Oracle extensions to SQL: DECODE

- **DECODE** is a function which Oracle provides for conditional branching in an SQL query. The usage of **DECODE** can be represented as follows:

\[
\text{DECODE}(\text{value}, \text{if}_1, \text{then}_1, \ldots, \text{if}_n, \text{then}_n, \text{else})
\]

To evaluate a **DECODE** expression, first **value** is evaluated, and if there exists some **if** such that \(\text{if}_i = \text{value}\), and there is no \(j < i\) such that \(\text{if}_j = \text{value}\), then the expression \(\text{then}_i\) is evaluated. Otherwise, **else** is evaluated.

```sql
SQL> select DECODE((10/5), 1, 'VAL1', 2, 'VAL2', 2, 'VAL3', NULL) Whichval

WHICH
----
VAL2
```
Oracle extensions to SQL: PL/SQL

The PL/SQL programming language is a full procedural extension to SQL. Variable assignment, conditional expressions, looping, etc., are all present, as is a sophisticated module mechanism. Benefits of PL/SQL include the following:

- Greatly increased powers of expression in “native” database language for defining triggers, system procedures, etc.

- Allows the developer to provide a uniform and easily manipulated interface for applications which use the database.

- Promotes data abstraction and modularity.

- Language features provide easy and effective integration with the database.
Programming PL/SQL

● To compile a PL/SQL program, all you need to do is enter the program at the SQL*Plus prompt, either by typing it in manually, cut and pasting it from a file, or by using the start command to load a file (give it a .sql extension).

● If your program has compilation errors, SQL*Plus will not be very informative at first:

```
SQL> create procedure empty is begin end; /

Warning: Procedure created with compilation errors.
```

Use the show errors command to display error messages:

```
SQL> show errors
Errors for PROCEDURE EMPTY:

LINE/COL  ERROR
----------  ----------------------------------
 1/26       PLS-00103: Encountered the symbol "END"
```

Alternatively, we could say show errors procedure empty in this case.
The source code for every PL/SQL procedure and function you define is stored in the \texttt{user_source} view. Likewise, all your triggers are stored in the \texttt{user_trigger} view:

\begin{verbatim}
SQL> desc user_source;
Name Null? Type
-------------- ------- ----
NAME NOT NULL VARCHAR2(30) 
TYPE NOT NULL VARCHAR2(12) 
LINE VARCHAR2(2000) 
TEXT NUMBER  

SQL> select text from user_source
where name = 'SHOWWEATHER';

TEXT
--------------

procedure showweather is
ignore boolean...
\end{verbatim}

Once they are created, procedures, functions and triggers exist in Oracle just like any other database object. They have access permissions, can be dropped, modified, etc.
Language Overview

- The block is the basic component of all PL/SQL programs. The basic structure of an anonymous block is as follows:

```sql
DECLARE
    variable declarations
BEGIN
    main program logic
EXCEPTION
    exception handling
END;
```

The variable declaration and exception handling sections may optionally be omitted.

Here’s the requisite first example:

```sql
DECLARE
    msg VARCHAR2(15);
BEGIN
    msg := ’Hello World!’;
    dbms_output.put_line(msg);
END;
```
• A procedure is just a named block; this naming will allow the DECLARE keyword to be omitted:

```sql
CREATE OR REPLACE PROCEDURE HiWorld
IS
    msg   VARCHAR2(15);
BEGIN
    msg := 'Hello World!';
    dbms_output.put_line(msg);
END;
```

Procedures may take arguments:

```sql
CREATE OR REPLACE PROCEDURE Hello(who IN VARCHAR2)
IS
    msg   VARCHAR2(15);
BEGIN
    msg := 'Hello ' || who || '!';
    dbms_output.put_line(msg);
END;
```

```
SQL> exec Hello('Chris');
Hello Chris!
```

Note the IN keyword; this specifies that the variable `who` is read-only. To specify a variable as write only, use `OUT`, and use `IN OUT` if the variable is to be read and written to.
• PL/SQL functions are very similar to procedures, except that parameters are read-only, and a return type must be specified. Most importantly, functions are used as parts of expressions, and are not themselves PL/SQL programs.

```
CREATE OR REPLACE FUNCTION fact(n number)
RETURN number
IS
BEGIN
  if n = 0 then
    return 1;
  else
    return n * fact(n-1);
  end if;
END;
```

```
SQL> select fact(4) from dual;

   FACT(4)
----------
      24
```

Note that PL/SQL supports recursion.
• Triggers can also be defined in PL/SQL. The syntax is similar to procedures and functions, with special conventions for specifying the trigger event (i.e., before or after an INSERT, UPDATE or DELETE on a particular table):

    CREATE OR REPLACE TRIGGER plays_for_insert
    AFTER INSERT ON plays_for
    FOR EACH ROW
    BEGIN
        UPDATE teamcount
        SET cnt = cnt + 1
        WHERE team_name = :new.team;
    END;

The FOR EACH ROW clause specifies that the trigger should be fired once for each row involved in the transaction (“row level”). The :new syntax allows the programmer to use transaction insert values.
Here's another example, which demonstrates the use of the :old keyword:

```sql
CREATE OR REPLACE TRIGGER account_update
BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
    UPDATE account
    SET old_balance = :old.balance
    WHERE acct_num = :new.account_num;
END;
```

In general, if FOR EACH ROW is not specified, then the trigger is “statement level”, and will be fired only once *per transaction*. If a trigger is statement level, then the usage of :new and :old is disallowed (why?).

Note the use of the BEFORE and AFTER keywords in these examples, which specify whether the trigger is to be fired before or after the transaction. In practice this ordering can be significant.
Types and variable declaration

- The syntax for declaring variables is:

  `<varname> <type>;

The datatypes in PL/SQL are a superset of those in SQL (i.e., NUMBER, VARCHAR2, etc., are all present). Additionally, PL/SQL provides the *table* type, which is similar to an array, and also the *record* type, which is like a Pascal record.

- Table types must be defined before a variable of that table type can be declared:

  ```declare
  TYPE num_table IS TABLE OF NUMBER
  INDEX BY BINARY INTEGER;
  ex_table num_table;
  begin
  ...```

It is important to note that PL/SQL tables do not have size constraints (i.e., they can grow arbitrarily large, modulo system limits), and PL/SQL table indices need not sequential.
• For example, the following insertions can be made in the execution section of the block:

```plaintext
ex_table(1000) := 333;
ex_table(-2) := 777;
```

After these expressions are evaluated, `ex_table` will contain two values (333 and 777), indexed by 1000 and -2.

• PL/SQL records must also be defined before they’re declared:

```plaintext
DECLARE
  TYPE plays_for_rec IS RECORD (
    pname VARCHAR2(20),
    tname VARCHAR2(20)
  )
  on_squad plays_for_rec;
BEGIN
  ...

Fields in a record are accessed by the usual syntax:

```plaintext
on_squad.pname := 'Jerome Bettis'
on_squad.tname := 'Steelers'
```
• A very nice feature of records is that they can allow for a very simple interaction with SQL statements in PL/SQL procedures:

```sql
DECLARE
    TYPE plays_for_rec IS RECORD (
        pname       VARCHAR2(20),
        tname       VARCHAR2(20)
    );

    on_squad   plays_for_rec;
BEGIN
    SELECT name, team
    INTO on_squad
    FROM plays_for
    WHERE name = 'Jerome Bettis'
END;
```

Note the use of the INTO keyword, which allows for variable assignment in the SQL statement.

Of course, concrete type definitions for variables which manipulate database values can lead to problems, if a relevant table’s type definitions change. For example, suppose we change the team attribute of plays_for to a number id at some point; what happens when you we try to execute this block?
• This problem is resolved with the use of the 
%TYPE construct, which allows type definitions 
which will always match the type of the spec-
ified column:

    DECLARE
        TYPE plays_for_rec IS RECORD (  
pname plas_for.name%TYPE,
tname plas_for.team%TYPE  
);
on_squad plays_for_rec;
BEGIN
    SELECT name, team
    INTO on_squad
    FROM plays_for
    WHERE name = 'Jerome Bettis';
END;

With the %ROWTYPE construct, things can be
made even simpler through the implicit defi-
nition of records which match the field names
and types of a database row:

    DECLARE
        on_squad plays_for%ROWTYPE;
BEGIN
    SELECT *
    INTO on_squad
    FROM plays_for ...
Control structures

- **PL/SQL** provides syntax for conditional expressions:

```sql
IF val < 100 THEN
    val_comment := 'small';
ELSEIF val BETWEEN 100 AND 1000;
    val_comment := 'medium';
ELSE
    val_comment := 'large';
END IF;
```

In general, the ELSE and each ELSEIF clause is optional. The expression is evaluated in the obvious manner.

- **PL/SQL** also provides looping constructs, which include the "simple" loop:

```sql
LOOP
    EXIT WHEN loop_counter >= 100;
    dbms_output.put_line('looping...');
    loop_counter := loop_counter + 1;
END LOOP;
```
• For more structured programming, PL/SQL provides FOR and WHILE loop constructs:

```
WHILE loop_counter < 100 LOOP
    dbms_output.put_line('looping...');
    loop_counter := loop_counter + 1;
END LOOP;

FOR loop_counter IN 1..99 LOOP
    dbms_output.put_line('looping...');
END LOOP;
```

In a numeric FOR loop, the loop counter is implicitly declared as a BINARY INTEGER, assigned the lowest number in the specified range, and incremented after each iteration.

• Recall that PL/SQL supports recursion. However, using recursion in PL/SQL programs is not advisable; an iterative approach is better.
Cursors

- Another helpful PL/SQL feature for db interaction is the cursor. A cursor is essentially a row pointer for the relation defined by an SQL select statement. For example:

```sql
DECLARE
    on_squad   plays_for%ROWTYPE;
CURSOR get_contract IS
    SELECT *
    FROM plays_for;
BEGIN
    OPEN get_contract;
    FETCH get_contract INTO on_squad;
    dbms_output.put_line(on_squad.team);
    CLOSE get_contract;
END;
```

When the cursor get_contract is OPENed, the relevant SQL statement is evaluated (yielding the “active set”), and the “active set pointer” associated with get_contract is assigned to the first row in the active set. Each subsequent FETCH statement assigns the values in the active row to the variable list or record following the INTO keyword, and then the active set pointer is incremented to the next row. Cursors should be CLOSEed at the end of the block.
• PL/SQL provides special constructs for looping with cursors:

```sql
DECLARE
    on_squad   plays_for%ROWTYPE;
CURSOR get_contract IS
    SELECT *
    FROM plays_for;
BEGIN
    OPEN get_contract;
    FETCH get_contract INTO on_squad;
    WHILE get_contract%FOUND LOOP
        dbms_output.put_line(on_squad.team);
        FETCH get_contract INTO on_squad;
    END LOOP;
    CLOSE get_contract;
END;
```

For any cursor curs, the expression curs%FOUND evaluates to true iff the last FETCH command returned a row. Hence, the above block will loop through every tuple in the plays_for table. Other attributes of cursors include %NOTFOUND (which is the negation of %FOUND), %ISOPEN (which is true iff the associated cursor is open), and %ROWCOUNT (which returns the number of rows fetched so far).
• Of course, the simplest loop construct for cursor iteration over some relation is the “cursor for loop”, which makes an implicit OPEN, FETCHes into an implicitly declared record of appropriate type, and implicitly CLOSEs on termination:

```
DECLARE
    CURSOR get_contract IS
        SELECT *
        FROM plays_for;
BEGIN
    FOR on_squad IN get_contract LOOP
        dbms_output.put_line(on_squad.team);
    END LOOP;
END;
```
Exceptions

- The exception handling segment of PL/SQL is easy to use:

```plsql
DECLARE
    program_var NUMBER;
    ...
BEGIN
    ...
EXCEPTION
    WHEN ZERO_DIVIDE
        INSERT INTO log_table VALUES (program_var, ...
END;
```

OTHERS Handles all exceptions.

Exceptions can be defined and explicitly raised:

```plsql
DECLARE
    my_exception EXCEPTION
    ...
BEGIN
    ...
    RAISE my_exception;
EXCEPTION
    WHEN my_exception
        INSERT INTO log_table ...
END;
```
Packages

- PL/SQL packages are essentially modules, and consist of a package specification and a package body, which must match the specification.

The following package specification provides an interface to a sports-oriented database:

```sql
CREATE OR REPLACE PACKAGE ShowStats AS

    PROCEDURE DefensiveStats
        (Player IN player.name%TYPE);

    PROCEDURE OffensiveStats
        (Player IN player.name%TYPE);

    PROCEDURE TeamStats(Team team.name%TYPE);

END;
```

The definitions for the procedures in this package must be made in a matching body.
A package body must contain definitions for at least the entities declared in the package specification:

CREATE OR REPLACE PACKAGE BODY ShowStats AS

PROCEDURE DefensiveStats
   (Player IN player.name%TYPE)
   IS
   ...
   END;

PROCEDURE OffensiveStats
   (Player IN player.name%TYPE)
   IS
   ...
   END;

PROCEDURE TeamStats(TEAM team.name%TYPE)
   IS
   ...
   END;

FUNCTION GetTeamId(TEAM team.name%TYPE)
   RETURNS NUMBER
   IS
   ...
   END;

END;
• Any package body which does not match the specification will not compile:

```sql
/* Error! body doesn’t satisfy specification */
CREATE OR REPLACE PACKAGE BODY ShowStats AS
  PROCEDURE DefensiveStats
    (Player IN player.name%TYPE)
    IS
    ...
  END;
END;
```

• In order to access the procedures, types and functions in a package:

```sql
ShowStats.TeamStats('Ravens')
```

• However, any procedures, functions or types which are defined in the package body, but not declared in the specification, cannot be accessed:

```sql
/* Error! GetTeamId is ‘private’ */
ShowStats.GetTeamId('Ravens')
```
Other hints

● Code commenting:
    -- Here’s the one line variety

    /* Comments can also be broken
     over multiple lines */

● A nasty fact: procedures can modify database tables, but functions cannot.

● The truth about dbms_output.put_line: to make it work, you need to run the command:
    SQL> set serveroutput on;

● Whenever possible, fully evaluate expressions before passing them to procedures and functions:
    some_procedure(var1 || var2); -- No!

    tmp := var1 || var2; -- Concatenate strings first
    some_procedure(tmp); -- Pass value to procedure

● Try to be extra careful about program termination (PL/SQL doesn’t handle memory overflow very well).