6

Dynamic SQL and Metadata

Objectives

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

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (that is, with `EXECUTE IMMEDIATE` statements)
- Compare Native Dynamic SQL with the `DBMS_SQL` package approach
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects

Execution Flow of SQL

- All SQL statements go through various stages:
  - Parse
  - Bind
  - Execute
  - Fetch
- Some stages may not be relevant for all statements—for example, the fetch phase is applicable to queries.

Note: For embedded SQL statements (`SELECT`, `DML`, `COMMIT`, and `ROLLBACK`), the parse and bind phases are done at compile time. For dynamic SQL statements, all phases are performed at run time.

Dynamic SQL

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- Is constructed and stored as a character string within the application
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)
- Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL
- Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package

Native Dynamic SQL

- Provides native support for dynamic SQL directly in the PL/SQL language
- Provides the ability to execute SQL statements whose structure is unknown until execution time
- Is supported by the following PL/SQL statements:
  - `EXECUTE IMMEDIATE`
  - `OPEN-FOR`
  - `FETCH`
  - `CLOSE`

Using the EXECUTE IMMEDIATE Statement

Use the EXECUTE IMMEDIATE statement for Native Dynamic SQL or PL/SQL anonymous blocks:

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.
- USING is used to hold all bind arguments. The default parameter mode is IN, if not specified.

Dynamic SQL with a DDL Statement

- Create a table:

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

- Call example:

```
BEGIN
  create_table('EMPLOYEE_NAMES',
   'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

Dynamic SQL with DML Statements

- Delete rows from any table:

```
CREATE FUNCTION del_rows(table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '||table_name;
  RETURN SQL%ROWCOUNT;
END;
```

```
BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
```

- Insert a row into a table with two columns:

```
CREATE PROCEDURE add_row(table_name VARCHAR2,
   id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

Dynamic SQL with a Single-Row Query

Example of a single-row query:

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/
```

```
DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```

Dynamic SQL with a Multirow Query

Use `OPEN-FOR`, `FETCH`, and `CLOSE` processing:

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv  emp_refcsr;
  emprec  employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                         ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```

Declaring Cursor Variables

- Declare a cursor type as `REF CURSOR`:

```
CREATE PROCEDURE process_data IS
  TYPE ref_ctype IS REF CURSOR; -- weak ref cursor
  TYPE emp_ref_ctype IS REF CURSOR -- strong
     RETURN employees%ROWTYPE;
  :
```

- Declare a cursor variable using the cursor type:

```
  :
  dept_csrvar ref_ctype;
  emp_csrvar  emp_ref_ctype;
BEGIN
  OPEN emp_csrvar FOR SELECT * FROM employees;
  OPEN dept_csrvar FOR SELECT * from departments;
  -- Then use as normal cursors
END;
```

Dynamically Executing a PL/SQL Block

Execute a PL/SQL anonymous block dynamically:

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
    'DECLARE '||
    ' emprec employees%ROWTYPE; '||
    'BEGIN '||
    ' emprec := get_emp(:empid); ' ||
    ' :res := emprec.salary * 12; ' ||
    'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
         USING IN emp_id, OUT result;
  RETURN result;
END;
/
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

Using Native Dynamic SQL to Compile PL/SQL Code

Compile PL/SQL code with the `ALTER` statement:

- `ALTER PROCEDURE name COMPILE`
- `ALTER FUNCTION name COMPILE`
- `ALTER PACKAGE name COMPILE SPECIFICATION`
- `ALTER PACKAGE name COMPILE BODY`

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

Using the DBMS_SQL Package

The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Using DBMS_SQL with a DML Statement

Example of deleting rows:

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (table_name VARCHAR2) RETURN NUMBER IS
  csr_id INTEGER;
  rows_del    NUMBER;
BEGIN
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id,
    'DELETE FROM '||table_name, DBMS_SQL.NATIVE);
  rows_del := DBMS_SQL.EXECUTE (csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  RETURN rows_del;
END;
/
```

```
CREATE table temp_emp as select * from employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

Using DBMS_SQL with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id      INTEGER;
  stmt        VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
          ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```

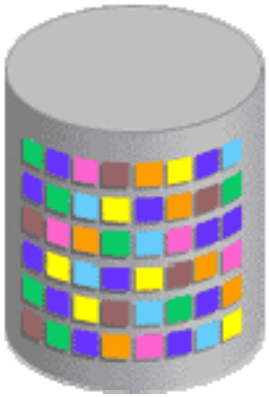
Comparison of Native Dynamic SQL and the `DBMS_SQL` Package

Native Dynamic SQL:

- Is easier to use than `DBMS_SQL`
- Requires less code than `DBMS_SQL`
- Enhances performance because the PL/SQL interpreter provides native support for it
- Supports all types supported by static SQL in PL/SQL, including user-defined types
- Can fetch rows directly into PL/SQL records

DBMS_METADATA Package

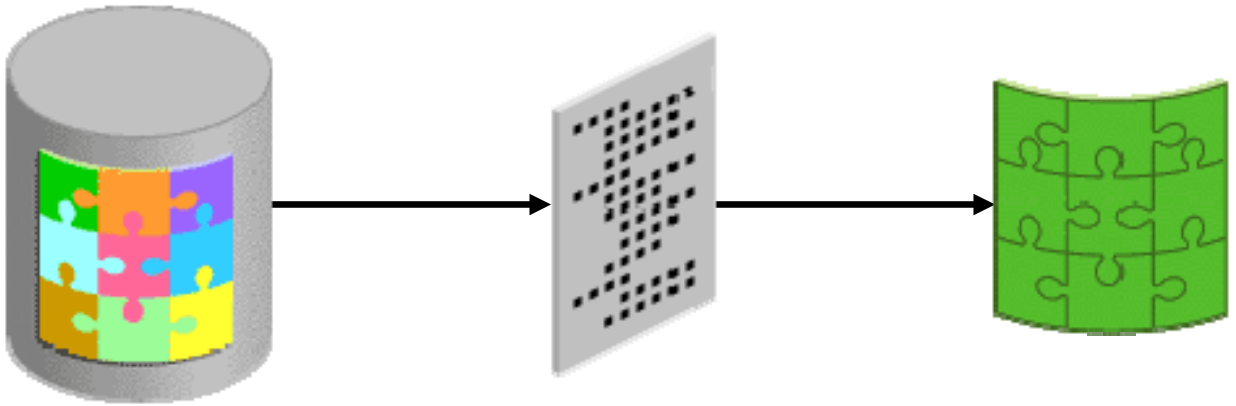
The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call
GET_QUERY	Returns the text of the queries that will be used by FETCH_xxx
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle
FETCH_*XXX*	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state

FETCH_xxx Subprograms

Name	Description
FETCH_XML	This function returns the XML metadata for an object as an XMLType.
FETCH_DDL	This function returns the DDL (either to create or drop the object) into a predefined nested table.
FETCH_CLOB	This function returns the objects (transformed or not) as a CLOB.
FETCH_XML_CLOB	This procedure returns the XML metadata for the objects as a CLOB in an IN OUT NOCOPY parameter to avoid expensive LOB copies.

SET_FILTER Procedure

- Syntax:

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- Example:

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME', 'HR');
...
```

Filters

There are over 70 filters, which are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export

Examples of Setting Filters

Set up the filter to fetch the `HR` schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain `PAYROLL` in the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
  'IN (''PAYROLL'', ''HR'')');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''FUNCTION''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PROCEDURE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PACKAGE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
  'LIKE ''PAYROLL%''', 'VIEW');
```

Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  h    NUMBER; th1  NUMBER; th2  NUMBER;
  doc  sys.ku$_ddls;                          (1)
BEGIN
  h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');   (2)
  DBMS_METADATA.SET_FILTER (h,'SCHEMA','HR'); (3)
  th1 := DBMS_METADATA.ADD_TRANSFORM (h,      (4)
    'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(th1,          (5)
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  th2 :=DBMS_METADATA.ADD_TRANSFORM(h, 'DDL');
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,
    'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,      (6)
    'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    doc := DBMS_METADATA.FETCH_DDL(h);        (7)
    EXIT WHEN doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(h);                     (8)
END;
```

Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
 h    NUMBER; -- returned by 'OPEN'
 th   NUMBER; -- returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 h := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','EMPLOYEES');
 -- request to be TRANSFORMED into creation DDL
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
 -- FETCH the object
 doc := DBMS_METADATA.FETCH_CLOB(h);
 -- release resources
 DBMS_METADATA.CLOSE(h);
 RETURN doc;
END;
/
```

Browsing APIs

Name	Description
GET_*XXX*	The GET_XML and GET_DDL functions return metadata for a single named object.
GET_DEPENDENT_*XXX*	This function returns metadata for a dependent object.
GET_GRANTED_*XXX*	This function returns metadata for a granted object.

Where ***xxx*** is:	**DDL** or **XML**

Browsing APIs: Examples

1. Get the XML representation of HR.EMPLOYEES:

```
SELECT DBMS_METADATA.GET_XML
                      ('TABLE', 'EMPLOYEES', 'HR')
FROM   dual;
```

2. Fetch the DDL for all object grants on HR.EMPLOYEES:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
                     ('OBJECT_GRANT', 'EMPLOYEES', 'HR')
FROM   dual;
```

3. Fetch the DDL for all system grants granted to HR:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
                   ('SYSTEM_GRANT', 'HR')
FROM   dual;
```

Browsing APIs: Examples

```
BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
   DBMS_METADATA.SESSION_TRANSFORM,
   'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT');
END;
/
```

Summary

In this lesson, you should have learned how to:

- Explain the execution flow of SQL statements
- Create SQL statements dynamically and execute them using either Native Dynamic SQL statements or the `DBMS_SQL` package
- Recognize the advantages of using Native Dynamic SQL compared to the `DBMS_SQL` package
- Use `DBMS_METADATA` subprograms to programmatically obtain metadata from the data dictionary

Practice 6: Overview

This practice covers the following topics:

- Creating a package that uses Native Dynamic SQL to create or drop a table and populate, modify, and delete rows from a table
- Creating a package that compiles the PL/SQL code in your schema
- Using `DBMS_METADATA` to display the statement to regenerate a PL/SQL subprogram