6

Using the Oracle Data Pump API

You can automate data movement operations by using the Oracle Data Pump PL/SQL API `DBMS_DATAPUMP`.

The Oracle Data Pump API `DBMS_DATAPUMP` provides a high-speed mechanism that you can use to move all or part of the data and metadata for a site from one Oracle Database to another. The Oracle Data Pump Export and Oracle Data Pump Import utilities are based on the Oracle Data Pump API.

Oracle Database PL/SQL Packages and Types Reference

- How Does the Oracle Data Pump Client Interface API Work?
  The main structure used in the client interface is a job handle, which appears to the caller as an integer.
- DBMS_DATAPUMP Job States
  Use Oracle Data Pump `DBMS_DATAPUMP` job states show to know which stage your data movement job is performing, and what options are available at each stage.
- What Are the Basic Steps in Using the Oracle Data Pump API?
  To use the Oracle Data Pump API, you use the procedures provided in the `DBMS_DATAPUMP` package.
- Examples of Using the Oracle Data Pump API
  To get started using the Oracle Data Pump API, review examples that show what you can do with Oracle Data Pump exports and imports.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

6.1 How Does the Oracle Data Pump Client Interface API Work?

The main structure used in the client interface is a job handle, which appears to the caller as an integer.

Handles are created using the `DBMS_DATAPUMP.OPEN` or `DBMS_DATAPUMP.ATTACH` function. Other sessions can attach to a job to monitor and control its progress. Handles are session specific. The same job can create different handles in different sessions. As a DBA, the benefit of this feature is that you can start up a job before departing from work, and then watch the progress of the job from home.

6.2 DBMS_DATAPUMP Job States

Use Oracle Data Pump `DBMS_DATAPUMP` job states show to know which stage your data movement job is performing, and what options are available at each stage.

Job State Definitions

Each phase of a job is associated with a state:

- **Undefined** — before a handle is created

- **Defining** — when the handle is first created
- **Executing** — when the `DBMS_DATAPUMP.START_JOB` procedure is running
- **Completing** — when the job has finished its work and the Oracle Data Pump processes are ending
- **Completed** — when the job is completed
- **Stop Pending** — when an orderly job shutdown has been requested
- **Stopping** — when the job is stopping
- **Idling** — the period between the time that a `DBMS_DATAPUMP.ATTACH` is run to attach to a stopped job, and the time that a `DBMS_DATAPUMP.START_JOB` is run to restart that job
- **Not Running** — when a Data Pump control job table exists for a job that is not running (has no Oracle Data Pump processes associated with it)

Usage Notes

Performing `DBMS_DATAPUMP.START_JOB` on a job in an **Idling** state returns that job to an **Executing** state.

If all users run `DBMS_DATAPUMP.DETACH` to detach from a job in the **Defining** state, then the job is totally removed from the database.

If a job terminates unexpectedly, or if an instance running the job is shut down, and the job was previously in an **Executing** or **Idling** state, then the job is placed in the **Not Running** state. You can then restart the job.

The Oracle Data Pump control job process is active in the **Defining**, **Idling**, **Executing**, **Stopping**, **Stop Pending**, and **Completing** states. It is also active briefly in the **Stopped** and **Completed** states. The Data Pump control table for the job exists in all states except the **Undefined** state. Child processes are only active in the **Executing** and **Stop Pending** states, and briefly in the **Defining** state for import jobs.

Detaching while a job is in the **Executing** state does not halt the job. You can reattach to a running job at any time to resume obtaining status information about the job.

A Detach can occur explicitly, when the `DBMS_DATAPUMP.DETACH` procedure is run, or it can occur implicitly when an Oracle Data Pump API session is run down, when the Oracle Data Pump API is unable to communicate with an Oracle Data Pump job, or when the `DBMS_DATAPUMP.STOP_JOB` procedure is run.

The **Not Running** state indicates that a Data Pump control job table exists outside the context of a running job. This state occurs if a job is stopped (and likely can restart later), or if a job has terminated in an unusual way. You can also see this state momentarily during job state transitions at the beginning of a job, and at the end of a job before the Oracle Data Pump control job table is dropped. Note that the **Not Running** state is shown only in the views `DBA_DATAPUMP_JOBS` and `USER_DATAPUMP_JOBS`. It is never returned by the `GET_STATUS` procedure.

The following table shows the valid job states in which `DBMS_DATAPUMP` procedures can be run. The states listed are valid for both export and import jobs, unless otherwise noted.

Table 6-1 Valid Job States in Which DBMS_DATAPUMP Procedures Can Be Run

Procedure Name	Valid States	Description
ADD_FILE	**Defining** (valid for both export and import jobs) **Executing** and **Idling** (valid only for specifying dump files for export jobs)	Specifies a file for the dump file set, the log file, or the SQLFILE output.
ATTACH	**Defining**, **Executing**, **Idling**, **Stopped**, **Completed**, **Completing**, **Not Running**	Enables a user session to monitor a job, or to restart a stopped job. If the dump file set or Data Pump control job table for the job have been deleted or altered in any way, then the attach fails.
DATA_FILTER	**Defining**	Restricts data processed by a job.
DETACH	All	Disconnects a user session from a job.
GET_DUMPFILE_INFO	All	Retrieves dump file header information.
GET_STATUS	All, except **Completed**, **Not Running**, **Stopped**, and **Undefined**	Obtains the status of a job.
LOG_ENTRY	**Defining**, **Executing**, **Idling**, Stop **Pending**, **Completing**	Adds an entry to the log file.
METADATA_FILTER	**Defining**	Restricts metadata processed by a job.
METADATA_REMAP	**Defining**	Remaps metadata processed by a job.
METADATA_TRANSFORM	**Defining**	Alters metadata processed by a job.
OPEN	**Undefined**	Creates a new job.
SET_PARALLEL	Defining, Executing, Idling	Specifies parallelism for a job.
SET_PARAMETER	**Defining** Note: You can enter the ENCRYPTION_PASSWORD parameter during the **Defining** and **Idling** states.	Alters default processing by a job.
START_JOB	**Defining**, **Idling**	Begins or resumes execution of a job.
STOP_JOB	**Defining**, **Executing**, **Idling**, Stop **Pending**	Initiates shutdown of a job.
WAIT_FOR_JOB	All, except **Completed**, **Not Running**, **Stopped**, and **Undefined**	Waits for a job to end.

6.3 What Are the Basic Steps in Using the Oracle Data Pump API?

To use the Oracle Data Pump API, you use the procedures provided in the `DBMS_DATAPUMP` package.

The following steps list the basic activities involved in using the Data Pump API, including the point in running an Oracle Data Pump job in which you can perform optional steps. The steps are presented in the order in which you would generally perform the activities.

1. To create an Oracle Data Pump job and its infrastructure, run the `DBMS_DATAPUMP.OPEN` procedure.

   When you run the procedure, the Oracle Data Pump job is started.
2. Define any parameters for the job.
3. Start the job.
4. (Optional) Monitor the job until it completes.
5. (Optional) Detach from the job, and reattach at a later time.
6. (Optional) Stop the job.
7. (Optional) Restart the job, if desired.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

6.4 Examples of Using the Oracle Data Pump API

To get started using the Oracle Data Pump API, review examples that show what you can do with Oracle Data Pump exports and imports.

- Using the Oracle Data Pump API Examples with Your Database
  If you want to copy these scripts and run them, then you must complete setup tasks on your database before you run the scripts.
- Performing a Simple Schema Export with Oracle Data Pump
  See an example of how you can create, start, and monitor an Oracle Data Pump job to perform a schema export.
- Performing a Table Mode Export to Object Store with Oracle Data Pump
  See an example of how you can use `DBMS_DATAPUMP.ADD_FILE` to perform a table mode export.
- Importing a Dump File and Remapping All Schema Objects
  See an example of how you can create, start, and monitor an Oracle Data Pump job to import a dump file.
- Importing a Table from an Object Store Using Oracle Data Pump
  See an example of how you can create, start, and monitor an Oracle Data Pump job to import a table from an object store.
- Using Exception Handling During a Simple Schema Export
  See an example of how you can create, start, and monitor an Oracle Data Pump job to perform a schema export.

- Displaying Dump File Information for Oracle Data Pump Jobs
  See an example of how you can display information about an Oracle Data Pump dump file outside the context of any Data Pump job.
- Network Mode and Schema Mode Import Over a Network Link
  See an example of how you can perform a schema mode import in Network mode, over a network link.

6.4.1 Using the Oracle Data Pump API Examples with Your Database

If you want to copy these scripts and run them, then you must complete setup tasks on your database before you run the scripts.

The Oracle Data Pump API examples are in the form of PL/SQL scripts. To run these example scripts on your own database, You have to ensure that you have the required directory objects. permissions, roles, and display settings configured.

Example 6-1 Create a Directory Object and Grant READ AND WRITE Access

In this example, you create a directory object named `dmpdir` to which you have access, and then replace *`user`* with your username.

```
SQL> CREATE DIRECTORY dmpdir AS '/rdbms/work';
SQL> GRANT READ, WRITE ON DIRECTORY dmpdir TO user;
```

Example 6-2 Ensure You Have `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` Roles

To see a list of all roles assigned to you within your security domain, enter the following statement:

```
SQL> SELECT * FROM SESSION_ROLES;
```

Review the roles that you see displayed. If you do not have the `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles assigned to you, then contact your database administrator for help.

Example 6-3 Turn on Server Display Output

To see output display on your screen, ensure that server output is turned on. To do this, enter the following command:

```
SQL> SET SERVEROUTPUT ON
```

If server display output is not turned on, then output is not displayed to your screen. You must set the display output to `ON` in the same session in which you run the example. If you exit SQL*Plus, then this setting is lost and must be reset when you begin a new session. If you connect to the database using a different user name, then you must also reset `SERVEROUTPUT` to `ON` for that user.

6.4.2 Performing a Simple Schema Export with Oracle Data Pump

See an example of how you can create, start, and monitor an Oracle Data Pump job to perform a schema export.

The PL/SQL script in this example shows how to use the Oracle Data Pump API to perform a simple schema export of the `HR` schema. The example shows how to create a job, start it, and monitor it. Additional information about the example is contained in the comments within the

script. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call `GET_STATUS` to retrieve more detailed error information when a failure occurs.

Example 6-4 Performing a Simple Schema Export

Connect as user `SYSTEM` to use this script.

```
DECLARE
  ind NUMBER;              -- Loop index
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a schema export.

  h1 := DBMS_DATAPUMP.OPEN('EXPORT','SCHEMA',NULL,'EXAMPLE1','LATEST');

-- Specify a single dump file for the job (using the handle just returned)
-- and a directory object, which must already be defined and accessible
-- to the user running this procedure.

  DBMS_DATAPUMP.ADD_FILE(h1,'example1.dmp','DMPDIR');

-- A metadata filter is used to specify the schema that will be exported.

  DBMS_DATAPUMP.METADATA_FILTER(h1,'SCHEMA_EXPR','IN (''HR'')');

-- Start the job. An exception will be generated if something is not set up
-- properly.

  DBMS_DATAPUMP.START_JOB(h1);

-- The export job should now be running. In the following loop, the job
-- is monitored until it completes. In the meantime, progress information is
-- displayed.

  percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

    if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
```

```
                            to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;

-- If any work-in-progress (WIP) or error messages were received for the job,
-- display them.

   if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
      else
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;

-- Indicate that the job finished and detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);
END;
/
```

6.4.3 Performing a Table Mode Export to Object Store with Oracle Data Pump

See an example of how you can use `DBMS_DATAPUMP.ADD_FILE` to perform a table mode export.

In this PL/SQL script, the Oracle Data Pump `DBMS_DATAPUMP` API uses the `ADD_FILE` call to specify the object-store URI, credential and filetype in a table export. It shows how to create a job, start it, and monitor it. Additional information about the example is contained in the comments within the script. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call `GET_STATUS` to retrieve more detailed error information when a failure occurs.

> **Note:**
>
> All credential, object-store, and network ACLS setup, and so on, are presumed to be in place, and therefore are not included in the scripts.

In comparison to an Oracle Data Pump script to perform an expert for an on oremises system, note the differences in the script in the call:

```
dbms_datapump.add_file(hdl, dumpFile, credName, '3MB', dumpType, 1);
```

Where the procedure parameter *filename* (`dumpFile`) contains the object store URI, *directory* (`credName`) contains the credential, and *filetype* (`dumpType`) contains a new filetype keyword

Note the following calls:

```
DBMS_DATAPUMP.ADD_FILE ( handle IN NUMBER, filename IN VARCHAR2,
directory IN VARCHAR2, filesize IN VARCHAR2 DEFAULT NULL, filetype IN NUMBER
DEFAULT
DBMS_DATAPUMP.KU$_FILE_TYPE_DUMP_FILE, reusefile IN NUMBER DEFAULT NULL);
```

And note the object store definitions in the script:

```
dumpFile     VARCHAR2(1024)  := 'https://example.oraclecloud.com/test/
den02ten_foo3b_split_%u.dat';
  dumpType     NUMBER          := dbms_datapump.ku$_file_type_uridump_file;
```

Example 6-5 Table Mode Export to Object Store

This table mode export example assumes that object store credentials, network ACLs, the database account and object-store information is already set up.

```
Rem
Rem
Rem tkdpose.sql
Rem
Rem    NAME
Rem      tkdpose.sql - <one-line expansion of the name>
Rem
Rem    DESCRIPTION
Rem      Performs a table mode export to the object store.
Rem
Rem    NOTES
Rem      Assumes that credentials, network ACLs, database account and
Rem      object-store information already been setup.
Rem

connect test/mypwd@CDB1_PDB1

SET SERVEROUTPUT ON
SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 80
SET TRIMSPOOL ON
SET TAB OFF
SET PAGESIZE 100

DECLARE
  hdl          NUMBER;         -- Datapump handle
```

```
  ind           NUMBER;         -- Loop index
  le            ku$_LogEntry;   -- For WIP and error messages
  js            ku$_JobStatus;  -- The job status from get_status
  jd            ku$_JobDesc;    -- The job description from get_status
  sts           ku$_Status;     -- The status object returned by get_status
  jobState      VARCHAR2(30);   -- To keep track of job state
  dumpType      NUMBER          := dbms_datapump.ku$_file_type_uridump_file;
  dumpFile      VARCHAR2(1024)  := 'https://example.oraclecloud.com/test/
den02ten_foo3b_split_%u.dat';
  dumpType      NUMBER          := dbms_datapump.ku$_file_type_uridump_file;
  credName      VARCHAR2(1024)  := 'BMCTEST';
  logFile       VARCHAR2(1024)  := 'tkopc_export3b_cdb2.log';
  logDir        VARCHAR2(9)     := 'WORK';
  logType       NUMBER          := dbms_datapump.ku$_file_type_log_file;

BEGIN

  --
  -- Open a schema-based export job and perform defining-phase initialization.
  --
  hdl := dbms_datapump.open('EXPORT', 'TABLE');
  dbms_datapump.set_parameter(hdl, 'COMPRESSION', 'ALL');
  dbms_datapump.set_parameter(hdl, 'CHECKSUM', 1);
  dbms_datapump.add_file(hdl, logfile, logdir, null, logType);
  dbms_datapump.add_file(hdl, dumpFile, credName, '3MB', dumpType, 1);
  dbms_datapump.data_filter(hdl, 'INCLUDE_ROWS', 1);
  dbms_datapump.metadata_filter(hdl, 'TABLE_FILTER', 'FOO', '');
  --
  -- Start the job.
  --
  dbms_datapump.start_job(hdl);

  --
  -- Now grab output from the job and write to standard out.
  --
  jobState := 'UNDEFINED';
  WHILE (jobState != 'COMPLETED') AND (jobState != 'STOPPED')
  LOOP
    dbms_datapump.get_status(hdl,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip, -1, jobState,sts);
    js := sts.job_status;

    --
    -- If we received any WIP or Error messages for the job, display them.
    --
    IF (BITAND(sts.mask,dbms_datapump.ku$_status_wip) != 0) THEN
      le := sts.wip;
    ELSE
      IF (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0) THEN
        le := sts.error;
      ELSE
        le := NULL;
      END IF;
    END IF;
```

```
    IF le IS NOT NULL THEN
      ind := le.FIRST;
      WHILE ind IS NOT NULL LOOP
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      END LOOP;
    END IF;
  END LOOP;

  --
  -- Detach from job.
  --
  dbms_datapump.detach(hdl);

  --
  -- Any exceptions that propagated to this point will be captured.
  -- The details are retrieved from get_status and displayed.
  --
  EXCEPTION
    WHEN OTHERS THEN
      BEGIN
        dbms_datapump.get_status(hdl, dbms_datapump.ku$_status_job_error, 0,
                                 jobState, sts);
        IF (BITAND(sts.mask,dbms_datapump.ku$_status_job_error) != 0) THEN
          le := sts.error;
          IF le IS NOT NULL THEN
            ind := le.FIRST;
            WHILE ind IS NOT NULL LOOP
              dbms_output.put_line(le(ind).LogText);
              ind := le.NEXT(ind);
            END LOOP;
          END IF;
        END IF;

      BEGIN
        dbms_datapump.stop_job (hdl, 1, 0, 0);
      EXCEPTION
        WHEN OTHERS THEN NULL;
      END;

      EXCEPTION
      WHEN OTHERS THEN
        dbms_output.put_line('Unexpected exception while in exception ' ||
                             'handler. sqlcode = ' || TO_CHAR(SQLCODE));
      END;
END;
/
EXIT;
```

The log reports the following information:

```
Starting "TEST"."SYS_EXPORT_TABLE_01":
Processing object type TABLE_EXPORT/TABLE/TABLE_DATA
```

```
Processing object type TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
Processing object type TABLE_EXPORT/TABLE/STATISTICS/MARKER
Processing object type TABLE_EXPORT/TABLE/TABLE
. . exported "TEST"."FOO"                                147.8 KB   70000 rows
Master table "TEST"."SYS_EXPORT_TABLE_01" successfully loaded/unloaded
Generating checksums for dump file set
******************************************************************************
Dump file set for TEST.SYS_EXPORT_TABLE_01 is:
  https://example.oraclecloud.com/test/den02ten_foo3b_split_01.dat
Job "TEST"."SYS_EXPORT_TABLE_01" successfully completed at Sun Dec 13
22:22:30 2020 elapsed 0 00:00:22
```

6.4.4 Importing a Dump File and Remapping All Schema Objects

See an example of how you can create, start, and monitor an Oracle Data Pump job to import a dump file.

The script in this example imports the dump file created in the Oracle Data Pump API example "Performing a Simple Schema Export with Oracle Data Pump" (an export of the `hr` schema). All schema objects are remapped from the `hr` schema to the `blake` schema. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call `GET_STATUS` to retrieve more detailed error information when a failure occurs.

Example 6-6 Importing the dump file and remapping all schema objects

Connect as user `SYSTEM` to use this script.

```
DECLARE
  ind NUMBER;              -- Loop index
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a "full" import (everything
-- in the dump file without filtering).

  h1 := DBMS_DATAPUMP.OPEN('IMPORT','FULL',NULL,'EXAMPLE2');

-- Specify the single dump file for the job (using the handle just returned)
-- and directory object, which must already be defined and accessible
-- to the user running this procedure. This is the dump file created by
-- the export operation in the first example.

  DBMS_DATAPUMP.ADD_FILE(h1,'example1.dmp','DMPDIR');

-- A metadata remap will map all schema objects from HR to BLAKE.

  DBMS_DATAPUMP.METADATA_REMAP(h1,'REMAP_SCHEMA','HR','BLAKE');

-- If a table already exists in the destination schema, skip it (leave
```

```
-- the preexisting table alone). This is the default, but it does not hurt
-- to specify it explicitly.

  DBMS_DATAPUMP.SET_PARAMETER(h1,'TABLE_EXISTS_ACTION','SKIP');

-- Start the job. An exception is returned if something is not set up
properly.

  DBMS_DATAPUMP.START_JOB(h1);

-- The import job should now be running. In the following loop, the job is
-- monitored until it completes. In the meantime, progress information is
-- displayed. Note: this is identical to the export example.

 percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

     if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
                           to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;

-- If any work-in-progress (WIP) or Error messages were received for the job,
-- display them.

       if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
      else
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;
```

```
-- Indicate that the job finished and gracefully detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);
END;
/
```

6.4.5 Importing a Table from an Object Store Using Oracle Data Pump

See an example of how you can create, start, and monitor an Oracle Data Pump job to import a table from an object store.

In this PL/SQL script, the Oracle Data Pump DBMS_DATAPUMP API uses the ADD_FILE call to specify the object-store URI, credential and filetype in a table export. It shows how to create a job, start it, and monitor it. Additional information about the example is contained in the comments within the script. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call `GET_STATUS` to retrieve more detailed error information when a failure occurs.

> **Note:**
>
> All credential, object-store, and network ACLS setup, and so on, are presumed to be in place, and therefore are not included in the scripts.

Example 6-7 Table Mode Import to Object Store

This table mode import example assumes that object store credentials, network ACLs, the database account and object-store information is already set up.

```
Rem    NAME
Rem      tkdposi.sql
Rem
Rem    DESCRIPTION
Rem      Performs a table mode import from the object-store.
Rem

connect test/mypwd@CDB1_PDB1

SET SERVEROUTPUT ON
SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 80
SET TRIMSPOOL ON
SET TAB OFF
SET PAGESIZE 100

DECLARE
  hdl          NUMBER;         -- Datapump handle
  ind          NUMBER;         -- Loop index
  le           ku$_LogEntry;   -- For WIP and error messages
```

```
  js           ku$_JobStatus;  -- The job status from get_status
  jd           ku$_JobDesc;    -- The job description from get_status
  sts          ku$_Status;     -- The status object returned by get_status
  jobState     VARCHAR2(30);   -- To keep track of job state
  dumpFile     VARCHAR2(1024)  := 'https://example.oraclecloud.com/test/
den02ten_foo3b_split_%u.dat';
  dumpType     NUMBER          := dbms_datapump.ku$_file_type_uridump_file;
  credName     VARCHAR2(1024)  := 'BMCTEST';
  logFile      VARCHAR2(1024)  := 'tkopc_import3b_cdb2.log';
  logDir       VARCHAR2(9)     := 'WORK';
  logType      NUMBER          := dbms_datapump.ku$_file_type_log_file;

BEGIN

  --
  -- Open a schema-based export job and perform defining-phase initialization.
  --
  hdl := dbms_datapump.open('IMPORT', 'TABLE', NULL, 'OSI');
  dbms_datapump.add_file(hdl, logfile, logdir, null, logType);
  dbms_datapump.add_file(hdl, dumpFile, credName, null, dumpType);
  dbms_datapump.metadata_filter(hdl, 'TABLE_FILTER', 'FOO', '');
  dbms_datapump.set_parameter(hdl, 'TABLE_EXISTS_ACTION', 'REPLACE');
  dbms_datapump.set_parameter(hdl, 'VERIFY_CHECKSUM', 1);

  --
  -- Start the job.
  --
  dbms_datapump.start_job(hdl);

  --
  -- Now grab output from the job and write to standard out.
  --
  jobState := 'UNDEFINED';
  WHILE (jobState != 'COMPLETED') AND (jobState != 'STOPPED')
  LOOP
    dbms_datapump.get_status(hdl,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip, -1, jobState,sts);
    js := sts.job_status;

    --
    -- If we received any WIP or Error messages for the job, display them.
    --
    IF (BITAND(sts.mask,dbms_datapump.ku$_status_wip) != 0) THEN
      le := sts.wip;
    ELSE
      IF (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0) THEN
        le := sts.error;
      ELSE
        le := NULL;
      END IF;
    END IF;

    IF le IS NOT NULL THEN
      ind := le.FIRST;
```

```
      WHILE ind IS NOT NULL LOOP
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      END LOOP;
    END IF;
  END LOOP;

  --
  -- Detach from job.
  --
  dbms_datapump.detach(hdl);

  --
  -- Any exceptions that propagated to this point will be captured.
  -- The details are retrieved from get_status and displayed.
  --
  EXCEPTION
    WHEN OTHERS THEN
      BEGIN
        dbms_datapump.get_status(hdl, dbms_datapump.ku$_status_job_error, 0,
                                 jobState, sts);
        IF (BITAND(sts.mask,dbms_datapump.ku$_status_job_error) != 0) THEN
          le := sts.error;
          IF le IS NOT NULL THEN
            ind := le.FIRST;
            WHILE ind IS NOT NULL LOOP
              dbms_output.put_line(le(ind).LogText);
              ind := le.NEXT(ind);
            END LOOP;
          END IF;
        END IF;

      BEGIN
        dbms_datapump.stop_job (hdl, 1, 0, 0);
      EXCEPTION
        WHEN OTHERS THEN NULL;
      END;

      EXCEPTION
      WHEN OTHERS THEN
        dbms_output.put_line('Unexpected exception while in exception ' ||
                             'handler. sqlcode = ' || TO_CHAR(SQLCODE));
      END;
END;
/
EXIT;
```

The log file reports the following information:

```
Verifying dump file checksums
Master table "TEST"."OSI" successfully loaded/unloaded
Starting "TEST"."OSI":
Processing object type TABLE_EXPORT/TABLE/TABLE
Processing object type TABLE_EXPORT/TABLE/TABLE_DATA
```

```
. . imported "TEST"."FOO"                                147.8 KB   70000 rows
Processing object type TABLE_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
Processing object type TABLE_EXPORT/TABLE/STATISTICS/MARKER
;;; Ext Tbl Query Coord.: worker id 1 for "SYS"."IMPDP_STATS"
;;; Ext Tbl Query Coord.: worker id 1 for "SYS"."IMPDP_STATS"
;;; Ext Tbl Shadow: worker id 1 for "SYS"."IMPDP_STATS"
Job "TEST"."OSI" successfully completed at Sun Dec 13 22:24:16 2020 elapsed 0
00:00:40
```

6.4.6 Using Exception Handling During a Simple Schema Export

See an example of how you can create, start, and monitor an Oracle Data Pump job to perform a schema export.

The script in this example shows a simple schema export using the Data Pump API. It extends the example shown in "Performing a Simple Schema Export with Oracle Data Pump" to show how to use exception handling to catch the `SUCCESS_WITH_INFO` case, and how to use the `GET_STATUS` procedure to retrieve additional information about errors. To obtain exception information about a `DBMS_DATAPUMP.OPEN` or `DBMS_DATAPUMP.ATTACH` failure, call `DBMS_DATAPUMP.GET_STATUS` with a `DBMS_DATAPUMP.KU$_STATUS_JOB_ERROR` information mask and a NULL job handle to retrieve the error details.

Example 6-8 Exception handing in simple schema export using the Data Pump API

Connect as user `SYSTEM` to use this script.

```
DECLARE
  ind NUMBER;              -- Loop index
  spos NUMBER;             -- String starting position
  slen NUMBER;             -- String length for output
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a schema export.

  h1 := dbms_datapump.open('EXPORT','SCHEMA',NULL,'EXAMPLE3','LATEST');

-- Specify a single dump file for the job (using the handle just returned)
-- and a directory object, which must already be defined and accessible
-- to the user running this procedure.

  dbms_datapump.add_file(h1,'example3.dmp','DMPDIR');

-- A metadata filter is used to specify the schema that will be exported.

  dbms_datapump.metadata_filter(h1,'SCHEMA_EXPR','IN (''HR'')');

-- Start the job. An exception will be returned if something is not set up
-- properly.One possible exception that will be handled differently is the
-- success_with_info exception. success_with_info means the job started
```

```
-- successfully, but more information is available through get_status about
-- conditions around the start_job that the user might want to be aware of.

    begin
    dbms_datapump.start_job(h1);
    dbms_output.put_line('Data Pump job started successfully');
    exception
      when others then
        if sqlcode = dbms_datapump.success_with_info_num
        then
          dbms_output.put_line('Data Pump job started with info available:');
          dbms_datapump.get_status(h1,
                                   dbms_datapump.ku$_status_job_error,0,
                                   job_state,sts);
          if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
          then
            le := sts.error;
            if le is not null
            then
              ind := le.FIRST;
              while ind is not null loop
                dbms_output.put_line(le(ind).LogText);
                ind := le.NEXT(ind);
              end loop;
            end if;
          end if;
        else
          raise;
        end if;
  end;

-- The export job should now be running. In the following loop,
-- the job is monitored until it completes. In the meantime, progress
information -- is displayed.

 percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

     if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
                           to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;

-- Display any work-in-progress (WIP) or error messages that were received for
-- the job.
```

```
      if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
      else
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;

-- Indicate that the job finished and detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);

-- Any exceptions that propagated to this point will be captured. The
-- details will be retrieved from get_status and displayed.

  exception
    when others then
      dbms_output.put_line('Exception in Data Pump job');
      dbms_datapump.get_status(h1,dbms_datapump.ku$_status_job_error,0,
                               job_state,sts);
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
        if le is not null
        then
          ind := le.FIRST;
          while ind is not null loop
            spos := 1;
            slen := length(le(ind).LogText);
            if slen > 255
            then
              slen := 255;
            end if;
            while slen > 0 loop
              dbms_output.put_line(substr(le(ind).LogText,spos,slen));
              spos := spos + 255;
              slen := length(le(ind).LogText) + 1 - spos;
            end loop;
            ind := le.NEXT(ind);
          end loop;
        end if;
```

```
      end if;
END;
/
```

6.4.7 Displaying Dump File Information for Oracle Data Pump Jobs

See an example of how you can display information about an Oracle Data Pump dump file outside the context of any Data Pump job.

The PL/SQL script in this example shows how to use the Oracle Data Pump API procedure `DBMS_DATAPUMP.GET_DUMPFILE_INFO` to display information about a Data Pump dump file at any point, not just when you are running the job. This example displays information contained in the dump file `example1.dmp` dump file created by the example PL/SQL script in "Performing a Simple Schema Export with Oracle Data Pump."

You can also use this PL/SQL script to display information for dump files created by original Export (the `exp` utility), as well as by the `ORACLE_DATAPUMP` external tables access driver.

Example 6-9 Using the Oracle Data Pump API procedure to display dumpfile information

Connect as user `SYSTEM` to use this script.

```
SET VERIFY OFF
SET FEEDBACK OFF

DECLARE
  ind        NUMBER;
  fileType   NUMBER;
  value      VARCHAR2(2048);
  infoTab    KU$_DUMPFILE_INFO := KU$_DUMPFILE_INFO();

BEGIN
  --
  -- Get the information about the dump file into the infoTab.
  --
  BEGIN
    DBMS_DATAPUMP.GET_DUMPFILE_INFO('example1.dmp','DMPDIR',infoTab,fileType);
    DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
    DBMS_OUTPUT.PUT_LINE('Information for file: example1.dmp');

    --
    -- Determine what type of file is being looked at.
    --
    CASE fileType
      WHEN 1 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is a Data Pump dump file');
      WHEN 2 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is an Original Export dump file');
      WHEN 3 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is an External Table dump file');
      ELSE
        DBMS_OUTPUT.PUT_LINE('example1.dmp is not a dump file');
        DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
    END CASE;
```

```
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('--------------------------------------------');
    DBMS_OUTPUT.PUT_LINE('Error retrieving information for file: ' ||
                         'example1.dmp');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    DBMS_OUTPUT.PUT_LINE('--------------------------------------------');
    fileType := 0;
END;

--
-- If a valid file type was returned, then loop through the infoTab and
-- display each item code and value returned.
--
IF fileType > 0
THEN
  DBMS_OUTPUT.PUT_LINE('The information table has ' ||
                       TO_CHAR(infoTab.COUNT) || ' entries');
  DBMS_OUTPUT.PUT_LINE('--------------------------------------------');

  ind := infoTab.FIRST;
  WHILE ind IS NOT NULL
  LOOP
    --
    -- The following item codes return boolean values in the form
    -- of a '1' or a '0'. Display them as 'Yes' or 'No'.
    --
    value := NVL(infoTab(ind).value, 'NULL');
    IF infoTab(ind).item_code IN
       (DBMS_DATAPUMP.KU$_DFHDR_MASTER_PRESENT,
        DBMS_DATAPUMP.KU$_DFHDR_DIRPATH,
        DBMS_DATAPUMP.KU$_DFHDR_METADATA_COMPRESSED,
        DBMS_DATAPUMP.KU$_DFHDR_DATA_COMPRESSED,
        DBMS_DATAPUMP.KU$_DFHDR_METADATA_ENCRYPTED,
        DBMS_DATAPUMP.KU$_DFHDR_DATA_ENCRYPTED,
        DBMS_DATAPUMP.KU$_DFHDR_COLUMNS_ENCRYPTED)
    THEN
      CASE value
        WHEN '1' THEN value := 'Yes';
        WHEN '0' THEN value := 'No';
      END CASE;
    END IF;

    --
    -- Display each item code with an appropriate name followed by
    -- its value.
    --
    CASE infoTab(ind).item_code
      --
      -- The following item codes have been available since Oracle
      -- Database 10g, Release 10.2.
      --
      WHEN DBMS_DATAPUMP.KU$_DFHDR_FILE_VERSION   THEN
        DBMS_OUTPUT.PUT_LINE('Dump File Version:         ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PRESENT THEN
        DBMS_OUTPUT.PUT_LINE('Master Table Present:      ' || value);
```

```
WHEN DBMS_DATAPUMP.KU$_DFHDR_GUID THEN
  DBMS_OUTPUT.PUT_LINE('Job Guid:                    ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_FILE_NUMBER THEN
  DBMS_OUTPUT.PUT_LINE('Dump File Number:            ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_CHARSET_ID  THEN
  DBMS_OUTPUT.PUT_LINE('Character Set ID:            ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_CREATION_DATE THEN
  DBMS_OUTPUT.PUT_LINE('Creation Date:               ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_FLAGS THEN
  DBMS_OUTPUT.PUT_LINE('Internal Dump Flags:         ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_JOB_NAME THEN
  DBMS_OUTPUT.PUT_LINE('Job Name:                    ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_PLATFORM THEN
  DBMS_OUTPUT.PUT_LINE('Platform Name:               ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_INSTANCE THEN
  DBMS_OUTPUT.PUT_LINE('Instance Name:               ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_LANGUAGE THEN
  DBMS_OUTPUT.PUT_LINE('Language Name:               ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_BLOCKSIZE THEN
  DBMS_OUTPUT.PUT_LINE('Dump File Block Size:        ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_DIRPATH THEN
  DBMS_OUTPUT.PUT_LINE('Direct Path Mode:            ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_METADATA_COMPRESSED THEN
  DBMS_OUTPUT.PUT_LINE('Metadata Compressed:         ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_DB_VERSION THEN
  DBMS_OUTPUT.PUT_LINE('Database Version:            ' || value);

--
-- The following item codes were introduced in Oracle Database 11g
-- Release 11.1
--

WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PIECE_COUNT THEN
  DBMS_OUTPUT.PUT_LINE('Master Table Piece Count:  ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PIECE_NUMBER THEN
  DBMS_OUTPUT.PUT_LINE('Master Table Piece Number: ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_DATA_COMPRESSED THEN
  DBMS_OUTPUT.PUT_LINE('Table Data Compressed:     ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_METADATA_ENCRYPTED THEN
  DBMS_OUTPUT.PUT_LINE('Metadata Encrypted:        ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_DATA_ENCRYPTED THEN
  DBMS_OUTPUT.PUT_LINE('Table Data Encrypted:      ' || value);
WHEN DBMS_DATAPUMP.KU$_DFHDR_COLUMNS_ENCRYPTED THEN
  DBMS_OUTPUT.PUT_LINE('TDE Columns Encrypted:     ' || value);

--
-- For the DBMS_DATAPUMP.KU$_DFHDR_ENCRYPTION_MODE item code a
-- numeric value is returned. So examine that numeric value
-- and display an appropriate name value for it.
--
WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCRYPTION_MODE THEN
  CASE TO_NUMBER(value)
    WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_NONE THEN
      DBMS_OUTPUT.PUT_LINE('Encryption Mode:           None');
    WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_PASSWORD THEN
      DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Password');
```

```
            WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_DUAL THEN
              DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Dual');
            WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_TRANS THEN
              DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Transparent');
          END CASE;

        --
        -- The following item codes were introduced in Oracle Database 12c
        -- Release 12.1
        --

        --
        -- For the DBMS_DATAPUMP.KU$_DFHDR_COMPRESSION_ALG item code a
        -- numeric value is returned. So examine that numeric value and
        -- display an appropriate name value for it.
        --
        WHEN DBMS_DATAPUMP.KU$_DFHDR_COMPRESSION_ALG THEN
          CASE TO_NUMBER(value)
            WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_NONE THEN
              DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     None');
            WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_BASIC THEN
              DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Basic');
            WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_LOW THEN
              DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Low');
            WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_MEDIUM THEN
              DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Medium');
            WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_HIGH THEN
              DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     High');
          END CASE;
        ELSE NULL;  -- Ignore other, unrecognized dump file attributes.
      END CASE;
      ind := infoTab.NEXT(ind);
    END LOOP;
  END IF;
END;
/
```

6.4.8 Network Mode and Schema Mode Import Over a Network Link

See an example of how you can perform a schema mode import in Network mode, over a network link.

To pass schemas between two databases over a network, you can create a network link and then use Oracle Data Pump API `DBMS_DATAPUMP` to perform a schema mode import over a network link.

Before you begin, you must have created a network link using the `CREATE DATABASE LINK` statement to create a database link.

Example 6-10 Performing an Oracle Data Pump schema mode import over a network link using DBMS_DATAPUMP

```
-- This example will perform a Data Pump schema mode import over a network
link, a network mode import

-- Define a handle for the job
```

```
Declare
 h1 number;

-- Print out an alert that the job is beginning
begin
  dbms_output.put_line('Starting import job over network link');

-- Define a schema level network mode import using the previously defined
network link, DBS1
  h1 := dbms_datapump.open('IMPORT','SCHEMA','DBS1',
'EXAMPLE_IMPORT_NETWORK_MODE);

-- Import the schema HR.
  dbms_datapump.metadata_filter(h1,'SCHEMA_EXPR','IN (''HR'')');

-- Start the job
  dbms_datapump.start_job(h1);
end;
exit
```

Related Topics

- CREATE DATABASE LINK
- DBMS_METADATA