59
DBMS_CUBE

DBMS_CUBE contains subprograms that create OLAP cubes and dimensions, and that load and process the data for querying.

> **See Also:**
>
> OLAP Technology in the Oracle Database in *Oracle OLAP User’s Guide* regarding use of the OLAP option to support business intelligence and analytical applications.

This chapter contains the following topics:

- Using DBMS_CUBE
- Using SQL Aggregation Management
- Upgrading 10g Analytic Workspaces
- Summary of DBMS_CUBE Subprograms

Using DBMS_CUBE

Cubes and cube dimensions are first class data objects that support multidimensional analytics. They are stored in a container called an analytic workspace. Multidimensional objects and analytics are available with the OLAP option to Oracle Database.

Cubes can be enabled as cube materialized views for automatic refresh of the cubes and dimensions, and for query rewrite. Several DBMS_CUBE subprograms support the creation and maintenance of cube materialized views as a replacement for relational materialized views. These subprograms are discussed in "Using SQL Aggregation Management".

The metadata for cubes and dimensions is defined in XML documents, called *templates*, which you can derive from relational materialized views using the CREATE_CUBE or DERIVE_FROM_MVIEW functions. Using a graphical tool named Analytic Workspace Manager, you can enhance the cube with analytic content or create the metadata for new cubes and cube dimensions from scratch.

Several other DBMS_CUBE subprograms provide a SQL alternative to Analytic Workspace Manager for creating an analytic workspace from an XML template and for refreshing the data stored in cubes and dimensions. The IMPORT_XML procedure creates an analytic workspace with its cubes and cube dimensions from an XML template. The BUILD procedure loads data into the cubes and dimensions from their data sources and performs whatever processing steps are needed to prepare the data for querying.

DBMS_CUBE Security Model

Certain roles and system privileges are required to use the DBMS_CUBE package.

To create dimensional objects in the user's own schema:

- `OLAP_USER` role
- `CREATE SESSION` privilege

To create dimensional objects in different schemas:

- `OLAP_DBA` role
- `CREATE SESSION` privilege

To create cube materialized views in the user's own schema:

- `CREATE MATERIALIZED VIEW` privilege
- `CREATE DIMENSION` privilege
- `ADVISOR` privilege

To create cube materialized views in different schemas:

- `CREATE ANY MATERIALIZED VIEW` privilege
- `CREATE ANY DIMENSION` privilege
- `ADVISOR` privilege

If the source tables are in a different schema, then the owner of the dimensional objects needs `SELECT` object privileges on those tables.

Using SQL Aggregation Management

SQL Aggregation Management is a group of PL/SQL subprograms in `DBMS_CUBE` that supports the rapid deployment of cube materialized views from existing relational materialized views.

Cube materialized views are cubes that have been enhanced to use the automatic refresh and query rewrite features of Oracle Database. A single cube materialized view can replace many of the relational materialized views of summaries on a fact table, providing uniform response time to all summary data.

Cube materialized views bring the fast update and fast query capabilities of the OLAP option to applications that query summaries of detail relational tables. The summary data is generated and stored in a cube, and query rewrite automatically redirects queries to the cube materialized views. Applications experience excellent querying performance.

In the process of creating the cube materialized views, `DBMS_CUBE` also creates a fully functional analytic workspace including a cube and the cube dimensions. The cube stores the data for a cube materialized view instead of the table that stores the data for a relational materialized view. A cube can also support a wide range of analytic functions that enhance the database with information-rich content.

Cube materialized views are registered in the data dictionary along with all other materialized views. A `CB$` prefix identifies a cube materialized view.

The `DBMS_CUBE` subprograms also support life-cycle management of cube materialized views.

> **See Also:**
>
> Adding Materialized View Capability to a Cube in *Oracle OLAP User's Guide* for more information about cube materialized views and enhanced OLAP analytics.

Subprograms in SQL Aggregation Management

SQL Aggregation Management includes four subprograms.

- CREATE_MVIEW Function
- DERIVE_FROM_MVIEW Function
- DROP_MVIEW Procedure
- REFRESH_MVIEW Procedure

Requirements for the Relational Materialized View

SQL Aggregation Management uses an existing relational materialized view to derive all the information needed to generate a cube materialized view. The relational materialized view determines the detail level of data that is stored in the cube materialized view. The related relational dimension objects determine the scope of the aggregates, from the lowest level specified in the GROUP BY clause of the materialized view subquery, to the highest level of the dimension hierarchy.

The relational materialized view must conform to these requirements:

- Explicit `GROUP BY` clause for one or more columns.
- No expressions in the select list or `GROUP BY` clause.
- At least one of these numeric aggregation methods: `SUM`, `MIN`, `MAX`, or `AVG`.
- No outer joins.
- Summary keys with at least one simple column associated with a relational dimension.

  or

  Summary keys with at least one simple column and no hierarchies or levels.
- Numeric datatype of any type for the fact columns. All facts are converted to `NUMBER`.
- Eligible for rewrite. `REWRITE_CAPABILITY` should be `GENERAL`; it cannot be `NONE`. Refer to the `ALL_MVIEWS` entry in the *Oracle Database Reference*.
- Cannot use the `DISTINCT` or `UNIQUE` keywords with an aggregate function in the defining query. For example, `AVG(DISTINCT units)` causes an error in `STRICT` mode and is ignored in `LOOSE` mode.

You can choose between two modes when rendering the cube materialized view, `LOOSE` and `STRICT`. In `STRICT` mode, any deviation from the requirements raises an exception and prevents the materialized view from being created. In `LOOSE` mode (the default), some deviations are allowed, but they affect the content of the materialized view. These elements in the relational materialized view generate warning messages:

- Complex expressions in the defining query are ignored and do not appear in the cube materialized view.

- The AVG function is changed to SUM and COUNT.
- The COUNT function without a SUM, MIN, MAX, or AVG function is ignored.
- The STDDEV and VARIANCE functions are ignored.

You can also choose how conditions in the WHERE clause are filtered. When filtering is turned off, the conditions are ignored. When turned on, valid conditions are rendered in the cube materialized view, but asymmetric conditions among dimension levels raise an exception.

Permissions for Managing and Querying Cube Materialized Views

Certain permissions are required to manage and query cube materialized views.

To create cube materialized views, you must have these privileges:

- CREATE [ANY] MATERIALIZED VIEW privilege
- CREATE [ANY] DIMENSION privilege
- ADVISOR privilege

To access cube materialized views from another schema using query rewrite, you must have these privileges:

- GLOBAL QUERY REWRITE privilege
- SELECT or READ privilege on the relational source tables
- SELECT or READ privilege on the analytic workspace (AW$*name*) that supports the cube materialized view
- SELECT or READ privilege on the cube
- SELECT or READ privilege on the dimensions of the cube

Note that you need SELECT or READ privileges on the database objects that *support* the cube materialized views, but not on the cube materialized views.

Example of SQL Aggregation Management

Six examples of SQL Aggregate Management are given. All these examples use the sample Sales History schema, which is installed in Oracle Database with two relational materialized views: CAL_MONTH_SALES_MV and FWEEK_PSCAT_SALES_MV.

- About Relational Materialized View CAL_MONTH_SALES_MV
- Creating the Cube Materialized View
- Disabling the Relational Materialized Views
- Creating Execution Plans for Cube Materialized Views
- Maintaining Cube Materialized Views
- New Database Objects

About Relational Materialized View CAL_MONTH_SALES_MV

This example uses CAL_MONTH_SALES_MV as the basis for creating a cube materialized view. The following query was used to create CAL_MONTH_SALES_MV. CAL_MONTH_SALES_MV summarizes the daily sales data stored in the SALES table by month.

```
SELECT query FROM user_mviews
     WHERE mview_name='CAL_MONTH_SALES_MV';

QUERY
--------------------------------------------
SELECT   t.calendar_month_desc
  ,        sum(s.amount_sold) AS dollars
  FROM     sales s
  ,        times t
  WHERE    s.time_id = t.time_id
  GROUP BY t.calendar_month_desc
```

DBMS_CUBE uses relational dimensions to derive levels and hierarchies for the cube materialized view. The SH schema has relational dimensions for most dimension tables in the schema, as shown by the following query.

```
SELECT dimension_name FROM user_dimensions;

DIMENSION_NAME
-------------------------------------
CUSTOMERS_DIM
PRODUCTS_DIM
TIMES_DIM
CHANNELS_DIM
PROMOTIONS_DIM
```

Creating the Cube Materialized View

This PL/SQL script uses the CREATE_MVIEW function to create a cube materialized view from CAL_MONTH_SALES_MV. CREATE_MVIEW sets the optional BUILD parameter to refresh the cube materialized view immediately.

```
SET serverout ON format wrapped

DECLARE
     salesaw  varchar2(30);

BEGIN
     salesaw := dbms_cube.create_mview('SH', 'CAL_MONTH_SALES_MV',
                'build=immediate');
END;
/
```

These messages confirm that the script created and refreshed CB$CAL_MONTH_SALES successfully:

```
Completed refresh of cube mview "SH"."CB$CAL_MONTH_SALES" at 20130212 08:42:58.0
03.
Created cube organized materialized view "CB$CAL_MONTH_SALES" for rewrite at 200
130212 08:42:58.004.
```

The following query lists the materialized views in the SH schema:

```
SELECT mview_name FROM user_mviews;

MVIEW_NAME
------------------------------
CB$CAL_MONTH_SALES
CB$TIMES_DIM_D1_CAL_ROLLUP
CAL_MONTH_SALES_MV
FWEEK_PSCAT_SALES_MV
```

Two new materialized views are registered in the data dictionary:

- CB$CAL_MONTH_SALES: Cube materialized view
- CB$TIMES_DIM_D1_CAL_ROLLUP: Cube dimension materialized view for the TIME_DIM Calendar Rollup hierarchy

Cube dimension materialized views support refresh of the cube materialized view. You do not directly administer dimension materialized views.

Disabling the Relational Materialized Views

After creating a cube materialized view, disable query rewrite on all relational materialized views for the facts now supported by the cube materialized view. You can drop them when you are sure that you created the cube materialized view with the optimal parameters.

```
ALTER MATERIALIZED VIEW cal_month_sales_mv DISABLE QUERY REWRITE;

Materialized view altered.
```

You can also use the DISABLEQRW parameter in the CREATE_MVIEW function, which disables query rewrite on the source materialized view as described in Table 59-7.

Creating Execution Plans for Cube Materialized Views

You can create execution plans for cube materialized views the same as for relational materialized views. The following command generates an execution plan for a query against the SALES table, which contains data at the day level. The answer set requires data summarized by quarter. Query rewrite would not use the original relational materialized view for this query, because its data is summarized by month. However, query rewrite can use the new cube materialized view for summary data for months, quarters, years, and all years.

```
EXPLAIN PLAN FOR SELECT
           t.calendar_quarter_desc,
           sum(s.amount_sold) AS dollars
  FROM     sales s,
           times t
  WHERE    s.time_id = t.time_id
  AND      t.calendar_quarter_desc LIKE '2001%'
  GROUP BY t.calendar_quarter_desc
  ORDER BY t.calendar_quarter_desc;
```

The query returns these results:

```
CALENDAR_QUARTER_DESC    DOLLARS
--------------------- ----------
2001-01                6547097.44
2001-02                6922468.39
2001-03                7195998.63
2001-04                7470897.52
```

The execution plan shows that query rewrite returned the summary data from the cube materialized view, CB$CAL_MONTH_SALES, instead of recalculating it from the SALES table.

```
SELECT plan_table_output FROM TABLE(dbms_xplan.display());

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------------------------------
Plan hash value: 2999729407

-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT              |                    |     1 |    30 |     3  (34)| 00:00:01 |
|   1 |  SORT GROUP BY                |                    |     1 |    30 |     3  (34)| 00:00:01 |
|*  2 |   MAT_VIEW REWRITE CUBE ACCESS | CB$CAL_MONTH_SALES |     1 |    30 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CB$CAL_MONTH_SALES"."D1_CALENDAR_QUARTER_DESC" LIKE '2001%' AND
              "CB$CAL_MONTH_SALES"."SYS_GID"=63)

15 rows selected.
```

Maintaining Cube Materialized Views

You can create a cube materialized view that refreshes automatically. However, you can force a refresh at any time using the REFRESH_MVIEW Procedure:

```
BEGIN
     dbms_cube.refresh_mview('SH', 'CB$CAL_MONTH_SALES');
END;
/

Completed refresh of cube mview "SH"."CB$CAL_MONTH_SALES" at 20130212
14:30:59.534.
```

If you want to drop a cube materialized view, use the DROP_MVIEW Procedure so that all supporting database objects (analytic workspace, cube, cube dimensions, and so forth) are also dropped:

```
BEGIN
     dbms_cube.drop_mview('SH', 'CB$CAL_MONTH_SALES');
END;
/

Dropped cube organized materialized view "SH"."CAL_MONTH_SALES" including
container analytic workspace "SH"."CAL_MONTH_SALES_AW" at 20130212 13:38:47.878.
```

New Database Objects

The CREATE_MVIEW function creates several first class database objects in addition to the cube materialized views. You can explore these objects through the data dictionary by querying views such as ALL_CUBES and ALL_CUBE_DIMENSIONS.

This example created the following supporting objects:

- Analytic workspace CAL_MONTH_SALES_AW (AW$CAL_MONTH_SALES_AW table)
- Cube CAL_MONTH_SALES
- Cube dimension TIMES_DIM_D1
- Dimension hierarchy CAL_ROLLUP
- Dimension levels ALL_TIMES_DIM, YEAR, QUARTER, and MONTH
- Numerous attributes for levels in the CAL_ROLLUP hierarchy

Upgrading 10*g* Analytic Workspaces

You can upgrade an OLAP 10*g* analytic workspace to OLAP 12*c* by saving the OLAP 10*g* objects as an XML template and importing the XML into a different schema. The original analytic workspace remains accessible and unchanged by the upgrade process.

Oracle OLAP metadata is the same in OLAP 11*g* and OLAP 12*c* so you do not need to upgrade an OLAP 11*g* analytic workspace to OLAP 12*c*. This topic describes upgrading an Oracle OLAP 10*g* analytic workspace to OLAP 12*c*.

> **Tip:**
>
> Oracle recommends using Analytic Workspace Manager for performing upgrades. See Upgrading Metadata From Oracle OLAP 10*g* in *Oracle OLAP User’s Guide*.

These subprograms in `DBMS_CUBE` support the upgrade process:

- CREATE_EXPORT_OPTIONS Procedure
- CREATE_IMPORT_OPTIONS Procedure
- EXPORT_XML Procedure
- EXPORT_XML_TO_FILE Procedure
- IMPORT_XML Procedure
- INITIALIZE_CUBE_UPGRADE Procedure
- UPGRADE_AW Procedure

Prerequisites:

- The OLAP 10*g* analytic workspace can use OLAP standard form metadata.
- Customizations to the OLAP 10*g* analytic workspace may not be exported to the XML template. You must re-create them in OLAP 12*c*.
- The original relational source data must be available to load into the new analytic workspace. If the data is in a different schema or the table names are different, then you must remap the dimensional objects to the new relational sources after the upgrade.
- You can create the OLAP 12*c* analytic workspace in the same schema as the OLAP 10*g* analytic workspace. However, if you prefer to create it in a different schema, then create a new user with the following privileges:
  - `SELECT` or `READ` privileges on the OLAP 10*g* analytic workspace (`GRANT SELECT ON` *`schema`*`.AW$`*`analytic_workspace`*).
  - `SELECT` or `READ` privileges on all database tables and views that contain the source data for the OLAP 10*g* analytic workspace.
  - Appropriate privileges for an OLAP administrator.
  - Same default tablespace as the Oracle 10*g* user.

  See the *Oracle OLAP User's Guide*.

Correcting Naming Conflicts

The namespaces are different in OLAP 10*g* than those in OLAP 12*c*. For a successful upgrade, you must identify any 10*g* object names that are used multiple times under the 12*c* naming rules and provide unique names for them.

The following namespaces control the uniqueness of OLAP object names in Oracle 12*c*:

- **Schema**: The names of cubes, dimensions, and measure folders must be unique within a schema. They cannot conflict with the names of tables, views, indexes, relational dimensions, or any other first class objects. However, these OLAP 12*c* object names do not need to be distinct from 10*g* object names, because they are in different namespaces.
- **Cube**: The names of measures must be unique within a cube.
- **Dimension**: The names of hierarchies, levels, and attributes must be unique within a dimension. For example, a dimension cannot have a hierarchy named Customers and a level named Customers.

You can use an initialization table and a rename table to rename objects in the upgraded 12*c* analytic workspace.

Initialization Table

The `INITIALIZE_CUBE_UPGRADE` procedure identifies ambiguous names under the OLAP 12*c* naming rules. For example, a 10*g* dimension might have a hierarchy and a level with the same name. Because hierarchies and levels are in the same 12*c* namespace, the name is not unique in 12*c*; to a 12*c* client, the hierarchy and the level cannot be differentiated by name.

`INITIALIZE_CUBE_UPGRADE` creates and populates a table named `CUBE_UPGRADE_INFO` with unique names for these levels, hierarchies, and attributes. By using the unique names provided in the table, a 12*c* client can browse the OLAP 12*c* metadata. You cannot attach an OLAP 12*c* client to the analytic workspace or perform an upgrade without a `CUBE_UPGRADE_INFO` table, if the 10*g* metadata contains ambiguous names.

You can edit `CUBE_UPGRADE_INFO` to change the default unique names to names of your choosing. You can also add rows to change the names of any other objects. When using an 12*c* client, you see the new object names. When using an 10*g* client, you see the original names. However, the `INITIALIZE_CUBE_UPGRADE` procedure overwrites this table, so you may prefer to enter customizations in a rename table.

During an upgrade from OLAP 10*g,* the unique object names in `CUBE_UPGRADE_INFO` are used as the names of 12*c* objects in the new analytic workspace. However, `INITIALIZE_CUBE_UPGRADE` does not automatically provide unique names for cubes, dimensions, and measure folders. To complete an upgrade, you must assure that these objects have unique names within the 12*c* namespace. You can provide these objects with new names in the `CUBE_UPGRADE_INFO` table or in a rename table.

OLAP 12*c* clients automatically use `CUBE_UPGRADE_INFO` when it exists in the same schema as the OLAP 10*g* analytic workspace.

> **See Also:**
>
> "INITIALIZE_CUBE_UPGRADE Procedure"

Rename Table

You can create a rename table that contains new object names for an OLAP 12*c* analytic workspace. You can then use the rename table in the `CREATE_IMPORT_OPTIONS` and `UPGRADE_AW` procedures.

When upgrading within the same schema, you must provide a unique name for the 12*c* analytic workspace. The `UPGRADE_AW` procedure provides a parameter for this purpose; otherwise, you must provide the new name in the rename table. The duplication of cube names does not create ambiguity because the 12*c* cubes are created in a different namespace than the 10*g* cubes.

The names provided in a rename table are used only during an upgrade and overwrite any names entered in the `CUBE_UPGRADE_INFO` table.

To create a rename table:

1. Open SQL*Plus or another SQL client, and connect to Oracle Database as the owner of the 10*g* analytic workspace.
2. Issue a command like the following:

```
CREATE TABLE table_name (
          source_id     VARCHAR2(300),
          new_name      VARCHAR2(30),
          object_type   VARCHAR2(30));
```

3. Populate the rename table with the appropriate values, as follows.

`table_name` is the name of the rename table.

`source_id` is the identifier for an object described in the XML document supplied to IMPORT_XML. The identifier must have this format:

schema_name.*object_name*[.*subobject_name*]

`new_name` is the object name given during the import to the object specified by `source_id`.

`object_type` is the object type as described in the XML, such as StandardDimension or DerivedMeasure.

For example, these SQL statements populate the table with new names for the analytic workspace, a cube, and four dimensions:

```
INSERT INTO my_object_map VALUES('GLOBAL_AW.GLOBAL10.AW', 'GLOBAL12', 'AW');
INSERT INTO my_object_map VALUES('GLOBAL_AW.UNITS_CUBE', 'UNIT_SALES_CUBE', 'Cube');
INSERT INTO my_object_map VALUES('GLOBAL_AW.CUSTOMER', 'CUSTOMERS', 'StandardDimension');
INSERT INTO my_object_map VALUES('GLOBAL_AW.CHANNEL', 'CHANNELS', 'StandardDimension');
INSERT INTO my_object_map VALUES('GLOBAL_AW.PRODUCT', 'PRODUCTS', 'StandardDimension');
INSERT INTO my_object_map VALUES('GLOBAL_AW.TIME', 'TIME_PERIODS', 'TimeDimension');
```

> **See Also:**
>
> "CREATE_IMPORT_OPTIONS Procedure"

Simple Upgrade

A simple upgrade creates an OLAP 12*c* analytic workspace from an OLAP 10*g* analytic workspace.

To perform a simple upgrade of an Oracle OLAP 10*g* analytic workspace:

1. Open SQL*Plus or a similar SQL command-line interface and connect to Oracle Database 12*c* as the schema owner for the OLAP 12*c* analytic workspace.
2. To rename any objects in the 12*c* analytic workspace, create a rename table as described in the Rename Table section. (Optional)
3. Perform the upgrade, as described in "UPGRADE_AW Procedure".
4. Use the `DBMS_CUBE.BUILD` procedure to load data into the cube.

Example 59-1 Performing a Simple Upgrade to the GLOBAL Analytic Workspace

This example creates an OLAP 12*c* analytic workspace named `GLOBAL12` from an OLAP 10*g* analytic workspace named `GLOBAL10`. `GLOBAL10` contains no naming conflicts between cubes, dimensions, measure folders, or tables in the schema, so a rename table is not needed in this example.

```
BEGIN

  -- Upgrade the analytic workspace
  dbms_cube.upgrade_aw(sourceaw =>'GLOBAL10', destaw => 'GLOBAL12');

  -- Load and aggregate the data
  dbms_cube.build(script => 'UNITS_CUBE, PRICE_AND_COST_CUBE');

END;
/
```

Custom Upgrade

A custom upgrade enables you to set the export and import options.

To perform a custom upgrade of an Oracle OLAP 10*g* analytic workspace:

1. Open SQL*Plus or a similar SQL command-line interface and connect to Oracle Database 12*c* as the schema owner of the OLAP 12*c* analytic workspace.
2. Generate an initialization table, as described in the Initialization Table section. Review the new, default object names and modify them as desired.
3. Create a rename table, as described in the Rename Table section. If you are upgrading in the same schema, you must use a rename table to provide a unique name for the 12*c* analytic workspace. Otherwise, a rename table is needed only if names are duplicated among the cubes, dimensions, and measure folders of the analytic workspace, or between those names and the existing cubes, dimensions, measure folders, or tables of the destination schema.
4. Create a SQL script that does the following:
   a. Create an XML document for the export options, as described in "CREATE_EXPORT_OPTIONS Procedure". The `SUPPRESS_NAMESPACE` option must be set to `TRUE` for the upgrade to occur.
   b. Create an XML document for the import options, as described in "CREATE_IMPORT_OPTIONS Procedure".

c. Create an XML template in OLAP 12*c* format, as described in "EXPORT_XML Procedure".

d. Create an OLAP 12*c* analytic workspace from the XML template, as described in "IMPORT_XML Procedure".

5. Load and aggregate the data in the new analytic workspace, as described in "BUILD Procedure".

Example 59-2 Performing a Custom Upgrade to the GLOBAL Analytic Workspace

This example upgrades the `GLOBAL10` analytic workspace from OLAP 10*g* metadata to OLAP 12*c* metadata in the `GLOBAL_AW` schema.

The rename table provides the new name of the analytic workspace. These commands define the rename table.

```
CREATE TABLE my_object_map(
       source_id    VARCHAR2(300),
       new_name     VARCHAR2(30),
       object_type  VARCHAR2(30));

INSERT INTO my_object_map VALUES('GLOBAL_AW.GLOBAL10.AW',  'GLOBAL12', 'AW');
COMMIT;
```

Following is the script for performing the upgrade.

```
set serverout on

DECLARE
  importClob    clob;
  exportClob    clob;
  exportOptClob clob;
  importOptClob clob;

BEGIN

  -- Create table of reconciled names
  dbms_cube.initialize_cube_upgrade;

  -- Create a CLOB containing the export options
  dbms_lob.createtemporary(exportOptClob, TRUE);
  dbms_cube.create_export_options(out_options_xml=>exportOptClob,
suppress_namespace=>TRUE, preserve_table_owners=>TRUE);

  -- Create a CLOB containing the import options
  dbms_lob.createtemporary(importOptClob, TRUE);
  dbms_cube.create_import_options(out_options_xml=>importOptClob, rename_table =>
'MY_OBJECT_MAP');

   -- Create CLOBs for the metadata
  dbms_lob.createtemporary(importClob, TRUE);
  dbms_lob.createtemporary(exportClob, TRUE);

  -- Export metadata from a 10g analytic workspace to a CLOB
  dbms_cube.export_xml(object_ids=>'GLOBAL_AW', options_xml=>exportOptClob,
out_xml=>exportClob);

  -- Import metadata from the CLOB
  dbms_cube.import_xml(in_xml => exportClob, options_xml=>importOptClob,
out_xml=>importClob);

  -- Load and aggregate the data
  dbms_cube.build('UNITS_CUBE, PRICE_AND_COST_CUBE');

END;
/
```

Summary of DBMS_CUBE Subprograms

This table lists and describes the `DBMS_CUBE` procedure subprograms.

Table 59-1 DBMS_CUBE Subprograms

Subprogram	Description
BUILD Procedure	Loads data into one or more cubes and dimensions, and prepares the data for querying.
CREATE_EXPORT_OPTIONS Procedure	Creates an input XML document of processing options for the EXPORT_XML procedure.
CREATE_IMPORT_OPTIONS Procedure	Creates an input XML document of processing options for the IMPORT_XML procedure.
CREATE_MVIEW Function	Creates a cube materialized view from the definition of a relational materialized view.
DERIVE_FROM_MVIEW Function	Creates an XML template for a cube materialized view from the definition of a relational materialized view.
DROP_MVIEW Procedure	Drops a cube materialized view.
EXPORT_XML Procedure	Exports the XML of an analytic workspace to a CLOB.
EXPORT_XML_TO_FILE Procedure	Exports the XML of an analytic workspace to a file.
IMPORT_XML Procedure	Creates, modifies, or drops an analytic workspace by using an XML template
INITIALIZE_CUBE_UPGRADE Procedure	Processes Oracle OLAP 10*g* objects with naming conflicts to enable Oracle 12*c* clients to access them.
REFRESH_MVIEW Procedure	Refreshes a cube materialized view.
UPGRADE_AW Procedure	Upgrades an analytic workspace from Oracle OLAP 10*g* to 12*c*.
VALIDATE_XML Procedure	Checks the XML to assure that it is valid, without committing the results to the database.

BUILD Procedure

This procedure loads data into one or more cubes and dimensions, and generates aggregate values in the cubes. The results are automatically committed to the database.

Syntax

```
DBMS_CUBE.BUILD (
       script                IN  VARCHAR2,
       method                IN  VARCHAR2        DEFAULT NULL,
       refresh_after_errors  IN  BOOLEAN         DEFAULT FALSE,
       parallelism           IN  BINARY_INTEGER  DEFAULT 0,
       atomic_refresh        IN  BOOLEAN         DEFAULT FALSE,
       automatic_order       IN  BOOLEAN         DEFAULT TRUE,
       add_dimensions        IN  BOOLEAN         DEFAULT TRUE,
       scheduler_job         IN  VARCHAR2        DEFAULT NULL,
       master_build_id       IN  BINARY_INTEGER  DEFAULT 0,
       nested                IN   BOOLEAN         DEFAULT FALSE);
       job_class             IN   VARCHAR2        DEFAULT 'DEFAULT_JOB_CLASS'
```

Parameters

Table 59-2   BUILD Procedure Parameters

Parameter	Description
script	A list of cubes and dimensions and their build options (see "SCRIPT Parameter").
method	A full or a fast (partial) refresh. In a fast refresh, only changed rows are inserted in the cube and the affected areas of the cube are re-aggregated. You can specify a method for each cube and dimension in sequential order, or a single method to apply to all cubes and dimensions. If you list more objects than methods, then the last method applies to the additional objects. • C: Complete refresh clears all dimension values before loading. (Default) • F: Fast refresh of a cube materialized view, which performs an incremental refresh and re-aggregation of only changed rows in the source table. • ?: Fast refresh if possible, and otherwise a complete refresh. • P: Recomputes rows in a cube materialized view that are affected by changed partitions in the detail tables. • S: Fast solve of a compressed cube. A fast solve reloads all the detail data and re-aggregates only the changed values. See the "Usage Notes" for additional details. Methods do not apply to dimensions.
refresh_after_errors	TRUE to roll back just the cube or dimension with errors, and then continue building the other objects. FALSE to roll back all objects in the build.
parallelism	Number of parallel processes to allocate to this job (see Usage Notes).
atomic_refresh	TRUE prevents users from accessing intermediate results during a build. It freezes the current state of an analytic workspace at the beginning of the build to provide current sessions with consistent data. This option thaws the analytic workspace at the end of the build to give new sessions access to the refreshed data. If an error occurs during the build, then all objects are rolled back to the frozen state. FALSE enables users to access intermediate results during an build.
automatic_order	TRUE enables optimization of the build order. Dimensions are loaded before cubes. FALSE builds objects in the order you list them in the script.
add_dimensions	TRUE automatically includes all the dimensions of the cubes in the build, whether or not you list them in the script. If a cube materialized view with a particular dimension is fresh, then that dimension is not reloaded. You can list a cube once in the script. FALSE includes only dimensions specifically listed in the script.
scheduler_job	Any text identifier for the job, which will appear in the log table. The string does not need to be unique.
master_build_id	A unique name for the build.

Table 59-2 (Cont.) BUILD Procedure Parameters

Parameter	Description
`nested`	`TRUE` performs nested refresh operations for the specified set of cube materialized views. Nested refresh operations refresh all the depending materialized views and the specified set of materialized views based on a dependency order to ensure the nested materialized views are truly fresh with respect to the underlying base tables. All objects must reside in a single analytic workspace.
`job_class`	The class this job is associated with.

SCRIPT Parameter

The `SCRIPT` parameter identifies the objects to include in the build, and specifies the type of processing to perform on each one. The parameter has this syntax:

```
[VALIDATE | NO COMMIT] objects [ USING ( commands ) ][,...]
```

Where:

`VALIDATE` checks all steps of the build and sends the planned steps to `CUBE_BUILD_LOG` without executing the steps. You can view all generated SQL in the `OUTPUT` column of the log table.

`NO COMMIT` builds the objects in the current attach mode (or Read Only when the analytic workspace is not attached) but does not commit the changes. This option supports what-if analysis, since it enables you to change data values temporarily. See "SCRIPT Parameter: USING Clause: SET command".

`objects` is the qualified name of one or more cubes or dimensions, separated by commas, in the form *`[aw_name.]object`*, such as `UNITS_CUBE` or `GLOBAL.UNITS_CUBE`.

SCRIPT Parameter: USING Clause

The `USING` clause specifies the processing options. It consists of one or more commands separated by commas.

> **Note:**
>
> A cube with a rewrite materialized view cannot have a `USING` clause, except for the `ANALYZE` command. It uses the default build options.

The `commands` can be any of the following.

- `AGGREGATE USING [MEASURE]`

  Generates aggregate values using the syntax described in "SCRIPT Parameter: USING Clause: AGGREGATE command".

- `ANALYZE`

  Runs `DBMS_AW_STATS.ANALYZE`, which generates and stores optimizer statistics for cubes and dimensions.

- `CLEAR [VALUES | LEAVES | AGGREGATES] [SERIAL | PARALLEL]`

Prepares the cube for a data refresh. It can also be used on dimensions, but `CLEAR` removes all dimension keys, and thus deletes all data values for cubes that use the dimension.

These optional arguments control the refresh method. If you omit the argument, then the behavior of `CLEAR` depends on the refresh method. The `'C'` (complete) refresh method runs `CLEAR VALUES`, and all other refresh methods run `CLEAR LEAVES`.

– `VALUES`: Clears all data in the cube. All facts must be reloaded and all aggregates must be recomputed. This option supports the `COMPLETE` refresh method. (Default for the `C` and `F` methods)

– `LEAVES`: Clears the detail data and retains the aggregates. All facts must be reloaded, and the aggregates for any new or changed facts must be computed. This option supports the `FAST` refresh method. (Default for the `?` method)

– `AGGREGATES`: Retains the detail data and clears the aggregates. All aggregates must be recomputed.

These optional arguments control the load method, and can be combined with any of the refresh options:

– `PARALLEL`: Each partition is cleared separately. (Default)

– `SERIAL`: All partitions are cleared together.

If you omit the `CLEAR` command, `DBMS_CUBE` loads new and updated facts, but does not delete any old detail data. This is equivalent to a `LOAD NO SYNC` for dimensions.

- `COMPILE [SORT | NO SORT | SORT ONLY]`

  Creates the supporting structures for the dimension. (Dimensions only)

  These options control the use of a sort order attribute:

  – `SORT`: The user-defined sort order attribute populates the sort column in the embedded-total (ET) view. (Default)

  – `NO SORT`: Any sort order attribute is ignored. This option is for very large dimensions where sorting could consume too many resources.

  – `SORT ONLY`: The compile step only runs the sort.

- `EXECUTE PLSQL string`

  Executes a PL/SQL command or script in the database.

- `EXECUTE OLAP DML string [PARALLEL | SERIAL]`

  Executes an OLAP DML command or program in the analytic workspace. The options control execution of the command or program:

  – `PARALLEL`: Execute the command or program once for each partition. This option can be used to provide a performance boost to complex DML operations, such as forecasts and models.

  – `SERIAL`: Execute the command or program once for the entire cube. (Default)

- `[INSERT | MERGE] INTO [ALL HIERARCHIES | HIERARCHIES (`*`dimension.hierarchy`*`)] VALUES (`*`dim_key, parent, level_name`*`)`

  Adds a dimension member to one or more hierarchies. `INSERT` throws an error if the member already exists, while `MERGE` does not. See "Dimension Maintenance Example".

  `dimension.hierarchy`: The name of a hierarchy the new member belongs to. Enclose each part of the name in double quotes, for example, `"PRODUCT"."PRIMARY"`.

dim_key: The DIM_KEY value of the dimension member.

parent: The parent of the dimension key.

level_name: The level of the dimension key.

- UPDATE [ALL HIERARCHIES | HIERARCHIES (*dimension.hierarchy*)] SET PARENT = *parent*, LEVEL=*level_name* WHERE MEMBER = *dim_key*

  Alters the level or parent of an existing dimension member. See INSERT for a description of the options. Also see "Dimension Maintenance Example".

- DELETE FROM DIMENSION WHERE MEMBER=*dim_key*

  Deletes a dimension member. See "Dimension Maintenance Example".

  dim_key: The DIM_KEY value of the dimension member to be deleted.

- SET *dimension.attribute[qdr]* = CAST('*attribute_value*' AS VARCHAR2))

  Sets the value of an attribute for a dimension member. See "Dimension Maintenance Example".

  dimension.attribute: The name of the attribute. Enclose each part of the name in double quotes, for example, "PRODUCT"."LONG_DESCRIPTION".

  qdr: The dimension member being given an attribute value in the form of a qualified data reference, such as "PRODUCT"='OPT MOUSE'.

  attribute_value: The value of the attribute, such as 'Optical Mouse'.

- FOR *dimension_clause measure_clause* BUILD (*commands*)

  Restricts the build to particular measures and dimension values, using the following arguments. See "FOR Clause Example".

  - *dimension_clause*:

    dimension ALL | NONE | WHERE *condition* | LEVELS (*level* [, *level*...])

    dimension is the name of a dimension of the cube.

    ALL sets the dimension status to all members before executing the list of commands.

    NONE loads values for no dimension members.

    WHERE loads values for those dimension members that match the condition.

    LEVELS loads values for dimension members in the named levels.

    level is a level of the named dimension.

  - *measure_clause*:

    MEASURES (*measure* [, *measure*...])

    measure is the name of a measure in the cube.

  - *commands*: Any of the other USING commands.

- LOAD [SYNCH | NO SYNCH | RETAIN] [PRUNE | PARALLEL | SERIAL] [WHERE *condition*]

  Loads data into the dimension or cube.

  - WHERE limits the load to those values in the mapped relational table that match *condition*.

  - *condition* is a valid predicate based on the columns of the mapped table. See the "Examples".

  These optional arguments apply only to dimensions:

  - `SYNCH` matches the dimension keys to the relational data source. (Default)
  - `NO SYNCH` loads new dimension keys but does not delete old keys.

    If the parent of a dimension key has changed in the relational data source, this option allows the load to change the parent/child relation in the analytic workspace.
  - `RETAIN` loads new dimension keys but does not delete old keys.

    This option does not allow the parent of a dimension key to change. If the parent has changed, the load rejects the record. The rejection generates an error in the rejected records log, if the log is enabled.

  These optional arguments apply only to cubes:

  - `PRUNE`: Runs a full table scan on the fact table to determine which partitions to load. For example, if a cube is partitioned by month and the fact table has values only for the last two months, then jobs are only started to load the partitions for the last two months.
  - `PARALLEL`: Each partition is loaded separately. (Default)
  - `SERIAL`: All partitions are loaded in one `SELECT` statement.

- `MODEL model_name [PARALLEL | SERIAL]`

  Executes a model previously created for the cube. It accepts these arguments:

  - `PARALLEL`: The model runs separately on each partition.
  - `SERIAL`: The model runs on all cubes at the same time. (Default)

- `SET`

  Supports write-back to the cube using the syntax described in "SCRIPT Parameter: USING Clause: SET command". (Cubes only)

- `SOLVE [PARALLEL | SERIAL]`

  Aggregates the cube using the rules defined for the cube, including the aggregation operator and the precompute specifications. (Cubes only)

  It accepts these arguments:

  - `PARALLEL`: Each partition is solved separately. (Default)
  - `SERIAL`: All partitions are solved at the same time.

SCRIPT Parameter: USING Clause: AGGREGATE command

The `AGGREGATE` command in a script specifies the aggregation rules for one or more measures.

> **Note:**
>
> The `AGGREGATE` command is available only for uncompressed cubes.

`AGGREGATE` has the following syntax:

```
{ AGGREGATE USING MEASURE
     WHEN measure1 THEN operator1
     WHEN measure2 THEN operator2...
```

```
         ELSE default_operator
|
 [AGGREGATE USING] operator_clause }
processing_options
OVER { ALL | dimension | dimension HIERARCHIES (hierarchy)}
```

USING MEASURE Clause

This clause enables you to specify different aggregation operators for different measures in the cube.

Operator Clause

The `operator_clause` has this syntax:

```
operator(WEIGHTBY expression | SCALEBY expression)
```

`WEIGHTBY` multiplies each data value by an expression before aggregation.

`SCALEBY` adds the value of an expression to each data value before aggregation.

Table 59-3 Aggregation Operators

Operator	Option	Description
`AVG`	`WEIGHTBY`	Adds data values, then divides the sum by the number of data values that were added together.
`FIRST`	`WEIGHTBY`	The first real data value.
`HIER_AVG`	`WEIGHTBY`	Adds data values, then divides the sum by the number of the children in the dimension hierarchy. Unlike AVERAGE, which counts only non-NA children, HAVERAGE counts all of the logical children of a parent, regardless of whether each child does or does not have a value.
`HIER_FIRST`	`WEIGHTBY`	The first data value in the hierarchy, even when that value is NA.
`HIER_LAST`	`WEIGHTBY`	The last data value in the hierarchy, even when that value is NA.
`LAST`	`WEIGHTBY`	The last real data value.
`MAX`	`WEIGHTBY`	The largest data value among the children of each parent.
`MIN`	`WEIGHTBY`	The smallest data value among the children of each parent.
`NO AGGREGATION`	No option	Do not aggregate the values of the dimension or dimensions. Leave all aggregated values as NA.
`SUM`	`SCALEBY | WEIGHTBY`	Adds data values. (Default)

Processing Options

You can specify these processing options for aggregation:

- `(ALLOW | DISALLOW) OVERFLOW`

  Specifies whether to allow decimal overflow, which occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the numerical representation.

  - `ALLOW`: A calculation that generates overflow executes without error and produces null results. (Default)

  - `DISALLOW`: A calculation involving overflow stops executing and generates an error message.

- `(ALLOW | DISALLOW) DIVISION BY ZERO`

  Specifies whether to allow division by zero.

  - `ALLOW`: A calculation involving division by zero executes without error but returns a null value. (Default)
  - `DISALLOW`: A calculation involving division by zero stops executing and generates an error message.

- `(CONSIDER | IGNORE) NULLS`

  Specifies whether nulls are included in the calculations.

  - `CONSIDER`: Nulls are included in the calculations. A calculation that includes a null value returns a null value.
  - `IGNORE`: Only actual data values are used in calculations. Nulls are treated as if they do not exist. (Default)

- `MAINTAIN COUNT`

  Stores an up-to-date count of the number of dimension members for use in calculating averages. Omit this option to count the members on the fly.

SCRIPT Parameter: USING Clause: SET command

The `SET` command in a script assigns values to one or more cells in a stored measure. It has this syntax:

```
SET target = expression
```

Where:

`target` is a a measure or a qualified data reference.

`expression` returns values of the appropriate datatype for *`target`*.

Qualified Data References

Qualified data references (QDRs) limit a dimensional object to a single member in one or more dimensions for the duration of a query.

A QDR has the following syntax:

```
expression [ { dimension = member }[ , { dimension = member } ...] ]
```

Where:

`expression` is a dimensional expression, typically the name of a measure.

`dimension` is a primary dimension of expression.

`member` is a value of dimension.

The outside square brackets shown in bold are literal syntax elements; they do not indicate an optional argument. The inside square brackets shown in regular text delimit an optional argument and are not syntax elements.

This example returns Sales values for calendar year 2007:

```
global.sales[global.time = 'CY2007'
]
```

The next example returns Sales values only for the United States in calendar year 2007:

```
sales[customer = 'US', time = 'CY2007'
]
```

See the Examples for qualified data references in SET commands.

Usage Notes

Build Methods

The `C`, `S`, and `?` methods always succeed and can be used on any cube.

The `F` and `P` methods require that the cube have a materialized view that was created as a fast or a rewrite materialized view.

Parallelism

Partitioned cubes can be loaded and aggregated in parallel processes. For example, a cube with five partitions can use up to five processes. Dimensions are always loaded serially.

The number of parallel processes actually allocated by a build is controlled by the smallest of these factors:

- Number of cubes in the build and the number of partitions in each cube.
- Setting of the `PARALLELISM` argument of the `BUILD` procedure.
- Setting of the `JOB_QUEUE_PROCESSES` database initialization parameter.

Suppose `UNITS_CUBE` has 12 partitions, `PARALLELISM` is set to 10, and `JOB_QUEUE_PROCESSES` is set to 4. OLAP uses four processes, which appear as slave processes in the build log.

The SQL engine may allocate additional processes when the PARALLEL_DEGREE_POLICY database initialization parameter is set to AUTO or LIMITED. For example, if OLAP allocates four processes, the SQL engine might determine that two of those processes should be done by four processes instead, for a total of six processes.

Build Logs

OLAP generates three logs that provide diagnostic information about builds:

- Cube build log
- Rejected records log
- Cube dimension compile log

Analytic Workspace Manager creates these logs automatically as tables in the same schema as the analytic workspace. If you do not use Analytic Workspace Manager, you can create and manage the logs in PL/SQL using the `DBMS_CUBE_LOG` package.

You can also create the cube log file by running `$ORACLE_HOME/olap/admin/utlolaplog.sql`. This script creates three additional views:

- `CUBE_BUILD_LATEST`: Returns rows only from the last build.
- `CUBE_BUILD_REPORT`: Returns one row for each command with elapsed times.
- `CUBE_BUILD_REPORT_LATEST`: Returns a report like `CUBE_BUILD_REPORT` only from the last build.

This report shows a successfully completed build of the objects in the GLOBAL analytic workspace, which has four dimensions and two cubes.

```
SELECT command, status, build_object, build_object_type type
     FROM cube_build_report_latest;

COMMAND                   STATUS     BUILD_OBJECT                   TYPE
------------------------- ---------- ------------------------------ ----------
BUILD                     COMPLETED                                 BUILD
FREEZE                    COMPLETED                                 BUILD
LOAD NO SYNCH             COMPLETED  CHANNEL                        DIMENSION
COMPILE                   COMPLETED  CHANNEL                        DIMENSION
UPDATE/COMMIT             COMPLETED  CHANNEL                        DIMENSION
LOAD NO SYNCH             COMPLETED  CUSTOMER                       DIMENSION
COMPILE                   COMPLETED  CUSTOMER                       DIMENSION
UPDATE/COMMIT             COMPLETED  CUSTOMER                       DIMENSION
LOAD NO SYNCH             COMPLETED  PRODUCT                        DIMENSION
COMPILE                   COMPLETED  PRODUCT                        DIMENSION
UPDATE/COMMIT             COMPLETED  PRODUCT                        DIMENSION
LOAD NO SYNCH             COMPLETED  TIME                           DIMENSION
COMPILE                   COMPLETED  TIME                           DIMENSION
UPDATE/COMMIT             COMPLETED  TIME                           DIMENSION
COMPILE AGGMAP            COMPLETED  PRICE_CUBE                     CUBE
UPDATE/COMMIT             COMPLETED  PRICE_CUBE                     CUBE
COMPILE AGGMAP            COMPLETED  UNITS_CUBE                     CUBE
UPDATE/COMMIT             COMPLETED  UNITS_CUBE                     CUBE
DBMS_SCHEDULER.CREATE_JOB COMPLETED  PRICE_CUBE                     CUBE
DBMS_SCHEDULER.CREATE_JOB COMPLETED  UNITS_CUBE                     CUBE
BUILD                     COMPLETED                                 BUILD
LOAD                      COMPLETED  PRICE_CUBE                     CUBE
SOLVE                     COMPLETED  PRICE_CUBE                     CUBE
UPDATE/COMMIT             COMPLETED  PRICE_CUBE                     CUBE
BUILD                     COMPLETED                                 BUILD
LOAD                      COMPLETED  UNITS_CUBE                     CUBE
SOLVE                     COMPLETED  UNITS_CUBE                     CUBE
UPDATE/COMMIT             COMPLETED  UNITS_CUBE                     CUBE
ANALYZE                   COMPLETED  PRICE_CUBE                     CUBE
ANALYZE                   COMPLETED  UNITS_CUBE                     CUBE
THAW                      COMPLETED                                 BUILD

31 rows selected.
```

Examples

This example uses the default parameters to build UNITS_CUBE.

```
EXECUTE DBMS_CUBE.BUILD('GLOBAL.UNITS_CUBE');
```

The next example builds UNITS_CUBE and explicitly builds two of its dimensions, TIME and CHANNEL. The dimensions use the complete (C) method, and the cube uses the fast solve (S) method.

```
BEGIN
  DBMS_CUBE.BUILD(
    script=>'GLOBAL."TIME", GLOBAL.CHANNEL, GLOBAL.UNITS_CUBE',
    method=>'CCS',
    parallelism=>2);
END;
/
```

The following example loads only the selection of data identified by the WHERE clause:

```
BEGIN
  DBMS_CUBE.BUILD(q'!
  GLOBAL."TIME",
  GLOBAL.CHANNEL,
  GLOBAL.CUSTOMER,
  GLOBAL.PRODUCT,
  GLOBAL.UNITS_CUBE USING (LOAD NO SYNCH
     WHERE UNITS_FACT.MONTH_ID LIKE '2006%'
     AND UNITS_FACT.SALES > 5000)!');
END;
/
```

FOR Clause Example

In this example, the Time dimension is partitioned by calendar year, and DBMS_CUBE builds only the partition identified by CY2006. The HIER_ANCESTOR is an analytic function in the OLAP expression syntax.

```
BEGIN
   dbms_cube.build(q'!
   UNITS_CUBE USING
   (
   FOR "TIME"
     WHERE HIER_ANCESTOR(WITHIN "TIME".CALENDAR LEVEL "TIME".CALENDAR_YEAR) = 'CY2006'
     BUILD (LOAD, SOLVE)
   )!',
   parallelism=>1);
END;
/
```

The next example uses a FOR clause to limit the build to the SALES measure in 2006. All objects are built using the complete (C) method.

```
BEGIN
  DBMS_CUBE.BUILD(
  script => '
  GLOBAL."TIME",
  GLOBAL.CHANNEL,
  GLOBAL.CUSTOMER,
  GLOBAL.PRODUCT,
  GLOBAL.UNITS_CUBE USING
  (
    FOR MEASURES(GLOBAL.UNITS_CUBE.SALES)
      BUILD(LOAD NO SYNCH WHERE GLOBAL.UNITS_FACT.MONTH_ID LIKE ''2006%'')
   )',
  method => 'C',
  parallelism => 2);
END;
/
```

Write-Back Examples

The following examples show various use of the SET command in a USING clause.

This example sets Sales Target to Sales increased by 5%:

```
DBMS_CUBE.BUILD('UNITS_CUBE USING(
   SET UNITS_CUBE.SALES_TARGET = UNITS_CUBE.SALES * 1.05, SOLVE)');
```

This example sets the price of the Deluxe Mouse in May 2007 to $29.99:

```
DBMS_CUBE.BUILD('PRICE_CUBE USING(
  SET PRICE_CUBE.UNIT_PRICE["TIME"=''2007.05'', "PRODUCT"=''DLX MOUSE'']
  = 29.99, SOLVE)');
```

The next example contains two SET commands, but does not reaggregate the cube:

```
DBMS_CUBE.BUILD('PRICE_CUBE USING(
   SET PRICE_CUBE.UNIT_PRICE["TIME"=''2006.12'', "PRODUCT"=''DLX MOUSE'']
   = 29.49,
   SET PRICE_CUBE.UNIT_PRICE["TIME"=''2007.05'', "PRODUCT"=''DLX MOUSE'']
   = 29.99)');
```

Dimension Maintenance Example

This script shows dimension maintenance. It adds a new dimension member named `OPT MOUSE` to all hierarchies, alters its position in the Primary hierarchy, assigns it a long description, then deletes it from the dimension.

```
BEGIN
dbms_output.put_line('Add optical mouse');
dbms_cube.build(q'!
   "PRODUCT" using (MERGE INTO ALL HIERARCHIES
   VALUES ('ITEM_OPT MOUSE', 'CLASS_SFT', "PRODUCT"."FAMILY"))
!');

dbms_output.put_line('Alter optical mouse');
dbms_cube.build(q'!
   "PRODUCT" using (UPDATE HIERARCHIES ("PRODUCT"."PRIMARY")
   SET PARENT = 'FAMILY_ACC', LEVEL = "PRODUCT"."ITEM"
   WHERE MEMBER = 'ITEM_OPT MOUSE')
!');

dbms_output.put_line('Provide attributes to optical mouse');
dbms_cube.build(q'!
   "PRODUCT" USING (SET "PRODUCT"."LONG_DESCRIPTION"["PRODUCT" = 'ITEM_OPT MOUSE']
    = CAST('Optical Mouse' AS VARCHAR2))
!');

dbms_output.put_line('Delete optical mouse');
dbms_cube.build(q'!
   "PRODUCT" USING (DELETE FROM DIMENSION WHERE MEMBER='ITEM_OPT MOUSE')
!');

END;
/
```

OLAP DML Example

This example uses the OLAP DML to add comments to the cube build log:

```
BEGIN
 DBMS_CUBE.BUILD(q'!
  global.units_cube USING (
   EXECUTE OLAP DML 'SHOW STATLEN(units_cube_prt_list)' PARALLEL,
   EXECUTE OLAP DML 'SHOW LIMIT(units_cube_prt_list KEEP ALL)' PARALLEL,
   EXECUTE OLAP DML 'SHOW STATLEN(time)' parallel,
   EXECUTE OLAP DML 'SHOW LIMIT(time KEEP time_levelrel ''CALENDAR_YEAR'')' parallel)!',
  parallelism=>2,
  add_dimensions=>false);
END;
/
```

This query shows the comments in the cube build log:

```
SELECT partition, slave_number, TO_CHAR(output) output
   FROM cube_build_log
   WHERE command = 'OLAP DML'
   AND status = 'COMPLETED'
   ORDER BY slave_number, time;

PARTITION    SLAVE_NUMBER OUTPUT
------------ ------------ -------------------------------------------------------
P10:CY2007              1 <OLAPDMLExpression
                            Expression="TO_CHAR(statlen(units_cube_prt_list))"
                            Value="1"/>

P10:CY2007              1 <OLAPDMLExpression
                            Expression="TO_CHAR(limit(units_cube_prt_list keep al
                          l))"
                            Value="P10"/>

P10:CY2007              1 <OLAPDMLExpression
                            Expression="TO_CHAR(statlen(time))"
                            Value="17"/>

P10:CY2007              1 <OLAPDMLExpression
                            Expression="TO_CHAR(limit(time keep time_levelrel &ap
                          os;CALENDAR_YEAR&apos;))"
                            Value="CALENDAR_YEAR_CY2007"/>

P9:CY2006               2 <OLAPDMLExpression
                            Expression="TO_CHAR(statlen(units_cube_prt_list))"
                            Value="1"/>

P9:CY2006               2 <OLAPDMLExpression
                            Expression="TO_CHAR(limit(units_cube_prt_list keep al
                          l))"
                            Value="P9"/>

P9:CY2006               2 <OLAPDMLExpression
                            Expression="TO_CHAR(statlen(time))"
                            Value="17"/>
               .
               .
               .
```

CREATE_EXPORT_OPTIONS Procedure

This procedure creates an input XML document that describes processing options for the EXPORT_XML Procedure and the EXPORT_XML_TO_FILE Procedure.

Syntax

```
DBMS_CUBE.CREATE_EXPORT_OPTIONS (
          out_options_xml        IN/OUT  CLOB,
          target_version         IN      VARCHAR2  DEFAULT NULL,
          suppress_owner         IN      BOOLEAN   DEFAULT FALSE,
          suppress_namespace     IN      BOOLEAN   DEFAULT FALSE,
          preserve_table_owners  IN      BOOLEAN   DEFAULT FALSE,
          metadata_changes       IN      CLOB      DEFAULT NULL);
```

Parameters

Table 59-4 CREATE_EXPORT_OPTIONS Procedure Parameters

Parameter	Description
`out_options_xml`	Contains the generated XML document, which can be passed into the `options_xml` parameter of the EXPORT_XML Procedure.
`target_version`	Specifies the version of Oracle Database in which the XML document generated by EXPORT_XML or EXPORT_XML_TO_FILE will be imported. You can specify two to five digits, such as 12.1 or 12.1.0.1.0. This parameter defaults to the current database version, and so can typically be omitted.
`suppress_owner`	Controls the use of the Owner attribute in XML elements and the owner qualifier in object names. Enter `True` to drop the owner from the XML, or enter `False` to retain it. Enter `True` if you plan to import the exported metadata into a different schema.
`suppress_namespace`	Controls the use of Namespace attributes in XML elements and the namespace qualifier in object names. Enter `True` to drop the namespace from the XML, or enter `False` to retain it (default). Enter `True` when upgrading to Oracle OLAP 12*c* metadata. Namespaces allow objects created in Oracle 10*g* to coexist with objects created in Oracle 12*c*. You cannot set or change namespaces.
`preserve_table_owners`	Controls the use of the owner in qualifying table names in the mapping elements, such as GLOBAL.UNITS_HISTORY_FACT instead of UNITS_HISTORY_FACT. Enter `True` to retain the table owner, or enter `False` to default to the current schema for table mappings. If you plan to import the exported metadata to a different schema, you must set this option to `True` to load data from tables and views in the original schema, unless the destination schema has its own copies of the tables and views.
`metadata_changes`	Contains an 12*c* XML description of an object that overwrites the exported object description. The XML document must contain all parent XML elements of the modified element with the attributes needed to uniquely identify them. Use the Name attribute if it exists. See the Examples.

Examples

The following example generates an XML document of export options:

```
DECLARE
   optionsClob  CLOB;

BEGIN
   dbms_lob.createtemporary(optionsClob, false, dbms_lob.CALL);
   dbms_cube.create_export_options(out_options_xml=>optionsClob,
suppress_namespace=>TRUE);
   dbms_output.put_line(optionsClob);
END;
/
```

The `DBMS_OUTPUT.PUT_LINE` procedure displays this XML document (formatted for readability:

```
<?xml version="1.0"?>
<Export TargetVersion="12.1.0.1">
  <ExportOptions>
```

```
    <Option Name="SuppressOwner" Value="FALSE"/>
    <Option Name="SuppressNamespace" Value="TRUE"/>
    <Option Name="PreserveTableOwners" Value="FALSE"/>
  </ExportOptions>
</Export>
```

The next example generates an XML document with a metadata change to the mapping of the American long description attribute of the `CHANNEL` dimension.

```
DECLARE
  importClob         clob;
  exportClob         clob;
  overClob           clob;
  exportOptClob      clob;
  importOptClob      clob;

BEGIN
  dbms_lob.createtemporary(overClob, TRUE);
  dbms_lob.open(overClob, DBMS_LOB.LOB_READWRITE);
  dbms_lob.writeappend(overClob,58, '<Metadata Version="1.3" 
MinimumDatabaseVersion="12.1.0.1">');
  dbms_lob.writeappend(overClob,34, '<StandardDimension Name="CHANNEL">');
  dbms_lob.writeappend(overClob,75, '<Description Type="Description" Language="AMERICAN" 
Value="Sales Channel"/>');
  dbms_lob.writeappend(overClob,20, '</StandardDimension>');
  dbms_lob.writeappend(overClob,11, '</Metadata>');
  dbms_lob.close(overClob);

  -- Enable Oracle Database 12c Release 1 (12.1) clients to access 10g metadata
  dbms_cube.initialize_cube_upgrade;

  -- Create a CLOB containing the export options
  dbms_lob.createtemporary(exportOptClob, TRUE);
  dbms_cube.create_export_options(out_options_xml=>exportOptClob, 
suppress_namespace=>TRUE, metadata_changes=>overClob);

  -- Create a CLOB containing the import options
  dbms_lob.createtemporary(importOptClob, TRUE);
  dbms_cube.create_import_options(out_options_xml=>importOptClob, rename_table => 
'MY_OBJECT_MAP');

   -- Create CLOBs for the metadata
  dbms_lob.createtemporary(importClob, TRUE);
  dbms_lob.createtemporary(exportClob, TRUE);

  -- Export metadata from a 10g analytic workspace to a CLOB
  dbms_cube.export_xml(object_ids=>'GLOBAL_AW', options_xml=>exportOptClob, 
out_xml=>exportClob);

  -- Import metadata from the CLOB
  dbms_cube.import_xml(in_xml => exportClob, options_xml=>importOptClob, 
out_xml=>importClob);

  -- Load and aggregate the data
  dbms_cube.build(script=>'UNITS_CUBE, PRICE_AND_COST_CUBE');

END;
/
```

The following is the content of `exportClob` (formatting added for readability). The XML document changes the description of Channel to Sales Channel.

```
<Metadata Version="1.3" MinimumDatabaseVersion="12.1.0.1">
  <StandardDimension Name="CHANNEL">
    <Description Type="Description" Language="AMERICAN" Value="Sales Channel"/>
  </StandardDimension>
</Metadata>
```

Related Topics

- EXPORT_XML Procedure
  This procedure writes OLAP metadata to a CLOB.
- EXPORT_XML_TO_FILE Procedure
  This procedure exports OLAP metadata to a file. This file can be imported into a new or existing analytic workspace using the IMPORT_XML procedure. In this way, you can create a copy of the analytic workspace in another schema or database.

CREATE_IMPORT_OPTIONS Procedure

This procedure creates an input XML document that describes processing options for the IMPORT_XML Procedure.

Syntax

```
DBMS_CUBE.CREATE_IMPORT_OPTIONS (
          out_options_xml  IN/OUT  CLOB,
          validate_only    IN      BOOLEAN   DEFAULT FALSE,
          rename_table     IN      VARCHAR2  DEFAULT NULL);
```

Parameters

Table 59-5 CREATE_IMPORT_OPTIONS Procedure Parameters

Parameter	Description
out_options_xml	Contains the generated XML document, which can be passed to the options_xml parameter of the IMPORT_XML Procedure.
validate_only	TRUE causes the IMPORT_XML procedure to validate the metadata described in the input file or the in_xml parameter, without committing the changes to the metadata.
rename_table	The name of a table identifying new names for the imported objects, in the form [*schema_name*.]*table_name*. The IMPORT_XML procedure creates objects using the names specified in the table instead of the ones specified in the XML document. See the Usage Notes for the format of the rename table.

Usage Notes

See the information about using a rename table in DBMS_CUBE - Upgrading 10g Analytic Workspaces.

Examples

This example specifies validation only and a rename table. For an example of the import CLOB being used in an import, see "IMPORT_XML Procedure".

```
DECLARE
importClob  clob;

BEGIN
```

```
  dbms_lob.createtemporary(importClob, TRUE);

  dbms_cube.create_import_options(out_options_xml => importClob, rename_table =>
'MY_OBJECT_MAP', validate_only => TRUE);

  dbms_output.put_line(importClob);
END;
/
```

It generates the following XML document:

```
<?xml version="1.0"?>
<Import>
  <ImportOptions>
    <Option Name="ValidateOnly" Value="TRUE"/>
    <Option Name="RenameTable" Value="MY_OBJECT_MAP"/>
  </ImportOptions>
</Import>
```

Related Topics

- IMPORT_XML Procedure
  This procedure creates, modifies, or drops an analytic workspace by using an XML template.

CREATE_MVIEW Function

This function creates a cube materialized view from the definition of a relational materialized view.

Syntax

```
DBMS_CUBE.CREATE_MVIEW (
          mvowner        IN  VARCHAR2,
          mvname         IN  VARCHAR2,
          sam_parameters IN  CLOB  DEFAULT NULL)
     RETURN VARCHAR2;
```

Parameters

Table 59-6   CREATE_MVIEW Function Parameters

Parameter	Description
mvowner	Owner of the relational materialized view.
mvname	Name of the relational materialized view. For restrictions, see "Requirements for the Relational Materialized View". A single cube materialized view can replace many of the relational materialized views for a table. Choose the materialized view that has the lowest levels of the dimension hierarchies that you want represented in the cube materialized view.
sam_parameters	Parameters in the form '*parameter1*=*value1*, *parameter2*=*value2*,...'. See "SQL Aggregation Management Parameters".

SQL Aggregation Management Parameters

The CREATE_MVIEW and DERIVE_FROM_MVIEW functions use the SQL aggregation management (SAM) parameters described in Table 59-7. Some parameters support the development of cubes with advanced analytics. Other parameters support the development of Java applications. The default settings are appropriate for cube materialized views that are direct replacements for relational materialized views.

Table 59-7 SQL Aggregation Management Parameters

Parameter	Description
ADDTOPS	Adds a top level and a level member to every dimension hierarchy in the cube. If the associated relational dimension has no hierarchy, then a dimension hierarchy is created. TRUE: Creates levels named ALL_*dimension* with level members All_*dimension*. (Default) FALSE: Creates only the hierarchies and levels identified by the relational dimensions.
ADDUNIQUEKEYPREFIX	Controls the creation of dimension keys. TRUE: Creates cube dimension keys by concatenating the level name with the relational dimension key. This practice assures that the dimension keys are unique across all levels, such as CITY_NEW_YORK and STATE_NEW_YORK. (Default) FALSE: Uses the relational dimension keys as cube dimension keys.
ATRMAPTYPE	Specifies whether attributes are mapped by hierarchy levels, dimension levels, or both. HIER_LEVEL: Maps attributes to the levels of a particular dimension hierarchy. (Default) DIM_LEVEL: Maps attributes to the levels of the dimension regardless of hierarchy. BOTH: Maps attributes to both dimension and hierarchy levels. AUTO: Maps attributes to the levels of the dimension for a star schema and to the levels of a particular dimension hierarchy for a snowflake schema.
AWNAME	Provides the name of the analytic workspace that owns the cube. Choose a simple database object name of 1 to 30 bytes. The default name is *fact_tablename*_AW*n*.
BUILD	Specifies whether a data refresh will immediately follow creation of the cube materialized view. IMMEDIATE: Refreshes immediately. DEFERRED: Does not perform a data refresh. (Default) **Note**: Only the CREATE_MVIEW function uses this parameter.

Table 59-7 (Cont.) SQL Aggregation Management Parameters

Parameter	Description
CUBEMVOPTION	Controls validation and creation of a cube materialized view. Regardless of this setting, the function creates an analytic workspace containing a cube and its related cube dimensions. COMPLETE_REFRESH: Creates a complete refresh cube materialized view (full update). FAST_REFRESH: Creates a fast refresh materialized view (incremental update). REWRITE_READY: Runs validation checks for a rewrite cube materialized view, but does not create it. REWRITE: Creates a rewrite cube materialized view. REWRITE_WITH_ATTRIBUTES: Creates a rewrite cube materialized view that includes columns with dimension attributes, resulting in faster query response times. (Default) **Note**: *The following settings do not create a cube materialized view.* Use Analytic Workspace Manager to drop an analytic workspace that does not have a cube materialized view. You can use the DROP_MVIEW procedure to delete an analytic workspace only when it supports a cube materialized view. NONE: Does not create a cube materialized view. COMPLETE_REFRESH_READY: Runs validation checks for a complete refresh cube materialized view, but does not create it. FAST_REFRESH_READY: Runs validation checks for fast refresh, but does not create the cube materialized view.
CUBENAME	Provides the name of the cube derived from the relational materialized view. Choose simple database object name of 1 to 30 bytes. The default name is *fact_tablename_Cn*.
DIMJAVABINDVARS	Supports access by Java programs to the XML document. TRUE: Generates an XML template that uses Java bind variable notation for the names of dimensions. No XML validation is performed. You cannot use the IMPORT_XML procedure to create a cube using this template. FALSE: Generates an XML template that does not support Java bind variables. (Default)
DISABLEQRW	Controls disabling of query rewrite on the source relational materialized view. TRUE: Issues an ALTER MATERIALIZED VIEW *mview_name* DISABLE QUERY REWRITE command. FALSE: No action. **Note**: Only the CREATE_MVIEW function with BUILD=IMMEDIATE uses this parameter.
EXPORTXML	Exports the XML that defines the dimensional objects to a file, which you specify as *dir/filename*. Both the directory and the file name are case sensitive. *dir*: Name of a database directory. *filename*: The name of the file, typically given an XML filename extension.

Table 59-7 (Cont.) SQL Aggregation Management Parameters

Parameter	Description
FILTERPARTITIONANCESTORLEVELS	Controls the generation of aggregate values above the partitioning level of a partitioned cube. TRUE: Removes levels above the partitioning level from the cube. Requests for summary values above the partitioning level are solved by SQL. FALSE: All levels are retained in the cube. Requests for summary values are solved by OLAP. (Default)
LOGDEST	Directs and stores log messages. By default, the messages are not available. SERVEROUT: Sends messages to server output (typically the screen), which is suitable when working interactively such as in SQL*Plus or SQL Developer. TRACEFILE: Sends messages to the session trace file.
PARTITIONOPTION	Controls partitioning of the cube. NONE: Prevents partitioning. DEFAULT: Allows the Sparsity Advisor to determine whether partitioning is needed and how to partition the cube. (Default) FORCE: Partitions the cube even when the Sparsity Advisor recommends against it. The Sparsity Advisor identifies the best dimension, hierarchy, and level to use for partitioning. *dimension.hierarchy.level*: Partitions the cube using the specified dimension, hierarchy, and level.
POPULATELINEAGE	Controls the appearance of attributes in a cube materialized view. TRUE: Includes all dimension attributes in the cube materialized view. (Default) FALSE: Omits all dimension attributes from the cube materialized view.
PRECOMPUTE	Identifies a percentage of the data that is aggregated and stored. The remaining values are calculated as required by queries during the session. *precompute_percentage*[:*precompute_top_percentage*] Specify the top percentage for partitioned cubes. The default value is 35:0, which specifies precomputing 35% of the bottom partition and 0% of the top partition. If the cube is not partitioned, then the second number is ignored.
REMAPCOMPOSITEKEYS	Controls how multicolumn keys are rendered in the cube. TRUE: Creates a unique key attribute whose values are concatenated string expressions with an underscore between the column values. For example, the value BOSTON_MA_USA might be an expression produced from a multicolumn key composed of CITY, STATE, and COUNTRY columns. In addition, an attribute is created for each individual column to store the relational keys. (Default) FALSE: Creates a unique key attribute for each column.

Table 59-7 (Cont.) SQL Aggregation Management Parameters

Parameter	Description
RENDERINGMODE	Controls whether a loss in fidelity between the relational materialized view and the cube materialized view results in a warning message or an exception. See "Requirements for the Relational Materialized View". LOOSE: Losses are noted in the optional logs generated by the CREATE_MVIEW Function and the DERIVE_FROM_MVIEW Function. No exceptions are raised. (Default) STRICT: Any loss in fidelity raises an exception so that no XML template is created.
SEEFILTERS	Controls whether conditions in the WHERE clause of the relational materialized view's defining query are retained or ignored. TRUE: Renders valid conditions in the XML template. (Default) FALSE: Ignores all conditions.
UNIQUENAMES	Controls whether top level dimensional objects have unique names. Cross namespace conflicts may occur because dimensional objects have different namespaces than relational objects. TRUE: Modifies all relational names when they are rendered in the cube.(Default) FALSE: Duplicates relational names in the cube unless a naming conflict is detected. In that case, a unique name is created.
UNKNOWNKEYASDIM	Controls handling of simple columns with no levels or hierarchies in the GROUP BY clause of the relational materialized view's defining query. TRUE: Renders a simple column without a relational dimension as a cube dimension with no levels or hierarchies. FALSE: Raises an exception when no relational dimension is found for the column. (Default)
VALIDATEXML	Controls whether the generated XML document is validated. TRUE: Validates the template using the VALIDATE_XML procedure. (Default) FALSE: No validation is done.

Returns

The name of the cube materialized view created by the function.

Usage Notes

See "Using SQL Aggregation Management"

Examples

All examples for the SQL Aggregate Management subprograms use the sample Sales History schema, which is installed in Oracle Database with two relational materialized views: CAL_MONTH_SALES_MV and FWEEK_PSCAT_SALES_MV.

The following script creates a cube materialized view using CAL_MONTH_SALES_MV as the relational materialized view. It uses all default options.

```
SET serverout ON format wrapped

DECLARE
     salesaw  varchar2(30);
```

```
BEGIN
     salesaw := dbms_cube.create_mview('SH', 'CAL_MONTH_SALES_MV');
END;
/
```

The next example sets several parameters for creating a cube materialized view from `FWEEK_PSCAT_SALES_MV`. These parameters change the cube materialized view in the following ways:

- `ADDTOPS`: Adds a top level consisting of a single value to the hierarchies. All of the dimensions in Sales History have a top level already.
- `PRECOMPUTE`: Changes the percentage of materialized aggregates from 35:0 to 40:10.
- `EXPORTXML`: Creates a text file for the XML document.
- `BUILD`: Performs a data refresh.

```
DECLARE
     salescubemv   varchar2(30);
     sam_param     clob := 'ADDTOPS=FALSE,
                            PRECOMPUTE=40:10,
                            EXPORTXML=WORK_DIR/sales.xml,
                            BUILD=IMMEDIATE';

BEGIN
     salescubemv := dbms_cube.create_mview('SH', 'FWEEK_PSCAT_SALES_MV',
                    sam_param);
END;
/
```

DERIVE_FROM_MVIEW Function

This function generates an XML template that defines a cube with materialized view capabilities, using the information derived from an existing relational materialized view.

Syntax

```
DBMS_CUBE.DERIVE_FROM_MVIEW (
          mvowner        IN   VARCHAR2,
          mvname         IN   VARCHAR2,
          sam_parameters IN   CLOB   DEFAULT NULL)
     RETURN CLOB;
```

Parameters

Table 59-8 DERIVE_FROM_MVIEW Function Parameters

Parameter	Description
`mvowner`	Owner of the relational materialized view.
`mvname`	Name of the relational materialized view. For restrictions, see "Requirements for the Relational Materialized View". A single cube materialized view can replace many of the relational materialized views for a table. Choose the materialized view that has the lowest levels of the dimension hierarchies that you want represented in the cube materialized view.

Table 59-8 (Cont.) DERIVE_FROM_MVIEW Function Parameters

Parameter	Description
`sam_parameters`	Optional list of parameters in the form '*parameter1*=*value1*, *parameter2*=*value2*,...'. See "SQL Aggregation Management Parameters".

Returns

An XML template that defines an analytic workspace containing a cube enabled as a materialized view.

Usage Notes

To create a cube materialized view from an XML template, use the `IMPORT_XML` procedure. Then use the `REFRESH_MVIEW` procedure to refresh the cube materialized view with data.

See "Using SQL Aggregation Management".

Examples

The following example generates an XML template named `sales_cube.xml` from the `CAL_MONTH_SALES_MV` relational materialized view in the `SH` schema.

```
DECLARE
     salescubexml  clob := null;
     sam_param     clob := 'exportXML=WORK_DIR/sales_cube.xml';

BEGIN
     salescubexml := dbms_cube.derive_from_mview('SH', 'CAL_MONTH_SALES_MV',
     sam_param);
END;
/
```

DROP_MVIEW Procedure

This procedure drops a cube materialized view and all associated objects from the database. These objects include the dimension materialized views, cubes, cube dimensions, levels, hierarchies, and the analytic workspace.

Syntax

```
DBMS_CUBE.DROP_MVIEW (
          mvowner        IN   VARCHAR2,
          mvname         IN   VARCHAR2,
          sam_parameters IN   CLOB   DEFAULT NULL);
```

Parameters

Table 59-9 DROP_MVIEW Procedure Parameters

Parameter	Description
`mvowner`	Owner of the cube materialized view
`mvname`	Name of the cube materialized view

Table 59-9 (Cont.) DROP_MVIEW Procedure Parameters

Parameter	Description
`sam_parameters`	`EXPORTXML`: Exports the XML that drops the dimensional objects to a file, which you specify as *`dir/filename`*. Both the directory and the file name are case sensitive. *`dir`*: Name of a database directory. *`filename`*: The name of the file, typically given an XML filename extension.

Usage Notes

Use this procedure to drop a cube materialized view that you created using the `CREATE_MVIEW` and `DERIVE_FROM_MVIEW` functions. If you make modifications to the cubes or dimensions, then `DROP_MVIEW` may not be able to drop the cube materialized view.

Some of the `CUBEMVOPTION` parameters used by the `CREATE_MVIEW` and `DERIVE_FROM_MVIEW` functions do not create a materialized view. Use Analytic Workspace Manager to drop the analytic workspace, cubes, and cube dimensions.

If you use the `EXPORTXML` parameter, then you can use the XML document to drop the cube materialized view, after you re-create it. Use the `IMPORT_XML` procedure.

See "Using SQL Aggregation Management".

Examples

The current schema has four materialized views. `CB$CAL_MONTH_SALES` is a cube materialized view for the `SALES` table. `CB$TIMES_DIM_D1_CAL_ROLLUP` is a cube dimension materialized view for the `TIMES_DIM` dimension on the `TIMES` dimension table. The others are relational materialized views.

```
SELECT mview_name FROM user_mviews;

MVIEW_NAME
------------------------------
CB$CAL_MONTH_SALES
CB$TIMES_DIM_D1_CAL_ROLLUP
CAL_MONTH_SALES_MV
FWEEK_PSCAT_SALES_MV
```

The following command drops both `CB$CAL_MONTH_SALES` and `CB$TIMES_DIM_D1_CAL_ROLLUP`.

```
EXECUTE dbms_cube.drop_mview('SH', 'CB$CAL_MONTH_SALES');

Dropped cube organized materialized view "SH"."CAL_MONTH_SALES"
including container analytic workspace "SH"."CAL_MONTH_SALES_AW"
at 20130213 16:31:40.056.
```

This query against the data dictionary confirms that the materialized views have been dropped.

```
SELECT mview_name FROM user_mviews;

MVIEW_NAME
------------------------------
CAL_MONTH_SALES_MV
FWEEK_PSCAT_SALES_MV
```

EXPORT_XML Procedure

This procedure writes OLAP metadata to a CLOB.

Syntax

```
DBMS_CUBE.EXPORT_XML
       (object_ids              IN       VARCHAR2,
        out_xml                 IN/OUT   CLOB;

DBMS_CUBE.EXPORT_XML
       (object_ids              IN       VARCHAR2,
        options_xml             IN       CLOB,
        out_xml                 IN/OUT   CLOB;

DBMS_CUBE.EXPORT_XML
       (object_ids              IN       VARCHAR2,
        options_dirname         IN       VARCHAR2,
        options_filename        IN       VARCHAR2,
        out_xml                 IN/OUT   CLOB;
```

Parameters

Table 59-10 EXPORT_XML Procedure Parameters

Parameter	Description
`object_ids`	Any of these identifiers. • The name of a schema, such as `GLOBAL`. • The fully qualified name of an analytic workspace in the form *`owner.aw_name`*`.AW`, such as `GLOBAL.GLOBAL.AW`. • Cube • Dimension • Named build process • Measure folder You can specify multiple objects by separating the names with commas. **Note**: When exporting an individual object, be sure to export any objects required to reconstruct it. For example, when exporting a cube, you must also export the dimensions of the cube.
`options_dirname`	The case-sensitive name of a database directory that contains *`options_filename`*.
`options_filename`	A file containing an XML document of export options.
`options_xml`	A CLOB variable that contains an XML document of export options. Use the CREATE_EXPORT_OPTIONS Procedure to generate this document.
`out_xml`	A CLOB variable that will store the XML document of OLAP metadata for the objects listed in *`object_ids`*.

Export Options

The default settings for the export options are appropriate in many cases, so you can omit the *`options_xml`* parameter or the *`options_dirname`* and *`options_filename`* parameters. However, when upgrading Oracle OLAP 10*g* metadata to OLAP 12*c*, you must specify an XML document that changes the default settings. This example changes all of the parameters from False to True; set them appropriately for your schema.

```
<?xml version="1.0"?>
<Export>
  <ExportOptions>
    <Option Name="SuppressNamespace" Value="True"/>
    <Option Name="SuppressOwner" Value="True"/>
    <Option Name="PreserveTableOwners" Value="True"/>
  </ExportOptions>
</Export>
```

You can create this XML document manually or by using the CREATE_EXPORT_OPTIONS Procedure.

Usage Notes

See "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

Example

For an example of using EXPORT_XML in an upgrade to the same schema, see "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

The following PL/SQL script copies an OLAP 12*c* analytic workspace named GLOBAL12 from the GLOBAL_AW schema to the GLOBAL schema. No upgrade is performed.

To upgrade into a different schema, change the example as follows:

- Call the INITIALIZE_CUBE_UPGRADE procedure.
- Call the CREATE_EXPORT_OPTIONS procedure with the additional parameter setting SUPPRESS_NAMESPACE=>TRUE.

The PL/SQL client must be connected to the database as GLOBAL. The GLOBAL user must have SELECT permissions on GLOBAL_AW.AW$GLOBAL and on all relational data sources.

```
BEGIN
  -- Create a CLOB for the export options
  dbms_lob.createtemporary(optionsClob, TRUE);
  dbms_cube.create_export_options(out_options_xml=>optionsClob, suppress_owner=>TRUE, 
preserve_table_owners=>TRUE);

  -- Create a CLOB for the XML template
  dbms_lob.createtemporary(exportClob, TRUE);

  -- Export metadata from an analytic workspace to a CLOB
  dbms_cube.export_xml(object_ids=>'GLOBAL_AW.GLOBAL12.AW', options_xml=>optionsClob, 
out_xml=>exportClob);

  -- Import metadata from the CLOB
  dbms_cube.import_xml(in_xml=>exportClob);

  -- Load and aggregate the data
  dbms_cube.build(script=>'GLOBAL.UNITS_CUBE, GLOBAL.PRICE_AND_COST_CUBE');

END;
/
```

EXPORT_XML_TO_FILE Procedure

This procedure exports OLAP metadata to a file. This file can be imported into a new or existing analytic workspace using the IMPORT_XML procedure. In this way, you can create a copy of the analytic workspace in another schema or database.

This procedure can also be used as part of the process for upgrading OLAP standard form metadata contained in an Oracle OLAP 10*g* analytic workspace to OLAP 12*c* format.

Syntax

```
DBMS_CUBE.EXPORT_XML_TO_FILE
       (object_ids           IN      VARCHAR2,
        output_dirname       IN      VARCHAR2,
        output_filename      IN      VARCHAR2;

DBMS_CUBE.EXPORT_XML_TO_FILE
       (object_ids           IN      VARCHAR2,
        options_dirname      IN      VARCHAR2,
        options_filename     IN      VARCHAR2,
        output_dirname       IN      VARCHAR2,
        output_filename      IN      VARCHAR2;
```

Parameters

Table 59-11   EXPORT_XML_TO_FILE Procedure Parameters

Parameter	Description
object_ids	Any of these identifiers. • The name of a schema, such as GLOBAL. • The fully qualified name of an analytic workspace in the form *owner.aw_name*.AW, such as GLOBAL.GLOBAL.AW. • Cube • Dimension • Named build process • Measure folder You can specify multiple objects by separating the names with commas. **Note**: When exporting an individual object, be sure to export any objects required to reconstruct it. For example, when you export a cube, you must also export the dimensions of the cube.
options_dirname	The case-sensitive name of a database directory that contains options_filename. See "Export Options".
options_filename	The name of a file containing an XML document of export options. See "Export Options".
output_dirname	The case-sensitive name of a database directory where *output_filename* is created.
output_filename	The name of the template file created by the procedure.

Export Options

The default settings for the export options are appropriate in most cases, and you can omit the *options_dirname* and *options_filename* parameters. However, when upgrading Oracle OLAP

10*g* metadata to OLAP 12*c*, you must specify an XML document that changes the default settings, like the following:

```
<?xml version="2.0"?>
<Export>
  <ExportOptions>
    <Option Name="SuppressNamespace" Value="True"/>
    <Option Name="SuppressOwner" Value="True"/>
    <Option Name="PreserveTableOwners" Value="True"/>
  </ExportOptions>
</Export>
```

You can create this XML document manually or by using the CREATE_EXPORT_OPTIONS Procedure.

Usage Notes

See "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

Examples

The following example generates an XML file named global.xml in OLAP 12*c* format using the default export settings. The metadata is derived from all analytic workspaces and CWM metadata in the `GLOBAL_AW` schema. The output file is generated in the `WORK_DIR` database directory.

```
execute dbms_cube.export_xml_to_file('GLOBAL_AW', 'WORK_DIR', 'global.xml');
```

The next example also generates an XML file named global.xml in OLAP 12*c* format using the export options set in options.xml. The metadata is derived from the GLOBAL analytic workspace in the `GLOBAL_AW` schema. Both the options file and the output file are in the `WORK_DIR` database directory.

```
execute dbms_cube.export_xml_to_file('GLOBAL_AW.GLOBAL.AW', 'WORK_DIR', 'options.xml',
'WORK_DIR', 'global.xml');
```

IMPORT_XML Procedure

This procedure creates, modifies, or drops an analytic workspace by using an XML template.

Syntax

```
DBMS_CUBE.IMPORT_XML
       (dirname               IN      VARCHAR2,
        filename              IN      VARCHAR2 );

DBMS_CUBE.IMPORT_XML
       (dirname               IN      VARCHAR2,
        filename              IN      VARCHAR2,
        out_xml               IN/OUT  CLOB );

DBMS_CUBE.IMPORT_XML
       (input_dirname         IN      VARCHAR2,
        input_filename        IN      VARCHAR2
        options_dirname       IN      VARCHAR2,
        options_filename      IN      VARCHAR2,
        out_xml               IN/OUT  CLOB );

DBMS_CUBE.IMPORT_XML
       (in_xml                IN      CLOB );
```

```
DBMS_CUBE.IMPORT_XML
       (in_xml                IN      CLOB,
        out_xml               IN/OUT  CLOB );

DBMS_CUBE.IMPORT_XML

       (in_xml                IN      CLOB,
        options_xml           IN      CLOB,
        out_xml               IN/OUT  CLOB );
```

Parameters

Table 59-12 IMPORT_XML Procedure Parameters

Parameter	Description
dirname	The case-sensitive name of a database directory containing the XML document describing an analytic workspace.
filename	A file containing an XML document describing an analytic workspace.
in_xml	A CLOB containing an XML document describing an analytic workspace.
input_dirname	The case-sensitive name of a database directory containing the XML document describing an analytic workspace.
input_filename	A file containing an XML document describing an analytic workspace.
options_dirname	The case-sensitive name of a database directory containing a file of import options.
options_filename	A file of import options.
options_xml	An XML document describing the import options. Use the CREATE_IMPORT_OPTIONS Procedure to generate this document.
out_xml	An XML document that either describes the analytic workspace or, for validation only, describes any errors. It may contain changes that DBMS_CUBE made to the imported XML, such as setting default values or making minor corrections to the XML.

Usage Notes

The XML can define, modify, or drop an entire analytic workspace, or one or more cubes or dimensions. When defining just cubes or dimensions, you must do so within an existing analytic workspace.

You can also use IMPORT_XML to drop an analytic workspace by using the XML document generated by the DROP_MVIEW procedure with the EXPORTXML parameter.

See "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

Example

This example loads an XML template from a file named GLOBAL.XML and located in a database directory named XML_DIR.

```
EXECUTE dbms_cube.import_xml('XML_DIR', 'GLOBAL.XML');
```

The next example exports an OLAP 10*g* template and uses IMPORT_XML to validate it before an upgrade to 12*c*.

```
DECLARE
```

```
  exportOptClob clob;
  importOptClob clob;
  importClob    clob;
  exportClob    clob;

BEGIN

  -- Create a CLOB for the export options
  dbms_lob.createtemporary(exportOptClob, TRUE);
  dbms_cube.create_export_options(out_options_xml=>exportOptClob,
suppress_namespace=>TRUE, preserve_table_owners=>TRUE);

  -- Create a CLOB for the XML template
  dbms_lob.createtemporary(exportClob, TRUE);

  -- Create a CLOB for import options
  dbms_lob.createtemporary(importOptClob, TRUE);
  dbms_cube.create_import_options(out_options_xml=>importOptClob, validate_only=>TRUE);

  -- Create a CLOB for the change log
  dbms_lob.createtemporary(importClob, TRUE);

  -- Enable Oracle Database 12c Release 1 (12.1) clients to access 10g metadata
  dbms_cube.initialize_cube_upgrade;

  -- Export metadata from an analytic workspace to a CLOB
  dbms_cube.export_xml(object_ids=>'GLOBAL_AW', options_xml=>exportOptClob,
out_xml=>exportClob);

  /* Import metadata from the CLOB. No objects are committed to the database
     because the validate_only parameter of CREATE_IMPORT_OPTIONS is set to
     TRUE.
  */

  dbms_cube.import_xml(in_xml=>exportClob, options_xml=>importOptClob,
out_xml=>importClob);

   -- Output the metadata changes
  dbms_output.put_line('This is the validation log:');
  dbms_output.put_line(importClob);

END;
/
```

The contents of `importClob` show that the XML is valid. Otherwise, error messages appear in the `<RootCommitResult>` element.

```
This is the validation log:
<?xml version="1.0" encoding="UTF-16"?>
<RootCommitResult>

</RootCommitResult>
```

For an example of `IMPORT_XML` within the context of an upgrade from 10*g* to 12*c* metadata, see the Custom Upgrade section of "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

INITIALIZE_CUBE_UPGRADE Procedure

This procedure processes analytic workspaces created in Oracle OLAP 10*g* so they can be used by Oracle OLAP 12*c* clients. It processes all analytic workspaces in the current schema. Run this procedure once for each schema in which there are 10*g* analytic workspaces.

Without this processing step, 12*c* clients cannot connect to a database containing a 10*g* analytic workspace with subobjects of a dimension or cube having the same name. Additionally, some DBMS_CUBE procedures and functions, such as EXPORT_XML and EXPORT_XML_TO_FILE, do not work on the 10*g* metadata.

After processing, OLAP 12*c* clients can connect and use the alternate names provided by INITIALIZE_CUBE_UPGRADE for the conflicting subobjects. OLAP 10*g* clients continue to use the original names.

INITIALIZE_CUBE_UPGRADE does not upgrade any OLAP 10*g* objects to OLAP 12*c* format.

See "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

Syntax

```
DBMS_CUBE.INITIALIZE_CUBE_UPGRADE;
```

Usage Notes

This procedure creates and populates a table named CUBE_UPGRADE_INFO. If it already exists, the table is truncated and repopulated.

While the 10*g* namespace allowed subobjects with the same name in the same dimension or cube, the 12*c* namespace does not. When INITIALIZE_CUBE_UPGRADE detects a name conflict among subobjects such as levels, hierarchies, and dimension attributes, it creates a row in CUBE_UPGRADE_INFO providing a new, unique name for each one. Rows may also be created for objects that do not require renaming; these rows are distinguished by a value of 0 or null in the CONFLICT column. Top-level objects, such as dimensions and cubes, are not listed.

You can edit the table using SQL INSERT and UPDATE if you want to customize the names of OLAP 10*g* objects on OLAP 12*c* clients.

The UPGRADE_AW, EXPORT_XML and EXPORT_XML_TO_FILE procedures use the names specified in the NEW_NAME column of the table to identify objects in CWM or OLAP standard form (AWXML) analytic workspaces, rather than the original names.

The following table describes the columns of CUBE_UPGRADE_INFO.

Column	Datatype	NULL	Description
OWNER	VARCHAR2	NOT NULL	Owner of the analytic workspace.
AW	VARCHAR2	NOT NULL	Name of the analytic workspace.
AWXML_ID	VARCHAR2	NOT NULL	Full logical name of the object requiring modification, in the form *simple_name.[subtype_name].object_type*. For example, TIME.DIMENSION and PRODUCT.COLOR.ATTRIBUTE.
NEW_NAME	VARCHAR2	NOT NULL	The name the object will have in Oracle 12*c* after the upgrade.
OBJECT_CLASS	VARCHAR2	--	DerivedMeasure for calculated measures, or empty for all other object types.

Column	Datatype	NULL	Description
CONFLICT	NUMBER	--	Indicates the reason that the row was added to CUBE_UPGRADE_INFO: • 0: The object does not have a naming conflict but appears in the table for other reasons. • 1: Two objects have the same name and would create a conflict in the OLAP 12*c* namespace. The object type (such as level or hierarchy) will be added to the names.

Examples

The following command creates and populates the CUBE_UPGRADE_INFO table:

```
EXECUTE dbms_cube.initialize_cube_upgrade;
```

The table shows that the OLAP 10*g* analytic workspace has a hierarchy and a level named MARKET_SEGMENT, which will be renamed. The table also contains rows for calculated measures, but these objects do not require renaming: The value of `CONFLICT` is `0`.

```
SELECT awxml_id, new_name, conflict FROM cube_upgrade_info;

AWXML_ID                                 NEW_NAME                    CONFLICT
---------------------------------------- ------------------------- ----------
CUSTOMER.MARKET_SEGMENT.HIERARCHY        MARKET_SEGMENT_HIERARCHY           1
CUSTOMER.MARKET_SEGMENT.LEVEL            MARKET_SEGMENT_LEVEL               1
UNITS_CUBE.EXTENDED_COST.MEASURE         EXTENDED_COST                      0
UNITS_CUBE.EXTENDED_MARGIN.MEASURE       EXTENDED_MARGIN                    0
UNITS_CUBE.CHG_SALES_PP.MEASURE          CHG_SALES_PP                       0
UNITS_CUBE.CHG_SALES_PY.MEASURE          CHG_SALES_PY                       0
UNITS_CUBE.PCTCHG_SALES_PP.MEASURE       PCTCHG_SALES_PP                    0
UNITS_CUBE.PCTCHG_SALES_PY.MEASURE       PCTCHG_SALES_PY                    0
UNITS_CUBE.PRODUCT_SHARE.MEASURE         PRODUCT_SHARE                      0
UNITS_CUBE.CHANNEL_SHARE.MEASURE         CHANNEL_SHARE                      0
UNITS_CUBE.MARKET_SHARE.MEASURE          MARKET_SHARE                       0
UNITS_CUBE.CHG_EXTMRGN_PP.MEASURE        CHG_EXTMRGN_PP                     0
UNITS_CUBE.CHG_EXTMRGN_PY.MEASURE        CHG_EXTMRGN_PY                     0
UNITS_CUBE.PCTCHG_EXTMRGN_PP.MEASURE     PCTCHG_EXTMRGN_PP                  0
UNITS_CUBE.PCTCHG_EXTMRGN_PY.MEASURE     PCTCHG_EXTMRGN_PY                  0
UNITS_CUBE.CHG_UNITS_PP.MEASURE          CHG_UNITS_PP                       0
UNITS_CUBE.EXTMRGN_PER_UNIT.MEASURE      EXTMRGN_PER_UNIT                   0
UNITS_CUBE.SALES_YTD.MEASURE             SALES_YTD                          0
UNITS_CUBE.SALES_YTD_PY.MEASURE          SALES_YTD_PY                       0
UNITS_CUBE.PCTCHG_SALES_YTD_PY.MEASURE   PCTCHG_SALES_YTD_PY                0
UNITS_CUBE.SALES_QTD.MEASURE             SALES_QTD                          0
UNITS_CUBE.CHG_UNITS_PY.MEASURE          CHG_UNITS_PY                       0
```

REFRESH_MVIEW Procedure

This procedure refreshes the data in a cube materialized view.

Syntax

```
DBMS_CUBE.REFRESH_MVIEW (
          mvowner              IN  VARCHAR2,
          mvname               IN  VARCHAR2,
          method               IN  VARCHAR2       DEFAULT NULL,
          refresh_after_errors IN  BOOLEAN        DEFAULT FALSE,
          parallelism          IN  BINARY_INTEGER DEFAULT 0,
          atomic_refresh       IN  BOOLEAN       DEFAULT FALSE,
          scheduler_job        IN  VARCHAR2      DEFAULT NULL,
          sam_parameters       IN  CLOB          DEFAULT NULL,
          nested               IN  BOOLEAN       DEFAULT FALSE );
```

Parameters

Table 59-13 REFRESH_MVIEW Procedure Parameters

Parameter	Description
`mvowner`	Owner of the cube materialized view.
`mvname`	Name of the cube materialized view.
`method`	A full or a fast (partial) refresh. In a fast refresh, only changed rows are inserted in the cube and the affected areas of the cube are re-aggregated. You can specify a method for each cube in sequential order, or a single method to apply to all cubes. If you list more cubes than methods, then the last method applies to the additional cubes. • `C`: Complete refresh clears all dimension values before loading. (Default) • `F`: Fast refresh of a cube materialized view, which performs an incremental refresh and re-aggregation of only changed rows in the source table. • `?`: Fast refresh if possible, and otherwise a complete refresh. • `P`: Recomputes rows in a cube materialized view that are affected by changed partitions in the detail tables. • `S`: Fast solve of a compressed cube. A fast solve reloads all the detail data and re-aggregates only the changed values. See the "Usage Notes" for the `BUILD` procedure for additional details.
`refresh_after_errors`	`TRUE` to roll back just the cube or dimension with errors, and then continue building the other objects. `FALSE` to roll back all objects in the build.
`parallelism`	Number of parallel processes to allocate to this job. See the "Usage Notes" for the `BUILD` procedure for additional details.
`atomic_refresh`	`TRUE` prevents users from accessing intermediate results during a build. It freezes the current state of an analytic workspace at the beginning of the build to provide current sessions with consistent data. This option thaws the analytic workspace at the end of the build to give new sessions access to the refreshed data. If an error occurs during the build, then all objects are rolled back to the frozen state. `FALSE` enables users to access intermediate results during an build.
`scheduler_job`	Any text identifier for the job, which will appear in the log table. The string does not need to be unique.
`sam_parameters`	None.
`nested`	`TRUE` performs nested refresh operations for the specified set of cube materialized views. Nested refresh operations refresh all the depending materialized views and the specified set of materialized views based on a dependency order to ensure the nested materialized views are truly fresh with respect to the underlying base tables. All objects must reside in a single analytic workspace.

Usage Notes

REFRESH_MVIEW changes *mvname* to the name of the cube, then passes the cube name and all parameters to the BUILD procedure. Thus, you can use the BUILD procedure to refresh a cube materialized view. See the "BUILD Procedure" for additional information about the parameters.

Examples

The following example uses the default settings to refresh a cube materialized view named CB$FWEEK_PSCAT_SALES.

```
SET serverout ON format wrapped

EXECUTE dbms_cube.refresh_mview('SH', 'CB$FWEEK_PSCAT_SALES');
```

The next example changes the refresh method to use fast refresh if possible, continue refreshing after an error, and use two parallel processes.

```
EXECUTE dbms_cube.refresh_mview('SH', 'CB$FWEEK_PSCAT_SALES', '?', TRUE, 2);
```

After successfully refreshing the cube materialized view, REFRESH_MVIEW returns a message like the following:

```
Completed refresh of cube mview "SH"."CB$FWEEK_PSCAT_SALES" at 20130212 15:04:46.370.
```

UPGRADE_AW Procedure

This procedure creates an Oracle OLAP 12*c* analytic workspace from a copy of the metadata contained in an OLAP 10*g* analytic workspace. The original OLAP 10*g* analytic workspace is not affected and can exist at the same time and in the same schema as the OLAP 12*c* analytic workspace.

UPGRADE_AW automatically runs INITIALIZE_CUBE_UPGRADE if the CUBE_UPGRADE_INFO table does not exist. If it does exist, then UPGRADE_AW does not overwrite it, thus preserving any changes you made to the table.

See "DBMS_CUBE - Upgrading 10g Analytic Workspaces".

Syntax

```
DBMS_CUBE.UPGRADE_AW
       (sourceaw              IN  VARCHAR2,
        destaw                IN  VARCHAR2,
        upgoptions            IN  CLOB DEFAULT NULL);
```

Parameters

Table 59-14 UPGRADE_AW Procedure Parameters

Parameter	Description
sourceaw	The name of a 10*g* analytic workspace.
destaw	A new name for the generated 12*c* analytic workspace. It cannot be the same as *sourceaw*.

Table 59-14 (Cont.) UPGRADE_AW Procedure Parameters

Parameter	Description
`upgoptions`	One or more of these upgrade options, as a string in the form '*OPTION*=*VALUE*'. Separate multiple options with commas. • `PRESERVE_TABLE_OWNERS`: `YES` preserves the original source table mappings. Use this option when creating an OLAP 12*c* analytic workspace in a different schema from the 10*g* analytic workspace, and you want the new objects mapped to tables in the original schema. (Default) `NO` removes the schema owner from the source table mappings. Use this option when creating an OLAP 12*c* analytic workspace in a different schema from the 10*g* analytic workspace, and you want the new objects mapped to tables in the destination schema. • `RENAME_TABLE`: The name of a table that specifies new names for objects as they are created in OLAP 12*c* format. These changes are in addition to those specified by the `INITIALIZE_CUBE_UPGRADE` procedure. See "CREATE_IMPORT_OPTIONS Procedure" for information about creating a rename table. • `TARGET_VERSION`: The version of the upgrade, specified by a 2- to 5-part number, such as 11.2 or 11.2.0.2.0. If you enter an unsupported version number, then the closest version below it is used.

Examples

This example upgrades an OLAP 10*g* analytic workspace named `GLOBAL10` to an OLAP 12*c* analytic workspace named `GLOBAL12`, using a rename table named `MY_OBJECT_MAP`:

```
BEGIN

  -- Upgrade the analytic workspace
  dbms_cube.upgrade_aw(sourceaw =>'GLOBAL10', destaw => 'GLOBAL12', upgoptions =>
'RENAME_TABLE=MY_OBJECT_MAP');

  -- Load and aggregate the data
  dbms_cube.build(script=>'UNITS_CUBE, PRICE_AND_COST_CUBE');

END;
/
```

VALIDATE_XML Procedure

This procedure checks the XML to assure that it is valid without committing the results to the database. It does not create an analytic workspace.

Syntax

```
DBMS_CUBE.VALIDATE_XML
       (dirname                IN  VARCHAR2,
        filename               IN  VARCHAR2 );

DBMS_CUBE.VALIDATE_XML
       (in_xml                 IN  CLOB );
```

Parameters

Table 59-15 VALIDATE_XML Procedure Parameters

Parameter	Description
`dirname`	The case-sensitive name of a database directory.
`filename`	The name of a file containing an XML template.
`IN_XML`	The name of a CLOB containing an XML template.

Usage Notes

You should always load a template into the same version and release of Oracle Database as the one used to generate the template. The XML may not be valid if it was generated by a different release of the software.

Example

This example reports a problem in the schema:

```
EXECUTE dbms_cube.validate_xml('UPGRADE_DIR', 'MYGLOBAL.XML');
BEGIN dbms_cube.validate_xml('UPGRADE_DIR', 'MYGLOBAL.XML'); END;

*
ERROR at line 1:
ORA-37162: OLAP error
'GLOBAL.PRICE_CUBE.$AW_ORGANIZATION': XOQ-01950: The AWCubeOrganization for
cube "GLOBAL.PRICE_CUBE" contains multiple BuildSpecifications with the same
name.
'GLOBAL.UNITS_CUBE.$AW_ORGANIZATION': XOQ-01950: The AWCubeOrganization for
cube "GLOBAL.UNITS_CUBE" contains multiple BuildSpecifications with the same
name.
XOQ-01400: invalid metadata objects
ORA-06512: at "SYS.DBMS_CUBE", line 411
ORA-06512: at "SYS.DBMS_CUBE", line 441
ORA-06512: at "SYS.DBMS_CUBE", line 501
ORA-06512: at "SYS.DBMS_CUBE", line 520
ORA-06512: at line 1
```

After the problems are corrected, the procedure reports no errors:

```
EXECUTE dbms_cube.validate_xml('UPGRADE_DIR', 'MYGLOBAL.XML');

PL/SQL procedure successfully completed.
```

This example loads an XML template into a temporary CLOB, then validates it. The script is named `GLOBAL.XML`, and it is located in a database directory named `XML_DIR`.

```
DEFINE xml_file = 'GLOBAL.XML';

SET ECHO ON;
SET SERVEROUT ON;


DECLARE
     xml_file     BFILE := bfilename('XML_DIR', '&xml_file');
     in_xml       CLOB;
     out_xml      CLOB := NULL;
     dest_offset  INTEGER := 1;
     src_offset   INTEGER := 1;
     lang_context INTEGER := 0;
     warning      INTEGER;
BEGIN
     -- Setup the clob from a file
     DBMS_LOB.CREATETEMPORARY(in_xml, TRUE);
     DBMS_LOB.OPEN(in_xml, DBMS_LOB.LOB_READWRITE);
     DBMS_LOB.OPEN(xml_file, DBMS_LOB.FILE_READONLY);
     DBMS_LOB.LOADCLOBFROMFILE(in_xml, xml_file, DBMS_LOB.LOBMAXSIZE,
        dest_offset, src_offset, 0, lang_context, warning);

     -- Validate the xml
     DBMS_CUBE.VALIDATE_XML(in_xml);
END;
/
```