PROGRAMMING ORACLE DATA PUMP

Sandi Reddick, OraPro.net

INTRODUCTION

Oracle Data Pump, first introduced with version 10g Release 1, enables high speed loading and unloading of data and metadata between databases. Data Pump is an integral feature of the Oracle Database and its robust feature set combined with ease of use has enabled it to quickly obsolete the older EXP and IMP utilities. Although some Data Pump features require specific licensing, such as parallelism which requires Enterprise Edition or compression which requires the Advanced Compression Option, the bulk of the Data Pump feature set is available with all database versions.

Data Pump can be used to export a filtered data subset to a file, import a table directly from another database or extract metadata in the form of SQL scripts. However, if you are only using the command line Data Pump clients impdp and expdp, then you aren't taking full advantage of all that Data Pump has to offer. This paper will provide a review of the Data Pump API, dbms_datapump, to show you how this API can be leveraged to create fully automated and robust export/import jobs using 100% PL/SQL. Note that while this paper will focus exclusively on the use of PL/SQL to interface with Data Pump, any programming environment that supports SQL function and procedure calls can be used to do the same tasks following similar methods.

BACKGROUND

The primary user interfaces for interacting with Data Pump are the command-line export and import utilities, expdp and impdp. These interfaces provide limited automation capability through the use of shell or batch scripts. There is also an Html user interface (HUI) available within Oracle Enterprise Manager (OEM). While job definition within OEM can be accomplished with point-and-click simplicity, it is impossible to create 'smart' jobs that can reconfigure themselves based on export/import conditions at run-time. This is where the use of the Data Pump API shines.

If you've ever wanted to:

- export a table *or* just a subset of data from the table *if* greater than a certain size,
- exclude a table from a data-only direct database import *if* the table doesn't exist at the destination,
- get out of the shell/batch scripting business and manage all your code in one place, in the database,

then dbms_datapump is the tool for you.

It is interesting to note that the underlying foundation for the Oracle provided Data Pump interfaces, expdp, impdp and the web-based interface in OEM, *is* dbms_datapump. When you run an expdp-based export job, expdp is making calls to dbms_datapump on your behalf. Many Oracle features also rely on dbms_datapump to do their work, such as Logical Standby, Transportable Tablespaces, Streams-based replication and others.

DBMS_DATAPUMP PREREQUISITES

Before beginning, there are a few housekeeping items to be concerned with. The first items to address are the permissions that will be required to execute functions and procedures within the dbms_datapump package. If your user is not the 'SYS' user, you will need to ensure that user is granted the EXP_FULL_DATABASE role to perform exports and/or the IMP_FULL_DATABASE role to perform imports or generate SQL files. You'll also need to ensure that at least one DIRECTORY object is created and that your user has the appropriate read/write privileges to it, since Data Pump requires the use of DIRECTORY objects when defining file paths.

DBMS_DATAPUMP PACKAGE OVERVIEW

Let's begin with a survey of the available functions and procedures within the dbms_datapump package and a brief description of each.

BASIC JOB CONTROL FUNCTIONS & PROCEDURES

Function Name	Return type	Description
OPEN	NUMBER	Call to create a new job. Returns job handle.
ATTACH	NUMBER	Establish access to a previously-created job. Returns job handle.

Procedure Name	Description
ADD_FILE	Define dump, log and/or SQL output files.
SET_PARALLEL	Define maximum number of worker processes than can be spawned for the job. EE only.
START_JOB	Begin or resume execution of a defined job and return program control.
STOP_JOB	Terminate a job, optionally preserving job state info.
WAIT_FOR_JOB	Begin or resume execution of a defined job but do not return program control until job halts.
DETACH	Releases job handle, ending access to job without terminating it.

JOB DEFINITION PROCEDURES

Procedure Name	Description
DATA_FILTER	Define filters to restrict table row data included during import or export.
METADATA_FILTER	Define filters to restrict objects included in job.
METADATA_REMAP	Define object remapping during import or SQL file generation such as schema or tablespace remapping.
METADATA_TRANSFORM	Define object transformations during import or SQL file generation such as storage parameter or object id reassignment.
SET_PARAMETER	Define optional job parameters such as those used to specify encryption or compression, SCN, estimate, table_exists_actions.

ADVANCED JOB CONTROL AND MONITORING

Procedure Name	Description
GET_DUMPFILE_INFO	Get information about a dump file such as file type, database version, character set.
GET_STATUS	Monitor the progress of a job and retrieve error messages.
LOG_ENTRY	Add text to a log file and optionally display to attached users.

DATA PUMP API BASICS

You can define and execute a Data Pump job using the functions and procedures defined by the dbms_datapump package with relatively little coding. Of the API calls available, only 3 must be made to define the simplest of jobs. Additional API calls can be made to refine the import or export object selection, to monitor job execution, attach or detach from jobs and stop/restart executing jobs. Begin by looking at the simplest example, a full database export. In the following example, only 3 calls are required: OPEN, to create the Data Pump job; ADD_FILE, to define the dumpfile filename; and START_JOB, to begin execution of the job.

STEP 1: CREATE THE DATA PUMP JOB

To create a Data Pump job, you must make a call to OPEN. Here is the function declaration for OPEN:

```
dbms_datapump.OPEN (
  operation     IN VARCHAR2,
  job_mode      IN VARCHAR2,
  remote_link   IN VARCHAR2 DEFAULT NULL,
  job_name      IN VARCHAR2 DEFAULT NULL,
  version       IN VARCHAR2 DEFAULT 'COMPATIBLE',
  compression   IN NUMBER DEFAULT  dbms_datapump.ku$_compress_metadata)
RETURN NUMBER;
```

*Note the compression parameter shown in italics. This parameter is undocumented.

When the OPEN function is called, the Data Pump master table for the job is created. If successful, OPEN will return a NUMBER data type. The value that is returned is a handle, used to reference this job in future API calls from within the same session. Only two input parameters are required, operation and job_mode, since the others have declared default values.

Valid operation types are:

Value	Description
EXPORT	Create data and/or metadata dump files, or simply estimate the size of an export dump job.
IMPORT	Restore data and/or metadata from dump files or a remote database.
SQL_FILE	Create a metadata SQL script from dump files or a remote database.

Valid job modes are:

Value	Description
FULL	Includes all schemas except: SYS, XDB, ORDSYS, MDSYS, CTXSYS, ORDPLUGINS and LBACSYS.
SCHEMA	Define job to include a set of schemas. If no schema is identified, defaults to current user's schema. SYS, XDB, ORDSYS, MDSYS, CTXSYS, ORDPLUGINS and LBACSYS schemas maybe not be specified for this mode.
TABLE	Define job to include a set of tables. If no tables are identified, default to all tables in the current user's schema.
TABLESPACE	Define job to include a set of tablespaces. All tables stored in the defined tablespace(s) will be included.
TRANSPORTABLE	Defines job to process metadata for objects required for a transportable tablespace export or import.

In this first dbms_datapump example, you will define a full database export. The call to open can be made using only the two required input parameters. However, if the optional job name parameter is not provided, one will be assigned. Since the job name will need to be known in order to reattach to the Data Pump job, it is a good idea to explicitly assign a name to the job. To create the job for a simple full export:

Code snippet #1:

```
dbms_datapump.OPEN (
   operation   => 'EXPORT',
   job_mode    => 'FULL',
   job_name    => 'FULL_DB_EXP');
```

STEP 2: DEFINE THE OUTPUT FILES

The next step in defining a basic Data Pump export job is to define the output file. The dump file is the only output file that *must* be declared for any Data Pump export. This is defined by calling the ADD_FILE procedure. Here is the procedure declaration for ADD_FILE:

```
dbms_datapump.ADD_FILE (
   handle          IN NUMBER,
   filename        IN VARCHAR2,
   directory       IN VARCHAR2 DEFAULT NULL,
   filesize        IN VARCHAR2 DEFAULT NULL,
   filetype        IN NUMBER DEFAULT dbms_datapump.ku$_file_type_dump_file,
   reusefile       IN NUMBER   DEFAULT NULL);
```

Similar to the call to OPEN above, only two input parameters are required. The first parameter should contain the handle that was returned from the call to OPEN the job. The second parameter is used to identify the filename for your export dumpfile. In its simplest usage, to define the export dumpfile:

Code snippet #2:

```
dbms_datapump.ADD_FILE(
   handle     => h1,
   filename   => 'FULL_DB_EXP.DMP');
```

If no directory is defined, the directory will default to NULL and Data Pump will use the Oracle-created default DATA_PUMP_DIR directory object to identify the file path. A filesize parameter can be defined, applicable to export only, to limit the size of the dump files generated. If unspecified, the generated dumpfile size is limited only by available disk space. When defining a dumpfile filename, filetype does not need to be declared since the dumpfile filetype is the declared default. When defining log or SQL_FILE files, the filetype parameter must be included.

Valid filetype parameters are:

File type	Value	Filetype name
Dump file	1	ku$_file_type_dump_file
Log file	3	ku$_file_type_log_file
SQL file	4	ku$_file_type_sql_file

All of the file types may be declared using either the filetype name (preceded by dbms_datapump) or its numeric value.

Code snippet #3:

```
dbms_datapump.ADD_FILE(
   handle     => h1,
   filename   => 'FULL_DB_EXP.LOG',
   filetype   => dbms_datapump.ku$_file_type_log_file);
```

STEP 3: START THE JOB

With the job created and files defined, the Data Pump job can be launched. The START_JOB procedure is used to start, or restart, Data Pump jobs. When called, this procedure will change the job state to 'EXECUTING'. Here is the procedure declaration for START_JOB:

```
dbms_datapump.START_JOB(
   handle          IN NUMBER,
   skip_current    IN NUMBER DEFAULT 0,
   abort_step      IN NUMBER DEFAULT 0,
   cluster_ok      IN NUMBER DEFAULT 1,
   service_name    IN VARCHAR2 DEFAULT NULL);
```

Code snippet #4:

```
dbms_datapump.start_job(handle => h1);
```

PUTTING IT ALL TOGETHER

Using everything you've learned so far, you will see that with just 4 calls to 3 API objects, you can create and launch a Data Pump job to perform a full database export which will generate both dump and log files.

CODE EXAMPLE #1: SIMPLE DATABASE EXPORT

```
declare
  h1   NUMBER;
begin
  h1 := dbms_datapump.open(operation => 'EXPORT', job_mode => 'FULL'
    job_name => 'FULL_DB_EXP');
  dbms_datapump.add_file(handle => h1, filename => 'FULL_DB_EXP.DMP');
  dbms_datapump.add_file(handle => h1, filename => 'FULL_DB_EXP.LOG',
    filetype => dbms_datapump.ku$_file_type_log_file);
  dbms_datapump.start_job(handle => h1);
end;
```

DEFINING JOB PARAMETERS

In the simple full database export example, you were only required to define the output dumpfile. The SET_PARAMETER procedure is used to define many other optional job processing preferences. Here are the procedure declarations for SET_PARAMETER:

```
dbms_datapump.SET_PARAMETER(
   handle       IN NUMBER,
   name         IN VARCHAR2,
   value        IN VARCHAR2);

dbms_datapump.SET_PARAMETER (
   handle       IN NUMBER,
   name         IN VARCHAR2,
   value        IN NUMBER);
```

Notice that the value parameter is overloaded; depending upon the parameter defined, the value passed may contain either a number or a string.

Basic Parameters

Parameter Name	Description
ESTIMATE	Define method for computing table size estimates: BLOCKS, STATISTICS.
ESTIMATE_ONLY	Calculate export dumpfile size without performing an actual export. Value must be 1.
FLASHBACK_SCN	Define consistent export based on SCN.
FLACHBACK_TIME	Define consistent export based on timestamp.
INCLUDE_METADATA	Define job as a data load or unload only operation. Default value of 1 will include object metadata. Set to 0 to perform data load/unload only.
SKIP_UNUSABLE_INDEXES	Ignore unusable indexes and load table data anyway. Default value of 1 will allow data to be loaded. Set to 0 to prevent load of data for tables with unusable indexes.
TABLE_EXISTS_ACTION	Define import behavior for existing tables: TRUNCATE, REPLACE, APPEND, SKIP.
USER_METADATA	Include metadata to recreate user schemas. Default value of 1 will include user metadata. Set to 0 to exclude.

Advanced Parameters

Parameter Name	Description
COMPRESSION	Trade speed for size: DATA_ONLY, METADATA_ONLY, ALL, NONE.
DATA_OPTIONS	Advanced data processing options such as 'skip rows that violate constraints and continue import.
ENCRYPTION	Define encryption options for export: DATA_ONLY, METADATA_ONLY, ENCRYPTED_COLUMNS_ONLY, ALL, NONE.
ENCRYPTION_ALGORITHM	Define algorithm to be used: AES128, AES192, AES256.
ENCRYPTION_MODE	Define encryption type: PASSWORD, TRANSPARENT, DUAL
ENCRYPTION_PASSWORD	Define 'key' used for re-encryption.
PARTITION_OPTIONS	Define handling for partitioned tables: DEPARTITION, MERGE, NONE.
TABLESPACE_DATAFILE	Define target datafile for transportable tablespace import.
TRANSPORTABLE	Enables transportable mode for table exports.
TTS_FULL_CHECK	Enables dependency validations for

Other Parameters

Parameter Name	Description
CLIENT_COMMAND	Clients call this procedure to record the original command line used to invoke a job.
SOURCE_EDITION	Define source system version
TARGET_EDITION	Define target system version.

Using SET_PARAMETER, you can now define many of the most common Data Pump job execution parameters.

Code snippet #5:

```
dbms_datapump.SET_PARAMETER(
  handle => h1, name => 'TABLE_EXISTS_ACTION', value => 'TRUNCATE');
```

```
dbms_datapump.SET_PARAMETER(
  handle => h1, name => 'ESTIMATE', value => 'BLOCKS');
```

GETTING SELECTIVE: ADDING OBJECT FILTERS

Frequently, a simple full database import or export is not all that is needed. Instead, you may need to dump all of the objects belonging to a particular schema, a single table, or a subset of tables. The dbms_datapump package provides a robust set of options for defining filters with fine-grained control. The METADATA_FILTER procedure is the primary mechanism by which object level filters are defined to refine the scope of your Data Pump operation. Here is the procedure declaration for METADATA_FILTER:

```
dbms_datapump.METADATA_FILTER (
   handle           IN NUMBER,
   name             IN VARCHAR2,
   value            IN VARCHAR2,
   object_path      IN VARCHAR2 DEFAULT NULL,
   object_type      IN VARCHAR2 DEFAULT NULL);

dbms_datapump.METADATA_FILTER (
   handle           IN NUMBER,
   name             IN VARCHAR2,
   value            IN CLOB,
   object_path      IN VARCHAR2 DEFAULT NULL,
   object_type      IN VARCHAR2 DEFAULT NULL);
```

*Note the object_type parameter shown in italics. This parameter is undocumented and is functionally equivalent to object_path.

When METADATA_FILTER is called, you are required to provide at least three input parameters: handle, name and value. Handle, as previously discussed, is the job handle returned from the previous call to OPEN. The name and value parameters are used to define the type of filter that will be applied and the actual filter value. If specified, the object_path or object_type parameters will restrict application of the defined filter to only objects in that object path.

There are five metadata filter types:

Filter Types	Description
NAME	Used to identify objects by name
SCHEMA	Used to restrict objects by schema owner
TABLESPACE	Used to restrict objects by storage location
INCLUDE_PATH	Used to include objects by type
EXCLUDE_PATH	Used to exclude objects by type

Each of these metadata filters supports two input versions:

Filter Version	Description
LIST	Value parameter contains an explicit comma separated list of quoted identifiers. Used to define filters with maximum validation. All objects identified in list must be present for job to succeed.
EXPR	Value parameter contains a SQL expression which may contain wildcards. Used to define filters with maximum flexibility.

The filter type and filter versions are combined as <type>_<version> (ex:NAME_LIST) to create the valid filters that can be used for the name parameter.

The following table lists the 10 possible combinations, with examples demonstrating the syntax for each:

Filter name	Value examples
NAME_LIST	`value => '''DEPARTMENTS'''` `value => '''EMP'',''DEPT'',''BONUS'''`
NAME_EXPR	`value => 'IN(''EMP'',''DEPT'')'` `value => 'LIKE ''PRODUCT_%'''`
SCHEMA_LIST	`value => '''SCOTT'''` `value => '''SH'',''OE'''`
SCHEMA_EXPR	`value => '!=''BI'''` `value => 'LIKE ''__'''`
TABLESPACE_LIST	`value => '''USERS'''` `value => '''EXAMPLE'',''USERS'''`
TABLESPACE_EXPR	`value => 'IN(''EXAMPLE'')'` `value => 'LIKE ''EX%'''`
INCLUDE_PATH_LIST	`value => '''TABLE/INDEX'''` `value => '''TABLE/COMMENT'',''TABLE/CONSTRAINT'''`
INCLUDE_PATH_EXPR	`value => 'LIKE(''/TABLE/%'')'` `value => 'LIKE ''PACKAGE/PACKAGE%'''`
EXCLUDE_PATH_LIST	`value => '''TABLE/INDEX'''` `value => '''TABLE/COMMENT'',''TABLE/CONSTRAINT'''`
EXCLUDE_PATH_EXPR	`value => '=''VIEW'''` `value => '!=''/TABLE/INDEX'''`

Code snippet #6:

```
dbms_datapump.METADATA_FILTER(
  handle => h1, name => 'TABLESPACE_EXPR', value => '=''EXAMPLE''');
```

PUTTING IT ALL TOGETHER

Working from the earlier example, you can see that it takes only one additional API call in order to restrict the export to include only a specific set of schemas. For this example, a schema export is defined. A single filter is added to include only the SH and HR schemas in the export.

CODE EXAMPLE #2: FILTERED SCHEMA EXPORT

```
declare
  h1   NUMBER;
begin
  h1 := dbms_datapump.open(operation => 'EXPORT', job_mode => 'SCHEMA',
    job_name => 'SCHEMA_EXP_JOB');
  dbms_datapump.add_file(handle => h1, filename => 'SCHEMA_EXP.DMP',
    filetype => dbms_datapump.ku$_file_type_dump_file);
  dbms_datapump.add_file(handle => h1, filename => 'SCHEMA_EXP.LOG',
    filetype => dbms_datapump.ku$_file_type_log_file);
  dbms_datapump.metadata_filter(handle => h1, name => 'SCHEMA_LIST',
    value => '''SH'',''HR''');
  dbms_datapump.start_job(handle => h1);
  dbms_datapump.detach(handle => h1);
end;
```

GETTING MORE SELECTIVE: ADDING DATA FILTERS

When filtering to a single table isn't enough, due to the large size of an individual table, data filters can be used to restrict the size of the export or import. The dbms_datapump package provides several options for restricting the amount of data imported, or exported, at the row, block, or partition levels. Optionally, all table data can be excluded. The DATA_FILTER procedure is called to define these data level filters. Here are the procedure declarations for DATA_FILTER:

```
dbms_datapump.DATA_FILTER(
   handle           IN NUMBER,
   name             IN VARCHAR2,
   value            IN NUMBER,
   table_name       IN VARCHAR2 DEFAULT NULL,
   schema_name      IN VARCHAR2 DEFAULT NULL);

dbms_datapump.DATA_FILTER(
   handle           IN NUMBER,
   name             IN VARCHAR2,
   value            IN VARCHAR2,
   table_name       IN VARCHAR2 DEFAULT NULL,
   schema_name      IN VARCHAR2 DEFAULT NULL);

dbms_datapump.DATA_FILTER(
   handle           IN NUMBER,
   name             IN VARCHAR2,
   value            IN CLOB,
   table_name       IN VARCHAR2 DEFAULT NULL,
   schema_name      IN VARCHAR2 DEFAULT NULL);
```

When calling the DATA_FILTER procedure, you will be required to provide at least three input parameters: handle, name and value. The name and value parameters are used define the data filter to be applied.

The name parameter can contain one of these five data filter types:

Filter Types	Description
INCLUDE_ROWS	Default value of 1 will include row data. Set to 0 to exclude rows and perform a metadata only export or import.
PARTITION_EXPR	Specify partitions to include or exclude using SQL expressions.
PARTITION_LIST	Specify partitions to include in processing explicitly by name.
SAMPLE	Define percentage of blocks to include in processing.
SUBQUERY	Define a where clause to be applied to one or more tables.

The optional table_name and schema_name parameters are used to specify which tables or schemas the filter should be applied to. If a schema_name is specified, a table_name must also be specified.

Code snippet #7:

```
dbms_datapump.DATA_FILTER(
   handle       => h1,
   name         => 'SUBQUERY',
   value        => 'WHERE CUST_ID > 50000',);
```

PUTTING IT ALL TOGETHER

With many databases now containing very large tables, it is useful to know how to export or import a table with just a subset of the data, rather than in an all-or-nothing fashion. Modifying the earlier example, you can see that it takes only two additional API calls to achieve this goal. For this example, a table export is defined. Filters are defined to restrict the output to include only a subset of the data from the 'CUSTOMERS' table in the 'SH' schema.

CODE EXAMPLE #3: FILTERED TABLE EXPORT

```
declare
  h1   NUMBER;
begin
  h1 := dbms_datapump.open(operation => 'EXPORT', job_mode => 'TABLE',
    job_name => 'TABLE_EXP_JOB');
  dbms_datapump.add_file(handle => h1, filename => 'TABLE_EXP.DMP',
    filetype => dbms_datapump.ku$_file_type_dump_file);
  dbms_datapump.add_file(handle => h1, filename => 'TABLE_EXP.LOG',
    filetype => dbms_datapump.ku$_file_type_log_file);
  dbms_datapump.metadata_filter(handle => h1, name => 'SCHEMA_LIST',
    value => '''SH''');
  dbms_datapump.metadata_filter(handle => h1, name => 'NAME_LIST',
    value => '''CUSTOMERS''');
  dbms_datapump.data_filter(handle => h1, name => 'SUBQUERY',
    value => 'WHERE CUST_ID > 50000');
  dbms_datapump.start_job(handle => h1);
  dbms_datapump.detach(handle => h1);
end;
```

MAKING CHANGES: REMAP AND TRANSFORM

Dbms_datapump also enables us to make certain *edits* to object definitions as part of export or import processing. The two primary means for modifying object definitions are remapping and transformation.

REMAPPING

METADATA_REMAP provides a mechanism to implement remapping of objects, enables modification of object names, schema ownership and physical storage. Here is the procedure declaration for METADATA_REMAP:

```
dbms_datapump.METADATA_REMAP(
   handle         IN NUMBER,
   name           IN VARCHAR2,
   old_value      IN VARCHAR2,
   value          IN VARCHAR2,
   object_type    IN VARCHAR2 DEFAULT NULL);
```

Calling METADATA_REMAP requires 4 parameters: handle, name, old_value and value. The name parameter is used to identify the scope of the remapping function.

Valid remap name parameters are:

Remap name	Description
REMAP_SCHEMA	Alter object ownership.
REMAP_TABLESPACE	Alter object tablespace assignments.
REMAP_DATAFILE	Alter datafile references.
REMAP_TABLE	Alter table names.

Code snippet #8:

```
dbms_datapump.METADATA_REMAP(handle => h1, name => 'REMAP_SCHEMA',
  old_value  => 'SH', value => 'SCOTT');
```

PUTTING IT ALL TOGETHER

This example will demonstrate a schema import in which a subset of tables are remapped into a different schema. The dump file created in example #2 will be used as the import source. Using the filter methods discussed earlier, it requires two filters to restrict the import job to import only those tables in the 'SH' schema that begin with the letter 'C'. By adding a call to METADATA_REMAP, the selected tables, previously owned by 'SH' will be imported into SCOTT's schema. Since the import will include only a subset of tables from the 'SH' schema and since those tables may include referential constraints to other tables that have not been included in the import, an additional filter to exclude referential constraints is defined.

CODE EXAMPLE #4: REMAPPED TABLE IMPORT

```
declare
  h1   NUMBER;
begin
  h1 := dbms_datapump.open(operation => 'IMPORT', job_mode => 'TABLE',
    job_name => 'TABLE_IMP_JOB');
  dbms_datapump.add_file(handle => h1, filename => 'SCHEMA_EXP.DMP');
  dbms_datapump.add_file(handle => h1, filename => 'TABLE_IMP.LOG', filetype => 3);
  dbms_datapump.metadata_filter(handle => h1, name => 'SCHEMA_EXPR',
    value => 'IN (''SH'')');
  dbms_datapump.metadata_filter(handle => h1, name => 'NAME_EXPR',
    value => 'LIKE ''C%''');
  dbms_datapump.metadata_filter(handle => h1, name => 'EXCLUDE_PATH_EXPR',
    value => 'LIKE ''%/TABLE/CONSTRAINT/REF_CONSTRAINT%''');
  dbms_datapump.metadata_remap(handle => h1, name => 'REMAP_SCHEMA',
    old_value => 'SH', value => 'SCOTT');
  dbms_datapump.start_job(handle => h1);
  dbms_datapump.detach(handle => h1);
end;
```

TRANSFORMATION

METADATA_TRANSFORM supports a limited number of pre-defined object definition changes. It can be used to shrink object allocations on import, exclude processing of storage or segment metadata and enable reassignment of certain object id (OIDs) during import. The OID transform can be extremely useful if you need to import type definitions created on a remote databases and found that the OIDs between the systems conflict. Here is the procedure declaration for METADATA_TRANSFORM:

```
dbms_datapump.METADATA_TRANSFORM(
  handle         IN NUMBER,
  name           IN VARCHAR2,
  value          IN VARCHAR2,
  object_type    IN VARCHAR2 DEFAULT NULL);
```

METADATA_TRANSFORM requires three parameters: handle, name, and value. The name parameter is used to identify the type of the transformation.

Valid transform name parameters are:

Transform name	Description
PCTSPACE	Alter object ownership.
SEGMENT_ATTRIBUTES	Omit or Include storage segment parameters.
STORAGE	Omit or Include storage clause.
OID	Force reassignment of certain object ids during object creation.

Code snippet #9:

```
dbms_datapump.METADATA_TRANSFORM(handle => h1, name => 'STORAGE', value => 0);
```

PUTTING IT ALL TOGETHER

In the final example, a Data Pump SQL_FILE job will be defined. The job will use the dumpfile created by the export in example #2 as its source. Using each of the filter methods discussed previously, the job is constructed to select only the SH.CUSTOMERS table and its dependent objects. The target schema for the table is remapped to 'SCOTT'. Using METADATA_TRANSFORM, the job is also defined to exclude the storage parameters in the generated SQL script for the renamed SH_CUSTOMERS table.

CODE EXAMPLE #5: REMAPPED TABLE IMPORT

```
declare
  h1   NUMBER;
begin
  h1 := dbms_datapump.open(operation => 'SQL_FILE', job_mode => 'TABLE',
    job_name => 'TRANSFORM_SQL_JOB');
  dbms_datapump.add_file(
    handle => h1, filename => 'SCHEMA_EXP.DMP', filetype => 1);
  dbms_datapump.add_file(
    handle => h1, filename => 'TRANSFORM_SQL.LOG', filetype => 3);
  dbms_datapump.add_file(
    handle => h1, filename => 'TRANSFORM_SQL.SQL', filetype => 4);
  dbms_datapump.metadata_filter(
    handle => h1, name => 'SCHEMA_EXPR', value => 'IN (''SH'')');
  dbms_datapump.metadata_filter(
    handle => h1, name => 'NAME_EXPR', value => 'IN (''CUSTOMERS'')');
  dbms_datapump.metadata_filter(
    handle => h1, name => 'INCLUDE_PATH_EXPR', value => 'IN (''TABLE'')');
  dbms_datapump.metadata_remap(
    handle => h1, name => 'REMAP_TABLE',
    old_value => 'CUSTOMERS', value => 'SH_CUSTOMERS');
  dbms_datapump.metadata_remap(
    handle => h1, name => 'REMAP_SCHEMA',
    old_value => 'SH', value => 'SCOTT');
  dbms_datapump.metadata_transform(
    handle => h1, name => 'STORAGE', value => 0, object_type => 'TABLE');
  dbms_datapump.start_job(handle => h1);
  dbms_datapump.detach(handle => h1);
end;
```

CONCLUSION

With relatively few lines of code, dbms_datapump provides a very rich interface for defining and executing robust Data Pump jobs using 100% PL/SQL (or Java, C#, or other programming languages). Whether you want to perform a basic export or create an automated process to move data directly between databases, having a solid understanding of the basic use of the functions and procedures provided within the dbms_datapump package is key. Building on that foundation, Data Pump job definition via the Data Pump API can be easily combined with real-time decision making to create extremely flexible automated import/export processes.

REFERENCES

"Data Pump in Oracle Database 11g Release 2: Foundation for Ultra High-Speed Data Movement Utilities", Oracle White Paper, 9/2010, Oracle Technology Network, http://download.oracle.com/otndocs/products/database/enterprise_edition/utilities/pdf/datapump11gr2_techover_1009.pdf

"Data Transformations with Oracle Data Pump" Oracle White Paper, 9/2010, Oracle Technology Network, http://download.oracle.com/otndocs/products/database/enterprise_edition/utilities/pdf/datapump11g2_transform_1009.pdf

"Oracle DBMS_DATAPUMP", Morgan's Library @ http://psoug.org/reference/dbms_datapump.html

"Oracle Database Utilities 11g Release 2 (11.2)", Chapters 1,2,3,6

"Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)", Chapter 48

"The Oracle 10g Data Pump API Speeds Up the ETL Process", Natalka Roshak, Devx.com, http://www.devx.com/dbzone/Article/30355/1954

ABOUT THE AUTHOR

Sandi Reddick has been an Oracle DBA for 14 years, starting with version 7.3.4 in 1997. She has worked as an Oracle DBA for Lockheed Martin, Hewlett Packard and most recently Fiserv. Sandi attended the University of Central Florida, graduating in 1996 with a Bachelor's of Science in Computer Science. She is the founder of OraPro.net, has previously presented at UKOUG on 'Tuning Oracle Text', and is an active member of her local user group, WWOUG in Seattle, WA. Please feel free to send any questions or comments to sandi@orapro.net.

