ||
'Yours,' ||CHR(10) ||user || '.'|| CHR(10)|| CHR(10);
ELSIF
:OLD.amount_in_stock<:NEW.amount_in_stock THEN NULL;
ELSE
v_msg_text := 'Product #'|| TO_CHAR(:NEW.product_id)
||' ordered. '|| CHR(10)|| CHR(10); END IF;
DBMS_PIPE.PACK_MESSAGE(v_msg_text);
stat_send := DBMS_PIPE.SEND_MESSAGE('INV_PIPE');
END;
```

Benefits of Database Triggers

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

Managing Triggers

In order t o create a trigger in your schema, you need the CREATE TRIGGER system privilege, and you must either own the table specified in the triggering statement, have the ALTER privilege for the table in the triggering statement, or have the ALTER ANY TABLE system privilege. You can alter or drop your triggers without any further privileges being required. If the ANY keywor d is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.
You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects t hat are not in your schema, the user creating the trigger must have the EXECUTE privilege on the referenced procedures, functions, or packages, and not through roles. As with stored procedures, the statement in the trigger body operates under the privilege domain of the trigger's owner, not that of the user issuing the triggering statement.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, you can drop the trigger, but you cannot alter it.

Viewing Trigger Information

The USER_OBJECTS view contains the name and status of the trigger and the date and time when the trigger was created.

The USER_ERRORS view contains the details of the compilation errors that occurred while a trigger was compiling. The contents of these views are si milar to t hose for subprograms.

The USER_TRIGGERS view c ontains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The SELECT Username FROM USER_USERS; statement gives the name of the owner of the trigger, not the na me of the user who is updating the table.

Column	**Column Description**
TRIGGER_NAME	**Name of the trigger**
TRIGGER_TYPE	**The type is BEFORE, AFTER, INSTEAD OF**
TRIGGERING_EVENT	**The DML operation firing the trigger**
TABLE_NAME	**Name of the database table**
REFERENCING_NAMES	**Name used for :OLD and :NEW**
WHEN_CLAUSE	**The when_clause used**
STATUS	**The status of the trigger**
TRIGGER_BODY	**Theactiontotake**

Listing the Code of Triggers

```
SELECT trigger_name, trigger_type, triggering_event, table_name,
referencing_names,
status, trigger_body FROM user_triggers WHERE trigger_name =
'RESTRICT_SALARY';
```

Managing Dependencies

Objectives

After completing this lesson, you should be able to do the following:

- Track procedural dependencies
- Predict the effect of changing a database object **upon stored procedures and functions**
- Manage procedural dependencies

Dependent and Referenced Objects

Some objects reference other objects as part of their definition. For example, a stored procedure could contain a SELECT statement that selects columns from a table. For this reason, the s tored procedure is called a dependent object, wherea s the table is called a referenced object.

Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. F or example, if t he table defi nition is changed, the procedure may or may not continue to work without error. The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or inval id) t hat is recorded in the dat a dict ionary, and you can view the status in the USER_OBJECTS dat a dictionary view.

Status Significance

VALID The schema object has been compiled and can be immediately used when referenced.
INVALID The schema object must be compiled before it ca n be used.

A proc edure or a function can directly or indirect ly (t hrough an intermediate view, procedure, function, or packaged procedure or function) reference the following objects:
•Tables
•Views
•Sequences
• Procedures
• Functions
• Packaged procedures or functions

Managing Local Dependencies

In the case of local dependencies, the objects are on the same node in the same da tabase. The Oracle server automatically manages all local dependencies, using the database's internal "depends-on" table. When a referenced object is modified, the dependent objects are invalidated. The next time an invalidated object is called, the Oracle server automatically recompiles it.

Assume that the structure of the table on which a view is based is modified. When you describe the view, you get an error message that states that the object
is invalid to describe. This is because the comma nd is not a SQL command and, at this sta ge, the view is invalid because the structure of its base table is changed. If you query the view now, the view is recompiled automatically and you can see the result if it is successfully recompil ed.

Example

The QUERY_EMP procedure directly references the EMPLOYEES table. The ADD_EMP procedure updates the EMPLOYEES table indirectly, by way of the EMP_VW vi ew. In each of the following cases, will the ADD_EMP procedure be invali dated, and will it successfully recompile?
1. The internal logic of the QUERY_EMP procedure is modified.
2. A new column is added to the EMPLOYEES table.
3. The EMP_VW view is dropped.

```
SELECT name, type, referenced_name , ref eren ced_type FROM user_dependencies
WHERE referenced_name IN ('EMPLOY EES', 'EMP _VW' );
…
…
```

Display Direct Dependencies by Using USER_DEPENDENCIES

Determine which database objects to recompile manually by displaying direct dependencies from the USER_DEPENDENCIES data dictionary view.
Examine the ALL_DEPENDENCIES and DBA_DEPENDENCIES views, each of which contains the additional c olumn OWNER, that reference the owner of the object .

Column	**Column Description**
NAME	The name of the dependent object
TYPE	The type of the dependent object (PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER,orVIEW)
REFERENCED_OWNER	The schema of the referenced object
REFERENCED_NAME	The name of the referenced object
REFERENCED_TYPE	The type of the re ferenced object

REFERENCED_LINK_NAME The database link used to access the referenced object

Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

Display direct and indirect dependencies from additional user views called DEPTREE and IDEPTREE; these view are provided by Oracle.

Example

1. Make sure the utldtree.sql script has been executed. This script is located in the $ORACLE_HOME/rdbms/admin folder. (This script is supplied in the lab folder of your class files.)
2. Populate the DEPTREE_TEMPTAB table with information for a particular referenced object by invoking the DEPTREE_FILL procedure. There are three parameters for this procedure:

object_type Is the type of the referenced object
object_owner Is the schema of the referenced object
object_name Is the name of the referenced object

DEPTREE View

SELECT nested_level, type, name FROM deptree
ORDER BY seq#;

…

…

Example

Display a tabular representation of all dependent objects by querying the DEPTREE view. Display an indent ed representation of the same information by querying the IDEPTREE view, which consist s of a singl e column named DEPENDENCIES.

For example,

SELECT * FROM ideptree;

provides a single column of indented output of the dependencies in a hierarchical structure.

Predicting the Effects of Changes on Dependent Objects

Example 1

Predict the effect that a change in the definition of a procedure has on the recompilation of a dependent procedure.

Suppose that the RAISE_SAL procedure updates the EMPLOYEES table directly, and that the REDUCE_SAL procedure updates the EMPLOYEES table indir ectly by way of RAISE_SAL.

In each of the following cases, will the REDUCE_SAL procedure successfully recompile?

1. The internal logic of the RAISE_SAL procedure is modified.
2. One of the formal parameters to the RAISE_SAL procedure is eliminat ed.

Predicting Effects of Changes on Dependent Objects (continued)

Example 2

Be aware of the subtle case in which the creation of a table, view, or synonym may unexpectedly invalida te a dependent object because it interferes with the Ora cle server hierarchy for resolving name refer ences. Predict the effect that the name of a new object has upon a dependent procedure.

Suppose that your QUERY_EMP procedure originally referenced a public synonym called EMPLOYEES. However, you have just created a new t able call ed EMPLOYEES within your own schema. Will this cha nge invalidate the procedure? Whi ch of t he two

EMPLOYEES objects will QUERY_EMP reference when the procedure recompiles? Now suppose that you drop your private EMPLOYEES table. Will this invalidate the procedure? What will happen when the procedure recompiles? You can track security dependencies within the USER_TAB_PRIVS data dicti onary view.

Understanding Remote Dependencies

In the case of remote dependencies, the objects are on separate nodes. The Oracle server does not ma nage dependencies among remote schema objects other than local-procedure-to-remote- procedure dependencies (including functions, packa ges, and triggers). The local stored proc edure and all of its dependent objects will be invalidated but will not automatically recompile when ca lled for the first time.

Recompilation of Dependent Objects: Local and Remote

• Verify successful explic it recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the USER_OBJECTS view.
• If an automatic implicit recompilation of the dependent local procedures fails, the status rema ins invalid and the Oracle server is sues a run-time error. Therefor e, to avoid dis rupting production, it is strongly recommended that you recompile local dependent objects manual ly, rather than relying on an automatic mechanism.

Concepts of Remote Dependencies

Remote dependencies are governed by the mode chosen by the user:
• TIMESTAMP checking
• SIGNATURE checking

TIMESTAMP Checking

Each PL/SQL program unit carries a time stamp that is set when it is created or recompiled. Whenever you alter a PL/SQL program unit or a relevant schema object, all of its dependent program units are marked as invalid and must be recompiled before they can execute. The actual time stamp comparison occurs when a statement in the body of a local procedure calls a remote procedure.

SIGNATURE Checking

For each PL/SQL program unit, both the time stamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:
• The name of the construct (procedure, function, or packa ge)
• The base types of the parameters of the construct
• The modes of the parameters (IN, OUT,orIN OUT)
• The number of the parameters
The recorded time stamp in the calling program unit is compared with the current time stamp in the called remot e program unit. If the time stamps mat ch, the call proceeds norma ll y. If they do not match, the Remote Procedure Calls (RPC) layer performs a simple test to compare the signature to determine whether the call is safe or not. If the signature has not been changed in an incompatible ma nner, execution continues; otherwise, an error status is returned.

REMOTE_DEPENDENCIES_MODE Parameter
Setting REMOTE_DEPENDENCIES_MODE:
- As an init.ora parameter

REMOTE_DEPENDENCIES_MODE = value
- At the system level

ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = value
- At the session level

ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = value
Note: The calling site determines the dependency model.

Using Time Stamp Mode for Automatic Recompilation of Local and Remote Objects

If time stamps are used to handle dependencies a mong PL/SQL program units, then whenever you alter a program unit or a relevant schema object, all of its dependent units are marked as invalid and must be recompiled before they can be run.

Using Time Stamp Mode for Automatic Recompilation of Local and Remote Objects

The definition of the table changes. Hence, all of its dependent units are marked as inva lid and must be recompiled before t hey can be run.

- When remote objects change, it is strongly recommended that you recompile local dependent obj ects manually in order to avoid disrupting production.
- The remote dependency mecha nism is different from the automatic local dependency mechanism already discussed. The first time a recompiled remote subprogram is invoked by a local subprogram, you get an execution error and the local subprogram is invalidated; the second time it is invoked, implicit automatic recompilation takes place.

Local Procedures Referencing Remote Procedures
A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is rec ompiled after the local procedure is compiled.
Automatic Remote Dependency Mechanism
When a procedure compiles, the Oracle server records the time stamp of that compilation within the P code of the procedure.
Automatic Remote Dependency Mechanism
When a local procedure referencing a remote procedure compiles, the Oracle server also records the time stamp of the remote procedure into the P code of the local procedure.
In the slide, local procedure A which is dependent on remote procedure B is compiled at 9:00 a.m. The time stamps of both procedure A and remote procedure B are recorded in the P code of procedure A.
Automatic Remote Dependency
When the local procedure is invoked, at run time the Oracle server compares the two time stamps of the referenced remot e procedur e. If the time stamps are equal (indicating that the remote procedure has not recompiled), the Ora cle server execut es the local procedur e.

Assume that the remote procedure B is successfully recompiled at 11a .m. The new time stamp is recorded along with its P code.
If the time stamps are not equal (indicating that the remote procedure has recompiled), the Oracle server invalidates the local procedure and returns a runtime error.
If the local procedur e, which is now tagged as invalid, is invoked a second time, the Oracle server recompiles it before executing, in accordance with the automatic local dependency mechanism.

Note: If a local procedure returns a run-time error the first time that it is invoked, indicating that the remote procedure's time stamp has changed, you should develop a strategy to reinvoke the local procedure.

Disadvantage of time stamp mode

A disadvantage of the time stamp mode is that it is unnecessarily restrictive. Recompilation of dependent objects across the network are often performed when not strictly necessary, leading to performance degradation.

Signatures

To alleviate some of the problems with the time stamp-only dependency model, you can use the signature model. This allows the remote procedure to be recompiled without affecting the local procedures. This is important if the database is distributed.

The signature of a subprogram contains the following information:

- The name of the subprogram
- The datatypes of the parameters
- The modes of the parameters
- The number of parameters
- The datatype of the return value for a function

If a remote program is changed and recompiled but the signature does not change, then the local procedure can execute the remote procedure. With the time stamp method, an error would have been raised because the time stamps would not have matched.

Recompiling PL/SQL Objects

If the recompilation is successful, the object becomes valid. If not, the Oracle server returns an error and the object remains invalid. When you recompile a PL/SQL object, the Oracle server first recompiles any invalid objects on which it depends.

Procedure

Any local objects that depend on a procedure (such as procedures that call the recompiled procedure or package bodies that define the procedures that call the recompiled procedure) are also invalidated.

Packages

The COMPILE PACKAGE option recompiles both the package specification and the body, regardless of whether it is invalid. The COMPILE BODY option recompiles only the package body. Recompiling a package specification invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. Note that the body of a package also depends on its specification.

Triggers

Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The DEBUG option instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

Unsuccessful Recompilation

Sometimes a recompilation of dependent procedures is unsuccessful, for example, when a referenced table is dropped or renamed. The success of any recompilation is based on the exact dependency. If a referenced view is recreated, any object that is dependent on the view needs to be recompiled. The success of the recompilation depends on the columns that the view now cont ains, as well as t he columns that the dependent obj ects require for their execution. If the required columns are not part of the new view, the object remains invalid.

Successful Recompilation

The recompilation of dependent objects is successful if:

- New columns are added to a referenced table
- AllINSERT statements include a column list
- No new column is defined as NOT NULL

When a private table is referenced by a dependent procedure, and the private table is dropped, the status of the dependent procedure becomes invalid. When the procedure is recompiled, either explicitly or implicitly, and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the sta tus of the procedure remains invalid.

Recompilation of Procedures

Minimize dependency failures by:

- Declaring records by using the %ROWTYPE attribute
- Declaring variables with the %TYPE attribute
- Querying with the SELECT * notation
- Including a column list with INSERT statements

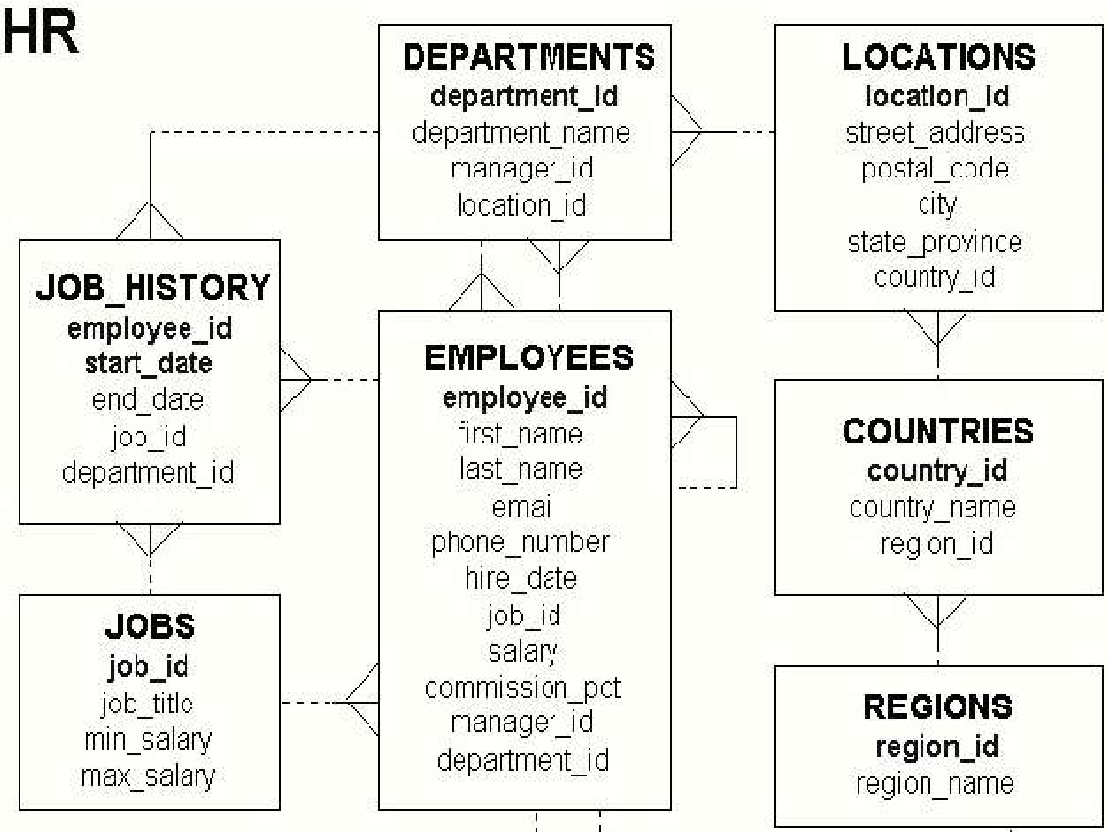
Managing Dependencies

You can greatly simplify dependency management with packages when referencing a package procedure or function from a stand-alone procedure or function.

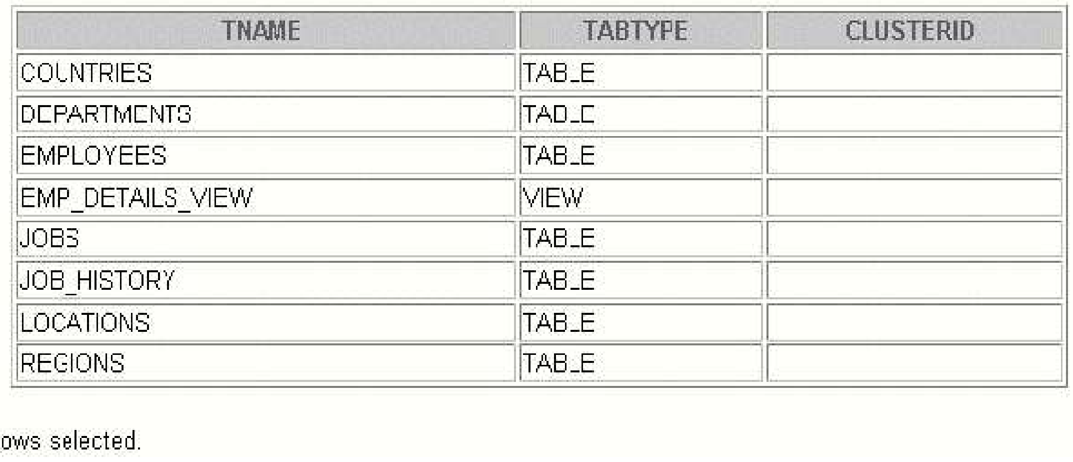
- If the packa ge body changes and the pac kage specification does not change, the stand-alone procedure referencing a package construct remains valid.
- If the packa ge specification changes, the outside procedure referencing a package construct is invalidated, as is the package body.

If a stand-alone procedure refer enced within the package changes, the entire pa ckage body is invalidated, but the package spec ification remai ns valid. Therefore, it is recommended t hat you bring the procedure into the packa ge.

Table Descriptions and Data



ENTITY RELATIONSHIP DIAGRAM

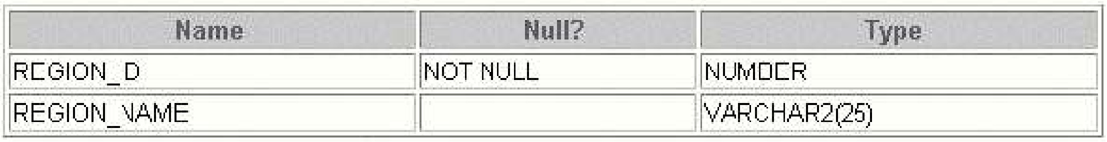


TNAME	TABTYPE	CLUSTERID
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

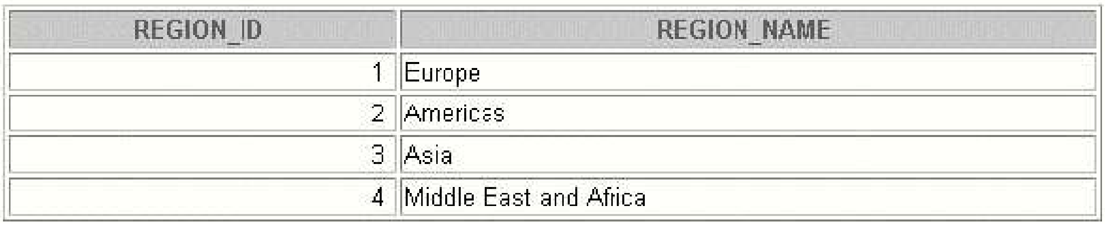
8 rows selected.

Tables in the Schema

SELECT * FROM tab;



Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

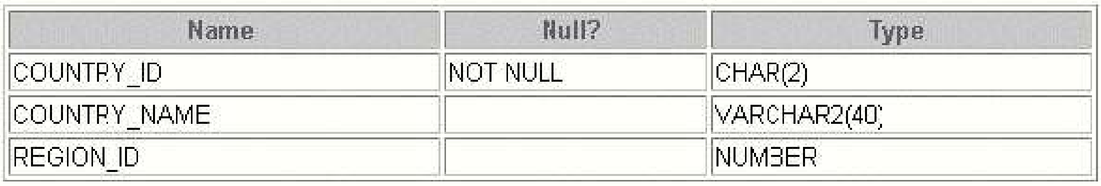


REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

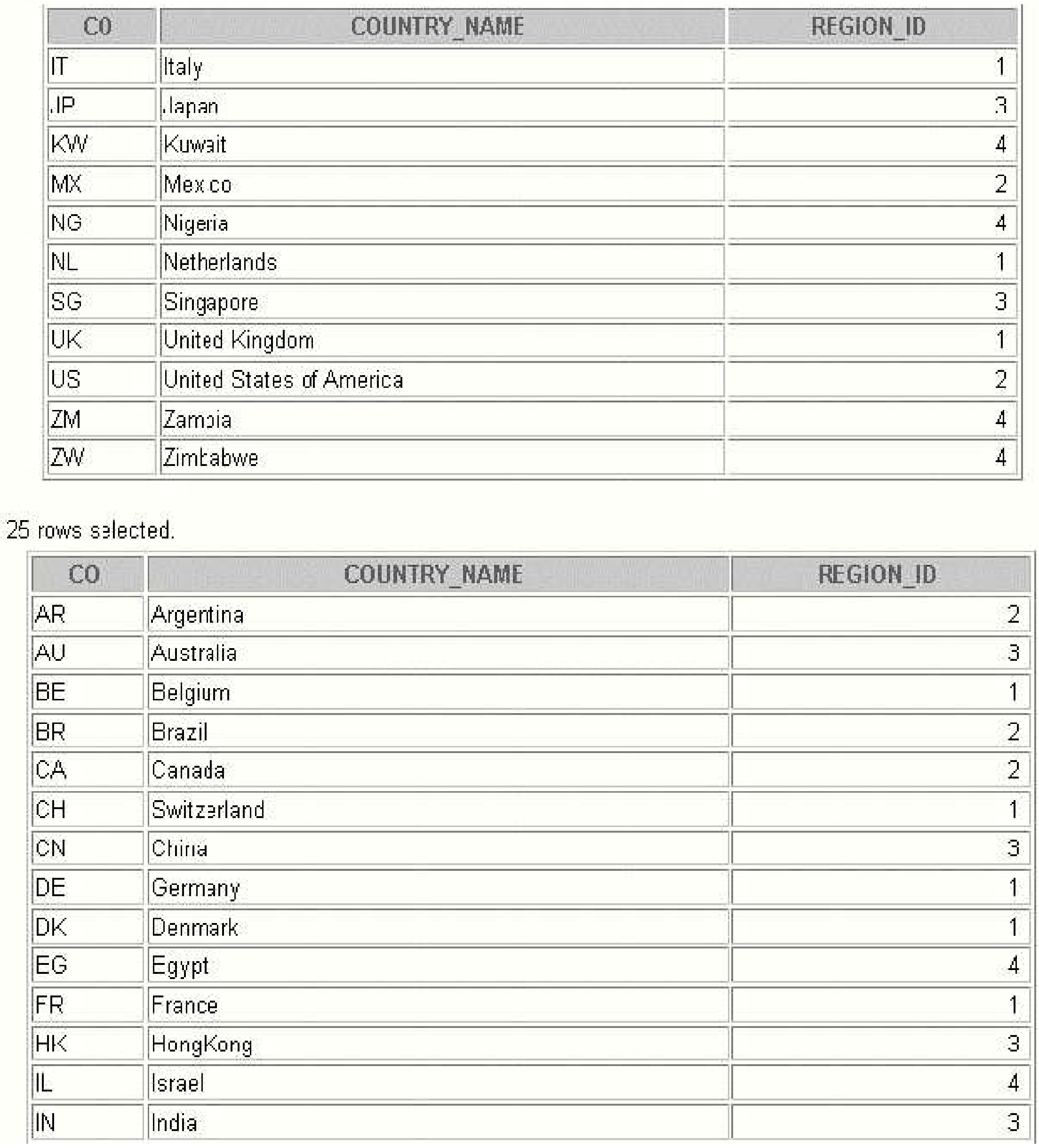
REGIONS Table

DESCRIBE regions

SELECT * FROM regions;



Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER



CO	COUNTRY_NAME	REGION_ID
IT	Italy	1
JP	Japan	3
KW	Kuwait	4
MX	Mexico	2
NG	Nigeria	4
NL	Netherlands	1
SG	Singapore	3
UK	United Kingdom	1
US	United States of America	2
ZM	Zambia	4
ZW	Zimbabwe	4

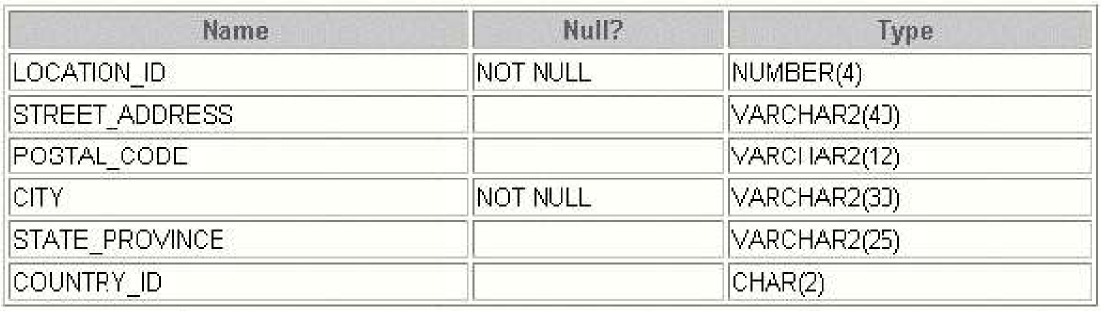
25 rows selected.

CO	COUNTRY_NAME	REGION_ID
AR	Argentina	2
AU	Australia	3
BE	Belgium	1
BR	Brazil	2
CA	Canada	2
CH	Switzerland	1
CN	China	3
DE	Germany	1
DK	Denmark	1
EG	Egypt	4
FR	France	1
HK	HongKong	3
IL	Israel	4
IN	India	3

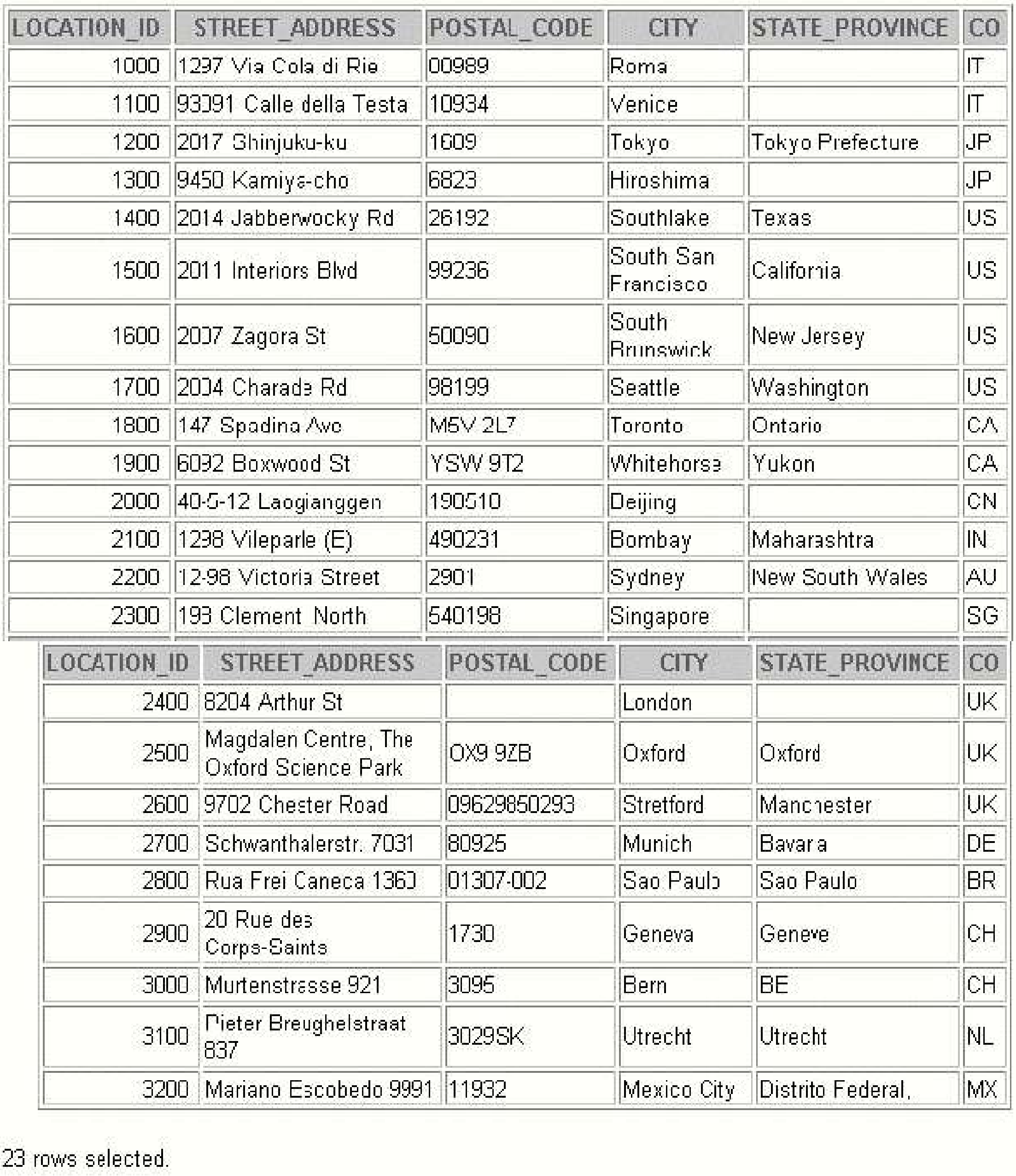
COUNTRIES Table

DESCRIBE countries

SELECT * FROM countries;



Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

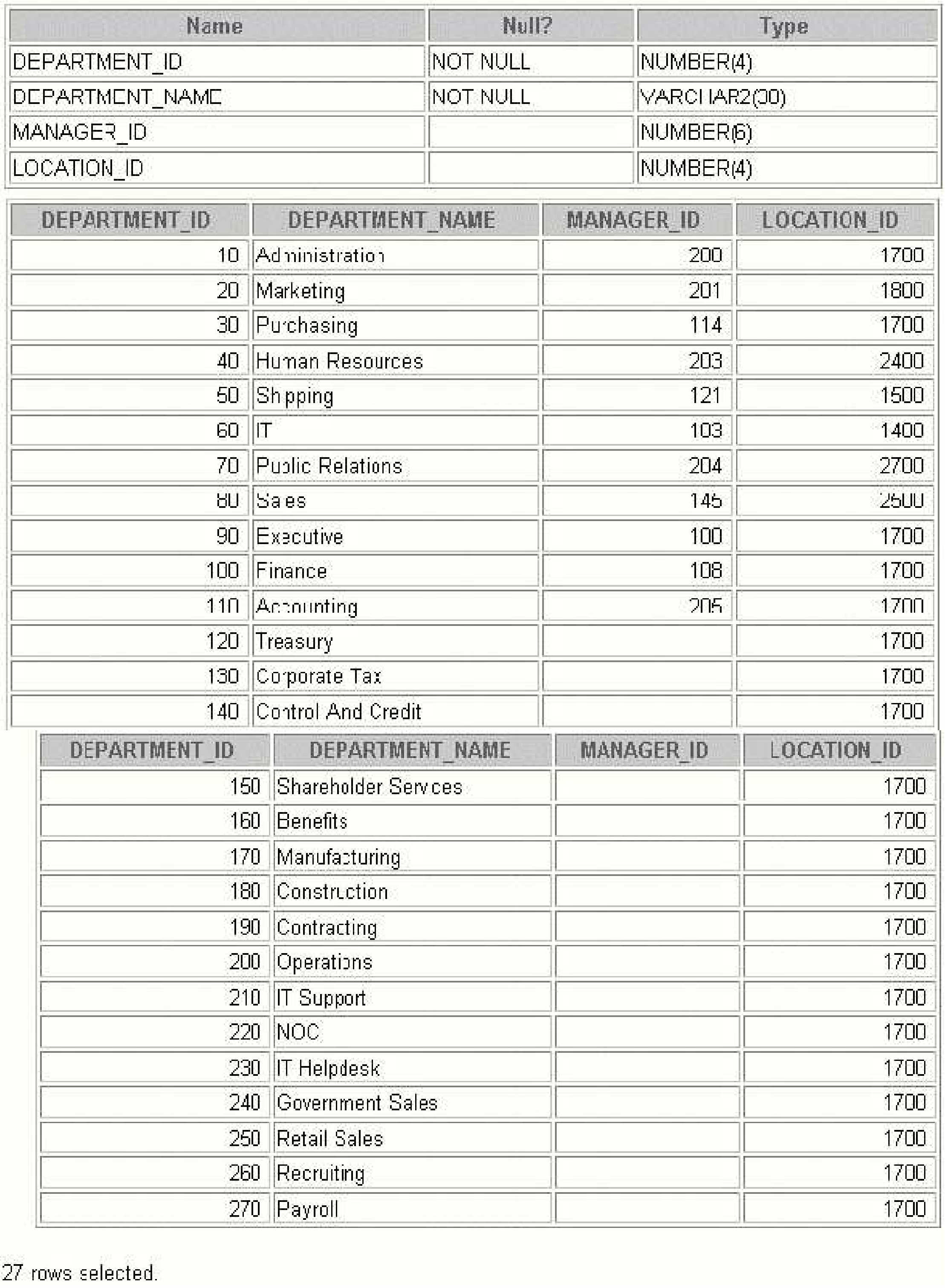


LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
2000	40-5-12 Laogianggen	190518	Beijing		CN
2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
2200	12-98 Victoria Street	2901	Sydney	New South Wales	AU
2300	198 Clement North	540198	Singapore		SG
2400	8204 Arthur St		London		UK
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
3000	Murtenstrasse 921	3095	Bern	BE	CH
3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

23 rows selected.

LOCATIONS Table

DESCRIBE locations;
SELECT * FROM locations;



Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

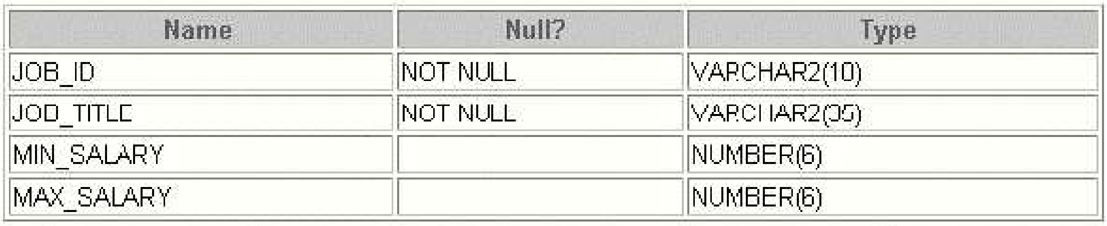
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

27 rows selected.

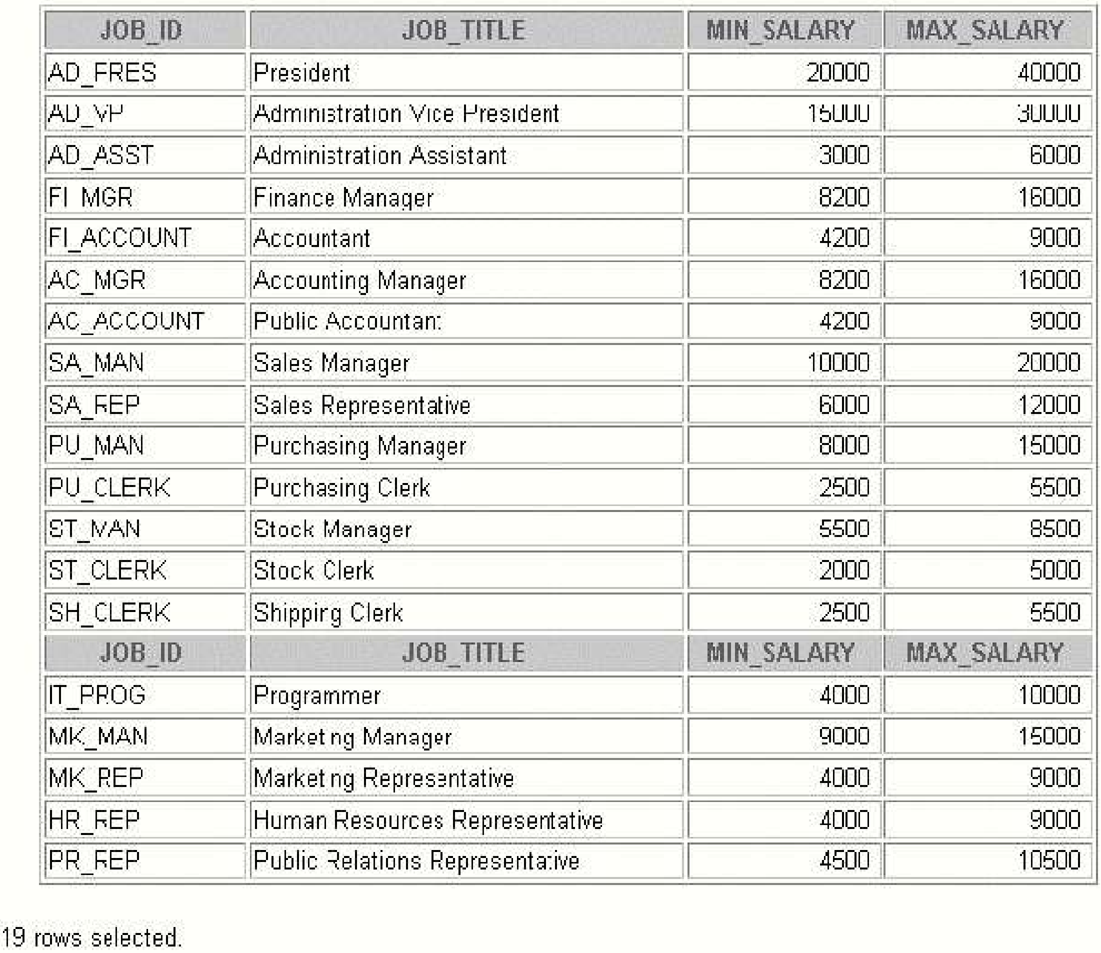
DEPARTMENTS Table

DESCRIBE departments

SELECT * FROM departments;



Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)



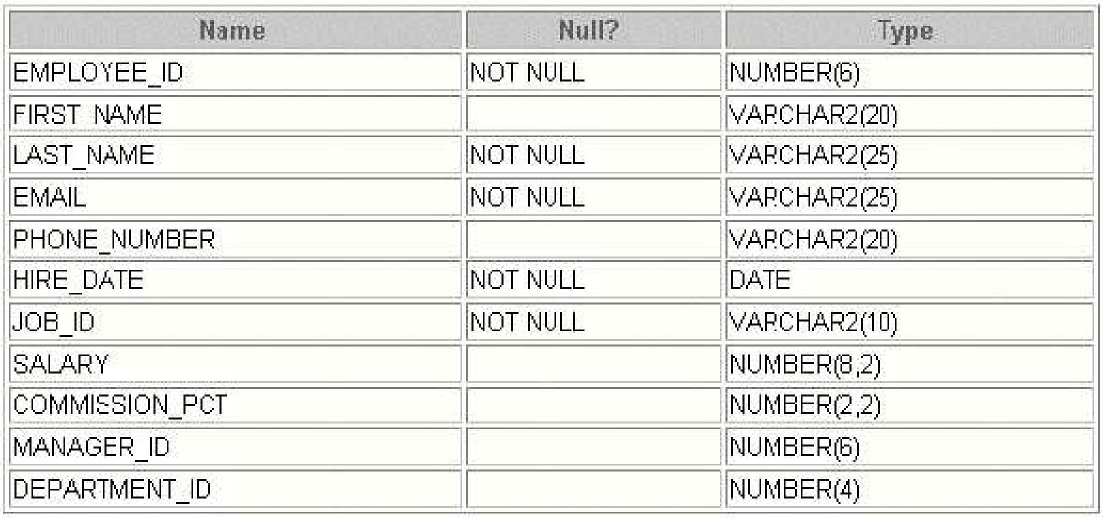
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.

JOBS Table

DESCRIBE jobs

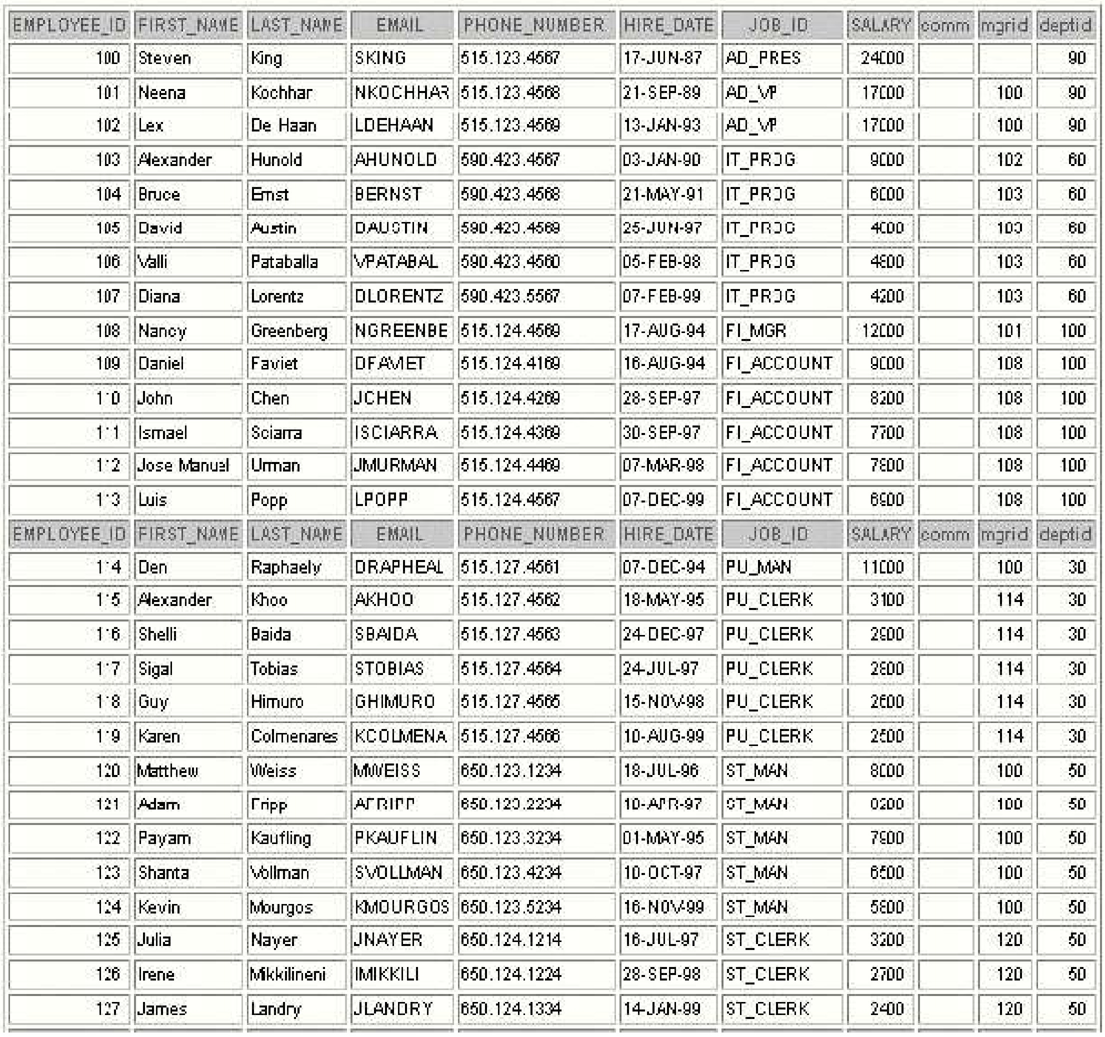
SELECT * FROM jobs;



Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

EMPLOYEES Table

DESCRIBE employees



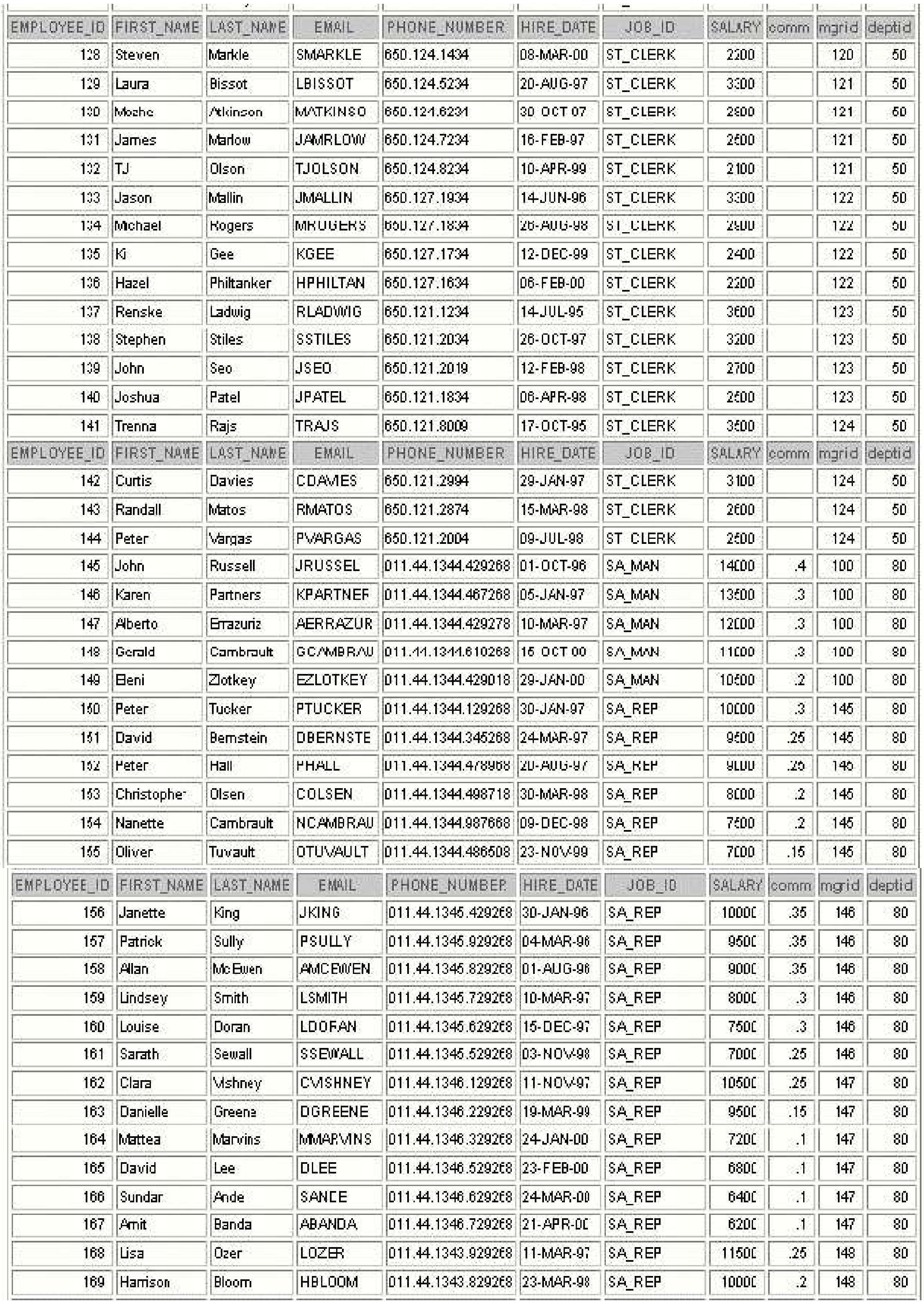
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700		108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900		108	100
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100		114	30
116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800		100	50
125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200		120	50
126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700		120	50
127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400		120	50

EMPLOYEES Table

The headings for columns COMMISSION_PCT, MANAGER_ID,andDEPARTMENT_ID are set to

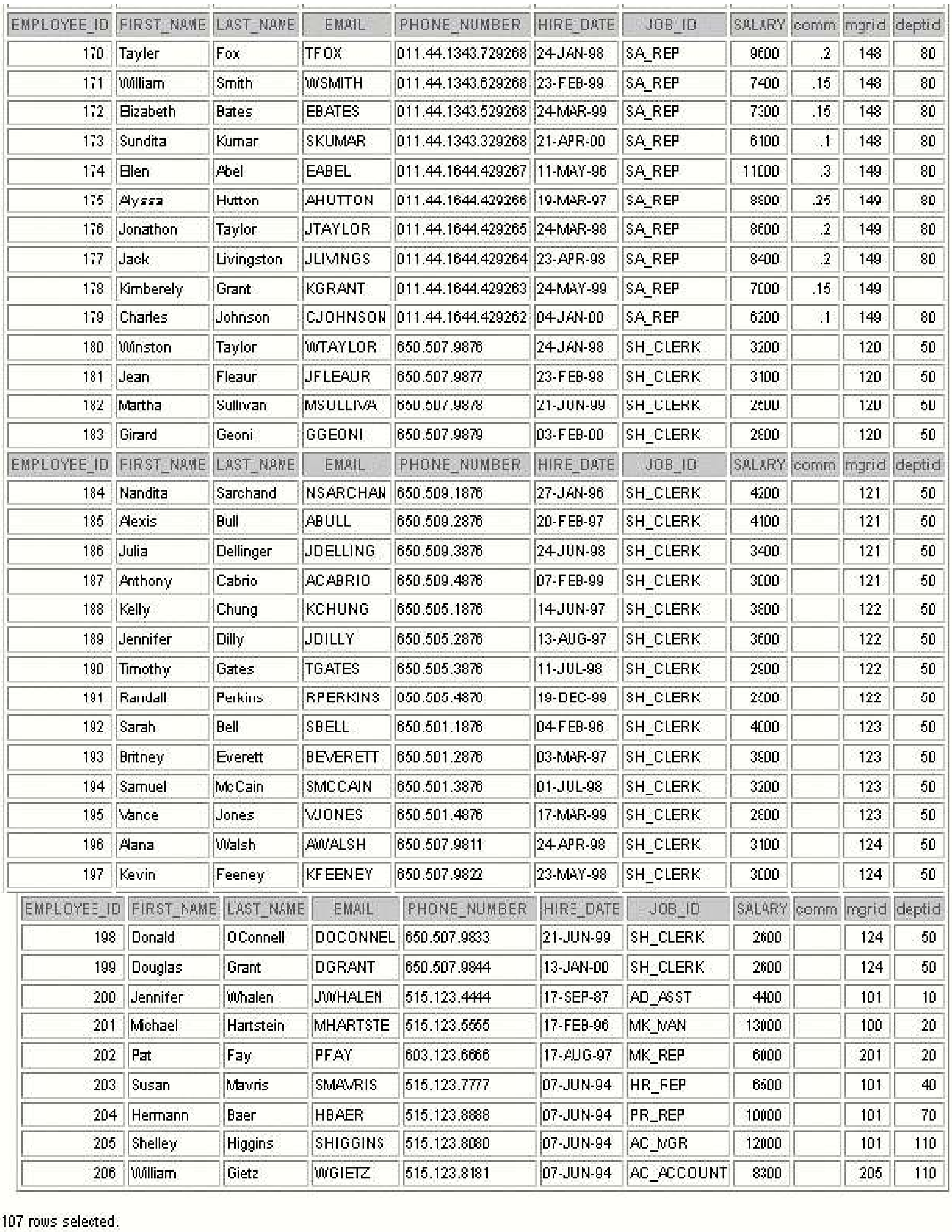
COMM, MGRID,andDEPTID in the following screenshot, to fit the table values across the page.

SELECT * FROM employees;



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200		120	50
129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300		121	50
130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800		121	50
131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500		121	50
132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100		121	50
133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300		122	50
134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900		122	50
135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400		122	50
136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200		122	50
137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600		123	50
138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200		123	50
139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700		123	50
140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500		123	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100		124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600		124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500		124	50
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	SA_MAN	14000	.4	100	80
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	SA_MAN	13500	.3	100	80
147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-97	SA_MAN	12000	.3	100	80
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	SA_MAN	11000	.3	100	80
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	10500	.2	100	80
150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-97	SA_REP	10000	.3	145	80
151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-97	SA_REP	9500	.25	145	80
152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-97	SA_REP	9000	.25	145	80
153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-98	SA_REP	8000	.2	145	80
154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-98	SA_REP	7500	.2	145	80
155	Oliver	Tuvault	OTUVAULT	011.44.1344.486508	23-NOV-99	SA_REP	7000	.15	145	80
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
156	Janette	King	JKING	011.44.1345.429268	30-JAN-96	SA_REP	10000	.35	146	80
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500	.35	146	80
158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-96	SA_REP	9000	.35	146	80
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	.3	146	80
160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-97	SA_REP	7500	.3	146	80
161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-98	SA_REP	7000	.25	146	80
162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-97	SA_REP	10500	.25	147	80
163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	SA_REP	9500	.15	147	80
164	Mattea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-00	SA_REP	7200	.1	147	80
165	David	Lee	DLEE	011.44.1346.529268	23-FEB-00	SA_REP	6800	.1	147	80
166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-00	SA_REP	6400	.1	147	80
167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-00	SA_REP	6200	.1	147	80
168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-97	SA_REP	11500	.25	148	80
169	Harrison	Bloom	HBLOOM	011.44.1343.829268	23-MAR-98	SA_REP	10000	.2	148	80

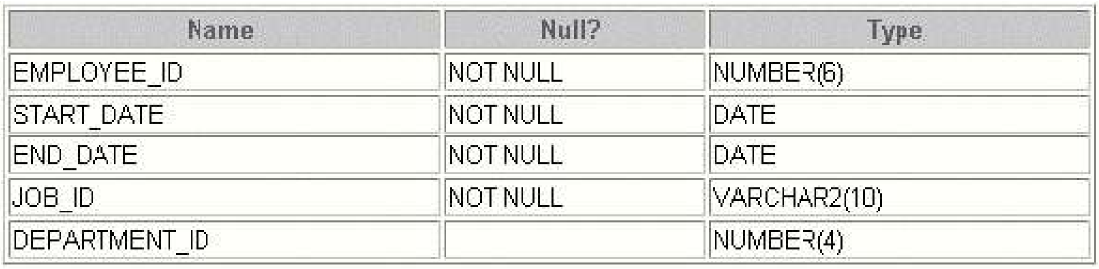
EMPLOYEES Table (continued)



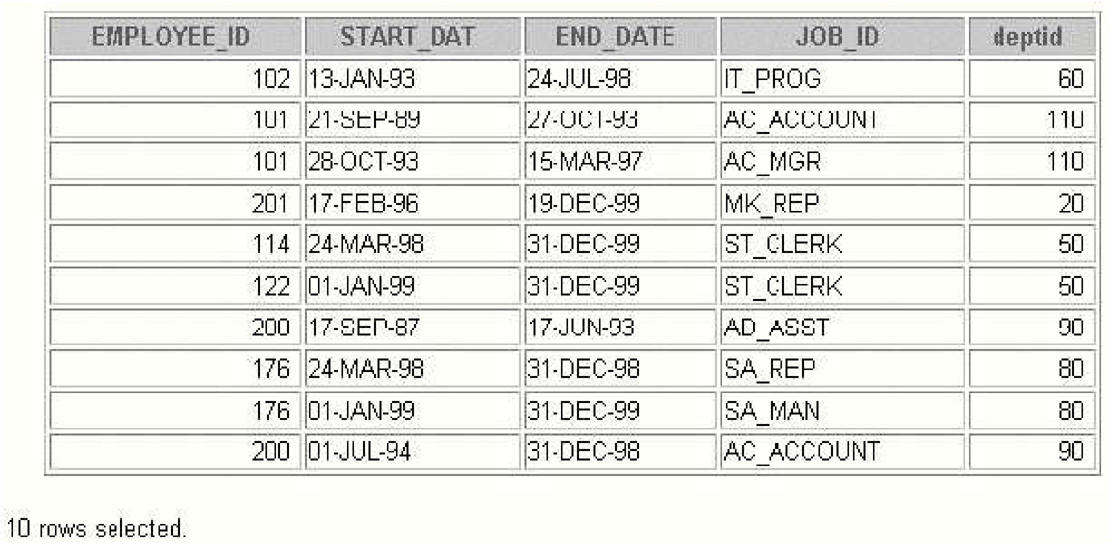
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
170	Tayler	Fox	TFOX	011.44.1343.729268	24-JAN-98	SA_REP	9600	.2	148	80
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	.15	148	80
172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-99	SA_REP	7300	.15	148	80
173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-00	SA_REP	6100	.1	148	80
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	11000	.3	149	80
175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	19-MAR-97	SA_REP	8800	.25	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	8600	.2	149	80
177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-98	SA_REP	8400	.2	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	.15	149	
179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-00	SA_REP	6200	.1	149	80
180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200		120	50
181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100		120	50
182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500		120	50
183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800		120	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200		121	50
185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100		121	50
186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400		121	50
187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000		121	50
188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800		122	50
189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600		122	50
190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900		122	50
191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500		122	50
192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000		123	50
193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900		123	50
194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200		123	50
195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800		123	50
196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100		124	50
197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600		124	50
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600		124	50
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400		101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000		100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000		201	20
203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500		101	40
204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000		101	70
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000		101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300		205	110

107 rows selected.

EMPLOYEES Table (continued)



Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)



EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	deptid
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

JOB_HISTORY Table
DESCRIBE job_history
SELECT * FROM job_history;

Creating Program Units by Using Procedure Builder

Objectives

After completing this appendix, you should be able to do the following:

- Describe the features of Oracle Procedure Builder
- Manage program units using the Object Navigator
- Create and compile program units using the

Program Unit Editor

- Invoke program units using the PL/SQL Interpreter
- Debug subprograms using the debugger
- Control execution of an interrupted PL/SQL

program unit

- Test possible solutions at run time

PL/SQL Block Structure

Every PL/SQL const ruct is composed of one or more blocks. These blocks can be entirely sepa rate or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Note: The PL/SQL blocks can be constructed on and use the Oracle server (stored PL/SQL program units). They can also be constructed using t he Oracle Developer tools such as Ora cle Forms Developer, Oracle Report Developer, and so on (application or client-side PL/SQL program units).

Object types are user-defined composite dat a types that encapsulate a data structure along wit h the functions and procedures needed to manipulate the data. You can create object types either on the Oracle server or using the Oracle Developer tools.

You can create both application program units and stored program units using

Oracle Procedure Builder. Application program units are used in graphical user environment tools such as Oracle Forms. Stored program units are stored on the database server and can be shared by multiple applications.

iSQL*Plus and Oracle Procedure Builder

PL/SQL is not an Oracle product in its own right. It is a technology employed by the Oracle Server and by certain Oracle development tools. Blocks of PL/SQL are passed to, and processed by, a PL/SQL engine. That engine may reside within the tool or within the Oracle Server.There a re two main development environments for PL/SQL: iSQL*Plus and Oracle Procedur e Builder.

About Procedure Builder

Oracle Proc edure Builder is a tool you can use to crea te, execute, and debug PL/SQL programs used in your application tools, such as a form or report, or on the Oracle server through its graphical interface.

Integrated PL/SQL Development Environment

Procedure Builder's development environment contains a build-in editor for you to create or edit subprograms. You can compile, test, and debug your code.

Unified Client-Server PL/SQL Development

Application partitioning through Procedure Builder is available to assist you with distribution of logic between client and server. Users can drag and drop a PL/SQL program unit between the client and the server.

Components of Procedure Builder

Procedure Builder is an integrated development environment. It enables you to edit, compile, test, and debug client-side and server-side PL/SQL program units within a single tool.

The Object Navigator

The Object Navigator provides an outline-style i nterface to browse obj ects, view the relat ionships bet ween them, and edit their properties.

The Interpreter Pane

The Interpreter pane is the c entral debugging workspace of the Oracle Procedure Builder. It is a window with t wo regions where you display, debug, and run PL/SQL progra m units. It also int eractively supports the evaluation of PL/SQL constructs, SQL comma nds, and Procedure Builder commands.

The Program Unit Editor

The easiest and most common place t o enter P L/S QL source code is in the Program Unit Editor. You can use it to edit, compile, and browse warning and error messages during application development. The Stored Program Unit Editor is a GUI environment for editing server-side packages and subprograms. The compile operation submits the source text to the server-side PL/SQL compiler.

TheDatabaseTriggerEditor

The Database Trigger Editor is a GUI environment for editing database triggers. The compile operation submits the source text to the server-side PL/SQL compiler.

Program Units and Stored Program Units

Use Procedure Builder to develop PL/SQL subprograms that ca n be used by client and server applications. Program units are cli ent-side PL/SQL subprograms that you use wit h client applicat ions, such as Oracle Developer. Stored program units are server-side PL/SQL subprograms that you use with all applications, client or server.

Developing PL/S QL Code

Client-side code:

• Create program unit s by using the Program Unit Editor

• Drag a server-side subprogram to the client by using the Object Na vigator Server-side code:

• Create stored programs by using the Stored Program Unit Editor

• Drag a client-side program unit to the server by using the Object Navigator

Components of the Object Navigator

1. Location indicat or: S hows your current location in the hierarchy.
2. Subobject indicator: Allows you to expand and collapse nodes to view or hide object informa tion. Different icons represent different classes of objects.
3. Type icon: Indicates the type of object, followed by the name of the object. If you double-click the icon, Procedure Builder opens the Program Unit Editor and displays the code of that object.
4. Object na me: Shows you the na mes of the objects.
5. Find field: Allows you to search for objects.

Object Navigator

The Object Navigator is Procedure Builder's browser for locating and working with bot h client and server program units, libraries, and triggers.

The Object Navigator allows you to expand and collapse nodes, cut and paste, search for an object, and drag P L/S QL program units between the client and the server side.

Components of the Object Navigator: Vertical Button Bar

The vertical button bar on the Object Navigator provides convenient access for ma ny of the actions frequent ly performed from the File, Edit, and Navigator menus.

1. Open: Opens a library from the file system or from the Oracle server. Sa ve: Saves a library in the file system or on the Oracle server.
2. Cut: Cuts the selected object and stores it in the clipboard. Cutting an object also cuts any objects owned by that object.

Copy: Makes a copy of the selected object and stored it in the clipboard. Copying an object a lso copies any objects owned by that object.

Paste: Pastes the cut or copied module into the selected location. Note that objects must be copied to a valid location in the object hierarchy.

3. Create: Creates a new instance of the currently selected object. Delete: Deletes the selected object with confirmation.
4. Expand, Collapse, Expand All, and Collapse All: Expands or collapses one or all levels of subobjects of the currently selected object.
5.

How to Develop Stored Program Units

Use the following steps to develop a stored progra m unit:

1. Enter t he synt ax in the Program Unit editor.
2. Click the Save button to compile and sa ve the code.

The sourc e code is compiled into P code.

Program Unit Editor

Use the Program Unit Editor to edit, compile, and browse warning and error messages during development of client-side PL/SQL subprograms. To bring a subprogram into the source text pane, select an option from the Name drop-down list. Use the buttons to decide which action to take once you are in the P rogram Unit Editor.

The Store d Program Unit Editor

Use the Stored P rogram Unit Edit or to edit server-side PL/SQL constructs. The Save opera tion submits thesourcetexttotheserver-sidePL/SQLcompiler.

How to Create a Client-Side Program Unit

1. Select the Program Units object or subobject.
2. Click the Create button. The New Program Unit dialog box appears.
3. Enter the name of your subprogram, select the subprogram type, and click the OK button to accept the entries.
4. The Program Unit editor is displayed. It cont ains the skelet on for your PL/SQL construct. The cursor is automatically positioned on the line beneath the BEGIN keyword. You can now write the code.
5. When you finish writing the code, click Compile in the Program Unit Editor.

Error messages generated during compilation are displayed in the compilation message pane in the Program Unit window. When you select an error message, the cursor moves to the location of the error in the program screen. When your PL/SQL code is error free, the compilation message disappears, and the Successfully Compiled message appears in the status line of the Program Unit Editor.

Note: Program units that reside in the Program Units node are lost when you exit Procedure Builder. You must export them to a file, sa ve them in a PL/SQL library, or store them in the database.

How to Create a Server-Side Program Unit

1. Select the Database Objects node in the Object Navigator, expand the schema name, and c lick Stored Program Units.
2. Click Create.
3. In the New Program Unit window, enter the name of the subprogram, select the subprogram type, and click OK to accept the entries.
4. The S tored Program Unit editor is displayed. It contains the skelet on for your PL/SQL construct. The cursor is automatically positioned on the line beneath the BEGIN keyword. You can now write the code.
5. When you finish writing the code, click Save in the Stored Program Unit Editor. Error messages generated during c ompilation are displayed in a compilation message at the bottom of the window. Click an error message to move to the location of the error. When the PL/SQL code is error-free, the compila tion message does not appear. The Successfully Compiled message appears in the status line at the bottom of the Stored Program Unit Editor window.

Application Partitioning

Using Procedure Builder you can creat e PL/SQL program units on both t he cli ent and the server. You can also use Procedure Builder to copy program units created on the client into stored program units on the server (or vice versa). You can do this by a dragging the program unit t o the destination S tored Program Units node in the a ppropriate schema.

PL/SQL code that is stored in the server is processed by the server-side PL/SQL engine; therefore, any SQL statements contained within the program unit do not have to be transferred between a client application a nd the server. Program units on the server are potentially accessible to all applications (subject to user security privileges).

Describing Procedures and Functions

To display a procedure or funct ion, its parameter list, and ot her information, use the . DESCRIBE command in Procedure Builder.

Listing Code of a Stored Procedure

1. Select File > Connect and enter your username, password, and database.
2. Select Database Objects and click the Expand button.
3. Select the schema of the procedure owner and click the Expand button.
4. Select Stored Program Units and click the Expand button.
5. Double-click the icon of the stored procedure. The Stored Program Unit editor appears in the window and contains the code of the procedure.

How to Resolve Compilation Errors

1. Click Compile.
2. Select an error message.

The cursor moves to the location of the error in the source pane.

3. Resolve the syntax error and click Compile.

TEXT_IO Built-in Package

You can use TEXT_IO packaged procedures t o output values and messages from a client-side procedure or function to the PL/SQL Interpreter window.

TEXT_IO is a built-in package that is part of Procedure Builder.

Use the Oracle supplied package DBMS_OUTPUT to debug server-side procedures, and the Procedure Builder built-in, TEXT_IO, to debug client-side procedures.

Note:

- You cannot use TEXT_IO to debug server-side procedures. The program will fail to compile successfully because TEXT_IO is not st ored i n the database.
- DBMS_OUTPUT does not display messages in the PL/SQL Interpreter window if you execute a procedure from Procedure Builder.

How to Create a Statement Trigger When Using Procedure Builder

You can also create the same BEFORE statement trigger in Procedure Builder.

1. Connect to the database.
2. Click the Database Objects node in the Object Navigator.
3. Select the Database Trigger edit or from the Program menu.
4. Select a table owner and a table from the Table owner and Table drop-down lists.
5. Click New to start creating the trigger.
6. Select one of the Triggering option buttons to choose the timing component.
7. Select Statement to choose the event component.
8. In the Trigger Body region, enter the trigger code.
9. Click Save. Your trigger code will now be compiled by the PL/SQL engine in the server. Once successfully compiled, your trigger is stored in the database and automa

tically enabled.

Note: If the trigger has compilation errors, the error message appears in a separate window.

How to Create a Row Trigger When Using Procedure Builder

You can also create the same BEFORE row trigger in Procedur e Builder.

1. Connect to the database.
2. Click the Database Objects node in the Object Navigator.
3. Sel ect the Database Trigger Editor from the Program menu.
4. Select a table owner and a table from the corresponding drop-down lists.
5. Click New to start creating the trigger.
6. Select the Triggering option button to choose the timing component.
7. Select the appropriate Statement check boxes to choose the events component.
8. In the For Ea ch region, select the Row option button to designate the trigger as a row trigger.
9. Complete the Refer encing OLD As and NEW As fields if you want to modify the correlation names. In the When field, enter a WHEN condition to restrict the execution of the trigger. These fields are optional and are available only with row triggers.
10. Enter the trigger code.
11. Click Save. The trigger code is now compiled by the PL/SQL engine in the server. When successfully compiled, the trigger is stored in the database and automatically enabled.

Removing a Server-Side Program Unit

When you decide to delete a stored program unit, an alert box displays with the following message: "Do you really want to drop stored program unit <program unit name>?". Click Yes to drop t he unit . In the Stored Program Units Edit or, you can also clic k DROP to remove the procedure from the server.

Re moving a Client-Side Program Unit

Follow the steps in the slide to remove a procedure from Procedure Builder.

If you have exported the code that built your procedure to a text file and you want to delete that file from the client, you must use the appropriate operating system comma nd.

Debugging Subprograms by Using Procedure Builder

You can perform debug actions on a server-side or client-side subprogram using Procedure Builder.

Use the following steps to load the subprogram:

1. From the Object Navigator, select Program > PL/SQL Int erpreter.
2. In the menu, select View > Naviga tor Pane.
3. From the Navigator pane, expand either the P rogram Units or the Database objects node.
4. Locat e the program unit that you want to debug and click it.

Listing Code in the Source Pane

Performing Debug Actions in the Interpreter

You can use the Object Navigator to exa mine and modify parameters in an interrupted program. By invoking the Object Navigator within the Interpreter, you can perform debugging actions entir ely within the Interpreter window. Alternatively, you can interact with the Object Navigator and Interpreter windows separately.

1. Invoking the Object Navigator Pane

- Select PL/SQL Interpreter from the Tools menu to open the Interpreter if it is not already open.
- Select Navi gator Pane from the View menu.
- The Navigator pane is inserted between the S ource and the Int erpreter panes.
- Dra g the split bars to adjust the size of each pane.

2. Listing Source Text in the Source Pane
- Click the Program Units node in the Navigator pane to expand the list.

The list of program units is displayed.
- Click the object icon of the program unit to be list ed.

3. The source code is listed in the Source pane of the Interpreter.

Setting a Breakpoint

If you encounter errors while compiling or running your application, you should test the code and determine the cause for the error. To determine the cause of the error effectively, review the code, line by line. Eventually, you should identify the exact line of code causing the error. You can use a breakpoint to halt execution at any given point and to permit you to examine the status of the code on a line-by-line basis.

Setting a Breakpoint

1. Double click the executa ble line of code on which to break. A "B(n)" is placed in the line where the break is set.
2. The message Breakpoint #n installed at line i of name isshowninthe Interpreter pane.

Note: Breakpoints also can be set using debugger commands in the Interpreter pane. Test breakpoints by entering the program unit name at the Interpreter PL/SQL prompt.

Monitoring Debug Actions

Debug actions, like breakpoints, can be viewed in the Object Navigator under the heading Debug Actions. Double-c lick the Debug Actions icon to view a description of the breakpoint. Remove breakpoints by double-clicking the breakpoint line number

Reviewing Code

When a breakpoint is reached, you can use a set of commands to step through the code. You can execute these commands by clicking the command buttons on the Interpreter toolbar or by entering the command at the Interpreter prompt.

Commands for Stepping through Code

Command	**Description**
Step Into	Advances executi on into the next executabl e line of code
Step Over	Bypasses the next executable line of code and advances to the subsequent line
Step Out	Resumes to the end of the current level of code, such as the subprogram
Go	Resumes execution until either the program unit ends or is interrupted again by a debug ac tion
Reset	Aborts the execution at the current levels of debugging

Stepping Through Code

Determining the Cause of Error

After the breakpoint is found at run time, you can begin stepping through the code. An arrow (=>) indicates the next line of code to execute.

1. Click the Step Into button.
2. A single line of code is executed. The arrow moves to the next line of code.
3. Repeat step 1 as necessary until the line causing the error is found.

The arrow continues to move forward until the erroneous line of code is found. At that time, PL/SQL displays an error message.

Changing a Value

Examining Local Variables

Using Procedure Builder, you can examine and modify local variables and parameters in an int errupted program. Use the Stack node in the Navigator pane t o view and change the values of local variables and parameters associated with the current program unit located in the call stack. When debugging code, check for the absence of values as well as incorrect values.

Examining Values and Testing the Possible Solution

1. Click the Stack node in the Object Navigator or Navigator pane to expand it.
2. Clock the value of the varia ble to edit. For example, select variable 1.

The value 1 becomes an editable field.

3. Enter the new value and click anywhere in the Navigator pane to end the variable editing, for example, enter 3.

The following statement is displayed in the Interpreter pane:

(debug1) PL/SQL> debug.seti('I',3);

4 Click the Go button to resume execution through the end of the program unit.

Note: Variables and parameters can also be changed by using commands at the Interpret er PL/SQL prompt.
REF Cursors

Cursor Variables

Cursor variables are like C or Pascal point ers, which hold the memory location (address) of some it em inst ead of the it em itself. Thus, declaring a cursor variable creates a pointer, not an item. In P L/ SQL, a pointer has the datatype REF X,whereREF is short for REFERENCE and X stands for a class of objects. A cursor variable has datatype REF CURSOR.

Like a cursor, a cursor variable point s to the current row i n the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied t o a specific query. You can open a cursor variable for a ny type-c ompatible quer y. This gives you more flexibility.

Cursor variables are available to every PL/ SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, and then pass it as an input host variable (bind variable) to PL/SQL. Moreover, applica tion development tools such as Oracle For ms and Oracle Report s, which have a PL/SQL engine, can use cursor variables entirely on the client side.

The Oracle server also has a PL/SQL engine. You can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

Why Use Cursor Variables?

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the res ult set is st ored. For example, an OCI cli ent, an Oracle F orms application, and the Oracle server can all refer to the same work area.

A query work area rema ins accessible as long as a ny cursor va riable points to it. Therefore, you can pa ss the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block that is embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from c lient to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variabl es in a single round t rip. A cursor variable holds a reference to the cursor work area in the PGA instead of addressing it with a

static name. Because you address this area by a reference, you gain the flexibility of a variable.

Defining REF CURSOR Types

To define a REF CURSOR, you perform two steps. F irst, you define a REF CURSOR type, and then you declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the following synta x:

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

in which: ref_type_name is a type specifier used in subsequent declarations of cursor variables

return_type represent s a record or a row in a database table

In the following exa mple, you specify a return type that represents a row in the database table DEPARTMENT.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next exa mple shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE; -- strong
TYPE GenericCurTyp IS REF CURSOR; -- weak
```

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

After you define a REF CURSOR type, you can declare cursor variables of that type in any P L/ SQL block or subprogram. In the following exa mple, you declare the cursor variable DEPT_CV:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
dept_cv DeptCurTyp; -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packa ged variables, cursor variables do not have persistent states. Remember, declaring a cursor variable creates a pointer, not an item. Cursor variables cannot be saved in the database; they follow the usual scoping and instantiation rules.

In the RETURN clause of a REF CURSOR type definition, you can use % ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
tmp_cv TmpCurTyp; -- declare cursor variable
```

```
TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
emp_cv EmpCurTyp; -- declare cursor variable
```

Likewise, you can use %TYPE to provi de the datatype of a record variable, as the following example shows:

```
DECLARE
dept_rec departments%ROWTYPE; -- declare record variable
TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
dept_cv DeptCurTyp; -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
TYPE EmpRecTyp IS RECORD ( empno NUMBER(4), ename VARCHAR2(1O), sal NUMBER(7,2));
TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
emp_cv EmpCurTyp; -- declare cursor variable
```

Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following exa mple, you define the REF CURSOR type EmpCurTyp, and then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Using the OPEN-FOR, FETCH,andCLOSE Statements

You use three statements to process a dyna mic multirow query: OPEN-FOR, FETCH,and CLOSE. First, you OPEN a cursor variable FOR a multirow query. Then, you FETCH rows from the result set one at a time. When all the rows are processed, you CLOSE the cursor variable.

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor varia ble with a multirow query, executes the query, ident ifies the result set, positions the cursor to point to the first row of the results set, then sets the rows-processed count kept by %ROWCOUNT to zero. Unlike the sta tic form of OPEN-FOR,the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT sta tement.

The syntax is:

```
OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
[USING bind_argument[, bind_argument]...];
```

where CURSOR_VARIABLE is a weakl y typed cursor variable (one without a return type),

HOST_CURSOR_VARIABLE is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multirow query.

In the following exa mple, the syntax declares a cursor variable, and then associates it with a dynamic SELECT statement that returns rows from t he EMPLOYEES table:

```
DECLARE
TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR type
emp_cv EmpCurTyp; -- declare cursor variable
my_ename VARCHAR2(15);
my_sal NUMBER := 1000;
BEGIN
```

```
OPEN emp_cv FOR -- open cursor variable
'SELECT last_name, salary FROM employees WHERE salary > :s'
USING my_sal;
...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. Thus, to fetch rows from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

Fetching from the Cursor Variable

The FETCH statement returns a row from the result set of a multirow query, assigns the values of select-list items to corresponding variables or fields in the INTO cla use, increments the count kept by %ROWCOUNT, and advances the cursor to the next row. Use the following syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
INTO {define_variable[, define_variable]... | record};
```

Continuing the example, fetch rows from cursor variable EMP_CV into define variables MY_ENAME and MY_SAL:

```
LOOP
FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
EXIT WHEN emp_cv%NOTFOUND; -- exit loop when last row is fetched
-- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-c ompatible variable or field in the INTO clause. You can use a different INTO clause on separate fetches with the same cursor variable. Ea ch fetch retrieves another row from the same result set. If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Closing the C ursor Variable

The CLOSE statement disables a cursor variable. Aft er that, the associated result set is undefined. Use the following syntax:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this exa mple, when the la st row is processed, close cursor variable EMP_CV:

```
LOOP
FETCH emp_cv INTO my_ename, my_sal;
EXIT WHEN emp_cv%NOTFOUND;
-- process row
END LOOP;
CLOSE emp_cv; -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

```
DECLARE
TYPE EmpCurTyp IS REF CURSOR;
emp_cv EmpCurTyp;
emp_rec employees%ROWTYPE;
sql_stmt VARCHAR2(200);
my_job VARCHAR2(10) := 'ST_CLERK';
BEGIN
sql_stmt := 'SELECT * FROM employees
WHERE job_id = :j';
OPEN emp_cv FOR sql_stmt USING my_job;
```

```
LOOP
FETCH emp_cv INTO emp_rec;
EXIT WHEN emp_cv%NOTFOUND;
-- process record
END LOOP;
CLOSE emp_cv;
END;
/
```

An Example of Fetching

The example shows that you can fetch rows from the result set of a dynamic multirow query i nto a record. F irst you must defi ne a REF CURSOR type, EmpCurTyp. Next you define a cursor variable emp_cv,ofthetype EmpcurTyp. In the executable s ection of the PL/SQL block, the OPEN-FOR statement associates the cursor variable EMP_CV with the multirow query, sql_stmt.TheFETCH statement returns a row from the result set of a multirow query and assigns the values of select-list items to EMP_REC in the INTO clause. When the last row is processed, close the cursor variable EMP_CV.