PL/SQL Overview

PL/SQL is Procedural Language extension to SQL. It is loosely based on Ada (a variant of Pascal developed for the US Dept of Defense). PL/SQL was first released in ١٩٩٢ as an optional extension to Oracle ٦. It is exclusive product of Oracle Corporation

PL/SQL, is an application-development language that is a superset of SQL, supplementing it with standard programming-language features that include the following:

- block (modular) structure
- flow-control statements and loops
- variables, constants, and types
- structured data
- customized error handling

Another feature of PL/SQL is that it allows to store compiled code directly in a database. This enables any number of applications or users to share same functions and procedures. In fact, once a given block of code is loaded into memory, any number of users can use copy of the code simultaneously. PL/SQL also enables to define triggers, which are subprograms that a database executes automatically in response to specified events.

Basic Structure and Syntax of PL/SQL

PL/SQL, groups the syntax of the programs into units called *blocks*. These blocks can either named or unnamed. The named blocks are called *subprograms and* unnamed blocks are called *anonymous blocks*. Subprograms can be referred as either functions or procedures. The difference between functions and procedures is that a function can be used in an expression and it returns a value to that expression. While a procedure is invoked as a standalone statement and passes values to the calling program only through parameters. Subprograms can be nested within one another and can be grouped in larger units called *packages*.

A block has three parts:

- A DECLARE section. In this section one can define local variables, constants, types, exceptions, and nested subprograms.

- An EXECUTABLE section. In this is the actual code of a block gets executed. This part of the block must be always present in the program.

- An EXCEPTION section. This section is used for handling runtime errors and warnings.

The DECLARE Section

The DECLARE section begins with the keyword DECLARE and ends when the keyword BEGIN. The next section that follows is the EXECUTABLE section. One can declare types, constants, variables, exceptions, and cursors in any order, as long as they are declared before they are referenced in the program. Subprograms declared last. A semi-colon terminates each definition.

Datatypes

PL/SQL provides a number of predefined data types for variables and constants. It also enables you to define your own types, which are subtypes of the predefined types. The types fall into the following three categories:

- Scalar. These include all string, number, and binary types.
- Composite. These are structured data types. The PL/SQL composite types are TABLE and RECORD.
- Reference. There is one kind of reference data type--REF CURSOR--which is a pointer to a cursor..

In many cases, one can convert from one data type to another, either explicitly or automatically.

One can also define a variable so that it inherits its data type from a database column or from another variable or constant.

Declaring Variables

Given below is the syntax of declaring a variable

cnum INTEGER(°) NOT NULL;

It declares a five-digit integer called *cnum* that will not accept nulls.

Inheriting Datatypes To inherit the data type of a database column or of another variable, one can use the %TYPE attribute in place of a declared data type, as follows:

```
snum cnum%TYPE;
```

This means that *snum* inherits the datatype of *cnum*. You can inherit datatypes from database columns in the same way, by using the notation *tablename.columname* in place of the variable name.

Declaring Constants

Constants can be declared the same way as variables, except adding the keyword CONSTANT and assigning a value. Constants do not take attributes other than the value.

An example of contact is follows:

```
interest CONSTANT REAL(٥,٢) := ٧٥٩.٣٢;
```

Defining Types

User-defined types in PL/SQL are subtypes of existing data types. They provide you with the ability to rename types and to constrain them by specifying for your subtype lengths, maximum lengths, scales, or precisions, as appropriate to the standard datatype on which the subtype is based.

For example:

SUBTYPE shortnum IS INTEGER(٣); This defines SHORTNUM as a ٣-digit version of INTEGER.

Scope and Visibility

Nested subprograms, defined in the DECLARE section, can be called from either of the other sections, but only from within the same block where they are defined or within blocks contained in that block. Variables, constants, types, and subprograms defined within a block are local to the block, and their definitions are not applicable outside of the block. Objects that are local to a block may be used by subprograms contained at any level of nesting in the block. Such objects are global to the block that calls them.

The area of a program within which an object can be used is called the object's scope. An object's scope is distinct from its visibility. The former is the area of the program that can reference the object; the latter is the, generally smaller, portion that can reference it without qualification.

Data Structures

PL/SQL provides two structured data types: TABLE and RECORD. It also provides a data structure called a cursor that holds the results of queries. Cursors are different from the other two in that you declare variables and constants to be of type TABLE or RECORD just as you would any other data type. Cursors, on the other hand, have their own syntax and their own operations. Explanations of these types follow:

PL/SQL Tables: These are similar to database tables, except that they always consist of two columns: a column of values and a primary key. This also makes them similar to one-dimensional arrays, with the primary key functioning as the array index. Like SQL tables, PL/SQL tables have no fixed allocation of rows, but grow dynamically. One of their main uses is to enable you to pass entire columns of values as parameters to subprograms. With a set of such parameters, you can pass an entire table. The primary key is always of type BINARY_INTEGER, and the values can be of any scalar type.

You declare objects of type TABLE in two stages:

١. You declare a subtype using the following syntax:

```
TYPE type_name IS TABLE OF

        datatype_spec

        [ NOT NULL ]

        INDEX BY BINARY INTEGER;
```

Where datatype_spec means the following:

```
datatype | variablename%TYPE | tablename.columname%TYPE
```

In other words, you can either specify the type of values directly or use the %TYPE attribute to inherit the data type from an existing variable or database column.

٢. You assign objects to this subtype in the usual way. You cannot assign initial values to tables, so the first reference to the table in the EXECUTABLE section must provide it at least one value.

When you reference PL/SQL tables, you use an array-like syntax of the form:

```
column_value(primary_key_value)
```

In other words, the third row (value) of a table called "Employees" would be referenced as follows:

```
Employees(٣)
```

You can use these as ordinary expressions. For example, to assign a value to a table row, use the following syntax:

```
Employees(٣) := 'Marsha';
```

Records As in many languages, these are data structures that contain one or more fields. Each record of a given type contains the same group of fields with different values. Each field has a datatype, which can be RECORD. In other words, you can nest records, creating data structures of arbitrary complexity. As with tables, you declare records by first declaring a subtype, using the following syntax:

```
TYPE record_type IS RECORD

        (fieldname datatype[, fieldname datatype]...);
```

The second line of the above indicates a parenthesized, comma-separated, list of fieldnames followed by data type specifications. The data type specifications can be direct or be inherited using the %TYPE attribute.

You can also define a record type that automatically mirrors the structure of a database table or of a cursor, so that each record of the type corresponds to a row, and each field in the record corresponds to a column. To do this, use the %ROWTYPE attribute with a table or cursor name in the same way you would the %TYPE attribute with a variable, or column. The fields of the record inherit the column names and data types from a cursor or table.

Cursors A cursor is a data structure that holds the results of a query (a SELECT statement) for processing by other statements. Since the output of any query has the structure of a table, you can think of a cursor as a temporary table whose content is the output of the query.

Exceptions

The DECLARATION section can also be used to define your own error conditions, called "exceptions".

Declaring Subprograms

You must place all subprogram declarations at the end of the declare section, following all variable, constant, type, and exception declarations for a block. The syntax is as follows:

```
PROCEDURE procedure_name (parameter_name datatype,

parameter_name datatype...) IS

        {local declarations}

        BEGIN {executable code}

        EXCEPTION

        END;
```

The names you give the parameters in the declaration are the names that the procedure itself uses to refer to them. These are called the *formal parameters*. When the procedure is invoked, different variables or constants may be used to pass values to or from the formal parameters; these are called the *actual parameters*.

When calling the procedure, you can use each parameter for input of a value to the procedure, output of a value from it, or both. These correspond to the three *parameter modes*: IN, OUT, and IN/OUT.

Functions are the same, except for the addition of a return value, specified as follows:

```
FUNCTION function_name (parameter_name, parameter_name datatype...)

        RETURN datatype IS

        {local declarations}

        BEGIN {executable code}

        EXCEPTION {local exception handlers}

        END;
```

A RETURN statement in the executable section actually determines what the return value is. This consists of the keyword RETURN followed by an expression. When the function executes the RETURN statement, it terminates and passes the value of that expression to whichever statement called it in the containing block.

The EXECUTABLE Section

The executable section is the main body of code. It consists primarily of SQL statements, flow control statements, and assignments.

Assignments

The assignment operator is :=. For example, the following statement assigns the value ٤٥ to the variable a:

```
a := ٤٥;
```

Character strings should be set off with single quotes (') as in all expressions. An example follows:

```
FNAME := 'Clair';
```

Flow Control

PL/SQL supports the following kinds of flow-control statements:

- IF statements. These execute a group of one or more statements based on whether a condition is TRUE.
- Basic loops. These repeatedly execute a group of one or more statements until an EXIT statement is reached.
- FOR loops. These repeatedly execute a group of one or more statements a given number of times or until an EXIT statement is reached.
- WHILE loops. These repeatedly execute a group of one or more statements until a particular condition is met or an EXIT statement is reached.
- GOTO statements. These pass execution directly to another point in the code, exiting loops and enclosing blocks as necessary. Use these sparsely, as they make code difficult to read and debug.

IF Statements

These are similar to the IF statement in many other programming languages.

The IF statement has the following forms:

```
١       IF <condition> THEN <statement-list>;

        END IF;
```

If the condition following IF is TRUE, PL/SQL executes the statements in the list following THEN. A semicolon terminates this list. END IF (not ENDIF) is mandatory and terminates the entire IF statement. Here is an example:

```
        IF balance > ٥٠٠ THEN send_bill(customer);

        END IF;
```

We are assuming that send_bill is a procedure taking a single parameter.

```
٢       IF <condition> THEN <statement-list>;

        ELSE <statement-list>;

        END IF;
```

This is the same as the preceding statement, except that, if that condition is FALSE or NULL, PL/SQL executes the statement list following ELSE instead of that following THEN.

```
٣       IF <condition> THEN <statement-list>;

        ELSIF <condition> THEN <statement-list>;
```

```
        ELSIF <condition> THEN <statement-list>;.....

        ELSE <statement-list>;

        END IF;
```

You can include any number of ELSIF (not ELSEIF) conditions. Each is tested only if the IF condition and all preceding ELSIF conditions are FALSE or NULL. As soon as PL/SQL finds an IF or ELSIF condition that is TRUE, it executes the associated THEN statement list and skips ahead to END IF. The ELSE clause is optional, but, if included, must come last. It is executed if all preceding IF and ELSIF conditions are FALSE or NULL.

Basic Loops

A basic loop is that keeps repeating the executing the program statements until an EXIT statement is reached. The EXIT statement must be within the loop itself. If there is no EXIT (or GOTO) statement present in the program, the loop will be infinite.

For example follows:

```
credit := ٠;

 LOOP

        IF c = ٥ THEN EXIT;

        END IF;

        credit := credit + ١;

END LOOP;
```

This loop keeps incrementing credit until it reaches ٥ and then exits. An alternative to placing an exit statement inside an IF statement is to use the EXIT-WHEN syntax, as follows:

```
EXIT WHEN credit = ٥;
```

FOR Loops

A FOR loop, as in most languages, repeats a group of statements a given number of times. The following FOR loop is equivalent to the example used for basic loops, except that it also changes a variable called interest.

```
FOR credit IN ١..٥ LOOP

        interest := interest * ١.٢;

END LOOP;
```

The numbers used to specify the range (in this case, ١ and ٥) can be variables, so you can let the number of iterations of the loop be determined at runtime if you wish.

WHILE Loops

A WHILE loop repeats a group of statements until a condition is met. Here is a WHILE loop that is the equivalent of the preceding example:

```
credit := ١;

WHILE credit <= ٥ LOOP

        interest := interest * ١.٢;

        credit := credit + ١;

END LOOP;
```

GOTO Statements

A GOTO statement immediately transfers execution to another point in a program. The point in the program where the statement is to arrive must be preceded by a label. A label is an identifier for a location in the code. It must be unique within its scope and must be enclosed in double angle brackets, as follows:

```
<<this_is_a_label>>
```

You only use the brackets at the target itself, not in the GOTO statement that references it, so a GOTO statement transferring execution to the above label would be:

```
GOTO this_is_a_label;
```

A GOTO statement is subject to the following restrictions:

- It must branch to an executable statement, not, for example, an END.
- It cannot branch to a point within the body of IF or a LOOP statement, unless it is contained in the body of that statement itself.
- It cannot branch to a subprogram or enclosing block of the present block (with one exception, explained shortly).
- It cannot branch from one IF statement clause to another. That is to say, it cannot jump between THEN, ELSIF, and ELSE clauses that are part of the same IF statement.

- It cannot branch from the EXCEPTION section to the EXECUTABLE section of the same block.

- It can, however, branch from the EXCEPTION section of a block to the EXECUTABLE section of an enclosing block, which is the exception to the third rule above.

The EXCEPTION Section

The EXCEPTION section follows the END that matches the BEGIN of the EXECUTABLE section and begins with the keyword EXCEPTION. It contains code that responds to runtime errors. An *exception* is a specific kind of runtime error. When that kind of error occurs, you say that the exception is *raised*. An *exception handler* is a body of code designed to handle a particular exception or group of exceptions. Exception handlers, like the rest of the code, are operative only once the code is compiled and therefore can do nothing about compilation errors.

There are two basic kinds of exceptions: predefined and user-defined. The predefined exceptions are provided by PL/SQL in a package called STANDARD. They correspond to various runtime problems that are known to arise often--for example, dividing by zero or running out of memory.

The Oracle Server can distinguish between and track many more kinds of errors than the limited set that STANDARD predefines. Each of Oracle's hundreds of messages are identified with a number, and STANDARD has simply provided labels for a few of the common ones. You can deal with the other messages in either or both of two ways:

- You can define your own exception labels for specified Oracle messages using a pragma (a compiler directive). This procedure will be explained shortly.

- You can define a handler for the default exception OTHERS. Within that handler, you can identify the specific error by accessing the built-in functions SQLCODE and SQLERRM, which contain, respectively, the numeric code and a prose description of the message.

You can also define your own exceptions as will be shown. It is usually better, however, to use Oracle exceptions where possible, because then the conditions are tested automatically when each statement is executed, and an exception is raised if the error occurs.

Declaring Exceptions

PL/SQL predefined exceptions, of course, need not be declared. You declare user-defined exceptions or user-defined labels for Oracle messages in the DECLARE section, similarly to variables. An example follows:

```
customer_deceased EXCEPTION;
```

In other words, an identifier you choose followed by the keyword EXCEPTION. Notice that all this declaration has done is provide a name. The program still has no idea when this exception should be raised. In fact, there is at this point no way of telling if this is to be a user-defined exception or simply a label for an Oracle message.

Labeling Oracle Messages

If a previously-declared exception is to be a label for an Oracle error, you must define it as such with a second statement in the DECLARE section, as follows:

```
PRAGMA EXCEPTION_INIT (exception_name, Oracle_error_number);
```

A PRAGMA is a instruction for the compiler, and EXCEPTION_INIT is the type of PRAGMA. This tells the compiler to associate the given exception name with the given Oracle error number. This is the same number to which SQLCODE is set when the error occurs. The advantage of this over defining your own error condition is that you pass the responsibility for determining when the error has occurred and raising the exception to Oracle..

User-Defined Exceptions

If a declared condition is to refer a user-defined error, in the EXECUTABLE section, you must test the situation that is intend the exception to handle whenever appropriate and raise the condition manually, if needed.

Here is an example:

```
IF cnum < · THEN RAISE customer_deceased;
```

You can also use the RAISE statement to force the raising of predefined exceptions.

Handling Exceptions

Once an exception is raised, whether explicitly with a RAISE statement or automatically by Oracle, execution passes to the EXCEPTION section of the block. If a handler for the raised exception is not found in the current block, enclosing blocks are searched until one is found. If PL/SQL finds an OTHERS handler in any block, execution passes to that handler.

This is the syntax of an exception handler:

```
WHEN exception_condition THEN statement_list;
```

The exception is the identifier for the raised condition. If desired, you can specify multiple exceptions for the same handler, separated by the keyword OR. The exception can be either one the package STANDARD provided or one you declared. The statement

list does what is appropriate to handle the error--writing information about it to a file, for example--and arranges to exit the block gracefully if possible. Although exceptions do not necessarily force program termination, they do force the program to exit the current block.

Storing Procedures and Functions in the Database:

To create a procedure or function that is to be stored as a database object, we issue a CREATE PROCEDURE or a CREATE FUNCTION statement directly on a server using SQL*PLUS or Server Manager. The easy way to do this is to use a ordinary text editor to write the CREATE statement and then to load it as a script. This approach is recommended because often a group of procedures and functions are created in a batch. These groups are called "packages".

The syntax for these statements is slightly different than that is used to declare subprograms in PL/SQL, as the following example shows:

```
CREATE PROCEDURE fire_employee (empno INTEGER) IS

        BEGIN

                DELETE FROM Employees WHERE enum = empno;

        END;
```

As you can see, the main difference is the addition of the keyword CREATE. You also have the option of replacing the keyword IS with AS, which does not affect the meaning. To replace an existing procedure of the same name with this procedure you can use CREATE OR REPLACE instead of simply CREATE. This destroys the old version, if any, without warning.

Privileges Required

A stored procedure or function is a database object like a table. It resides in a schema, and its use is controlled by privileges. To create a procedure and it compile it the following conditions should be fulfilled.

- To create a users own procedure an user should have CREATE PROCEDURE or the CREATE ANY PROCEDURE system privilege. Same privileges also apply for creating functions too.

- If a procedure is in a schema that an user owns the user must have the CREATE ANY PROCEDURE system privilege.

- You must have the object privileges necessary to perform all operations contained in the procedure. You must have these privileges as a user, not through roles. If

your privileges change after you have created the procedure, the procedure may no longer be executable.

To enable others to use the procedure, grant them the EXECUTE privilege on it using the SQL statement GRANT .When these users execute the procedure, they do so under your privileges, not their own. Therefore, you do not have to grant them the privileges to perform these actions outside the control of the procedure, which is a useful security feature. To enable all users to use the procedure, grant EXECUTE to PUBLIC. The following example permits all users to execute a procedure called show_product.

```
GRANT EXECUTE ON show_product TO PUBLIC;
```

Of course, the public normally does not execute such a procedure directly. This statement enables you to use the procedure in your PL/SQL code that is to be publicly executable. If multiple users access the same procedure simultaneously, each gets his own instance. This means that the setting of variables and other activities by different users do not affect one another.

Packages

A package is a group of related PL/SQL objects (variables, constants, types, and cursors) and subprograms that is stored in the database as a unit. Being a database object, a package resides in a schema, and its use is controlled by privileges. Among its differences from regular PL/SQL programs are that a package as such does not do anything. It is a collection of subprograms and objects, at least some of which are accessible to applications outside of it. It is the subprograms in the package that contain the executable code. A package has the following two parts:

- The package specification is the public interface to the package. It declares all objects and subprograms that are to be accessible from outside the package. Packages do not take parameters, so these constitute the entire public interface.

- The package body is the internal portion of the package. It contains all objects and subprograms that are to be local to the package. It also contains definitions of the public cursors and subprograms. The package specification declares but does not define these.

One of the advantages of using packages is that the package specification is independent of the body. You can change the body and, so long as it still matches the specification, no changes to other code are needed, nor will any other references become invalid.

Packages cannot be nested, but they can call one another's public subprograms and reference one another's public objects.

Instantiation of Packages

It is important to realize that a package is instantiated once for a given user session. That is to say, the values of all variables and constants, as well as the contents and state of all cursors, in a package, once set, persist for the duration of the session, even if you exit the package. When you reenter the package, these objects retain the values and state they had before, unless they are explicitly reinitialized. Of course, another user has another session and therefore another set of values. Nonetheless, a global reinitialization of a package's objects for you does not take place until you disconnect from the database.

There is an exception, however. When one package calls another, execution of the second has a dependency on the first. If the first is invalidated, for example because its creator loses a privilege that the package requires, the second, while not necessarily invalidated, becomes deinstantiated. That is to say, all its objects are reinitialized.

Creating Packages

To create a package, you use the SQL statement CREATE PACKAGE for the specification and CREATE PACKAGE BODY for the body. You must create the specification first. Sometimes, a package may consist of only public variables, types, and constants, in which case no body is necessary. Generally, however, you use both parts.

Creating the Package Specification The syntax of the CREATE PACKAGE statement is as follows:

```
CREATE [OR REPLACE] PACKAGE package_name IS

        {PL/SQL declarations}

        END;
```

The optional OR REPLACE clause operates just as it does for stored procedures, as explained elsewhere in this chapter. The PL/SQL declarations are as outlined under DECLARE SECTION elsewhere in this chapter, except that the keyword DECLARE is not used and that the subprogram and cursor declarations are incomplete. For subprograms, you provide only the name, parameters, and, in the case of functions, the datatype of the return value. For cursors, provide the name and a new item called the return type. This approach hides the implementation of these objects from the public while making the objects themselves accessible.

The syntax for declaring a cursor with a return type is as follows:

```
CURSOR c\ IS RETURN return_type;
```

The return type is always some sort of record type that provides a description of the cursor's output. The structure of this record is to mirror the structure of the cursor's rows. You can specify it using any of the following:

- A record subtype previously defined and in scope. For more information, see "Records" elsewhere in this chapter.

- A type inherited from such a record subtype using the %TYPE attribute. For more information, see "Declaring Variables" elsewhere in this chapter.

- A type inherited from a table, most likely the table the cursor queries, using the %ROWTYPE attribute. For more information, see "Records" elsewhere in this chapter.

- A type inherited from a cursor using the %ROWTYPE attribute. For more information, see "Records" elsewhere in this chapter.

Creating the Package Body To create the package body, use the CREATE PACKAGE BODY statement. The syntax is as follows:

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS

        {PL/SQL declarations}

        END;
```

Since a package as such does not do anything, the PL/SQL code still consists only of a DECLARE section with the keyword DECLARE omitted. It is the subprograms within the package that contain the executable code. Variables, constants, types, and cursors declared directly in the declare section have a global scope within a package body. Variables, constants, and types that are already declared in the package specification are public and should not be declared again.

Public cursors and subprograms, however, must be declared again here, as their declarations in the specification is incomplete. This time the declarations must include the PL/SQL code (in the case of subprograms) or the query (in the case of cursors) that is to be executed. For subprograms, the parameter list must match that given in the package specification word for word .This means, for example, that you cannot specify a data type directly in the specification and use the %TYPE attribute to specify it in the body.

You can create an initialization section at the end of the package body. This is a body of executable code--chiefly assignments--enclosed with the keywords BEGIN and END. Use this to initialize constants and variables that are global to the package, since otherwise they could be initialized only within subprograms, and you have no control of the order in which subprograms are called by outside applications. Initialization is performed only once per session.

Overloading Subprograms

Within a package, subprogram names need not be unique, even at the same level of scope. There can be multiple like-named subprograms in the same declare section,

provided that the parameters that they take differ in number, order, or datatype and that, when the procedures are called, the values passed by the calling procedure (the actual parameters) match or can be automatically converted to the datatypes specified in the declaration (the formal parameters).

The reason this is permitted is so you can overload subprograms. Overloading permits to have several versions of a procedure that are conceptually similar but behave differently with different parameters.

Database Triggers

Triggers are blocks of PL/SQL code that execute automatically in response to events. Database triggers reside in a database and respond to changes in data. These triggers need not to be confused with application triggers. Database triggers are a technology that superseded application triggers.

Triggers are created the way stored procedures and packages are created, by using text editors and run them using SQL*Plus or Server Manager. A trigger is like a package in that:

- It takes no parameters as such. It refers to, responds to, and possibly affects the data in a database.

- It cannot be directly called like a procedure. To fire (execute) a trigger, you must make a change in a database to respond.

Triggers can be classified in three ways:

- INSERT triggers, UPDATE triggers, and DELETE triggers. This is a classification based on a statement to which a trigger responds. The categories are not mutually exclusive, meaning one trigger can respond to any or all of these statements.

- Row triggers and statement triggers. Any of the above statements can affect any number of rows in a table at once. A row trigger is fired once for each row affected. A statement trigger is fired once for each statement, however many rows it affects.

- BEFORE triggers and AFTER triggers. This specifies whether the trigger is fired before or after the data modification occurs.

All three of these classifications apply to all triggers. So there are, for example, BEFORE DELETE OR INSERT statement triggers and AFTER UPDATE row triggers.

Creating Triggers

The syntax of the CREATE TRIGGER statement is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name

        BEFORE | AFTER

        DELETE | INSERT | UPDATE [OF column_list]

        ON table_name

        [ FOR EACH ROW [ WHEN predicate ] ]

        {PL/SQL block};
```

In the above, square brackets ([ ]) enclose optional elements. Vertical bars (|) indicate that what precedes may be replaced by what follows.

In other words, you must specify the following:

- A trigger name. This is used to alter or drop the trigger. The trigger name must be unique within the schema.
- BEFORE or AFTER. This specifies whether this is a BEFORE or AFTER trigger.
- INSERT, UPDATE, or DELETE. This specifies the type of statement that fires the trigger. If it is UPDATE, you optionally can specify a list of one or more columns, and only updates to those columns fire the trigger. In such a list, separate the column names with commas and spaces. You may specify this clause more than once for triggers that are to respond to multiple statements; if you do, separate the occurrences with the keyword OR surrounded by white space.
- ON table_name. This identifies the table with which the trigger is associated.
- PL/SQL Block. This is an anonymous PL/SQL block containing the code the trigger executes.

You optionally can specify the following:

- OR REPLACE. This has the usual effect.
- FOR EACH ROW [WHEN predicate]. This identifies a trigger as a row trigger. If omitted, it will be a statement trigger. Even if this clause is included, the WHEN clause remains optional. The WHEN clause contains a SQL predicate that is tested against each row the triggering statement alters. If the values in that row make the predicate TRUE, the trigger is fired; else it is not. If the WHEN clause is omitted, a trigger will fire for each altered row.

Here is an example:

```
CREATE TRIGGER give_bonus

        AFTER UPDATE OF sales

        ON salespeople

        FOR EACH ROW WHEN sales > ٨٠٠٠.٠٠

        BEGIN

        UPDATE salescommissions SET bonus = bonus + ١٥٠.٠٠;

        END;
```

This creates a row trigger called give_bonus. Every time the sales column of the salespeople table is updated, the trigger checks to see if it is over ٨٠٠٠.٠٠. If so, it executes the PL/SQL block, consisting in this case of a single SQL statement that increments the bonus column in the salescommissions table by ١٥٠.٠٠.

Privileges Required

To create a trigger in your own schema, you must have the CREATE TRIGGER system privilege and one of the following must be true:

- You own the table associated with a trigger.
- You have the ALTER privilege on the table associated with a trigger.
- You have the ALTER ANY TABLE system privilege.

To create a trigger in another user's schema, you must have the CREATE ANY TRIGGER system privilege. To create such a trigger, you precede the trigger name in the CREATE TRIGGER statement with the name of the schema wherein it will reside, using the conventional dot notation.

Referring to Altered and Unaltered States

You can use the correlation variables OLD and NEW in the PL/SQL block to refer to values in the table before and after the triggering statement had its effect. Simply precede the column names with these variables using the dot notation.

If these names are not suitable, you can define others using the REFERENCING clause of the CREATE TRIGGER statement, which is omitted from the syntax diagram above for the sake of simplicity..

Enabling and Disabling Triggers

Just because a trigger exists does not mean it is effective. If a trigger is disabled, it does not fire. By default, all triggers are enabled when created, but you can disable a trigger using the ALTER TRIGGER statement. To do this, a trigger must be in your schema, or you must have the ALTER ANY TRIGGER system privilege.

Here is the syntax:

```
ALTER TRIGGER trigger_name DISABLE;
```

Later you can enable the trigger again by issuing the same statement with ENABLE in place of DISABLE. The ALTER TRIGGER statement does not alter the trigger in any other way. To do that you must replace the trigger with a new version using CREATE OR REPLACE TRIGGER..

Source: http://lambda.uta.edu/cse٥٣٣١/spring٩٨/plsql.html