27 Using the Metadata APIs

The `DBMS_METADATA` APIs enable you to check and update object metadata.

The `DBMS_METADATA` API enables you to do the following:

- Retrieve an object's metadata as XML
- Transform the XML in a variety of ways, including transforming it into SQL DDL
- Submit the XML to re-create the object extracted by the retrieval

The `DBMS_METADATA_DIFF` API lets you compare objects between databases to identify metadata changes over time in objects of the same type.

- Why Use the DBMS_METADATA API?
  The `DBMS_METADATA` API eliminates the need for you to write and maintain your own code for metadata extraction.
- Overview of the DBMS_METADATA API
  Learn how to take advantage of the `DBMS_METADATA` API features.
- Using the DBMS_METADATA API to Retrieve an Object's Metadata
  The retrieval interface of the `DBMS_METADATA` API lets you specify the kind of object to be retrieved.
- Using the DBMS_METADATA API to Recreate a Retrieved Object
  When you fetch metadata for an object, you can choose to use it to recreate the object in a different database or schema.
- Using the DBMS_METADATA API to Retrieve Collections of Different Object Types
  To retrieve collections of objects in which the objects are of different types, but comprise a logical unit, you can use the heterogeneous object types in the `DBMS_METADATA` API.
- Filtering the Return of Heterogeneous Object Types
  Learn how you can use the `SET_FILTER` procedure to enable you to filter the return of heterogeneous object types.
- Using the DBMS_METADATA_DIFF API to Compare Object Metadata
  Description and example that uses the retrieval, comparison, and submit interfaces of `DBMS_METADATA` and `DBMS_METADATA_DIFF` to fetch metadata for two tables, compare the metadata, and generate `ALTER` statements which make one table like the other.
- Performance Tips for the Programmatic Interface of the DBMS_METADATA API
  Describes how to enhance performance when using the programmatic interface of the `DBMS_METADATA` API.
- Example Usage of the DBMS_METADATA API
  Example of how the `DBMS_METADATA` API could be used.
- Summary of DBMS_METADATA Procedures
  Provides brief descriptions of the procedures provided by the `DBMS_METADATA` API.
- Summary of DBMS_METADATA_DIFF Procedures
  Provides brief descriptions of the procedures and functions provided by the `DBMS_METADATA_DIFF` API.

27.1 Why Use the DBMS_METADATA API?

The `DBMS_METADATA` API eliminates the need for you to write and maintain your own code for metadata extraction.

If you have developed your own code for Oracle Database for extracting metadata from the dictionary, or for manipulating the metadata (adding columns, changing column data types, and so on), and converting the metadata to DDL so that you could recreate the object on the same or another database, then maintenance is an issue. Keeping that code updated to support new dictionary features has probably proven to be challenging.

Oracle Database provides a centralized facility for the extraction, manipulation, and recreation of dictionary metadata. Oracle Database also supports all dictionary objects at their most current level.

Although the `DBMS_METADATA` API can dramatically decrease the amount of custom code you are writing and maintaining, it does not involve any changes to your normal database procedures. You can install the `DBMS_METADATA` API in the same way as data dictionary views, by running `catproc.sql` to run a SQL script at database installation time. After you have installed `DBMS_METADATA`, it is available whenever the instance is operational, even in restricted mode.

When you change database releases using the `DBMS_METADATA` API, you are not required to make any source code changes. The DBMS_METADATA API enables the code to be upwardly compatible across different Oracle Database releases. XML documents retrieved by one release can be processed by the submit interface on the same or later releases. For example, XML documents retrieved by an Oracle Database 10g Release 2 (10.2) database can be submitted to Oracle Database 12c.

27.2 Overview of the DBMS_METADATA API

Learn how to take advantage of the `DBMS_METADATA` API features.

For the purposes of the `DBMS_METADATA` API, every entity in the database is modeled as an object that belongs to an object type. For example, the table `scott.emp` is an object. Its object type is `TABLE`. When you fetch an object's metadata, you must specify the object type.

Using Filters to Search for Objects By Object Type

To fetch a particular object or set of objects within an object type, you specify a filter. Different filters are defined for each object type. For example, two of the filters defined for the `TABLE` object type are `SCHEMA` and `NAME`. These filters enable you to say, for example, that you want the table whose schema is `scott`, and whose name is `emp`.

The `DBMS_METADATA` API makes use of XML (Extensible Markup Language) and XSLT (Extensible Stylesheet Language Transformation). The `DBMS_METADATA` API represents object metadata as XML, because it is a universal format that can be easily parsed and transformed. The `DBMS_METADATA` API uses XSLT to transform XML documents either into other XML documents, or into SQL DDL.

You can use the `DBMS_METADATA` API to specify one or more transforms (XSLT scripts) to be applied to the XML when the metadata is fetched (or when it is resubmitted). The API provides some predefined transforms, including one named DDL, which transforms the XML document into SQL creation DDL.

You can then specify conditions on the transform by using transform parameters. You can also specify optional parse items to access specific attributes of an object's metadata.

Using Views to Determine Valid DBMS_METADATA Options

You can use the following views to determine which `DBMS_METADATA` transforms are allowed for each object type transformation, the parameters for each transform, and their parse items.

- `DBMS_METADATA_TRANSFORMS` - documents all valid Oracle-supplied transforms that are used with the `DBMS_METADATA` package.
- `DBMS_METADATA_TRANSFORM_PARAMS` - documents the valid transform parameters for each transform.
- `DBMS_METADATA_PARSE_ITEMS` - documents the valid parse items.

For example, suppose that you want to know which transforms are allowed for `INDEX` objects. The following query returns the transforms that are valid for `INDEX` objects, the required input types, and the resulting output types:

```
SQL> SELECT transform, output_type, input_type, description
2 FROM dbms_metadata_transforms
3 WHERE object_type='INDEX';


TRANSFORM  OUTPUT_TYP INPUT_TYPE           DESCRIPTION
---------- ---------- --------------------
----------------------------------------------------------------------
ALTERXML   ALTER_XML  SXML difference doc  Generate ALTER_XML from SXML
difference document
SXMLDDL    DDL        SXML                 Convert SXML to DDL
MODIFY     XML        XML                  Modify XML document according to
transform parameters
SXML       SXML       XML                  Convert XML to SXML
DDL        DDL        XML                  Convert XML to SQL to create the
object
ALTERDDL   ALTER_DDL  ALTER_XML            Convert ALTER_XML to ALTER_DDL
MODIFYSXML SXML       SXML                 Modify SXML document
```

If you want to know which transform parameters are valid for the DDL transform, then you can run this query:

```
SQL> SELECT param, datatype, default_val, description
2 FROM dbms_metadata_transform_params
3 WHERE object_type='INDEX' and transform='DDL'
4 ORDER BY param;


PARAM                     DATATYPE   DEFAULT_VA  DESCRIPTION
------------------------- ---------- ----------
----------------------------------------------------------------------
INCLUDE_PARTITIONS        TEXT                   Include generated interval
and list partitions in DDL
                                                      transformation
INDEX_COMPRESSION_CLAUSE  TEXT       ""          Text of user-specified index
compression clause
PARTITIONING              BOOLEAN    TRUE        Include partitioning clauses
```

```
in transformation
PARTITION_NAME            TEXT       ""          Name of partition selected
for the transformation
PCTSPACE                  NUMBER     ""          Percentage by which space
allocation is to be modified
SEGMENT_ATTRIBUTES        BOOLEAN    TRUE        Include segment attribute
clauses (physical attributes, storage
                                                      attribues, tablespace,
logging) in transformation
STORAGE                   BOOLEAN    TRUE        Include storage clauses in
transformation
SUBPARTITION_NAME         TEXT       ""          Name of subpartition
selected for the transformation
TABLESPACE                BOOLEAN    TRUE        Include tablespace clauses
in transformation
```

You can also perform the following query which returns specific metadata about the `INDEX` object type:

```
SQL> SELECT parse_item, description
2 FROM dbms_metadata_parse_items
3 WHERE object_type='INDEX' and convert='Y';

PARSE_ITEM           DESCRIPTION
--------------------
--------------------------------------------------------------
OBJECT_TYPE          Object type
TABLESPACE           Object tablespace (default tablespace for partitioned
objects)
BASE_OBJECT_SCHEMA   Schema of the base object
SCHEMA               Object schema, if any
NAME                 Object name
BASE_OBJECT_NAME     Name of the base object
BASE_OBJECT_TYPE     Object type of the base object
SYSTEM_GENERATED     Y = system-generated object; N = not system-generated
```

Related Topics

- DBMS_METADATA_TRANSFORMS
- DBMS_METADATA_TRANSFORM_PARAMS
- DBMS_METADATA_PARSE_ITEMS

27.3 Using the DBMS_METADATA API to Retrieve an Object's Metadata

The retrieval interface of the `DBMS_METADATA` API lets you specify the kind of object to be retrieved.

- How to Use the DBMS_METADATA API to Retrieve Object Metadata
  Learn about the kinds of Oracle Database objects that you can query, and decide what interface you want to use for the query.

- Typical Steps Used for Basic Metadata Retrieval
  When you retrieve metadata, you use the DBMS_METADATA PL/SQL API.
- Retrieving Multiple Objects
  Description and example of retrieving multiple objects.
- Placing Conditions on Transforms
  To specify conditions on the transforms that you add with DBMS_METADATA, you can use transform parameters.
- Accessing Specific Metadata Attributes
  See how you can access specific metadata attributes of an object's metadata with the DBMS_METADATA API.

27.3.1 How to Use the DBMS_METADATA API to Retrieve Object Metadata

Learn about the kinds of Oracle Database objects that you can query, and decide what interface you want to use for the query.

This can be either a particular object type (such as a table, index, or procedure) or a heterogeneous collection of object types that form a logical unit (such as a database export or schema export). By default, metadata that you fetch is returned in an XML document.

Note:

To access objects that are not in your own schema, you must have the SELECT_CATALOG_ROLE role. However, roles are disabled within many PL/SQL objects (stored procedures, functions, definer's rights APIs). Therefore, if you are writing a PL/SQL program that will access objects in another schema (or, in general, any objects for which you need the SELECT_CATALOG_ROLE role), then you must put the code in an invoker's rights API.

You can use the programmatic interface for casual browsing, or you can use it to develop applications. You can use the browsing interface if you simply want to make quick queries of the system metadata. You can use the programmatic interface when you want to extract dictionary metadata as part of an application. In such cases, you can choose to use the procedures provided by the DBMS_METADATA API, instead of using SQL scripts or customized code that you may be currently using to do the same thing.

27.3.2 Typical Steps Used for Basic Metadata Retrieval

When you retrieve metadata, you use the DBMS_METADATA PL/SQL API.

The following examples illustrate the programmatic and browsing interfaces.

The DBMS_METADATA programmatic interface example provides a basic demonstration of using the DBMS_METADATA programmatic interface to retrieve metadata for one table. It creates a DBMS_METADATA program that creates a function named get_table_md. This function returns metadata for one table.

The DBMS_METADATA browsing interface example demonstrates how you can use the browsing interface to obtain the same results.

Example 27-1 Using the DBMS_METADATA Programmatic Interface to Retrieve Data

1. Create a `DBMS_METADATA` program that creates a function named `get_table_md`, which will return the metadata for one table, `timecards`, in the `hr` schema. The content of such a program looks as follows. (For this example, name the program `metadata_program.sql`.)

```
CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
-- Define local variables.
h NUMBER; --handle returned by OPEN
th NUMBER; -- handle returned by ADD_TRANSFORM
doc CLOB;
BEGIN

-- Specify the object type.
h := DBMS_METADATA.OPEN('TABLE');

-- Use filters to specify the particular object desired.
DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

 -- Request that the metadata 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;
/
```

2. Connect as user `hr`.

3. Run the program to create the `get_table_md` function:

```
SQL> @metadata_program
```

4. Use the newly created `get_table_md` function in a select operation. To generate complete, uninterrupted output, set the `PAGESIZE` to 0 and set `LONG` to some large number, as shown, before executing your query:

```
SQL> SET PAGESIZE 0
SQL> SET LONG 1000000
SQL> SELECT get_table_md FROM dual;
```

5. The output, which shows the metadata for the `timecards` table in the `hr` schema, looks similar to the following:

```
  CREATE TABLE "HR"."TIMECARDS"
   (    "EMPLOYEE_ID" NUMBER(6,0),
        "WEEK" NUMBER(2,0),
        "JOB_ID" VARCHAR2(10),
        "HOURS_WORKED" NUMBER(4,2),
         FOREIGN KEY ("EMPLOYEE_ID")
          REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "EXAMPLE"
```

Example 27-2   Using the DBMS_METADATA Browsing Interface to Retrieve Data

```
SQL> SET PAGESIZE 0
SQL> SET LONG 1000000
SQL> SELECT DBMS_METADATA.GET_DDL('TABLE','TIMECARDS','HR') FROM dual;
```

The results of this query are same as shown in step 5 in the `DBMS_METADATA` programmatic interface example.

27.3.3 Retrieving Multiple Objects

Description and example of retrieving multiple objects.

In the previous example “Using the `DBMS_METADATA` Programmatic Interface to Retrieve Data,” the `FETCH_CLOB` procedure was called only once, because it was known that there was only one object. However, you can also retrieve multiple objects, for example, all the tables in schema `scott`. To do this, you need to use the following construct:

```
  LOOP
    doc := DBMS_METADATA.FETCH_CLOB(h);
    --
    -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
    --
    EXIT WHEN doc IS NULL;
  END LOOP;
```

The following example demonstrates use of this construct and retrieving multiple objects. Connect as user `scott` for this example. The password is `tiger`.

Example 27-3   Retrieving Multiple Objects

1. Create a table named `my_metadata` and a procedure named `get_tables_md`, as follows. Because not all objects can be returned, they are stored in a table and queried at the end.

```
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md clob);
CREATE OR REPLACE PROCEDURE get_tables_md IS
-- Define local variables
h       NUMBER;         -- handle returned by 'OPEN'
th      NUMBER;         -- handle returned by 'ADD_TRANSFORM'
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN

 -- Specify the object type.
 h := DBMS_METADATA.OPEN('TABLE');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');

 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Fetch the objects.
 LOOP
   doc := DBMS_METADATA.FETCH_CLOB(h);

  -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in a table.
   INSERT INTO my_metadata(md) VALUES (doc);
```

```
    COMMIT;
  END LOOP;

  -- Release resources.
  DBMS_METADATA.CLOSE(h);
END;
/
```

2. Execute the procedure:

   ```
   EXECUTE get_tables_md;
   ```

3. Query the `my_metadata` table to see what was retrieved:

   ```
   SET LONG 9000000
   SET PAGES 0
   SELECT * FROM my_metadata;
   ```

27.3.4 Placing Conditions on Transforms

To specify conditions on the transforms that you add with `DBMS_METADATA`, you can use transform parameters.

To use transform parameters, you use the `SET_TRANSFORM_PARAM` procedure. For example, if you have added the `DDL` transform for a `TABLE` object, then you can specify the `SEGMENT_ATTRIBUTES` transform parameter to indicate that you do not want segment attributes (physical, storage, logging, and so on) to appear in the DDL. The default is that segment attributes do appear in the DDL.

Example 27-4   Placing Conditions on Transforms

This example shows how to use the `SET_TRANSFORM_PARAM` procedure.

1. Create a function named `get_table_md`, as follows:

   ```
   CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
    -- Define local variables.
    h    NUMBER;   -- handle returned by 'OPEN'
    th   NUMBER;   -- handle returned by 'ADD_TRANSFORM'
    doc  CLOB;
   BEGIN

    -- Specify the object type.
    h := DBMS_METADATA.OPEN('TABLE');

    -- Use filters to specify the particular object desired.
    DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
    DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

    -- Request that the metadata be transformed into creation DDL.
    th := dbms_metadata.add_transform(h,'DDL');

    -- Specify that segment attributes are not to be returned.
    -- Note that this call uses the TRANSFORM handle, not the OPEN handle.
   DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false);

    -- Fetch the object.
    doc := DBMS_METADATA.FETCH_CLOB(h);
   ```

```
 -- Release resources.
 DBMS_METADATA.CLOSE(h);

 RETURN doc;
END;
/
```

2. Perform the following query:

   ```
   SQL> SELECT get_table_md FROM dual;
   ```

   The output looks similar to the following:

   ```
     CREATE TABLE "HR"."TIMECARDS"
      (    "EMPLOYEE_ID" NUMBER(6,0),
           "WEEK" NUMBER(2,0),
           "JOB_ID" VARCHAR2(10),
           "HOURS_WORKED" NUMBER(4,2),
            FOREIGN KEY ("EMPLOYEE_ID")
             REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
      )
   ```

The examples shown up to this point have used a single transform, the `DDL` transform. The `DBMS_METADATA` API also enables you to specify multiple transforms, with the output of the first being the input to the next and so on.

Oracle supplies a transform called `MODIFY` that modifies an XML document. You can do things like change schema names or tablespace names. To do this, you use remap parameters and the `SET_REMAP_PARAM` procedure.

Example 27-5   Modifying an XML Document

This example shows how you can use the `SET_REMAP_PARAM` procedure. It first adds the `MODIFY` transform and specifies remap parameters to change the schema name from `hr` to `scott`. It then adds the `DDL` transform. The output of the `MODIFY` transform is an XML document that becomes the input to the `DDL` transform. The end result is the creation DDL for the `timecards` table with all instances of schema `hr` changed to `scott`.

1. Create a function named `remap_schema`:

   ```
   CREATE OR REPLACE FUNCTION remap_schema RETURN CLOB IS
   -- Define local variables.
   h NUMBER; --handle returned by OPEN
   th NUMBER; -- handle returned by ADD_TRANSFORM
   doc CLOB;
   BEGIN

   -- Specify the object type.
   h := DBMS_METADATA.OPEN('TABLE');

   -- Use filters to specify the particular object desired.
   DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
   DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

   -- Request that the schema name be modified.
   th := DBMS_METADATA.ADD_TRANSFORM(h,'MODIFY');
   ```

```
DBMS_METADATA.SET_REMAP_PARAM(th,'REMAP_SCHEMA','HR','SCOTT');

-- Request that the metadata be transformed into creation DDL.
th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

-- Specify that segment attributes are not to be returned.
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false);

-- Fetch the object.
doc := DBMS_METADATA.FETCH_CLOB(h);

-- Release resources.
DBMS_METADATA.CLOSE(h);
RETURN doc;
END;
/
```

2. Perform the following query:

   ```
   SELECT remap_schema FROM dual;
   ```

   The output looks similar to the following:

   ```
     CREATE TABLE "SCOTT"."TIMECARDS"
      (    "EMPLOYEE_ID" NUMBER(6,0),
           "WEEK" NUMBER(2,0),
           "JOB_ID" VARCHAR2(10),
           "HOURS_WORKED" NUMBER(4,2),
            FOREIGN KEY ("EMPLOYEE_ID")
             REFERENCES "SCOTT"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
      )
   ```

   If you are familiar with XSLT, then you can add your own user-written transforms to process the XML.

Example 27-6 INCLUDE_SHARDING_CLAUSES and PARTITION BY or PARTITIONS AUTO Keywords

Starting with Oracle Database 23ai, you can set the API transform parameter `INCLUDE_SHARDING_CLAUSES` using `dbms_metedata.set_transform_param()`. If it is set to `TRUE`, then `get_ddl()` will generate shard syntax as described below.

Create with a `SHARDED` keyword:

In the following example, a sharded table is created using the keyword `customers`:

```
CREATE SHARDED TABLE customers (
   custno   NUMBER NOT NULL,
   region  char(2) NOT NULL,
   name  VARCHAR2(20),
   zip number)
PARTITION BY CONSISTENT HASH (custno, region)
PARTITIONS AUTO
TABLESPACE SET ts1;
```

When the `INCLUDE_SHARDING_CLAUSES` parameter is set to `FALSE`, the DDL will contain `PARTITION BY RANGE` and not include the `PARTITIONS AUTO` clause. For example:

Partition by a consistent hash:

```
CREATE SHARDED TABLE customers (
   custno   NUMBER NOT NULL,
   region  char(2) NOT NULL,
   name  VARCHAR2(20),
   zip number)
PARTITION BY CONSISTENT HASH (custno, region)
PARTITIONS AUTO
TABLESPACE SET ts1;
```

27.3.5 Accessing Specific Metadata Attributes

See how you can access specific metadata attributes of an object's metadata with the `DBMS_METADATA` API.

It is often desirable to access specific attributes of an object's metadata, for example, its name or schema. You could get this information by parsing the returned metadata, but the `DBMS_METADATA` API provides another mechanism; you can specify parse items, specific attributes that will be parsed out of the metadata and returned in a separate data structure. To do this, you use the `SET_PARSE_ITEM` procedure.

Example 27-7   Using Parse Items to Access Specific Metadata Attributes

This example shows how to check all tables in a schema. For each table, a parse item is used to obtain its name. The name is then used to obtain all indexes on the table. In this example, you can see how to use the `FETCH_DDL` function, which returns metadata in a `sys.ku$_ddls` object.

In this example, we assume that you are connected to a schema that contains some tables and indexes. The outcome of this series of steps creates a table named `my_metadata`.

1. Create a table named `my_metadata` and a procedure named `get_tables_and_indexes`, as follows:

```
DROP TABLE my_metadata;
CREATE TABLE my_metadata (
  object_type   VARCHAR2(30),
  name          VARCHAR2(30),
  md            CLOB);
CREATE OR REPLACE PROCEDURE get_tables_and_indexes IS
-- Define local variables.
h1      NUMBER;         -- handle returned by OPEN for tables
h2      NUMBER;         -- handle returned by OPEN for indexes
th1     NUMBER;         -- handle returned by ADD_TRANSFORM for tables
th2     NUMBER;         -- handle returned by ADD_TRANSFORM for indexes
doc     sys.ku$_ddls;   -- metadata is returned in sys.ku$_ddls,
                        --  a nested table of sys.ku$_ddl objects
ddl     CLOB;           -- creation DDL for an object
pi      sys.ku$_parsed_items;   -- parse items are returned in this object
                                -- which is contained in sys.ku$_ddl
objname VARCHAR2(30);   -- the parsed object name
idxddls sys.ku$_ddls;   -- metadata is returned in sys.ku$_ddls,
                        --  a nested table of sys.ku$_ddl objects
idxname VARCHAR2(30);   -- the parsed index name
BEGIN
```

```
 -- This procedure has an outer loop that fetches tables,
 -- and an inner loop that fetches indexes.

 -- Specify the object type: TABLE.
 h1 := DBMS_METADATA.OPEN('TABLE');

 -- Request that the table name be returned as a parse item.
 DBMS_METADATA.SET_PARSE_ITEM(h1,'NAME');

 -- Request that the metadata be transformed into creation DDL.
 th1 := DBMS_METADATA.ADD_TRANSFORM(h1,'DDL');

 -- Specify that segment attributes are not to be returned.
 DBMS_METADATA.SET_TRANSFORM_PARAM(th1,'SEGMENT_ATTRIBUTES',false);

 -- Set up the outer loop: fetch the TABLE objects.
 LOOP
   doc := dbms_metadata.fetch_ddl(h1);

-- When there are no more objects to be retrieved, FETCH_DDL returns NULL.
   EXIT WHEN doc IS NULL;

-- Loop through the rows of the ku$_ddls nested table.
   FOR i IN doc.FIRST..doc.LAST LOOP
     ddl := doc(i).ddlText;
     pi := doc(i).parsedItems;
     -- Loop through the returned parse items.
     IF pi IS NOT NULL AND pi.COUNT > 0 THEN
       FOR j IN pi.FIRST..pi.LAST LOOP
         IF pi(j).item='NAME' THEN
           objname := pi(j).value;
         END IF;
       END LOOP;
     END IF;
     -- Insert information about this object into our table.
     INSERT INTO my_metadata(object_type, name, md)
       VALUES ('TABLE',objname,ddl);
     COMMIT;
   END LOOP;

   -- Now fetch indexes using the parsed table name as
   --  a BASE_OBJECT_NAME filter.

   -- Specify the object type.
   h2 := DBMS_METADATA.OPEN('INDEX');

   -- The base object is the table retrieved in the outer loop.
   DBMS_METADATA.SET_FILTER(h2,'BASE_OBJECT_NAME',objname);

   -- Exclude system-generated indexes.
   DBMS_METADATA.SET_FILTER(h2,'SYSTEM_GENERATED',false);

   -- Request that the index name be returned as a parse item.
   DBMS_METADATA.SET_PARSE_ITEM(h2,'NAME');

   -- Request that the metadata be transformed into creation DDL.
   th2 := DBMS_METADATA.ADD_TRANSFORM(h2,'DDL');

   -- Specify that segment attributes are not to be returned.
   DBMS_METADATA.SET_TRANSFORM_PARAM(th2,'SEGMENT_ATTRIBUTES',false);
```

```
     LOOP
      idxddls := dbms_metadata.fetch_ddl(h2);

      -- When there are no more objects to  be retrieved, FETCH_DDL returns NULL.
      EXIT WHEN idxddls IS NULL;

        FOR i in idxddls.FIRST..idxddls.LAST LOOP
          ddl  := idxddls(i).ddlText;
          pi   := idxddls(i).parsedItems;
          -- Loop through the returned parse items.
          IF pi IS NOT NULL AND pi.COUNT > 0 THEN
            FOR j IN pi.FIRST..pi.LAST LOOP
              IF pi(j).item='NAME' THEN
                idxname := pi(j).value;
              END IF;
            END LOOP;
           END IF;

           -- Store the metadata in our table.
            INSERT INTO my_metadata(object_type, name, md)
              VALUES ('INDEX',idxname,ddl);
           COMMIT;
         END LOOP;  -- for loop
    END LOOP;
    DBMS_METADATA.CLOSE(h2);
   END LOOP;
   DBMS_METADATA.CLOSE(h1);
  END;
  /
```

2. Execute the procedure:

```
EXECUTE get_tables_and_indexes;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGES 0
SELECT * FROM my_metadata;
```

27.4 Using the DBMS_METADATA API to Recreate a Retrieved Object

When you fetch metadata for an object, you can choose to use it to recreate the object in a different database or schema.

When you fetch metadata, suppose that you are not ready to make remapping decisions, and you want to defer these decisions until later. To defer your decision about remapping, you can fetch the metadata as XML, and store it in a file or table. Later, you can use that file or table with the submit interface to recreate the object.

The submit interface is similar in form to the retrieval interface. It has an `OPENW` procedure, in which you specify the object type of the object that you want to create. You can specify transforms, transform parameters, and parse items. You can call the `CONVERT` function to convert the XML to DDL, or you can call the `PUT` function to both convert XML to DDL, and to submit the DDL to create the object.

Example 27-8   Using the Submit Interface to Re-Create a Retrieved Object

This example shows how to fetch the XML for a table in one schema, and then use the submit interface to recreate the table in another schema.

1. Connect as a privileged user:

```
CONNECT system
Enter password: password
```

2. Because access to objects in another schema requires the `SELECT_CATALOG_ROLE` role, create an invoker's rights package to hold the procedure. In a definer's rights PL/SQL object (such as a procedure or function), roles are disabled.

```
CREATE OR REPLACE PACKAGE example_pkg AUTHID current_user IS
  PROCEDURE move_table(
        table_name  in VARCHAR2,
        from_schema in VARCHAR2,
        to_schema   in VARCHAR2 );
END example_pkg;
/
CREATE OR REPLACE PACKAGE BODY example_pkg IS
PROCEDURE move_table(
        table_name  in VARCHAR2,
        from_schema in VARCHAR2,
        to_schema   in VARCHAR2 ) IS

-- Define local variables.
h1      NUMBER;         -- handle returned by OPEN
h2      NUMBER;         -- handle returned by OPENW
th1     NUMBER;         -- handle returned by ADD_TRANSFORM for MODIFY
th2     NUMBER;         -- handle returned by ADD_TRANSFORM for DDL
xml     CLOB;           -- XML document
errs    sys.ku$_SubmitResults := sys.ku$_SubmitResults();
err     sys.ku$_SubmitResult;
result  BOOLEAN;
BEGIN

-- Specify the object type.
h1 := DBMS_METADATA.OPEN('TABLE');

-- Use filters to specify the name and schema of the table.
DBMS_METADATA.SET_FILTER(h1,'NAME',table_name);
DBMS_METADATA.SET_FILTER(h1,'SCHEMA',from_schema);

-- Fetch the XML.
xml := DBMS_METADATA.FETCH_CLOB(h1);
IF xml IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Table ' || from_schema || '.' || table_name
|| ' not found');
    RETURN;
  END IF;

-- Release resources.
DBMS_METADATA.CLOSE(h1);
```

```
-- Use the submit interface to re-create the object in another schema.

-- Specify the object type using OPENW (instead of OPEN).
h2 := DBMS_METADATA.OPENW('TABLE');

-- First, add the MODIFY transform.
th1 := DBMS_METADATA.ADD_TRANSFORM(h2,'MODIFY');

-- Specify the desired modification: remap the schema name.
DBMS_METADATA.SET_REMAP_PARAM(th1,'REMAP_SCHEMA',from_schema,to_schema);

-- Now add the DDL transform so that the modified XML can be
--  transformed into creation DDL.
th2 := DBMS_METADATA.ADD_TRANSFORM(h2,'DDL');

-- Call PUT to re-create the object.
result := DBMS_METADATA.PUT(h2,xml,0,errs);

DBMS_METADATA.CLOSE(h2);
  IF NOT result THEN
    -- Process the error information.
    FOR i IN errs.FIRST..errs.LAST LOOP
      err := errs(i);
      FOR j IN err.errorLines.FIRST..err.errorLines.LAST LOOP
        dbms_output.put_line(err.errorLines(j).errorText);
      END LOOP;
    END LOOP;
  END IF;
END;
END example_pkg;
/
```

3. Next, create a table named `my_example` in the schema `SCOTT`:

```
CONNECT scott
Enter password:
-- The password is tiger.

DROP TABLE my_example;
CREATE TABLE my_example (a NUMBER, b VARCHAR2(30));

CONNECT system
Enter password: password

SET LONG 9000000
SET PAGESIZE 0
SET SERVEROUTPUT ON SIZE 100000
```

4. Copy the `my_example` table to the `SYSTEM` schema:

```
DROP TABLE my_example;
EXECUTE example_pkg.move_table('MY_EXAMPLE','SCOTT','SYSTEM');
```

5. Perform the following query to verify that it worked:

```
SELECT DBMS_METADATA.GET_DDL('TABLE','MY_EXAMPLE') FROM dual;
```

27.5 Using the DBMS_METADATA API to Retrieve Collections of Different Object Types

To retrieve collections of objects in which the objects are of different types, but comprise a logical unit, you can use the heterogeneous object types in the DBMS_METADATA API.

There can be times when you need to retrieve collections of Oracle Database objects in which the objects are of different types, but comprise a logical unit. For example, you might need to retrieve all the objects in a database or a schema, or a table and all its dependent indexes, constraints, grants, audits, and so on. To make such a retrieval possible, the DBMS_METADATA API provides several heterogeneous object types. A heterogeneous object type is an ordered set of object types.

Oracle supplies the following heterogeneous object types:

- TABLE_EXPORT - a table and its dependent objects
- SCHEMA_EXPORT - a schema and its contents
- DATABASE_EXPORT - the objects in the database

These object types were developed for use by the Oracle Data Pump Export utility, but you can use them in your own applications.

You can use only the programmatic retrieval interface (OPEN, FETCH, CLOSE) with these types, not the browsing interface or the submit interface.

You can specify filters for heterogeneous object types, just as you do for the homogeneous types. For example, you can specify the SCHEMA and NAME filters for TABLE_EXPORT, or the SCHEMA filter for SCHEMA_EXPORT.

Example 27-9 Retrieving Heterogeneous Object Types

This example shows you how to retrieve the object types in the user scott schema. Connect as user scott. The password is tiger.

1. Create a table to store the retrieved objects:

```
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md CLOB);
CREATE OR REPLACE PROCEDURE get_schema_md IS

-- Define local variables.
h       NUMBER;         -- handle returned by OPEN
th      NUMBER;         -- handle returned by ADD_TRANSFORM
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN

-- Specify the object type.
 h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');
 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Fetch the objects.
 LOOP
   doc := DBMS_METADATA.FETCH_CLOB(h);

   -- When there are no more objects to be retrieved, FETCH_CLOB returns
NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in the table.
   INSERT INTO my_metadata(md) VALUES (doc);
   COMMIT;
 END LOOP;

 -- Release resources.
 DBMS_METADATA.CLOSE(h);
END;
/
```

2. Execute the procedure:

```
EXECUTE get_schema_md;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

In this example, objects are returned ordered by object type; for example, all tables are returned, then all grants on tables, then all indexes on tables, and so on. The order is, generally speaking, a valid creation order. Thus, if you take the objects in the order in which they were returned and use the submit interface to recreate them in the same order in another schema or database, then there usually should be no errors. (The exceptions usually involve circular references; for example, if package A contains a call to package B, and package B contains a call to package A, then one of the packages must be recompiled a second time.)

27.6 Filtering the Return of Heterogeneous Object Types

Learn how you can use the SET_FILTER procedure to enable you to filter the return of heterogeneous object types.

For finer control of the objects returned, use the SET_FILTER procedure and specify that the filter apply only to a specific member type. You do this by specifying the path name of the member type as the fourth parameter to SET_FILTER. In addition, you can use the EXCLUDE_PATH_EXPR filter to exclude all objects of an object type. For a list of valid path names, see the TABLE_EXPORT_OBJECTS catalog view.

Example 27-10   Filtering the Return of Heterogeneous Object Types

In this example, SET_FILTER is used to specify finer control on the objects returned.:

1. Create a table, `my_metadata`, to store the retrieved objects, and create a procedure, `get_schema_md2`:

```
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md CLOB);
CREATE OR REPLACE PROCEDURE get_schema_md2 IS

-- Define local variables.
h       NUMBER;         -- handle returned by 'OPEN'
th      NUMBER;         -- handle returned by 'ADD_TRANSFORM'
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN

 -- Specify the object type.
 h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');

 -- Use the fourth parameter to SET_FILTER to specify a filter
 -- that applies to a specific member object type.
 DBMS_METADATA.SET_FILTER(h,'NAME_EXPR','!=''MY_METADATA''','TABLE');

 -- Use the EXCLUDE_PATH_EXPR filter to exclude procedures.
 DBMS_METADATA.SET_FILTER(h,'EXCLUDE_PATH_EXPR','=''PROCEDURE''');

 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Use the fourth parameter to SET_TRANSFORM_PARAM to specify a parameter
 --  that applies to a specific member object type.
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false,'TABLE');

 -- Fetch the objects.
 LOOP
   doc := dbms_metadata.fetch_clob(h);

   -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in the table.
   INSERT INTO my_metadata(md) VALUES (doc);
   COMMIT;
 END LOOP;

 -- Release resources.
 DBMS_METADATA.CLOSE(h);
END;
/
```

2. Run the procedure:

```
EXECUTE get_schema_md2;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

27.7 Using the DBMS_METADATA_DIFF API to Compare Object Metadata

Description and example that uses the retrieval, comparison, and submit interfaces of `DBMS_METADATA` and `DBMS_METADATA_DIFF` to fetch metadata for two tables, compare the metadata, and generate `ALTER` statements which make one table like the other.

For simplicity, function variants are used throughout the example.

Example 27-11   Comparing Object Metadata

1. Create two tables, `TAB1` and `TAB2`:

```
SQL> CREATE TABLE TAB1
  2     (     "EMPNO" NUMBER(4,0),
  3           "ENAME" VARCHAR2(10),
  4           "JOB" VARCHAR2(9),
  5           "DEPTNO" NUMBER(2,0)
  6     ) ;

Table created.

SQL> CREATE TABLE TAB2
  2     (     "EMPNO" NUMBER(4,0) PRIMARY KEY ENABLE,
  3           "ENAME" VARCHAR2(20),
  4           "MGR" NUMBER(4,0),
  5           "DEPTNO" NUMBER(2,0)
  6     ) ;

Table created.
```

   Note the differences between `TAB1` and `TAB2`:

   - The table names are different
   - `TAB2` has a primary key constraint; `TAB1` does not
   - The length of the `ENAME` column is different in each table
   - `TAB1` has a `JOB` column; `TAB2` does not
   - `TAB2` has a `MGR` column; `TAB1` does not

2. Create a function to return the table metadata in SXML format. The following are some key points to keep in mind about SXML when you are using the `DBMS_METADATA_DIFF` API:

   - SXML is an XML representation of object metadata.
   - The SXML returned is not the same as the XML returned by `DBMS_METADATA.GET_XML`, which is complex and opaque and contains binary values, instance-specific values, and so on.
   - SXML looks like a direct translation of SQL creation DDL into XML. The tag names and structure correspond to names in the *Oracle Database SQL Language Reference*.
   - SXML is designed to support editing and comparison.

   To keep this example simple, a transform parameter is used to suppress physical properties:

```
SQL> CREATE OR REPLACE FUNCTION get_table_sxml(name IN VARCHAR2) RETURN CLOB IS
  2   open_handle NUMBER;
  3   transform_handle NUMBER;
  4   doc CLOB;
  5  BEGIN
  6   open_handle := DBMS_METADATA.OPEN('TABLE');
  7   DBMS_METADATA.SET_FILTER(open_handle,'NAME',name);
  8   --
  9   -- Use the 'SXML' transform to convert XML to SXML
 10   --
 11   transform_handle := DBMS_METADATA.ADD_TRANSFORM(open_handle,'SXML');
 12   --
 13   -- Use this transform parameter to suppress physical properties
 14   --
 15   DBMS_METADATA.SET_TRANSFORM_PARAM(transform_handle,'PHYSICAL_PROPERTIES',
 16                                     FALSE);
 17   doc := DBMS_METADATA.FETCH_CLOB(open_handle);
 18   DBMS_METADATA.CLOSE(open_handle);
 19   RETURN doc;
 20  END;
 21  /

Function created.
```

3. Use the `get_table_sxml` function to fetch the table SXML for the two tables:

```
SQL> SELECT get_table_sxml('TAB1') FROM dual;

  <TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
   <SCHEMA>SCOTT</SCHEMA>
   <NAME>TAB1</NAME>
   <RELATIONAL_TABLE>
      <COL_LIST>
         <COL_LIST_ITEM>
            <NAME>EMPNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>4</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>ENAME</NAME>
            <DATATYPE>VARCHAR2</DATATYPE>
            <LENGTH>10</LENGTH>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>JOB</NAME>
            <DATATYPE>VARCHAR2</DATATYPE>
            <LENGTH>9</LENGTH>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>DEPTNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>2</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
      </COL_LIST>
   </RELATIONAL_TABLE>
</TABLE>

1 row selected.

SQL> SELECT get_table_sxml('TAB2') FROM dual;
```

```
  <TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
   <SCHEMA>SCOTT</SCHEMA>
   <NAME>TAB2</NAME>
   <RELATIONAL_TABLE>
      <COL_LIST>
         <COL_LIST_ITEM>
            <NAME>EMPNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>4</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>ENAME</NAME>
            <DATATYPE>VARCHAR2</DATATYPE>
            <LENGTH>20</LENGTH>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>MGR</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>4</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>DEPTNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>2</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
      </COL_LIST>
      <PRIMARY_KEY_CONSTRAINT_LIST>
         <PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
            <COL_LIST>
               <COL_LIST_ITEM>
                  <NAME>EMPNO</NAME>
               </COL_LIST_ITEM>
            </COL_LIST>
         </PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
      </PRIMARY_KEY_CONSTRAINT_LIST>
   </RELATIONAL_TABLE>
</TABLE>

1 row selected.
```

4. Compare the results using the `DBMS_METADATA` browsing APIs:

   ```
   SQL> SELECT dbms_metadata.get_sxml('TABLE','TAB1') FROM dual;
   SQL> SELECT dbms_metadata.get_sxml('TABLE','TAB2') FROM dual;
   ```

5. Create a function using the `DBMS_METADATA_DIFF` API to compare the metadata for the two tables. In this function, the `get_table_sxml` function that was just defined in step 2 is used.

   ```
   SQL> CREATE OR REPLACE FUNCTION compare_table_sxml(name1 IN VARCHAR2,
     2                                         name2 IN VARCHAR2) RETURN CLOB IS
     3   doc1 CLOB;
     4   doc2 CLOB;
     5   diffdoc CLOB;
     6   openc_handle NUMBER;
     7  BEGIN
     8   --
     9   -- Fetch the SXML for the two tables
    10   --
   ```

```
 11   doc1 := get_table_sxml(name1);
 12   doc2 := get_table_sxml(name2);
 13   --
 14   -- Specify the object type in the OPENC call
 15   --
 16   openc_handle := DBMS_METADATA_DIFF.OPENC('TABLE');
 17   --
 18   -- Add each document
 19   --
 20   DBMS_METADATA_DIFF.ADD_DOCUMENT(openc_handle,doc1);
 21   DBMS_METADATA_DIFF.ADD_DOCUMENT(openc_handle,doc2);
 22   --
 23   -- Fetch the SXML difference document
 24   --
 25   diffdoc := DBMS_METADATA_DIFF.FETCH_CLOB(openc_handle);
 26   DBMS_METADATA_DIFF.CLOSE(openc_handle);
 27   RETURN diffdoc;
 28  END;
 29  /

Function created.
```

6. Use the function to fetch the SXML difference document for the two tables:

```
SQL> SELECT compare_table_sxml('TAB1','TAB2') FROM dual;

<TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
  <SCHEMA>SCOTT</SCHEMA>
  <NAME value1="TAB1">TAB2</NAME>
  <RELATIONAL_TABLE>
    <COL_LIST>
      <COL_LIST_ITEM>
        <NAME>EMPNO</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>4</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM>
        <NAME>ENAME</NAME>
        <DATATYPE>VARCHAR2</DATATYPE>
        <LENGTH value1="10">20</LENGTH>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM src="1">
        <NAME>JOB</NAME>
        <DATATYPE>VARCHAR2</DATATYPE>
        <LENGTH>9</LENGTH>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM>
        <NAME>DEPTNO</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>2</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM src="2">
        <NAME>MGR</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>4</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
    </COL_LIST>
    <PRIMARY_KEY_CONSTRAINT_LIST src="2">
      <PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
        <COL_LIST>
```

```
          <COL_LIST_ITEM>
            <NAME>EMPNO</NAME>
          </COL_LIST_ITEM>
        </COL_LIST>
      </PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
    </PRIMARY_KEY_CONSTRAINT_LIST>
  </RELATIONAL_TABLE>
</TABLE>

1 row selected.
```

The SXML difference document shows the union of the two SXML documents, with the XML attributes `value1` and `src` identifying the differences. When an element exists in only one document it is marked with `src`. Thus, `<COL_LIST_ITEM src="1">` means that this element is in the first document (`TAB1`) but not in the second. When an element is present in both documents but with different values, the element's value is the value in the second document and the `value1` gives its value in the first. For example, `<LENGTH value1="10">20</LENGTH>` means that the length is 10 in `TAB1` (the first document) and 20 in `TAB2`.

7. Compare the result using the `DBMS_METADATA_DIFF` browsing APIs:

```
SQL> SELECT dbms_metadata_diff.compare_sxml('TABLE','TAB1','TAB2') FROM dual;
```

8. Create a function using the `DBMS_METADATA.CONVERT` API to generate an ALTERXML document. This is an XML document containing `ALTER` statements to make one object like another. You can also use parse items to get information about the individual `ALTER` statements. (This example uses the functions defined thus far.)

```
SQL> CREATE OR REPLACE FUNCTION get_table_alterxml(name1 IN VARCHAR2,
  2                                         name2 IN VARCHAR2) RETURN CLOB IS
  3   diffdoc CLOB;
  4   openw_handle NUMBER;
  5   transform_handle NUMBER;
  6   alterxml CLOB;
  7  BEGIN
  8   --
  9   -- Use the function just defined to get the difference document
 10   --
 11   diffdoc := compare_table_sxml(name1,name2);
 12   --
 13   -- Specify the object type in the OPENW call
 14   --
 15   openw_handle := DBMS_METADATA.OPENW('TABLE');
 16   --
 17   -- Use the ALTERXML transform to generate the ALTER_XML document
 18   --
 19   transform_handle := DBMS_METADATA.ADD_TRANSFORM(openw_handle,'ALTERXML');
 20   --
 21   -- Request parse items
 22   --
 23   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'CLAUSE_TYPE');
 24   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'NAME');
 25   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'COLUMN_ATTRIBUTE');
 26   --
 27   -- Create a temporary LOB
 28   --
 29   DBMS_LOB.CREATETEMPORARY(alterxml, TRUE );
 30   --
 31   -- Call CONVERT to do the transform
 32   --
 DBMS_METADATA.CONVERT(openw_handle,diffdoc,alterxml);
 --
 -- Close context and return the result
 --
 DBMS_METADATA.CLOSE(openw_handle);
 RETURN alterxml;
END;
/

Function created.
```

9. Use the function to fetch the ALTER_XML document:

```
SQL> SELECT get_table_alterxml('TAB1','TAB2') FROM dual;

<ALTER_XML xmlns="http://xmlns.oracle.com/ku" version="1.0">
   <OBJECT_TYPE>TABLE</OBJECT_TYPE>
   <OBJECT1>
      <SCHEMA>SCOTT</SCHEMA>
      <NAME>TAB1</NAME>
   </OBJECT1>
   <OBJECT2>
      <SCHEMA>SCOTT</SCHEMA>
      <NAME>TAB2</NAME>
   </OBJECT2>
   <ALTER_LIST>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>NAME</ITEM>
               <VALUE>MGR</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>ADD_COLUMN</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))</TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>NAME</ITEM>
               <VALUE>JOB</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>DROP_COLUMN</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")</TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
```

```
               <ITEM>NAME</ITEM>
               <VALUE>ENAME</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>MODIFY_COLUMN</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>COLUMN_ATTRIBUTE</ITEM>
               <VALUE> SIZE_INCREASE</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" MODIFY
                    ("ENAME" VARCHAR2(20))
               </TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>ADD_CONSTRAINT</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY
                     ("EMPNO") ENABLE
               </TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>NAME</ITEM>
               <VALUE>TAB1</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>RENAME_TABLE</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"</TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
   </ALTER_LIST>
</ALTER_XML>


1 row selected.
```

10. Compare the result using the `DBMS_METADATA_DIFF` browsing API:

```
SQL> SELECT dbms_metadata_diff.compare_alter_xml('TABLE','TAB1','TAB2') FROM dual;
```

11. The ALTER_XML document contains an ALTER_LIST of each of the alters. Each ALTER_LIST_ITEM has a PARSE_LIST containing the parse items as name-value pairs and a SQL_LIST containing the SQL for the particular alter. You can parse this document and decide which of the SQL statements to execute, using the information in the PARSE_LIST. (Note, for example, that in this case one of the alters is a DROP_COLUMN, and you might choose not to execute that.)

12. Create one last function that uses the `DBMS_METADATA.CONVERT` API and the `ALTER DDL` transform to convert the ALTER_XML document into SQL DDL:

```
SQL> CREATE OR REPLACE FUNCTION get_table_alterddl(name1 IN VARCHAR2,
                                              name2 IN VARCHAR2) RETURN CLOB IS
      alterxml CLOB;
      openw_handle NUMBER;
      transform_handle NUMBER;
      alterddl CLOB;
     BEGIN
      --
      -- Use the function just defined to get the ALTER_XML document
      --
      alterxml := get_table_alterxml(name1,name2);
      --
      -- Specify the object type in the OPENW call
      --
      openw_handle := DBMS_METADATA.OPENW('TABLE');
      --
      -- Use ALTERDDL transform to convert the ALTER_XML document to SQL DDL
      --
      transform_handle := DBMS_METADATA.ADD_TRANSFORM(openw_handle,'ALTERDDL');
      --
      -- Use the SQLTERMINATOR transform parameter to append a terminator
      -- to each SQL statement
      --
      DBMS_METADATA.SET_TRANSFORM_PARAM(transform_handle,'SQLTERMINATOR',true);
      --
      -- Create a temporary lob
      --
      DBMS_LOB.CREATETEMPORARY(alterddl, TRUE );
      --
      -- Call CONVERT to do the transform
      --
      DBMS_METADATA.CONVERT(openw_handle,alterxml,alterddl);
      --
      -- Close context and return the result
      --
      DBMS_METADATA.CLOSE(openw_handle);
      RETURN alterddl;
     END;
     /

Function created.
```

13. Use the function to fetch the SQL ALTER statements:

```
SQL> SELECT get_table_alterddl('TAB1','TAB2') FROM dual;
ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))
/
  ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")
/
  ALTER TABLE "SCOTT"."TAB1" MODIFY ("ENAME" VARCHAR2(20))
/
  ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY ("EMPNO") ENABLE
```

```
/
  ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"
/

1 row selected.
```

14. Compare the results using the `DBMS_METADATA_DIFF` browsing API:

```
SQL> SELECT dbms_metadata_diff.compare_alter('TABLE','TAB1','TAB2') FROM dual;
ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))
  ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")
  ALTER TABLE "SCOTT"."TAB1" MODIFY ("ENAME" VARCHAR2(20))
  ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY ("EMPNO") USING INDEX
  PCTFREE 10 INITRANS 2 STORAGE ( INITIAL 16384 NEXT 16384 MINEXTENTS 1
  MAXEXTENTS 505 PCTINCREASE 50 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
  DEFAULT)  ENABLE ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"

1 row selected.
```

27.8 Performance Tips for the Programmatic Interface of the DBMS_METADATA API

Describes how to enhance performance when using the programmatic interface of the `DBMS_METADATA` API.

Specifically:

1. Fetch all of one type of object before fetching the next. For example, if you are retrieving the definitions of all objects in your schema, first fetch all tables, then all indexes, then all triggers, and so on. This will be much faster than nesting `OPEN` contexts; that is, fetch one table then all of its indexes, grants, and triggers, then the next table and all of its indexes, grants, and triggers, and so on. Example Usage of the DBMS_METADATA API reflects this second, less efficient means, but its purpose is to demonstrate most of the programmatic calls, which are best shown by this method.
2. Use the `SET_COUNT` procedure to retrieve more than one object at a time. This minimizes server round trips and eliminates many redundant function calls.
3. When writing a PL/SQL package that calls the `DBMS_METADATA` API, declare LOB variables and objects that contain LOBs (such as `SYS.KU$_DDLS`) at package scope rather than within individual functions. This eliminates the creation and deletion of LOB duration structures upon function entrance and exit, which are very expensive operations.

27.9 Example Usage of the DBMS_METADATA API

Example of how the `DBMS_METADATA` API could be used.

A script is provided that automatically runs the demo for you by performing the following actions:

- Establishes a schema (`MDDEMO`) and some payroll users.
- Creates three payroll-like tables within the schema and any associated indexes, triggers, and grants.
- Creates a package, `PAYROLL_DEMO`, that uses the `DBMS_METADATA` API. The `PAYROLL_DEMO` package contains a procedure, `GET_PAYROLL_TABLES`, that retrieves the DDL for the two tables in the `MDDEMO` schema that start with `PAYROLL`. For each table, it retrieves the DDL for

the table's associated dependent objects; indexes, grants, and triggers. All the DDL is written to a table named `MDDEMO.DDL`.

To execute the example, do the following:

1. Start SQL*Plus as user `system`. You will be prompted for a password.

   ```
   sqlplus system
   ```

2. Install the demo, which is located in the file `mddemo.sql` in `rdbms`/`demo`:

   ```
   SQL> @mddemo
   ```

   For an explanation of what happens during this step, see What Does the DBMS_METADATA Example Do?.

3. Connect as user `mddemo`. You will be prompted for a password, which is also `mddemo`.

   ```
   SQL> CONNECT mddemo
   Enter password:
   ```

4. Set the following parameters so that query output will be complete and readable:

   ```
   SQL> SET PAGESIZE 0
   SQL> SET LONG 1000000
   ```

5. Execute the `GET_PAYROLL_TABLES` procedure, as follows:

   ```
   SQL> CALL payroll_demo.get_payroll_tables();
   ```

6. Execute the following SQL query:

   ```
   SQL> SELECT ddl FROM DDL ORDER BY SEQNO;
   ```

   The output generated is the result of the execution of the `GET_PAYROLL_TABLES` procedure. It shows all the DDL that was performed in Step 2 when the demo was installed. See Output Generated from the GET_PAYROLL_TABLES Procedure for a listing of the actual output.

- What Does the DBMS_METADATA Example Do?
  Explanation of the `DBMS_METADATA` example.
- Output Generated from the GET_PAYROLL_TABLES Procedure
  Explanation of the output generated from the `GET_PAYROLL_TABLES` procedure.

27.9.1 What Does the DBMS_METADATA Example Do?

Explanation of the `DBMS_METADATA` example.

When the `mddemo` script is run, the following steps take place. You can adapt these steps to your own situation.

1. Drops users as follows, if they exist. This will ensure that you are starting out with fresh data. If the users do not exist, then a message to that effect is displayed, no harm is done, and the demo continues to execute.

   ```
   CONNECT system
   Enter password: password
   SQL> DROP USER mddemo CASCADE;
   SQL> DROP USER mddemo_clerk CASCADE;
   SQL> DROP USER mddemo_mgr CASCADE;
   ```

2. Creates user `mddemo`, identified by `mddemo`:

```
SQL> CREATE USER mddemo IDENTIFIED BY mddemo;
SQL> GRANT resource, connect, create session,
  1     create table,
  2     create procedure,
  3     create sequence,
  4     create trigger,
  5     create view,
  6     create synonym,
  7     alter session,
  8  TO mddemo;
```

3. Creates user `mddemo_clerk`, identified by `clerk`:

```
CREATE USER mddemo_clerk IDENTIFIED BY clerk;
```

4. Creates user `mddemo_mgr`, identified by `mgr`:

```
CREATE USER mddemo_mgr IDENTIFIED BY mgr;
```

5. Connect to SQL*Plus as `mddemo` (the password is also `mddemo`):

```
CONNECT mddemo
Enter password:
```

6. Creates some payroll-type tables:

```
SQL> CREATE TABLE payroll_emps
  2  ( lastname VARCHAR2(60) NOT NULL,
  3  firstname VARCHAR2(20) NOT NULL,
  4  mi VARCHAR2(2),
  5  suffix VARCHAR2(10),
  6  dob DATE NOT NULL,
  7  badge_no NUMBER(6) PRIMARY KEY,
  8  exempt VARCHAR(1) NOT NULL,
  9  salary NUMBER (9,2),
  10 hourly_rate NUMBER (7,2) )
  11 /

SQL> CREATE TABLE payroll_timecards
  2  (badge_no NUMBER(6) REFERENCES payroll_emps (badge_no),
  3  week NUMBER(2),
  4  job_id NUMBER(5),
  5  hours_worked NUMBER(4,2) )
  6 /
```

7. Creates a dummy table, `audit_trail`. This table is used to show that tables that do not start with `payroll` are not retrieved by the `GET_PAYROLL_TABLES` procedure.

```
SQL> CREATE TABLE audit_trail
  2  (action_time DATE,
  3  lastname VARCHAR2(60),
  4  action LONG )
  5  /
```

8. Creates some grants on the tables just created:

```
SQL> GRANT UPDATE (salary,hourly_rate) ON payroll_emps TO mddemo_clerk;
SQL> GRANT ALL ON payroll_emps TO mddemo_mgr WITH GRANT OPTION;

SQL> GRANT INSERT,UPDATE ON payroll_timecards TO mddemo_clerk;
SQL> GRANT ALL ON payroll_timecards TO mddemo_mgr WITH GRANT OPTION;
```

9. Creates some indexes on the tables just created:

```
SQL> CREATE INDEX i_payroll_emps_name ON payroll_emps(lastname);
SQL> CREATE INDEX i_payroll_emps_dob ON payroll_emps(dob);
SQL> CREATE INDEX i_payroll_timecards_badge ON payroll_timecards(badge_no);
```

10. Creates some triggers on the tables just created:

```
SQL> CREATE OR REPLACE PROCEDURE check_sal( salary in number) AS BEGIN
  2  RETURN;
  3  END;
  4  /
```

Note that the security is kept fairly loose to keep the example simple.

```
SQL> CREATE OR REPLACE TRIGGER salary_trigger BEFORE INSERT OR UPDATE OF salary
ON payroll_emps
FOR EACH ROW WHEN (new.salary > 150000)
CALL check_sal(:new.salary)
/

SQL> CREATE OR REPLACE TRIGGER hourly_trigger BEFORE UPDATE OF hourly_rate ON
payroll_emps
FOR EACH ROW
BEGIN :new.hourly_rate:=:old.hourly_rate;END;
/
```

11. Sets up a table to hold the generated DDL:

```
CREATE TABLE ddl (ddl CLOB, seqno NUMBER);
```

12. Creates the `PAYROLL_DEMO` package, which provides examples of how `DBMS_METADATA` procedures can be used.

```
SQL> CREATE OR REPLACE PACKAGE payroll_demo AS PROCEDURE get_payroll_tables;
END;
/
```

> **Note:**
>
> To see the entire script for this example, including the contents of the `PAYROLL_DEMO` package, see the file `mddemo.sql` located in your `$ORACLE_HOME/rdbms/demo` directory.

27.9.2 Output Generated from the GET_PAYROLL_TABLES Procedure

Explanation of the output generated from the `GET_PAYROLL_TABLES` procedure.

After you execute the `mddemo.payroll_demo.get_payroll_tables` procedure, you can execute the following query:

```
SQL> SELECT ddl FROM ddl ORDER BY seqno;
```

The results are as follows, which reflect all the DDL executed by the script as described in the previous section.

```
CREATE TABLE "MDDEMO"."PAYROLL_EMPS"
   (    "LASTNAME" VARCHAR2(60) NOT NULL ENABLE,
        "FIRSTNAME" VARCHAR2(20) NOT NULL ENABLE,
        "MI" VARCHAR2(2),
        "SUFFIX" VARCHAR2(10),
        "DOB" DATE NOT NULL ENABLE,
        "BADGE_NO" NUMBER(6,0),
        "EXEMPT" VARCHAR2(1) NOT NULL ENABLE,
        "SALARY" NUMBER(9,2),
        "HOURLY_RATE" NUMBER(7,2),
 PRIMARY KEY ("BADGE_NO") ENABLE
   ) ;

  GRANT UPDATE ("SALARY") ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_CLERK";
  GRANT UPDATE ("HOURLY_RATE") ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_CLERK";
  GRANT ALTER ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT DELETE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INDEX ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INSERT ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT SELECT ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT UPDATE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT REFERENCES ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT ON COMMIT REFRESH ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT QUERY REWRITE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;

  CREATE INDEX "MDDEMO"."I_PAYROLL_EMPS_DOB" ON "MDDEMO"."PAYROLL_EMPS" ("DOB")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;


  CREATE INDEX "MDDEMO"."I_PAYROLL_EMPS_NAME" ON "MDDEMO"."PAYROLL_EMPS" ("LASTNAME")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;

  CREATE OR REPLACE TRIGGER hourly_trigger before update of hourly_rate on payroll_emps
for each row
begin :new.hourly_rate:=:old.hourly_rate;end;
/
ALTER TRIGGER "MDDEMO"."HOURLY_TRIGGER" ENABLE;

  CREATE OR REPLACE TRIGGER salary_trigger before insert or update of salary on payroll_emps
for each row
WHEN (new.salary > 150000)  CALL check_sal(:new.salary)
/
ALTER TRIGGER "MDDEMO"."SALARY_TRIGGER" ENABLE;


CREATE TABLE "MDDEMO"."PAYROLL_TIMECARDS"
   (    "BADGE_NO" NUMBER(6,0),
        "WEEK" NUMBER(2,0),
        "JOB_ID" NUMBER(5,0),
        "HOURS_WORKED" NUMBER(4,2),
 FOREIGN KEY ("BADGE_NO")
  REFERENCES "MDDEMO"."PAYROLL_EMPS" ("BADGE_NO") ENABLE
   ) ;

  GRANT INSERT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_CLERK";
  GRANT UPDATE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_CLERK";
  GRANT ALTER ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT DELETE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INDEX ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INSERT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT SELECT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT UPDATE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT REFERENCES ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT ON COMMIT REFRESH ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT QUERY REWRITE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;

  CREATE INDEX "MDDEMO"."I_PAYROLL_TIMECARDS_BADGE" ON "MDDEMO"."PAYROLL_TIMECARDS" ("BADGE_NO")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;
```

27.10 Summary of DBMS_METADATA Procedures

Provides brief descriptions of the procedures provided by the `DBMS_METADATA` API.

For detailed descriptions of these procedures, see *Oracle Database PL/SQL Packages and Types Reference.*

The following table provides a brief description of the procedures provided by the `DBMS_METADATA` programmatic interface for retrieving multiple objects.

Table 27-1 DBMS_METADATA Procedures Used for Retrieving Multiple Objects

PL/SQL Procedure Name	Description
`DBMS_METADATA.OPEN()`	Specifies the type of object to be retrieved, the version of its metadata, and the object model.
`DBMS_METADATA.SET_FILTER()`	Specifies restrictions on the objects to be retrieved, for example, the object name or schema.
`DBMS_METADATA.SET_COUNT()`	Specifies the maximum number of objects to be retrieved in a single `FETCH_xxx` call.
`DBMS_METADATA.GET_QUERY()`	Returns the text of the queries that are used by `FETCH_xxx`. You can use this as a debugging aid.
`DBMS_METADATA.SET_PARSE_ITEM()`	Enables output parsing by specifying an object attribute to be parsed and returned. You can query the `DBMS_METADATA_PARSE_ITEMS` to see all valid parse items.
`DBMS_METADATA.ADD_TRANSFORM()`	Specifies a transform that `FETCH_xxx` applies to the XML representation of the retrieved objects. You can query the `DBMS_METADATA_TRANSFORMS` view to see all valid Oracle-supplied transforms.
`DBMS_METADATA.SET_TRANSFORM_PARAM()`	Specifies parameters to the XSLT stylesheet identified by `transform_handle.` You can query the `DBMS_METADATA_TRANSFORM_PARAMS` view to see all the valid transform parameters for each transform.
`DBMS_METADATA.SET_REMAP_PARAM()`	Specifies parameters to the XSLT stylesheet identified by `transform_handle`.
`DBMS_METADATA.FETCH_xxx()`	Returns metadata for objects meeting the criteria established by `OPEN`, `SET_FILTER`, `SET_COUNT`, `ADD_TRANSFORM`, and so on.
`DBMS_METADATA.CLOSE()`	Invalidates the handle returned by `OPEN` and cleans up the associated state.

The following table lists the procedures provided by the `DBMS_METADATA` browsing interface and provides a brief description of each one. These functions return metadata for one or more dependent or granted objects. These procedures do not support heterogeneous object types.

Table 27-2 DBMS_METADATA Procedures Used for the Browsing Interface

PL/SQL Procedure Name	Description
`DBMS_METADATA.GET_xxx()`	Provides a way to return metadata for a single object. Each `GET_xxx` call consists of an `OPEN` procedure, one or two `SET_FILTER` calls, optionally an `ADD_TRANSFORM` procedure, a `FETCH_xxx` call, and a `CLOSE` procedure. The *`object_type`* parameter has the same semantics as in the `OPEN procedure`. *`schema`* and *`name`* are used for filtering. If a transform is specified, then session-level transform flags are inherited.
`DBMS_METADATA.GET_DEPENDENT_xxx()`	Returns the metadata for one or more dependent objects, specified as XML or DDL.
`DBMS_METADATA.GET_GRANTED_xxx()`	Returns the metadata for one or more granted objects, specified as XML or DDL.

The following table provides a brief description of the `DBMS_METADATA` procedures and functions used for XML submission.

Table 27-3 DBMS_METADATA Procedures and Functions for Submitting XML Data

PL/SQL Name	Description
`DBMS_METADATA.OPENW()`	Opens a write context.
`DBMS_METADATA.ADD_TRANSFORM()`	Specifies a transform for the XML documents
`DBMS_METADATA.SET_TRANSFORM_PARAM() and DBMS_METADATA.SET_REMAP_PARAM()`	`SET_TRANSFORM_PARAM` specifies a parameter to a transform. `SET_REMAP_PARAM` specifies a remapping for a transform.
`DBMS_METADATA.SET_PARSE_ITEM()`	Specifies an object attribute to be parsed.
`DBMS_METADATA.CONVERT()`	Converts an XML document to DDL.
`DBMS_METADATA.PUT()`	Submits an XML document to the database.
`DBMS_METADATA.CLOSE()`	Closes the context opened with `OPENW`.

27.11 Summary of DBMS_METADATA_DIFF Procedures

Provides brief descriptions of the procedures and functions provided by the `DBMS_METADATA_DIFF` API.

For detailed descriptions of these procedures, see *Oracle Database PL/SQL Packages and Types Reference.*

Table 27-4 DBMS_METADATA_DIFF Procedures and Functions

PL/SQL Procedure Name	Description
`OPENC` function	Specifies the type of objects to be compared.
`ADD_DOCUMENT` procedure	Specifies an SXML document to be compared.
`FETCH_CLOB` functions and procedures	Returns a CLOB showing the differences between the two documents specified by `ADD_DOCUMENT`.
`CLOSE` procedure	Invalidates the handle returned by `OPENC` and cleans up associated state.