|  |
| --- |
| , RD Dep. |
| **Oracle Database Security Basics** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial Version | Siarhei Kandrashevich | 15-Jan-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Oracle Non-Schema Objects 3](#_Toc380991675)

[1.1. Oracle Schemas and Users 3](#_Toc380991676)

[1.2. Predefined User Accounts 4](#_Toc380991677)

[2. Oracle database security 4](#_Toc380991678)

[2.1. Managing Users and Resources 5](#_Toc380991679)

[2.2. Predefined Roles: CONNECT, RESOURCE and DBA 5](#_Toc380991680)

[3. Managing Privileges 5](#_Toc380991681)

[3.1. GRANT and REVOKE Statements 5](#_Toc380991682)

[4. PL/SQL Dynamic SQL 7](#_Toc380991683)

[4.1. When You Need Dynamic SQL 7](#_Toc380991684)

[5. External Sources 15](#_Toc380991685)

# 

# Oracle Non-Schema Objects

## Oracle Schemas and Users

A database schema is a logical container for data structures, called schema objects. Examples of schema objects are tables and indexes. Schema objects are created and manipulated with SQL.

A database user has a password and various database privileges. Each user owns a single schema, which has the same name as the user. The schema contains the data for the user owning the schema. For example, the HR user owns the HR schema, which contains schema objects such as the employees table.

In a production database, the schema owner usually represents a database application rather than a person.

Within a schema, each schema object of a particular type has a unique name. For example, hr.employees refers to the table employees in the HR schema.

Non-schema objects are stored in the database and can be created and manipulated with SQL but are not contained in a schema.

Non-schema objects:

* Contexts
* Directories
* Parameter files (PFILEs) and server parameter files (SPFILEs)
* Profiles
* Roles
* Rollback segments
* Tablespaces
* Users

**Privileges**

A user privilege is the right to run a particular type of SQL statement, or the right to access an object that belongs to another user, run a PL/SQL package, and so on. The types of privileges are defined by Oracle Database.

*System privileges.* These privileges allow the grantee to perform standard administrator tasks in the database. Restrict them only to trusted users. A system privilege is the right to perform a particular action or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges.

*Object privileges.* Each type of object has privileges associated with it. GRANT and REVOKE statements are used to manage privileges for different types of objects.

**Roles**

Roles are created by users (usually administrators) to group together privileges or other roles. They are a way to facilitate the granting of multiple privileges or roles to users.

You grant privileges (directly or through roles) to users so they can accomplish tasks required for their jobs. You should grant a privilege only to a user who requires that privilege to accomplish the necessary work. Excessive granting of unnecessary privileges can compromise security. *For example, you never should grant SYSDBA or SYSOPER privilege to users who do not perform administrative tasks.*

A user can receive a privilege in two ways:

* You can grant privileges to users explicitly. For example, you can explicitly grant to user psmith the privilege to insert records into the employees table.
* You can grant privileges to a role (a named group of privileges), and then grant the role to one or more users. For example, you can grant the privileges to select, insert, update, and delete records from the employees table to the role named clerk, which in turn you can grant to users psmith and robert.

Because roles allow for easier and better management of privileges, you should usually grant privileges to roles and not to specific users.

## Predefined User Accounts

Oracle Database includes a number of predefined user accounts. The three types of predefined accounts are:

* Administrative accounts (SYS, SYSTEM, SYSMAN, and DBSNMP).
  + SYS. When you create an Oracle Database, the user SYS is automatically created and granted the DBA role. All of the base tables and views for the database data dictionary are stored in the schema SYS. These base tables and views are critical for the operation of Oracle Database. To maintain the integrity of the data dictionary, tables in the SYS schema are manipulated only by the database. *They should never be modified by any user or database administrator, and no one should create any tables in the schema of user SYS.* (However, you can change the storage parameters of the data dictionary settings if necessary.)

Ensure that most database users are never able to connect to Oracle Database using the SYS account.

* + SYSTEM. When you create an Oracle Database, the user SYSTEM is also automatically created and granted the DBA role. The SYSTEM username is used to create additional tables and views that display administrative information, and internal tables and views used by various Oracle Database options and tools. *Never use the SYSTEM schema to store tables of interest to non-administrative users.*
  + SYSMAN is used to perform Oracle Enterprise Manager administration tasks.
  + The management agent of Enterprise Manager uses the DBSNMP account to monitor and manage the database. You must not delete these accounts.
* Sample schema accounts.  
  These accounts are used for examples in Oracle Database documentation and instructional materials. Examples are HR, SH, and OE. You must unlock these accounts and reset their passwords before using them.
* Internal accounts.  
  These accounts are created so that individual Oracle Database features or components can have their own schemas. You must not delete internal accounts, and you must not attempt to log in with them.

# Oracle database security

It is important to develop a security policy for every database. The security policy establishes methods for protecting your database from accidental or malicious destruction of data or damage to the database infrastructure.

Each database can have an administrator, referred to as the security administrator, who is responsible for implementing and maintaining the database security policy If the database system is small, the database administrator can have the responsibilities of the security administrator. However, if the database system is large, a designated person or group of people may have sole responsibility as security administrator.

Each database has a list of valid database users. To access a database, a user must run a database application, and connect to the database instance using a valid user name defined in the database. Oracle Database enables you to set up security for your users in a variety of ways. When you create user accounts, you can specify limits to the user account. You can also set limits on the amount of various system resources available to each user as part of the security domain of that user. Oracle Database provides a set of database views that you can query to find information such as resource and session information.

## Managing Users and Resources

To connect to the database, each user must specify a valid user name that has been previously defined to the database. An account must have been established for the user, with information about the user being stored in the data dictionary.

When you create a database user (account), you specify the following attributes of the user:

* User name
* Authentication method
* Default tablespace
* Temporary tablespace
* Other tablespaces and quotas
* User profile

The CREATE USER statement is used to create a new user (you must have the CREATE USER system privilege to do this). Because it is a powerful privilege, a database administrator or security administrator is usually the only user who has the CREATE USER system privilege.

## Predefined Roles: CONNECT, RESOURCE and DBA

**RESOURCE**

Provides the following system privileges: CREATE CLUSTER, CREATE INDEXTYPE, CREATE OPERATOR, CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER, CREATE TYPE.

This role is provided for compatibility with previous releases of Oracle Database. You can determine the privileges encompassed by this role by querying the DBA\_SYS\_PRIVS data dictionary view.

**CONNECT**

Provides the CREATE SESSION system privilege.

This role is provided for compatibility with previous releases of Oracle Database. You can determine the privileges encompassed by this role by querying the DBA\_SYS\_PRIVS data dictionary view.

**DBA**

Provides all system privileges WITH ADMIN OPTION.

This role is provided for compatibility with previous releases of Oracle Database. You can determine the privileges encompassed by this role by querying the DBA\_SYS\_PRIVS data dictionary view.

Note: Oracle recommends that you design your own roles for database security rather than relying on these predefined roles. These roles may not be created automatically by future releases of Oracle Database.

# Managing Privileges

## GRANT and REVOKE Statements

**Grant**

Use the GRANT statement to grant:

* Roles to users and roles. The granted roles can be either user-defined (local or external) or predefined. For a list of predefined roles, refer to Oracle Database Security Guide.
* System privileges to users and roles.
* Object privileges for a particular object to users, roles, *and PUBLIC.*

**Prerequisites**

To grant a system privilege, one of the following conditions must be met:

* You must have been granted the GRANT ANY PRIVILEGE system privilege. In this case, if you grant the system privilege to a role, then a user to whom the role has been granted does not have the privilege unless the role is enabled in user's session.
* You must have been granted the system privilege with the ADMIN OPTION. In this case, if you grant the system privilege to a role, then a user to whom the role has been granted has the privilege regardless whether the role is enabled in the user's session.

To grant a role, you must either have been granted the role with the ADMIN OPTION or have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

*To grant an object privilege,* you must own the object, or the owner of the object must have granted you the object privileges with the GRANT OPTION, or you must have been granted the GRANT ANY OBJECT PRIVILEGE system privilege. If you have the GRANT ANY OBJECT PRIVILEGE, then you can grant the object privilege only if the object owner could have granted the same object privilege. In this case, the GRANTOR column of the DBA\_TAB\_PRIVS view displays the object owner rather than the user who issued the GRANT statement.

**Revoke**

Use the REVOKE statement to:

* Revoke system privileges from users and roles
* Revoke roles from users and roles
* Revoke object privileges for a particular object from users and roles

**Prerequisites**

To revoke a system privilege, you must have been granted the privilege with the ADMIN OPTION.

You can revoke any privilege if you have the GRANT ANY PRIVILEGE system privilege.

To revoke a role, you must have been granted the role with the ADMIN OPTION. You can revoke any role if you have the GRANT ANY ROLE system privilege.

*To revoke an object privilege,* you must previously have granted the object privilege to the user and role or you must have the GRANT ANY OBJECT PRIVILEGE system privilege. In the latter case, you can revoke any object privilege that was granted by the object owner or on behalf of the owner by a user with the GRANT ANY OBJECT PRIVILEGE. However, you cannot revoke an object privilege that was granted by way of a WITH GRANT OPTION grant.

The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement.

You cannot use this statement to revoke:

* Privileges or roles not granted to the revokee
* Roles or object privileges granted through the operating system
* Privileges or roles granted to the revokee through roles

# PL/SQL Dynamic SQL

Dynamic SQL is a programming methodology for generating and running SQL statements at run time. It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compile time the full text of a SQL statement or the number or data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements

DBMS\_SQL package, an API for building, running, and describing dynamic SQL statements

Native dynamic SQL code is easier to read and write than equivalent code that uses the DBMS\_SQL package, and runs noticeably faster (especially when it can be optimized by the compiler). However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the DBMS\_SQL package. You must also use the DBMS\_SQL package if you want a stored subprogram to return a query result implicitly (not through an OUT REF CURSOR parameter).

When you need both the DBMS\_SQL package and native dynamic SQL, you can switch between them, using the "DBMS\_SQL.TO\_REFCURSOR Function" and "DBMS\_SQL.TO\_CURSOR\_NUMBER Function".

## When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL to run:

SQL whose text is unknown at compile time

For example, a SELECT statement that includes an identifier that is unknown at compile time (such as a table name) or a WHERE clause in which the number of subclauses is unknown at compile time.

SQL that is not supported as static SQL

That is, any SQL construct not included in "Description of Static SQL".

If you do not need dynamic SQL, use static SQL, which has these advantages:

Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.

Successful compilation creates schema object dependencies.

Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements with the EXECUTE IMMEDIATE statement.

If the dynamic SQL statement is a SELECT statement that returns multiple rows, native dynamic SQL gives you these choices:

Use the EXECUTE IMMEDIATE statement with the BULK COLLECT INTO clause.

Use the OPEN FOR, FETCH, and CLOSE statements.

The SQL cursor attributes work the same way after native dynamic SQL INSERT, UPDATE, DELETE, MERGE, and single-row SELECT statements as they do for their static SQL counterparts. For more information about SQL cursor attributes, see "Cursors".

Topics

* EXECUTE IMMEDIATE Statement
* OPEN FOR, FETCH, and CLOSE Statements
* Repeated Placeholder Names in Dynamic SQL Statements
* EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement is the means by which native dynamic SQL processes most dynamic SQL statements.

* If the dynamic SQL statement is self-contained (that is, if it has no placeholders for bind variables and the only result that it can possibly return is an error), then the EXECUTE IMMEDIATE statement needs no clauses.
* If the dynamic SQL statement includes placeholders for bind variables, each placeholder must have a corresponding bind variable in the appropriate clause of the EXECUTE IMMEDIATE statement, as follows:
* If the dynamic SQL statement is a SELECT statement that can return at most one row, put out-bind variables (defines) in the INTO clause and in-bind variables in the USING clause.
* If the dynamic SQL statement is a SELECT statement that can return multiple rows, put out-bind variables (defines) in the BULK COLLECT INTO clause and in-bind variables in the USING clause.
* If the dynamic SQL statement is a DML statement without a RETURNING INTO clause, other than SELECT, put all bind variables in the USING clause.
* If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind variables in the USING clause and out-bind variables in the RETURNING INTO clause.
* If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind variables in the USING clause.
* If the dynamic SQL statement invokes a subprogram, ensure that:
* The subprogram is either created at schema level or declared and defined in a package specification.

For information about creating subprograms at schema level, see "CREATE FUNCTION Statement" and "CREATE PROCEDURE Statement". For information about declaring and defining subprograms in packages, see "CREATE PACKAGE Statement" and "CREATE PACKAGE BODY Statement".

* Every bind variable that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter and a data type that is compatible with that of the subprogram parameter.
* No bind variable is the reserved word NULL.
* To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 5-7.
* No bind variable has a data type that SQL does not support (such as associative array indexed by string).
* If the data type is a collection or record type, then it must be declared in a package specification. For information about declaring types in a package specification, see "CREATE PACKAGE Statement".

Note:

Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

For syntax details of the EXECUTE IMMEDIATE statement, see "EXECUTE IMMEDIATE Statement".

In Example 5-1, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram created at schema level.

Example 5-1 Invoking Subprogram from Dynamic PL/SQL Block

-- Subprogram that dynamic PL/SQL block invokes:

CREATE OR REPLACE PROCEDURE create\_dept (

deptid IN OUT NUMBER,

dname IN VARCHAR2,

mgrid IN NUMBER,

locid IN NUMBER

) AUTHID DEFINER AS

BEGIN

deptid := departments\_seq.NEXTVAL;

INSERT INTO departments (

department\_id,

department\_name,

manager\_id,

location\_id

)

VALUES (deptid, dname, mgrid, locid);

END;

/

DECLARE

plsql\_block VARCHAR2(500);

new\_deptid NUMBER(4);

new\_dname VARCHAR2(30) := 'Advertising';

new\_mgrid NUMBER(6) := 200;

new\_locid NUMBER(4) := 1700;

BEGIN

-- Dynamic PL/SQL block invokes subprogram:

plsql\_block := 'BEGIN create\_dept(:a, :b, :c, :d); END;';

/\* Specify bind variables in USING clause.

Specify mode for first parameter.

Modes of other parameters are correct by default. \*/

EXECUTE IMMEDIATE plsql\_block

USING IN OUT new\_deptid, new\_dname, new\_mgrid, new\_locid;

END;

/

In Example 5-2, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL (but not SQL) data type BOOLEAN.

Example 5-2 Dynamically Invoking Subprogram with BOOLEAN Formal Parameter

CREATE OR REPLACE PROCEDURE p (x BOOLEAN) AUTHID DEFINER AS

BEGIN

IF x THEN

DBMS\_OUTPUT.PUT\_LINE('x is true');

END IF;

END;

/

DECLARE

dyn\_stmt VARCHAR2(200);

b BOOLEAN := TRUE;

BEGIN

dyn\_stmt := 'BEGIN p(:x); END;';

EXECUTE IMMEDIATE dyn\_stmt USING b;

END;

/

Result:

x is true

In Example 5-3, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL (but not SQL) data type RECORD. The record type is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body. (For information about packages, see Chapter 10, "PL/SQL Packages.")

Example 5-3 Dynamically Invoking Subprogram with RECORD Formal Parameter

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

TYPE rec IS RECORD (n1 NUMBER, n2 NUMBER);

PROCEDURE p (x OUT rec, y NUMBER, z NUMBER);

END pkg;

/

CREATE OR REPLACE PACKAGE BODY pkg AS

PROCEDURE p (x OUT rec, y NUMBER, z NUMBER) AS

BEGIN

x.n1 := y;

x.n2 := z;

END p;

END pkg;

/

DECLARE

r pkg.rec;

dyn\_str VARCHAR2(3000);

BEGIN

dyn\_str := 'BEGIN pkg.p(:x, 6, 8); END;';

EXECUTE IMMEDIATE dyn\_str USING OUT r;

DBMS\_OUTPUT.PUT\_LINE('r.n1 = ' || r.n1);

DBMS\_OUTPUT.PUT\_LINE('r.n2 = ' || r.n2);

END;

/

In Example 5-4, Example 5-5, and Example 5-6, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of a PL/SQL collection type. Collection types are not SQL data types. In each example, the collection type is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body. (For information about collection types, see Chapter 5, "PL/SQL Collections and Records." For information about packages, see Chapter 10, "PL/SQL Packages.")

In Example 5-4, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type associative array indexed by PLS\_INTEGER.

Note:

An associative array type used in this context must be indexed by PLS\_INTEGER.

Example 5-4 Dynamically Invoking Subprogram with Assoc. Array Formal Parameter

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

TYPE number\_names IS TABLE OF VARCHAR2(5)

INDEX BY PLS\_INTEGER;

PROCEDURE print\_number\_names (x number\_names);

END pkg;

/

CREATE OR REPLACE PACKAGE BODY pkg AS

PROCEDURE print\_number\_names (x number\_names) IS

BEGIN

FOR i IN x.FIRST .. x.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE(x(i));

END LOOP;

END;

END pkg;

/

DECLARE

digit\_names pkg.number\_names;

dyn\_stmt VARCHAR2(3000);

BEGIN

digit\_names(0) := 'zero';

digit\_names(1) := 'one';

digit\_names(2) := 'two';

digit\_names(3) := 'three';

digit\_names(4) := 'four';

digit\_names(5) := 'five';

digit\_names(6) := 'six';

digit\_names(7) := 'seven';

digit\_names(8) := 'eight';

digit\_names(9) := 'nine';

dyn\_stmt := 'BEGIN pkg.print\_number\_names(:x); END;';

EXECUTE IMMEDIATE dyn\_stmt USING digit\_names;

END;

/

In Example 5-5, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type nested table.

Example 5-5 Dynamically Invoking Subprogram with Nested Table Formal Parameter

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

TYPE names IS TABLE OF VARCHAR2(10);

PROCEDURE print\_names (x names);

END pkg;

/

CREATE OR REPLACE PACKAGE BODY pkg AS

PROCEDURE print\_names (x names) IS

BEGIN

FOR i IN x.FIRST .. x.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE(x(i));

END LOOP;

END;

END pkg;

/

DECLARE

fruits pkg.names;

dyn\_stmt VARCHAR2(3000);

BEGIN

fruits := pkg.names('apple', 'banana', 'cherry');

dyn\_stmt := 'BEGIN pkg.print\_names(:x); END;';

EXECUTE IMMEDIATE dyn\_stmt USING fruits;

END;

/

In Example 5-6, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type varray.

Example 5-6 Dynamically Invoking Subprogram with Varray Formal Parameter

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

TYPE foursome IS VARRAY(4) OF VARCHAR2(5);

PROCEDURE print\_foursome (x foursome);

END pkg;

/

CREATE OR REPLACE PACKAGE BODY pkg AS

PROCEDURE print\_foursome (x foursome) IS

BEGIN

IF x.COUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Empty');

ELSE

FOR i IN x.FIRST .. x.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE(x(i));

END LOOP;

END IF;

END;

END pkg;

/

DECLARE

directions pkg.foursome;

dyn\_stmt VARCHAR2(3000);

BEGIN

directions := pkg.foursome('north', 'south', 'east', 'west');

dyn\_stmt := 'BEGIN pkg.print\_foursome(:x); END;';

EXECUTE IMMEDIATE dyn\_stmt USING directions;

END;

/

Example 5-7 uses an uninitialized variable to represent the reserved word NULL in the USING clause.

Example 5-7 Uninitialized Variable Represents NULL in USING Clause

CREATE TABLE employees\_temp AS SELECT \* FROM EMPLOYEES;

DECLARE

a\_null CHAR(1); -- Set to NULL automatically at run time

BEGIN

EXECUTE IMMEDIATE 'UPDATE employees\_temp SET commission\_pct = :x'

USING a\_null;

END;

/

OPEN FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a SELECT statement that returns multiple rows, you can process it with native dynamic SQL as follows:

Use an OPEN FOR statement to associate a cursor variable with the dynamic SQL statement. In the USING clause of the OPEN FOR statement, specify a bind variable for each placeholder in the dynamic SQL statement.

The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 5-7.

For syntax details, see "OPEN FOR Statement".

Use the FETCH statement to retrieve result set rows one at a time, several at a time, or all at once.

For syntax details, see "FETCH Statement".

Use the CLOSE statement to close the cursor variable.

For syntax details, see "CLOSE Statement".

Example 5-8 lists all employees who are managers, retrieving result set rows one at a time.

Example 5-8 Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements

DECLARE

TYPE EmpCurTyp IS REF CURSOR;

v\_emp\_cursor EmpCurTyp;

emp\_record employees%ROWTYPE;

v\_stmt\_str VARCHAR2(200);

v\_e\_job employees.job%TYPE;

BEGIN

-- Dynamic SQL statement with placeholder:

v\_stmt\_str := 'SELECT \* FROM employees WHERE job\_id = :j';

-- Open cursor & specify bind variable in USING clause:

OPEN v\_emp\_cursor FOR v\_stmt\_str USING 'MANAGER';

-- Fetch rows from result set one at a time:

LOOP

FETCH v\_emp\_cursor INTO emp\_record;

EXIT WHEN v\_emp\_cursor%NOTFOUND;

END LOOP;

-- Close cursor:

CLOSE v\_emp\_cursor;

END;

/

The dynamic SQL statement can query a collection if the collection meets the criteria in "Querying a Collection". Example 5-9 is like Example 6-30 except that the collection variable v1 is a bind variable.

Example 5-9 Querying a Collection with Native Dynamic SQL

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));

TYPE mytab IS TABLE OF rec INDEX BY pls\_integer;

END;

/

DECLARE

v1 pkg.mytab; -- collection of records

v2 pkg.rec;

c1 SYS\_REFCURSOR;

BEGIN

OPEN c1 FOR 'SELECT \* FROM TABLE(:1)' USING v1;

FETCH c1 INTO v2;

CLOSE c1;

DBMS\_OUTPUT.PUT\_LINE('Values in record are ' || v2.f1 || ' and ' || v2.f2);

END;

/

Repeated Placeholder Names in Dynamic SQL Statements

If you repeat placeholder names in dynamic SQL statements, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement.

Topics

* Dynamic SQL Statement is Not Anonymous Block or CALL Statement
* Dynamic SQL Statement is Anonymous Block or CALL Statement
* Dynamic SQL Statement is Not Anonymous Block or CALL Statement

If the dynamic SQL statement does not represent an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is insignificant. Placeholders are associated with bind variables in the USING clause by position, not by name.

For example, in this dynamic SQL statement, the repetition of the name :x is insignificant:

sql\_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';

In the corresponding USING clause, you must supply four bind variables. They can be different; for example:

EXECUTE IMMEDIATE sql\_stmt USING a, b, c, d;

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

INSERT INTO payroll VALUES (a, b, c, d)

To associate the same bind variable with each occurrence of :x, you must repeat that bind variable; for example:

EXECUTE IMMEDIATE sql\_stmt USING a, a, b, a;

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

INSERT INTO payroll VALUES (a, a, b, a)

Dynamic SQL Statement is Anonymous Block or CALL Statement

If the dynamic SQL statement represents an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is significant. Each unique placeholder name must have a corresponding bind variable in the USING clause. If you repeat a placeholder name, you need not repeat its corresponding bind variable. All references to that placeholder name correspond to one bind variable in the USING clause.

In Example 5-10, all references to the first unique placeholder name, :x, are associated with the first bind variable in the USING clause, a, and the second unique placeholder name, :y, is associated with the second bind variable in the USING clause, b.

Example 5-10 Repeated Placeholder Names in Dynamic PL/SQL Block

CREATE PROCEDURE calc\_stats (

w NUMBER,

x NUMBER,

y NUMBER,

z NUMBER )

IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE(w + x + y + z);

END;

/

DECLARE

a NUMBER := 4;

b NUMBER := 7;

plsql\_block VARCHAR2(100);

BEGIN

plsql\_block := 'BEGIN calc\_stats(:x, :x, :y, :x); END;';

EXECUTE IMMEDIATE plsql\_block USING a, b; -- calc\_stats(a, a, b, a)

END;

/

# External Sources

1. Oracle Database SQL Language Reference 11g Release 2 (11.2) E26088-02
2. Oracle Database Concepts 11gRelease 2 (11.2) E25789-01
3. Oracle Database Administrator's Guide 11g Release 2 (11.2) E10595-04
4. ISO/IEC 9075-1 Part 1 Framework (SQL/Framework)
5. ISO/IEC 9075-2 Part 2 Foundation (SQL/Foundation)
6. ISO/IEC 9075-11 Part 11 Information and Definition Schemas (SQL/Schemata)