Subject: **Using DBMS_SQL Package to Execute DDL Statements and Dynamic SQL from PL/SQL**

Doc ID: **1008453.6** Type: **BULLETIN**

Modified Date : **29-APR-2009** Status: **PUBLISHED**

In this Document

Applies to:

PL/SQL - Version: 8.1.7.0 to 11.1.0.7
Information in this document applies to any platform.

Purpose

This article describes how to use the DBMS_SQL package provided with PL/SQL 2.1 to execute Data Definition Language (DDL) statements and dynamic SQL statements from PL/SQL.

Dynamic SQL allows you to use SQL statements where all or part of the statement is unknown untill runtime.

Scope and Application

Compiling a PL/SQL program involves resolving references to Oracle objects by looking up their definitions in the data dictionary and then binding storage addresses to program variables that will hold Oracle data.

PL/SQL uses static or early binding. This means that binding is done at compile time which increases efficiency because the definitions of database objects are looked up at compile time, rather than at run time.

The limitation of this method is that the names of database objects must be known at compile time. In the past, this prevented data definition language (DDL) statements and dynamic SQL from being executed directly from PL/SQL.

The DBMS_SQL package, which was released with PL/SQL 2.1 (RDBMS 7.1), allows you to parse any data manipulation language (DML) or DDL statement, and it allows you to use dynamic SQL statements in your code. Dynamic SQL statements are stored in character strings that are input to, or built by,
the program at run time. This means that the names of database objects do not need to be known until run time. It also means that all or part of the SQL statement to be executed can be input at run time. This enables you to create more general purpose procedures.

The following bulletin describes the DBMS_SQL package and gives some examples of how to use it. For additional information, refer to the "Oracle® Database PL/SQL Packages and Types Reference ".

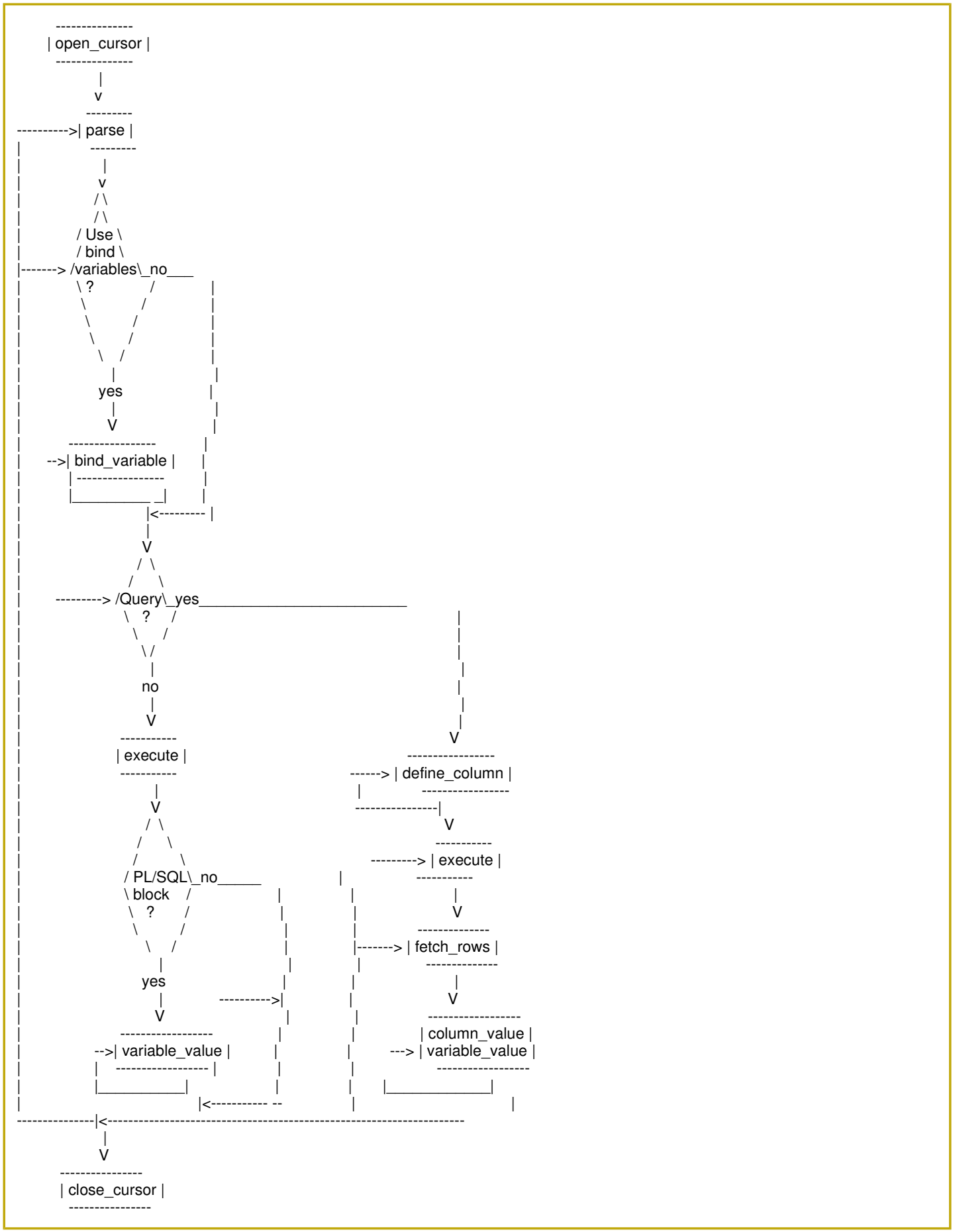
Using DBMS_SQL Package to Execute DDL Statements and Dynamic SQL from PL/SQL

Using DBMS_SQL to execute dynamic SQL is similar to the method used by the Oracle Call Interfaces (see the "Programmer's Guide to the ORACLE Call Interfaces") and dynamic SQL method 4 used by the Oracle Precompilers (see the "Programmer's Guide to the Oracle Precompilers").

DBMS_SQL Command	OCI Command	Precompiler Method 4 Command
open_cursor	oopen	EXEC SQL DECLARE...
parse	oparse	EXEC SQL PREPARE...
bind_variable	obndrn,obndrv,obndra	bind descriptor
---	odescr	EXEC SQL DESCRIBE SELECT LIST...
define_column	odefin	select descriptor
execute	oexec, oexn	EXEC SQL EXECUTE...
		EXEC SQL OPEN...
fetch_rows	ofetch, ofen	EXEC SQL FETCH
execute_and_fetch	oexfet	---

variable_value	---	bind descriptor
column_value	---	select descriptor
close_cursor	oclose	EXEC SQL CLOSE

The typical flow of procedure calls using DBMS_SQL is as follows:



DBMS_SQL Procedures and Functions

The following describes what the different procedures and functions inside the DBMS_SQL package do.

function open_cursor return integer :-

This function opens a new cursor. An open cursor is needed to process a SQL statement. The cursor can be used to execute the same SQL statement repeatedly or to execute a new SQL statement. When a cursor is reused, the parsing of the new SQL statement resets the contents of the corresponding cursor data area.

You can reuse a cursor without closing it and reopening it. The function returns a cursor ID number for the data structure representing a valid cursor maintained by Oracle. When you are finished with the cursor, you must explicitly close it by calling the procedure close_cursor.

function is_open (cursor in integer) return boolean :-

--cursor

The ID number of the cursor to check.

This function returns TRUE if the cursor is currently open, or FALSE if it is not.

procedure parse (cursor in integer, statement in varchar2, language_flag in integer)
-- cursor

The ID number of the cursor in which to parse the statement.

-- statement

The SQL statement to be parsed.

-- language_flag

This parameter determines how Oracle handles the SQL statement and can have one of the following values:

V6 specifies Version 6 behavior
V7 specified Oracle7 behavior
NATIVE specifies normal behavior for the database to which the program is connected

This procedure parses the given statement in the given cursor. Parsing the statement checks the statement's syntax and associates it with the cursor in your program. All statements must be parsed. Note that
parsing a DDL statement actually executes the statement and performs an implicit commit.

Note: When parsing a DDL statement to drop a package, a deadlock can occur if a procedure in the package is still in use by you. After a call to a procedure, that procedure is considered to be in use
until execution has returned to the user side. Any such deadlock will timeout after five minutes.

procedure bind_variable (cursor in integer,
name in varchar2,
value in <datatype>)
where <datatype> can be any one of the following types:
number, date, mlslabel

procedure bind_variable (cursor in integer,
name in varchar2,
value in varchar2 [, out_value_size in integer])

procedure bind_variable_char (cursor in integer,
name in varchar2,
value in char [, out_value_size in integer])

procedure bind_variable_raw (cursor in integer,
name in varchar2,
value in raw [, out_value_size in integer])

procedure bind_variable_rowid (cursor in integer,
name in varchar2,
value in rowid)

-- cursor

The ID number of the cursor to which you want to bind a value.

-- name

The name of the variable in the statement.

-- value

The value that you want to bind to the variable in the cursor. For IN and IN/OUT variables, the value has the same type as the type of the value being passed in for this parameter.

-- out_value_size

The maximum expected OUT value size, in bytes, for the VARCHAR2, RAW, and CHAR OUT or IN/OUT variable. If no size is given, the length of the current value parameter is used.

If you want to define a DML statement that contains input data to be supplied at runtime, you can use placeholders in the SQL statement to mark where data must be supplied. You then call the bind_variable
procedure for each placeholder in the SQL statement to supply the value of a variable in your program to the placeholder.

The bind_variable procedure allows you to bind a given value to a given variable in a cursor, based on the name of the variable in the statement. If the variable is an IN or IN/OUT variable, the given bind value must be valid for the variable type. Bind values for OUT variables are ignored. Bind variables in SQL statements are identified by name. When binding a value to a bind variable, the string identifying the bind variable in the statement must contain a leading colon, as shown in the following example:

SELECT emp_name FROM emp WHERE SAL > :x;

The bind call for the statement above would look like the following:

bind_variable (cursor, ':x', 3500);

procedure define_column (cursor in integer,
position in integer,
column in <datatype>
where <datatype> is one of the following:
number, date, mlslabel

procedure define_column (cursor in integer,
position in integer,
column in varchar2,
column_size in integer)

procedure define_column_char (cursor in integer,
position in integer,
column in char,
column_size in integer)

procedure define_column_raw (cursor in integer,
position in integer,
column in raw,
column_size in integer)

procedure define_column_rowid (cursor in integer,
position in integer,
column in rowid)

-- cursor

The ID number of the cursor for the row being defined to be selected.

-- position

The relative position of the column in the row being defined. The first column in a statement has position 1.

-- column

The value of the column being defined. The type of this value determines the type for the column being defined.

-- column_size

The maximum expected size of the column value, in bytes, for columns of type VARCHAR2, CHAR, and RAW.

This procedure is used only with SELECT cursors to define a column to be selected from the given cursor. The column being defined is identified by its relative position in the SELECT list of the statement in the given cursor. The type of the column parameter determines the type of the column being defined. The define_column procedure specifies the variables that are to receive the SELECT values for the dynamic query the way an INTO clause does for a static query.

function execute (cursor in integer) return integer :-

-- cursor

The ID number of the cursor to execute

This function executes the given cursor and returns the number of rows processed. The return value is only valid for INSERT, UPDATE, and DELETE statements; for other types of statements, the return value is undefined and should be ignored.

function fetch_rows (cursor in integer) return integer :-

-- cursor

The ID number of the cursor to fetch

This function tries to fetch a row from the given cursor. You can call FETCH_ROWS repeatedly as long as there are rows remaining to be fetched.

These rows are retrieved into a buffer, and must be read by calling COLUMN_VALUE, for each column, after each call to FETCH_ROWS. Function fetch_rows returns the number of rows actually fetched.

function execute_and_fetch (cursor in integer,
exact in boolean default false)
return integer

-- cursor

The ID number of the cursor to execute and fetch

-- exact

Set TRUE to raise an exception if the number of rows actually matching the query differs from one. Even if an exception is raised, the rows are still fetched and available.

This function executes the given cursor and fetches rows. It returns the number of rows actually fetched. Calling this function is the same as calling EXECUTE and then calling FETCH_ROWS. Calling EXECUTE_AND_FETCH instead, however, may cut down on the number of message round_trips when
used against a remote database.

procedure variable_value (cursor in integer,
name in varchar2,
value out <datatype>)
where <datatype> can be one of the following types:
number, date, mlslabel, varchar2

procedure variable_value_char (cursor in integer,
name in varchar2,
value out char)

procedure variable_value_raw (cursor in integer,
name in varchar2,
value out raw)

procedure variable_value_rowid (cursor in integer,
name in varchar2,
value out rowid)

-- cursor

The ID number of the cursor from which to get the values

-- name

The name of the variable for which you are retrieving the value

-- value

The value of the variable for the specified position

This procedure returns the value of the named variable for a given cursor.
For anonymous blocks containing calls to PL/SQL procedures, call procedure variable_value to retrieve the values assigned to the output variables of the PL/SQL procedures when they were executed.

Note: The exception, inconsistent_type (ORA-06562), is raised if the type of the output parameter, value, differs from the actual type of the value as it was originally defined by the call to the procedure
bind_variable.

procedure column_value (cursor in integer,
position in integer,
value out <datatype>
[, column_error out number]
[, actual_length out integer])

where <datatype> can be one of the following types: number, date, mlslabel, varchar2

procedure column_value_char (cursor in integer,
position in integer,
value out char
[, column_error out number]
[, actual_length out integer])

procedure column_value_raw (cursor in integer,
position in integer,
value out raw
[, column_error out number]
[, actual_length out integer])

procedure column_value_rowid (cursor in integer,
position in integer,
value out rowid
[, column_error out number]
[, actual_length out integer])

-- cursor

The ID number of the cursor from which you are fetching the values

-- position

The relative position of the column in the cursor. The first column in a statement has position 1.

-- value

The value of the column

-- column_error

Any error code for the specified column value

-- actual_length

The actual length, before any truncation, of the value in the specified column.

This procedure returns the value of the cursor element for a given position in a given cursor. For queries, call procedure column_value to determine the value of a column retrieved by a call to function fetch_rows.

Note: The exception, inconsistent_type (ORA-06562), is raised if the type of the output parameter differs from the actual type of the value, as defined by the call to procedure define_column.

procedure close_cursor (cursor in out integer)

-- cursor

The ID number of the cursor that you want to close

This procedure closes the given cursor. It should be called when you no longer need a cursor for a session.
Note: If you are using a gateway, you may need to close cursors at other times as well. Consult your gateways documentation for additional information.

When you call procedure close_cursor, the cursor is set to null, and the memory allocated to the cursor is released. If you neglect to close a cursor, the memory used by the cursor remains allocated even though it is no longer needed.

function last_error_position return integer :-

This function returns the byte offset in the SQL statement text where the error occurred. The first character in the SQL statement is at position 0.

Note: The value returned by this function is only meaningful immediately after a SQL statement is executed.
function last_row_count return integer :-

This function returns the cumulative count of the number of rows fetched.

Note: The value returned by this function is only meaningful immediately after a SQL statement is executed.
function last_row_id return rowid :-

This function returns the rowid of the last row processed.

Note: The value returned by this function is only meaningful immediately after a SQL statement is executed.
function last_sql_function code return integer :-

This function returns the SQL function code for the statement. You should only check this function immediately after the SQL statement is executed; otherwise, the return value is undefined.

Location of the DBMS_SQL Package

The DBMS_SQL package is owned by SYS. The script to create the package can be found on UNIX systems in $ORACLE_HOME/rdbms/admin/dbmssql.sql. When this script is run by SYS, a public synonym dbms_sql is created for sys.dbms_sql and execute on dbms_sql is granted to public. This script is automatically run by catproc.sql.

Roles and the DBMS_SQL Package

Roles are disabled inside stored procedures. For example, if the CREATE TABLE privilege is granted to you via a role, and you use the DBMS_SQL package to create a table, you will receive the Oracle error 'ORA-01031: insufficient privileges'. The solution is to grant the required privileges to the owner of the subprogram.

The reason for this is, unlike other Oracle supplied packages, the dbms_sql package, when used inside a stored subprogram, executes with the privileges of the owner of the subprogram, and not with the privileges of the caller.

Oracle 8i introduces invoker-rights for named PL/SQL. Therefore, roles are enabled for procedures created with AUTHID CURRENT_USER (i.e. Oracle supplied packages DBMS_SQL and DBMS_SYS_SQL). Thus, in Oracle 8i/above, the required privilege(s) can either be granted directly to the user or to the user via a role, including PUBLIC. Since Oracle supplied packages (i.e. DBMS_SQL and DBMS_SYS_SQL) are created with invoker-rights in 8i/above, roles can be granted the associated privileges.

Using the DBMS_SQL Package to Execute DDL statements

The DBMS_SQL package can be used to execute DDL statements directly from PL/SQL.

Example 1:

The following is a simple example of a procedure that creates any table. It takes two parameters: the name of the table and a list of columns and types.

```
CREATE OR REPLACE PROCEDURE ddlproc (tablename varchar2, cols varchar2) AS
  cursor1 INTEGER;
BEGIN
  cursor1 := dbms_sql.open_cursor;
  dbms_sql.parse(cursor1, 'CREATE TABLE ' || tablename || ' ( ' || cols || ' )', dbms_sql.v7);
  dbms_sql.close_cursor(cursor1);
end;
/

SQL> execute ddlproc ('MYTABLE','COL1 NUMBER, COL2 VARCHAR2(10)');

PL/SQL procedure successfully completed.

SQL> desc mytable;
 Name                           Null?    Type
 ------------------------------ -------- ----
 COL1                                    NUMBER
 COL2                                    VARCHAR2(10)
```

Note that DDL statements are executed by the parse command. Therefore, you cannot use bind variables with DDL statements or you will receive an error.

The example below is incorrect because it tries to use bind variables in a DDL statement.

**** Incorrect Example ****

```
CREATE OR REPLACE PROCEDURE ddlproc (tablename VARCHAR2,
                                     colname   VARCHAR2,
                                     coltype   VARCHAR2) AS
  cursor1 INTEGER;
  ignore  INTEGER;
BEGIN
  cursor1 := dbms_sql.open_cursor;
  dbms_sql.parse(cursor1, 'CREATE TABLE :x1 (:y1 :z1)', dbms_sql.v7);
  dbms_sql.bind_variable(cursor1, ':x1', tablename);
  dbms_sql.bind_variable(cursor1, ':y1', colname);
  dbms_sql.bind_variable(cursor1, ':z1', coltype);
  ignore := dbms_sql.execute(cursor1);
  dbms_sql.close_cursor(cursor1);
end;
/

Although you do not receive errors when the procedure is created, you
receive the error "ORA-00903: invalid table name" at run time.

SQL> execute ddlproc ('MYTABLE', 'COL1', 'NUMBER');
begin ddlproc ('MYTABLE', 'COL1', 'NUMBER'); end;

*
ERROR at line 1:
ORA-00903: invalid table name
ORA-06512: at "SYS.DBMS_SYS_SQL", line 239
ORA-06512: at "SYS.DBMS_SQL", line 25
ORA-06512: at "SCOTT.DDLPROC", line 8
ORA-06512: at line 1
```

Example 2:

The following is a procedure that drops any table. It takes the table name as a parameter.

```
create or replace procedure droptable (table_name varchar2) as
  cursor1 integer;
begin
  cursor1 := dbms_sql.open_cursor;
  dbms_sql.parse(cursor1, 'DROP TABLE ' || table_name, dbms_sql.v7);
  dbms_sql.close_cursor(cursor1);
end;
/

SQL> begin
  2    droptable('MYTABLE');
  3  end;
  4  /

PL/SQL procedure successfully completed.
```

Example 3:

The following is a procedure that executes any DDL statement. It takes the DDL statement as a parameter.

```
create procedure anyddl (s1 varchar2) as
  cursor1 integer;
begin
  cursor1 := dbms_sql.open_cursor;
  dbms_sql.parse(cursor1, s1, dbms_sql.v7);
  dbms_sql.close_cursor(cursor1);
end;
/

SQL> execute anyddl('CREATE TABLE MYTABLE (COL1 NUMBER)');

PL/SQL procedure successfully completed.

SQL> desc mytable;
 Name                           Null?    Type
 ------------------------------ -------- ----
 COL1                                    NUMBER

SQL> execute anyddl('drop table mytable');

PL/SQL procedure successfully completed.
```

Using the DBMS_SQL Package to Execute Dynamic SQL Statements:

The DBMS_SQL package can be used to execute dynamic SQL statements. These are statements in which all or part of the statement is unknown until runtime.

Example 4:

The following is a procedure that returns the name and employee number of all employees whose employee number is higher than a number that is supplied at run time. This is a fairly simple example that contains only one bind variable.

```
CREATE or REPLACE PROCEDURE rows_greater_than (low_value number) AS
cursor1 integer;
rows_processed integer;
myempno number;
myename varchar2(20);
BEGIN
cursor1 := dbms_sql.open_cursor;
dbms_sql.parse (cursor1, 'select empno, ename from emp where empno > :x',
dbms_sql.v7);
dbms_sql.bind_variable(cursor1, 'x', low_value);
dbms_sql.define_column (cursor1, 1, myempno);
dbms_sql.define_column (cursor1, 2, myename, 20);
rows_processed := dbms_sql.execute (cursor1);
loop
if dbms_sql.fetch_rows (cursor1) > 0 then
dbms_sql.column_value (cursor1, 1, myempno);
dbms_sql.column_value (cursor1, 2, myename);
dbms_output.put_line(to_char(myempno) || ' ' || myename);
else
exit;
end if;
end loop;
dbms_sql.close_cursor (cursor1);
EXCEPTION
WHEN OTHERS THEN
dbms_output.put_line(sqlerrm);
if dbms_sql.is_open (cursor1) then
dbms_sql.close_cursor (cursor1);
end if;
END;
/

Since this procedure uses the DBMS_OUTPUT package, you must first enter the
command SET SERVEROUTPUT ON.

SQL> set serveroutput on
SQL> execute rows_greater_than(7500);
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD
8100 MILLER

PL/SQL procedure successfully completed.

SQL> execute rows_greater_than(8000);
8100 MILLER

PL/SQL procedure successfully completed.
```

Example 5:

This example also returns employee names and numbers, but the WHERE clause is of the form:

column_name operator new_value

where:

'column_name' is the name of a column supplied at run time.

'operator' is supplied at run time and is one of the following:

<, <=, =, >=, >

'new_value' is a number supplied at runtime
Example 6:

You can also alter the SQL statement in the previous example so that the entire WHERE clause is unknown at run time.

```
CREATE or REPLACE PROCEDURE get_rows (column_name varchar2,
comparison_type varchar2,
new_value number) AS
cursor1 integer;
rows_processed integer;
myempno number;
myename varchar2(20);
BEGIN
cursor1 := dbms_sql.open_cursor;
dbms_sql.parse (cursor1, 'select empno, ename from emp
where ' || column_name ||
' ' || comparison_type || ' :x',
dbms_sql.v7);
dbms_sql.bind_variable(cursor1, 'x', new_value);
dbms_sql.define_column (cursor1, 1, myempno);
dbms_sql.define_column (cursor1, 2, myename, 20);
rows_processed := dbms_sql.execute (cursor1);
loop
if dbms_sql.fetch_rows (cursor1) > 0 then
dbms_sql.column_value (cursor1, 1, myempno);
dbms_sql.column_value (cursor1, 2, myename);
dbms_output.put_line(to_char(myempno) || ' ' || myename);
else
exit;
end if;
end loop;
dbms_sql.close_cursor (cursor1);
EXCEPTION
WHEN OTHERS THEN
dbms_output.put_line(sqlerrm);
if dbms_sql.is_open (cursor1) then
dbms_sql.close_cursor (cursor1);
end if;
END;
/

SQL> begin
get_rows('EMPNO', '<', 2000);
end;
/
1111

PL/SQL procedure successfully completed.

SQL> execute get_rows('SAL', '>', 3000);
7566 JONES
7788 SCOTT
7839 KING
7902 FORD

PL/SQL procedure successfully completed.

SQL> begin
get_rows('DEPTNO', '>=', 20);
end;
/
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7788 SCOTT
7844 TURNER
7876 ADAMS
7900 JAMES
7902 FORD

PL/SQL procedure successfully completed.
```

```
CREATE or REPLACE PROCEDURE get_rows (where_clause varchar2) AS
cursor1 integer;
rows_processed integer;
myempno number;
myename varchar2(20);
BEGIN
cursor1 := dbms_sql.open_cursor;
dbms_sql.parse (cursor1, 'select empno, ename from emp where ' ||
where_clause,
dbms_sql.v7);
dbms_sql.define_column (cursor1, 1, myempno);
dbms_sql.define_column (cursor1, 2, myename, 20);
rows_processed := dbms_sql.execute (cursor1);
loop
if dbms_sql.fetch_rows (cursor1) > 0 then
dbms_sql.column_value (cursor1, 1, myempno);
dbms_sql.column_value (cursor1, 2, myename);
dbms_output.put_line(to_char(myempno) || ' ' || myename);
else
exit;
end if;
end loop;
dbms_sql.close_cursor (cursor1);
EXCEPTION
WHEN OTHERS THEN
dbms_output.put_line(sqlerrm);
if dbms_sql.is_open (cursor1) then
dbms_sql.close_cursor (cursor1);
end if;
END;
/

SQL> execute get_rows('ENAME = ''KING'' ');
7839 KING

PL/SQL procedure successfully completed.

SQL> execute get_rows('SAL > 1000 AND DEPTNO = 10');
7782 CLARK
7839 KING
8100 MILLER

PL/SQL procedure successfully completed.
```

Example 7:

This procedure can execute any non-query SQL statement. It can even execute a PL/SQL block.

```
create procedure anysql (s1 varchar2) as
cursor1 integer;
return_value integer;
begin
cursor1 := dbms_sql.open_cursor;
dbms_sql.parse(cursor1, s1, dbms_sql.v7);
return_value := dbms_sql.execute(cursor1);
dbms_sql.close_cursor(cursor1);
end;
/

SQL> execute anysql('CREATE TABLE MYTABLE (COL1 number, col2 varchar2(3))');

PL/SQL procedure successfully completed.

SQL> desc mytable;
Name Null? Type
------------------------------- -------- ----
COL1 NUMBER
COL2 VARCHAR2(3)

SQL> execute anysql('INSERT INTO MYTABLE VALUES(1, ''ABC'')');

PL/SQL procedure successfully completed.

SQL> begin
anysql( 'declare
var1 varchar2(3);
begin
select col2  into var1  from mytable  where col1 = 1;
dbms_output.put_line(''var1 = '' || var1);
end;');
end;
/
var1 = ABC

PL/SQL procedure successfully completed.
```

Keywords

DBMS_SQL ; DDL ; PL/SQL ;

Help us improve our service. Please email us your comments for this document. .