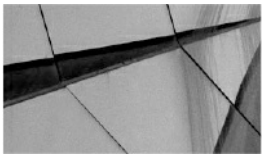
```
SQL> select spid, stid, pname, execution_type from v$process where spid=4634 and stid
in (4666,15873);

SPID  STID   PNAME EXECUTION_TYPE
----- ------ ----- ------------------
4634  4666   N000  THREAD
4634  15873  N001  THREAD
```



NOTE

Do not attempt to kill thread processes via the command line (using the `kill` command) because you might end up killing a number of other thread connections as well. Instead, consider using `ALTER SYSTEM KILL SESSION` by passing in the correct SID and SERIAL# values. Make sure you are working with the correct session by first looking at V$SESSION and V$PROCESS.

If you want to configure certain clients to use the threaded option and others to use processes, you will need to configure two listeners, each one using different ports. It is also possible to create a dedicated listener for a specific PDB, but you'll learn more about this a bit later in this chapter.

Service Names

When using a multitenant configuration, you will need to be aware of the changes introduced with services. In this section we will cover the important changes you should be aware of when creating and maintaining Oracle Database 12*c* Multitenant environments.

Default Services and Connecting to PDBs

When creating a new PDB, a new default service is automatically generated for it, with the same name as the PDB. The service will be registered with the listener and client connections, and connections to the PDB can begin to make use of the new service once the PDB is opened. The automatic registration might take a few seconds, although you can run `alter system`

`register` to force the registration to occur immediately.

So, for example, if a new PDB called PDB1 is created, a new default service of the same name will also be created. The new service details can be viewed by looking at v$services or cdb_services:

```
SQL> select service_id, name, network_name, PDB from v$services
           where network_name is not null order by 1;

SERVICE_ID NAME                      NETWORK NAME                 PDB
---------- ------------------------  ---------------------------- ----------
         5 CDB2XDB                   CDB2XDB                      CDB$ROOT
         6 CDB2.orademo.net          CDB2.orademo.net             CDB$ROOT
         7 pdb1.orademo.net          pdb1.orademo.net             PDB1
```

Reviewing the listener status and services will indicate whether the new service name is registered:

```
# lsnrctl status listener
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 27-FEB-2016 20:19:33
Copyright (c) 1991, 2015, Oracle.  All rights reserved.
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=linux2.orademo.net)
(PORT=1521)))
…
Services Summary...
…
Service "CDB2.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 1 handler(s) for this service...
Service "pdb1.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 1 handler(s) for this service...
The command completed successfully
# lsnrctl service listener
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 27-FEB-2016 20:28:00
Copyright (c) 1991, 2015, Oracle.  All rights reserved.
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=linux2.orademo.net)
(PORT=1521)))
Services Summary...
…
…
Service "CDB2.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:14 refused:0 state:ready
         LOCAL SERVER
Service "pdb1.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:14 refused:0 state:ready
         LOCAL SERVER
The command completed successfully
```

The end user/application can now connect to this PDB using the new service name via a number of methods, such as via the Oracle Net Services name using the tnsnames.ora file or an easy connect string. The basic easy connect string takes the format @[//`Host`[:`Port`]/<`service_name`>], and when this method is used, no entry is required in the tnsnames.ora file.

Here's an example of using the easy connect method to connect to the newly created PDB1:

```
SQL> connect aels/aelspassword@//linux2.orademo.net/pdb1.orademo.net
```

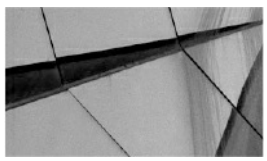
You can also add and use an Oracle Net Services name entry in the tnsnames.ora file:

```
PDB1 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = linux2.orademo.net)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = PDB1.orademo.net)
    )
  )
```

Once the entry is added to the tnsnames.ora file, it can be used to connect to the PDB:

```
SQL> connect aels/aelspassword@PDB1
Connected.
SQL> show con_name
CON_NAME
------------------------------
PDB1
```

Figure 5-2 illustrates two basic concepts. When each PDB is created, each will have a service name created that matches the PDB name; the service is automatically registered with the default listener. When a client connection is requested to the listener for a specific service name for a PDB, the listener prompts a server process to be spawned and the connection between the client and the PDB will be established, with the listener no longer involved.



NOTE

When running more than one CDB on a single system with PDBs using the same service names, it is recommended that you use

separate listeners for each CDB. If only one listener is used, you will end up with both CDB databases having service names for the PDBs registered with the same listener. The end result could be that an incorrect connection may be established, which may lead to undesirable results. In view of this, it is recommended that all service names on a system should be unique to avoid such collisions or, alternatively, separate listeners configured for each CDB.

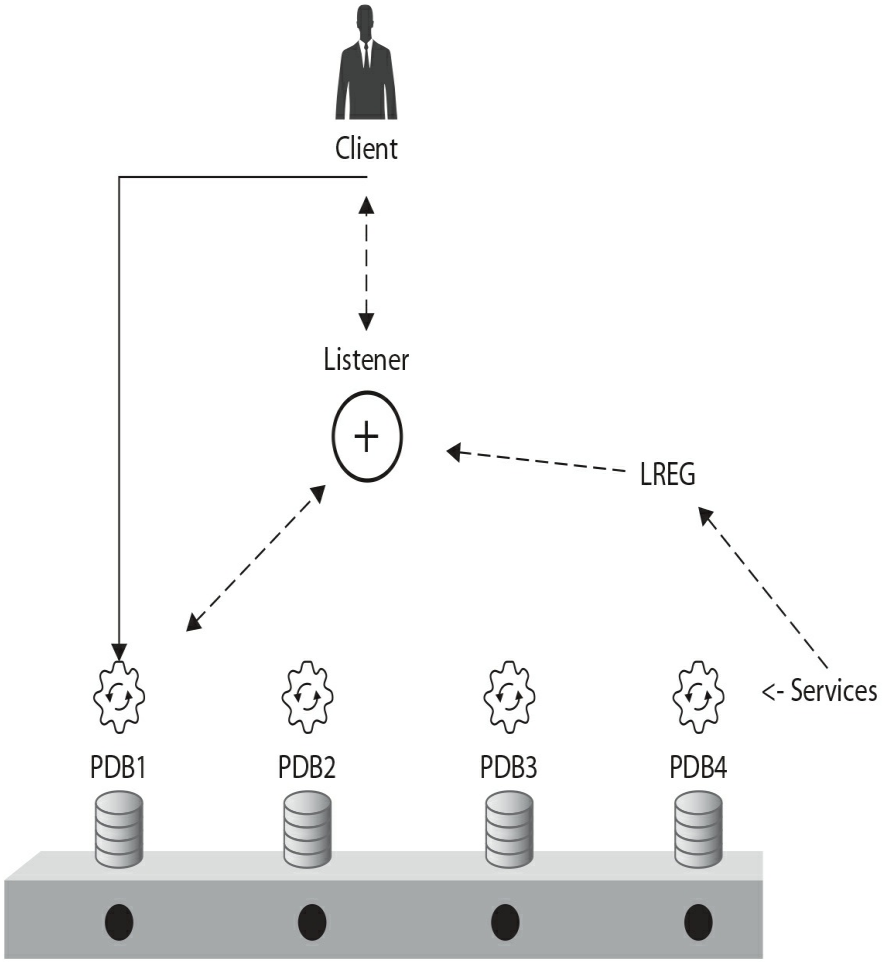


FIGURE 5-2. *Service name registration (LREG) and client connection*

Before moving on to the next section it is worth mentioning that you can also use the local environment variable `TWO_TASK` (on UNIX) or `LOCAL` (on Windows) to specify a default connect identifier (connect string). When you

set this variable, the user will be able to connect to a database without explicitly specifying the connect string. Here's an example:

```
#  export TWO_TASK=pdb1
#  sqlplus aels/aelspassword

SQL*Plus: Release 12.2.0.0.2 Beta on Sun Feb 28 12:37:07 2016
Copyright (c) 1982, 2015, Oracle.  All rights reserved.
Last Successful login time: Sun Feb 28 2016 05:52:55 +13:00
Connected to:
Oracle Database 12c Enterprise Edition Release 12.2.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options

SQL> show con_name
CON_NAME
------------------------------
PDB1
```

This method might be required by various applications. Also note that the easy connect string can be used as well; this removes the dependency of having a required entry in the tnsnames.ora file (example: `export TWO_TASK=//linux3.orademo.net/PDB1.orademo.net`).

Creating Services

By default, a service is generated for each PDB on creation. In many cases, the requirement will be to create additional services and associate them with the particular PDB. This can be especially useful in Oracle RAC configurations, where you might want a specific application to connect to its PDB only from one of the instances. Creating a service and setting up the rules to run on only one particular node in the cluster can be an extremely useful capability to have.

When creating a new service, you can set an optional PDB property at creation time, which can be modified at any time after. The PDB property is important because it associates the service to a particular PDB.

If a user connects to a service that does not have a value specified for the PDB property, the user name would be resolved in the context of the root container. However, if the value is specified for the PDB property, the user

name will be resolved in the context of the specified PDB.

Note the following regarding service creation:

- Services become active (listed in v$active_services and registered with the listener) only when the PDB is opened.
- Service names must be unique within the CDB, but they must also be unique between all databases using a specific listener.
- When using Oracle Restart or Oracle Clusterware, you can use either the SRVCTL utility or the DBMS_SERVICE package to add, modify, and manage new services. Using the SRVCTL utility in this case is recommended.
- The PDB property must be set using `-pdb <PDB>` when you're using SRVCTL. If you're using the DBMS_SERVICE package, the PDB property will automatically be set to the current connected container, so make sure you connect to the correct PDB before creating the service using DBMS_SERVICE.
- The PDB property cannot be changed with the DBMS_SERVICE package. The service will need to be re-created from within the correct PDB.
- If you are not using Oracle Restart or Oracle Clusterware, the DBMS_SERVICE package is used to add, modify, or remove new services.
- Stopping a service using the SRVCTL utility does not change the status of the PDB it is associated with. The SRVCTL `stop` command will affect only the service, not the PDB.
- When you unplug a PDB, the service will not be removed, so this should be managed manually. The same applies when a PDB is dropped. If the service is no longer required, it should be removed manually.

Creating a Service with SRVCTL

Creating services with the SRVCTL command line utility is very easy, and you can quickly understand why you should be using this with Oracle Restart or Oracle Clusterware.

In the next two examples, two new services will be created: the first uses a single-instance configuration, and the second uses a two-node Oracle RAC cluster.

Adding a Service for a PDB in a Single-Instance Database The CDB database CDB1 consists of PDB1 and PDB2. It is a single-instance database in which Oracle Restart is used. Each of the PDBs already has the default service created and registered with the default listener. In addition, the following services are required:

- PDB1 will use service CRMDEV.
- PDB2 will use service HRDEV.

As the Oracle database software owner, which in this installation is the user name *oracle* for the UNIX environment, set the environment to the correct Oracle Home and use the `srvctl` command to add the two services. Then review their status and start these two services. See the code blocks that follow for more detail on these steps.

1. Create the services.
```
#  srvctl add service -db CDB1 -pdb PDB1 -service CRMDEV
#  srvctl add service -db CDB1 -pdb PDB2 -service HRDEV
```
2. Review the service status.
```
#  srvctl status service -db CDB1
Service CRMDEV is not running.
Service HRDEV is not running.
```
3. Start the services.
```
# srvctl start service -db CDB1 -service CRMDEV
# srvctl start service -db CDB1 -service HRDEV
```
4. Show the status of the services following the startup.
```
# srvctl status service -db CDB1
Service CRMDEV is running
Service HRDEV is running
```

Now that the services are created, review what the listener knows. Notice that these newly added services are also registered with the listener, as per the extracts from these listener status commands. Here’s the first one:

```
# lsnrctl status listener
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 28-FEB-2016 07:44:14
Copyright (c) 1991, 2015, Oracle.  All rights reserved.

Connecting to (ADDRESS=(PROTOCOL=tcp)(HOST=)(PORT=1521))
…
Service "crmdev.orademo.net" has 1 instance(s).
  Instance "CDB1", status READY, has 1 handler(s) for this service...
Service "hrdev.orademo.net" has 1 instance(s).
  Instance "CDB1", status READY, has 1 handler(s) for this service...
…
The command completed successfully
```

And here's the second one:

```
# lsnrctl service listener
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 28-FEB-2016 07:44:48
Copyright (c) 1991, 2015, Oracle.  All rights reserved.

Connecting to (ADDRESS=(PROTOCOL=tcp)(HOST=)(PORT=1521))
Services Summary...
…
Service "crmdev.orademo.net" has 1 instance(s).
  Instance "CDB1", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:1688 refused:0 state:ready
         LOCAL SERVER
Service "hrdev.orademo.net" has 1 instance(s).
  Instance "CDB1", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:1688 refused:0 state:ready
         LOCAL SERVER
…
The command completed successfully
```

Reviewing V$SERVICES, you can also see the newly created services:

```
SQL> select con_id, service_id, name, network_name, pdb
     from v$services
     where pdb in ('PDB1','PDB2') order by 1,2;

 CON_ID SERVICE_ID NAME              NETWORK NAME         PDB
------- ---------- ----------------- -----------------    ------
      3          1 CRMDEV            CRMDEV               PDB1
      3          7 pdb1.orademo.net  pdb1.orademo.net     PDB1
      4          1 HRDEV             HRDEV                PDB2
      4          9 pdb2.orademo.net  pdb2.orademo.net     PDB2
```

You can now use these new services to connect to the respective PDBs—here's an example:

```
SQL> connect aels/aelspassword@//linux3/CRMDEV.orademo.net
Connected.
SQL> show con_name
CON_NAME
------------------------------
PDB1
```

Adding a Service for an Oracle RAC PDB The steps for adding a new service for a PDB under Oracle RAC are almost identical to those used in the preceding example for a single-instance database: the SRVCTL utility is invoked to add the service. In this case, however, you need to keep in mind one particular question: Do you want the service to be enabled on all the instances, or just one?

In the next example, the Oracle RAC database, RCDB, consists of two nodes running two PDB databases: RPDB1 and RPDB2. The requirement is that RPDB1 be accessible only via node 1 (instance 1, RCDB1) and that RPDB2 be accessible from both nodes. By creating two services, this can easily be achieved.

To create the service for RPDB1 to run from only instance 1, we use the following commands, executed as the Oracle Database software owner, which is the *oracle* UNIX user in this case.

1 Create new service CRMPRT to run on one instance only.

```
# srvctl add service -db RCDB -service CRMRPT -preferred "RCDB1"
-available "RCDB2" -pdb RPDB1
```

2 Create new service CRMPRD to run on both instances.

```
# srvctl add service -db RCDB -service CRMPRD -preferred
"RCDB1,RCDB2" -pdb RPDB2
```

3 Review the status of the service.

```
# srvctl status service -db RCDB
Service CRMPRD is not running.
Service CRMRPT is not running.
```

4 Start both services.

```
# srvctl start service -db RCDB -service CRMRPT
# srvctl start service -db RCDB -service CRMPRD
```

5 Review statuses following service startup:

```
# srvctl status service -db RCDB
Service CRMPRD is running on instance(s) RCDB1,RCDB2
Service CRMRPT is running on instance(s) RCDB1
```

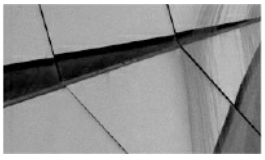
The status output shows that the service CRMPRD is now running and available on both instances in the Oracle RAC cluster, whereas the service CRMRPT runs only on the preferred instance, RCDB1. Users or clients will be able to start using them for connections—here's an example:

```
SQL> connect reportadmin/reportadmin@//rac12-scan/CRMRPT.orademo.net
Connected.
SQL> show con_name
CON_NAME
------------------------------
RPDB1

SQL> select instance_name from v$instance;
INSTANCE_NAME
---------
RCDB1
```

To modify or remove services, you can use the `modify service` or `remove service` option provided by the `srvctl` command:

```
# srvctl modify service -db RCDB -service CRMRPT -pdb RPDB2
# srvctl stop service -db RCDB -service CRMRPT
# srvctl remove service -db RCDB -service CRMRP
```



NOTE

For more information on using the SRVCTL utility, use the `srvctl -h` command to obtain detailed help options. If you specify `srvctl add service -h`, you can also obtain help specific to the addition of new services. Substitute with the `modify` or `remove` keyword to list information on the respective help option.

Creating a Service with DBMS_SERVICE

When using environments in which Oracle Restart or Oracle Clusterware is not installed, you need to use the DBMS_SERVICE package to create and manage new services. As mentioned, when creating new services for a specific PDB, you need to ensure that you are connected to that PDB when performing this operation. Otherwise, the service will be created in the context of the PDB you are connected to at the time, as the PDB property is

set to the connected PDB.

1. Switch to the PDB and create the new service.

```
SQL> alter session set container=PDB1;
SQL> begin
       dbms_service.create_service (service_name => 'CRMPROD'
                                    , network_name=> 'CRMPROD.
oradem o.net');
     end;
/
PL/SQL procedure successfully completed.
```

2. Review the services.

```
SQL> select service_id, name, network_name, creation_date, pdb from dba_services;

SERVICE_ID NAME             NETWORK NAME         CREATION_DATE       PDB
---------- ---------------- -----------------    ------------------- -------
         7  pdb1.oradem o.net  pdb1.oradem o.net     7/02/2016:18:31:40  PDB1
         1  CRMPROD          CRMPROD.oradem o.net  28/02/2016:04:01:50 PDB1
```

3. List all the active services.

```
SQL> select service_id, name, network_name from v$active_services;

SERVICE_ID NAME               NETWORK NAME
---------- ----------------   ---------------------------------------
         7  pdb1.oradem o.net  pdb1.oradem o.net
```

4. Start the service and review active services.

```
SQL> exec dbms_service.start_service ('CRMPROD');
PL/SQL procedure successfully completed.

SQL> select service_id, name, network_name from v$active_services;

SERVICE_ID NAME                 NETWORK NAME
---------- -------------------  ------------------------------------
         7  pdb1.oradem o.net     pdb1.oradem o.net
         1  CRMPROD              CRMPROD.oradem o.net
```

In this section we have demonstrated how easy it is to create new services for a PDB, using either the SRVCTL utility or the DBMS_SERVICE package.

Create a Dedicated Listener for a PDB

In some cases, you may need to use a specific dedicated listener port for one or more PDBs. This will require that you create a new listener and then ensure that the PDB is registered with it. In this section, we will show you how this can be done.

In a consolidated database environment with a large number of PDBs within a CDB, you may need to segment some PDBs off or, from a security point of view, maintain both encrypted and unencrypted connections. To do this, you first create a new listener.

In the following example, we will call the listener LISTENER_PDB and have it listen on port 1531. The following entry is added to the listener.ora file to introduce the new listener:

```
LISTENER_PDB =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = linux2.orademo.net)(PORT =
1531))
    )
  )
```

Once the new listener entry has been added, we can start it with the `lsnrctl start listener_pdb` command. When the listener is started, we can then add a net alias to the tnsnames.ora file:

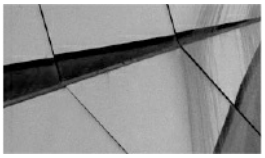
```
LISTENER_PDB =
  (ADDRESS = (PROTOCOL = TCP)(HOST = linux2.orademo.net)(PORT = 1531)
```

We are now ready to configure our PDB, which happens to be PDB1 in this case, to use this listener. We update the `LISTENER_NETWORKS` parameter specific to the PDB:

```
SQL> alter session set container=PDB1;
SQL> alter system set listener_networks='((NAME=PDB_NETWORK1)(LOCAL_
LISTENER=LISTENER_PDB))' scope=both;
```

The syntax for the `LISTENER_NETWORKS` parameter is as follows:

```
LISTENER_NETWORKS =
 '((NAME=network_name)(LOCAL_LISTENER=["]listener_address[,...]["])
[(REMOTE_LISTENER=["]listener_address[,...]["])])'  [,...]
```



NOTE

The `listener_address` string is an address (or address list) that resolves to a specific listener. An alias can be used, which is the case in the example used here, although it requires that you add an address entry into the tnsnames.ora file prior, as shown in the code block `Add net alias to tnsnames.ora file`.

If we now review the listener, we will notice that the PDB1 services are registered automatically by the LREG process, following the execution of the preceding commands.

```
# lsnrctl status listener_pdb
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 28-FEB-2016 05:45:36
Copyright (c) 1991, 2015, Oracle.  All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=linux2.orademo.net)
(PORT=1531)))
STATUS of the LISTENER
------------------------
Alias                     listener_pdb
Version                   TNSLSNR for Linux: Version 12.2.0.0.2 - Beta
Start Date                28-FEB-2016 05:34:25
Uptime                    0 days 0 hr. 11 min. 11 sec
Trace Level               off
Security                  ON: Local OS Authentication
SNMP                      OFF
Listener Parameter File   /u01/app/oracle/product/12.2.0/dbhome_1/network/admin/
listener.ora
Listener Log File         /u01/app/oracle/diag/tnslsnr/linux2/listener_pdb/alert/log.
xml
Listening Endpoints Summary...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=linux2.orademo.net)(PORT=1531)))
Services Summary...
Service "2cbaba1bf0dc4817e0538428a8c0ec8a.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 2 handler(s) for this service...
Service "pdb1.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 2 handler(s) for this service...
The command completed successfully

# lsnrctl service listener_pdb
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 28-FEB-2016 05:45:38
Copyright (c) 1991, 2015, Oracle.  All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=linux2.orademo.net)
(PORT=1531)))
Services Summary...
Service "2cbaba1bf0dc4817e0538428a8c0ec8a.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 2 handler(s) for this service...
    Handler(s):
      "D000" established:0 refused:0 current:0 max:1022 state:ready
         DISPATCHER <machine: linux2.orademo.net, pid: 15376>
         (ADDRESS=(PROTOCOL=tcp)(HOST=linux2.orademo.net)(PORT=36434))
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
Service "pdb1.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 2 handler(s) for this service...
    Handler(s):
      "D000" established:0 refused:0 current:0 max:1022 state:ready
         DISPATCHER <machine: linux2.orademo.net, pid: 15376>
         (ADDRESS=(PROTOCOL=tcp)(HOST=linux2.orademo.net)(PORT=36434))
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
The command completed successfully
```

Now that the PDB is registered with the listener, we can connect to it using the new listener that is running on port 1531, as follows:

```
SQL> connect aels/aelspassword@//linux2:1531/pdb1.orademo.net
Connected.
SQL> show con_name
CON_NAME
------------------------------
PDB1
```

If you are not using the easy connect string connection method, but are instead using a Net Service name using the tnsnames.ora file entries, you need to make sure to add the following entry to the tnsnames.ora file to reflect the new port change for this PDB:

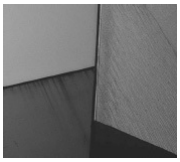
```
PDB1=
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = linux2.orademo.net)(PORT = 1531))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = PDB1.orademo.net)
    )
  )
```

Summary

In this chapter, we covered only the tip of the iceberg of information available regarding Oracle networking to highlight key features and functionalities you should be aware of when using the Oracle Database 12*c* Multitenant option. Creating and managing services in a multitenant configuration turns out to be less complex than you might have thought, even when using Oracle RAC. Using the SRVCTL utility or the DBMS_SERVICE package can help you achieve the required results quickly.

Now that you know how to connect, the next logical topic to address is

security. For many DBAs, this is a daunting and intimidating topic, but in the next chapter, we will systematically walk you through its key aspects, including explaining the difference between different types of users, roles, and permissions, along with discussions about encryption, isolation, and the lockdown of profiles.



CHAPTER 6

Security

So far in Part II we have described a number of operations you can perform on pluggable databases (PDBs) and the many different ways to connect to them. At this point we also need to factor security considerations into the mix, as the most prevalent attack vector is an abuse of database privileges. Rather than granting administrator rights too widely, we should instead apply the principle of minimal privileges. In 12*c*, with the Enterprise Edition and the Advanced Security option, the privilege analysis functionality offered can be a great help in this area.

With respect to multitenant, your effective user security administration starts by thinking about which users should have access only to specific PDBs, as users and roles can be created common or locally. Privileges can also be granted common or locally, and beyond this, multitenant (including single-tenant) brings additional fine-grained control via powerful commands, courtesy of lockdown profiles. Of course, when you issue powerful privileges locally to a PDB, you need to prevent any side effects on the container database (CDB), and 12.2 introduced a number of PDB isolation features to mitigate this.

In terms of data security, Oracle Virtual Private Database is still an option to limit access at row level, as is the Oracle Database Vault to protect against rogue database administrators (but we will not detail them here, because they're not specific to multitenant). Another type of possible attack is network sniffing—that is, reading data directly off the network—and network encryption is available in all Oracle 12*c* editions without options. Bulk data sets that leave the confines of the company premises, such as backups stored off site or in the cloud, are also potentially vulnerable. We will cover backup encryption in Chapter 7.

Finally, we may want to protect against unauthorized access at disk level, and this is highly recommended in a multitenant database, where data from different sources will be consolidated on the same CDB. This protection becomes mandatory when we put our data on a public cloud, and for these reasons we will cover Transparent Data Encryption at the end of this chapter.

Users, Roles, and Permissions

At a high level, when you connect to an Oracle Database, you do so with a database user that is declared within the database, in the dictionary, along

with the user's privilege definition. We have not yet defined which container this user information is stored in, and they can actually be common and thereby stored in CDB$ROOT or local and stored in a PDB.

Common or Local?

A common user's information is stored in CDB$ROOT and exists in every single PDB. You create common users for the CDB administrators or for users that have the same identity in all tenants. In both cases, they must connect to CDB$ROOT to change their passwords. In contrast, a local user is stored in a single PDB and exists only in that PDB.

For users, roles, profiles, and privileges, we have to work with the key words *common* and *local*. But before we go any further, let's ensure that we are all on the same page in terms of our definitions of a user and a role in the Oracle Database, because this is not an obvious given.

What Is a User?

In Oracle, a *user* and a *schema* are synonymous—or at least that was the case before the introduction of multitenant. When you create a user, you implicitly create a schema for the user objects, and when you have a schema, its owner is a user. Because this chapter is about security and not database objects, we will use the term "user" here. However, keep in mind that the one-to-one relationship between users and schemas has changed with multitenant, and one common user is now a different schema in each PDB.

A *user* is just a name that is employed when you connect to the database, and a session is always associated with a user. No matter which way you connect to the database, you will have a username, and this user is the vehicle that enables a session to perform operations on the database. You grant privileges to the user, and the connected user can enact whatever is permitted by those privileges.

Because it is highly likely that you will have several users with the same allotted privileges, or users with requirements for the same groups of privileges, you can define roles. In this way, you grant specific privileges to a role, and then grant the role to the users. Roles are also useful to enable the switching of users from one group of privileges to another. For example, the

same user may have a read-only role when connecting with SQL Developer and a read/write role when connected through the application, because the application can encapsulate some form of access control.

User and role definitions are actually stored in the same data dictionary table, USER$, where the TYPE# defines whether it is a user (1) or a role (0):

```
SQL> select user,name,type# from user$ order by 1;

USER                 NAME                             TYPE#
-------------------- ------------------------------ ----------
SYS                  SYS                                 1
SYS                  PUBLIC                              0
SYS                  CONNECT                             0
SYS                  RESOURCE                            0
SYS                  DBA                                 0
SYS                  AUDIT_ADMIN                         0
SYS                  AUDIT_VIEWER                        0
SYS                  AUDSYS                              1
SYS                  SYSTEM                              1
SYS                  SELECT_CATALOG_ROLE                 0
…
```

You might think that when you have no database, you would have no user definitions, because there is no dictionary yet; however, as it turns out, some users are actually hard-coded. Here we connect to an “idle instance” as we would do when wanting to create a database:

```
$ ORACLE_SID=DUMMY sqlplus / as sysdba
SQL*Plus: Release 12.1.0.2.0 Production on Tue Mar 8 16:00:45 2016
Copyright (c) 1982, 2014, Oracle.  All rights reserved.
Connected to an idle instance.
SQL> show user
USER is "SYS"
SQL> connect / as sysoper
Connected to an idle instance.
SQL> show user
USER is "PUBLIC"
```

SYSDBA privilege is mapped to the SYS user and SYSOPER privilege is mapped to the PUBLIC role. The authentication is handled through OS groups or via a password file, because there is no dictionary at this stage.

A number of users and roles are generated when you create the database, and these are maintained by Oracle. Starting in 12*c*, multitenant or not, there is an easy way to identify system users, because they have the `ORACLE_MAINTAINED` column set to TRUE in DBA_USERS. The creation of new users and roles is handled with the `CREATE USER` and `CREATE ROLE` statements. Both statements have a new optional clause in 12*c* multitenant, `CONTAINER=CURRENT` or `CONTAINER=ALL`, to define whether they are to be created for the current container or are to be common to all of them.

CONTAINER=CURRENT

When connecting to a PDB, we can connect to the root and `ALTER SYSTEM SET CONTAINER`, but, as we saw in the previous chapter, connecting via the listener to a service switches the session directly to the service's container:

```
SQL> connect sys/oracle@//localhost/PDB as sysdba
Connected.
```

Once connected, we can then create a user:

```
SQL> create user PDBUSER1 identified by oracle container=current;
User created.
```

In fact, when connected to a PDB, we can create only local users, so the container clause is not mandatory and defaults to local:

```
SQL> create user PDBUSER2 identified by oracle;
User created.
```

If we try something else, we get an error:

```
SQL> create user PDBUSER3 identified by oracle container=all;
create user PDBUSER3 identified by oracle container=all
                                          *
ERROR at line 1:
ORA-65050: Common DDLs only allowed in CDB$ROOT
```

So basically nothing changes here when working in a PDB; we simply create users in the way we are used to, and we can add the `container=current` just to be explicit.

When we want to create a user with Oracle Enterprise Manager, connected to a PDB, we don't have the choice. We receive a message, "Note: Created user will be a local user since you are in PDB container," as shown in Figure 6-1.

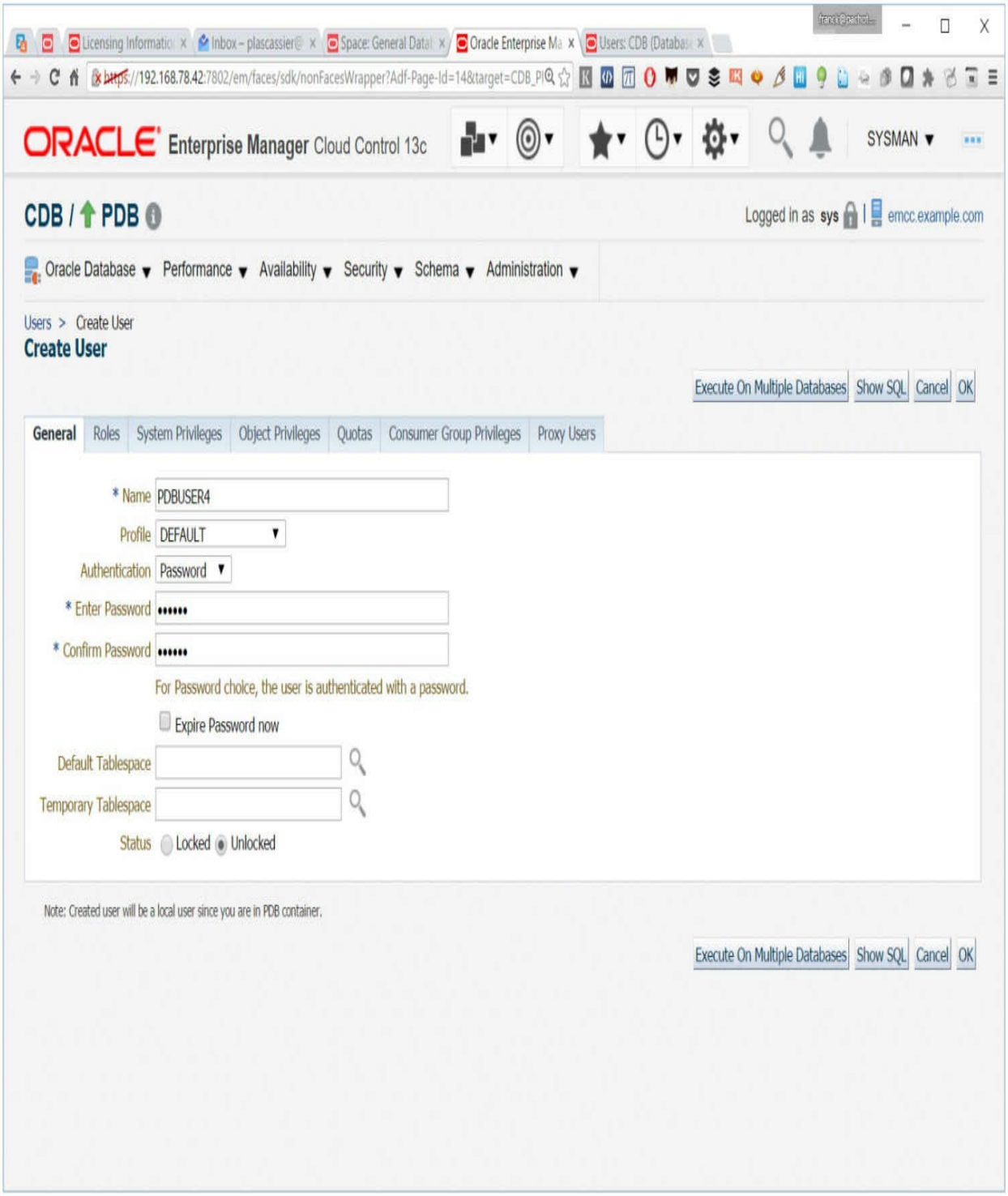


FIGURE 6-1. *Creating a local user in OEM 13*c

CONTAINER=COMMON

Now let's see what we can do from CDB$ROOT:

```
SQL> connect sys/oracle@//localhost/CDB as sysdba
Connected.
SQL> show con_name

CON_NAME
------------------------------
CDB$ROOT
```

First of all, trying to create a local user fails:

```
SQL> create user CDBUSER1 identified by oracle container=current;
create user CDBUSER1 identified by oracle container=current
                                    *
ERROR at line 1:
ORA-65049: creation of local user or role is not allowed in CDB$ROOT
```

When dealing with security objects (users, roles, profiles), what we create in the root container must be common. But there is something else to be aware of:

```
SQL> create user CDBUSER1 identified by oracle container=all;
create user CDBUSER1 identified by oracle container=all
            *
ERROR at line 1:
ORA-65096: invalid common user or role name
```

The best practice is to avoid mixing common and local users in the same namespace by setting a prefix for all common users. The prefix is set by the parameter `COMMON_USER_PREFIX` which defaults to `C##`:

```
SQL> show parameter common
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------
common_user_prefix                   string      C##
```

You can change this prefix and can even set it to null string, although we don't recommend this. The problem with doing this can occur after some PDB movement operation, when the same name is used by both a common and a local user. Having a prefix reserved for common users brings clarity and prevents such conflicts.

Here is how we create common users connected to CDB$ROOT. Note that the `CONTAINER=ALL` clause is not mandatory because it is implicit, and it is the only valid value in this context:

```
SQL> create user C##USER1 identified by oracle container=all;
User created.
SQL> create user C##USER2 identified by oracle;
User created.
```

Figure 6-2 shows this user creation within OEM, clearly displaying the information, "Common user name must begin with 'C##'"; it is a common user because it is created from root. Note that this message is static and does not take into account changes made to `COMMON_USER_PREFIX`.

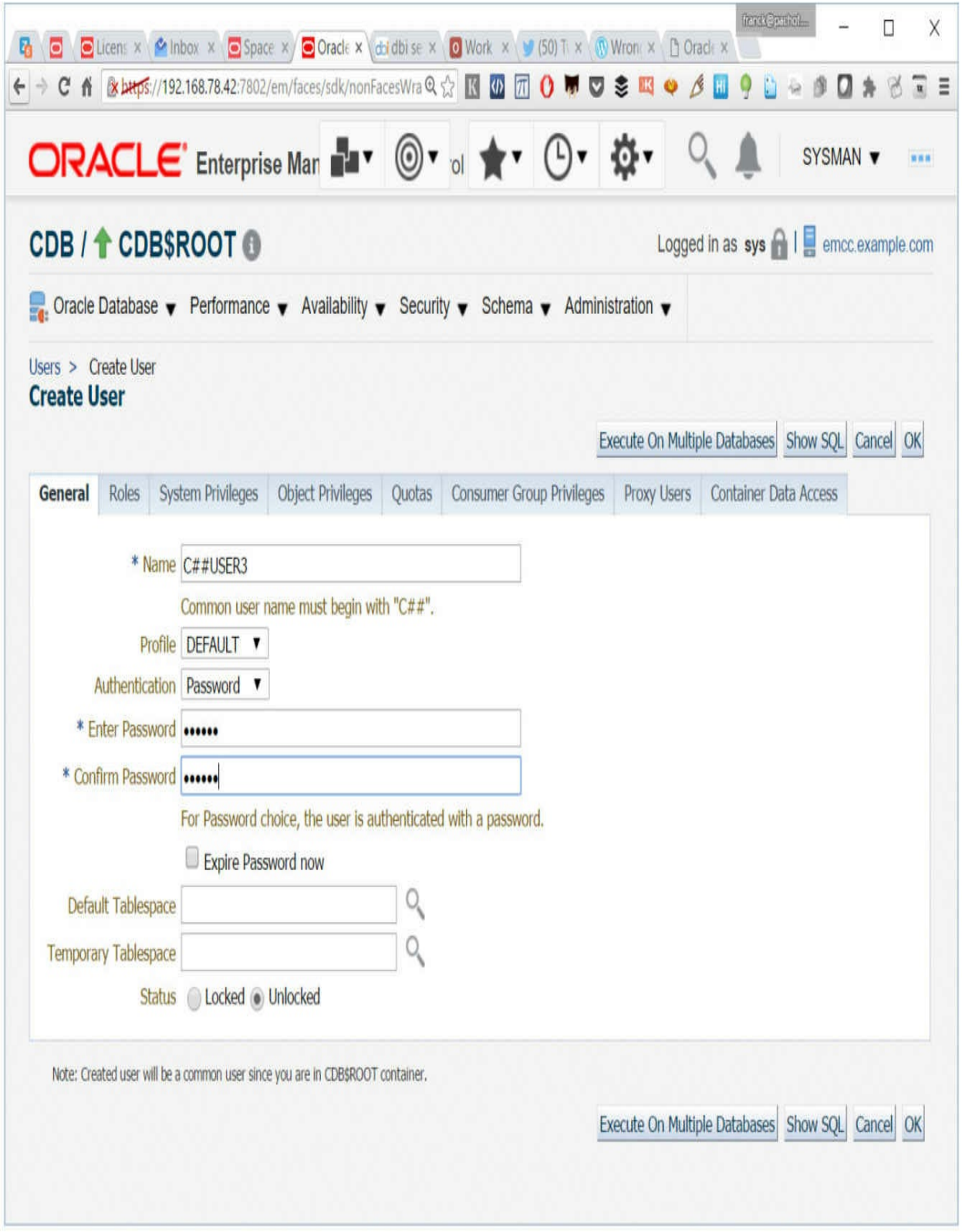

https://192.168.78.42:7802/em/faces/sdk/nonFacesWra
ORACLE Enterprise Man
SYSMAN
CDB / CDB$ROOT
Logged in as sys | emcc.example.com
Oracle Database
Performance
Availability
Security
Schema
Administration
Users > Create User
Create User
Execute On Multiple Databases
Show SQL
Cancel
OK
General
Roles
System Privileges
Object Privileges
Quotas
Consumer Group Privileges
Proxy Users
Container Data Access
* Name C##USER3
Common user name must begin with "C##".
Profile DEFAULT
Authentication Password
* Enter Password
* Confirm Password
For Password choice, the user is authenticated with a password.
Expire Password now
Default Tablespace
Temporary Tablespace
Status Locked Unlocked
Note: Created user will be a common user since you are in CDB$ROOT container.
Execute On Multiple Databases
Show SQL
Cancel
OK

FIGURE 6-2. *Creating a common user in OEM 13*c

We can check the users from CDB$ROOT:

```
SQL> select username, common from dba_users where oracle_maintained='N';

USERNAME   COM
---------- ---
C##USER1   YES
C##USER2   YES
C##USER3   YES
```

In this example, we query only for users we have created recently. Note that the Oracle-maintained users (such as SYS, SYSTEM, and so on) are common users even if they don't have the common prefix.

And here's the same query from our PDB:

```
SQL> alter session set container=PDB;
SQL> select username,common from cdb_users where oracle_maintained='N';

USERNAME   COM
---------- ---
C##USER1   YES
C##USER2   YES
C##USER3   YES
PDBUSER1   NO
PDBUSER2   NO
PDBUSER4   NO
```

In the PDB, we inherit the common users along with those defined locally. Once again, we have not displayed the Oracle-maintained users or the admin user that is generated when the PDB is created.

Local Grant

When in a PDB, you see local users in addition to common ones and you can grant them privileges. Of course, as you are in a PDB, the privileges are granted only at the PDB level. So, for example, if you grant CREATE SESSION to PDBUSER1, this user will be able to connect to its PDB, with the ability to grant that privilege to others, only when done so with the admin option, as follows:

```
SQL> alter session set container=PDB;
Session altered.
SQL> grant create session to PDBUSER1 with admin option container=current;
Grant succeeded.
SQL> connect PDBUSER1/oracle@//localhost/PDB
Connected.
SQL> show user
USER is "PDBUSER1"
```

This is, in fact, the only possibility, because in a PDB you can grant privileges only locally. The `CONTAINER=CURRENT` clause is the default, so it is not mandatory, and if you try something else you get this:

```
ORA-65029: a Local User may not grant or revoke a Common Privilege or Role
```

You can grant to a common user, but this will be for the local context only. For example, here in a PDB we grant the common user C##USER1 the right to connect to our PDB:

```
SQL> grant create session to C##USER1;
Grant succeeded.
```

The default, and only possibility, is `CONTAINER=CURRENT`, which we have omitted. We can see that the grant is there in PDB with `COMMON=NO`:

```
SQL> connect C##USER1/oracle@//localhost/PDB
Connected.
SQL> select * from user_sys_privs;
USERNAME     PRIVILEGE                ADM COMMON
------------ ------------------------ --- ------
C##USER1     CREATE SESSION           YES NO
```

However, the C##USER1 is known to all containers but does not have the CREATE SESSION privilege set for these:

```
SQL> alter session set container=CDB$ROOT;
ERROR:
ORA-01031: insufficient privileges
```

In a PDB, you can create users and roles and grant privileges to them locally. In addition, you inherit common users and roles and can also grant privileges to them locally. It is entirely possible for a common user to have different privileges specified in every PDB; the user is the same, because it is common, but it has different behaviors and privileges in each PDB.

Common Grant

In addition to local grants, we can also grant *common* privileges from CDB$ROOT. For the moment, our three users have no such privileges defined, so let’s look at an example. As we did in our PDB, we can grant CREATE SESSION to the C##USER1 when in CDB$ROOT:

```
SQL> connect / as sysdba
Connected.
SQL> grant create session to C##USER1 container=current;
Grant succeeded.
```

This means that C##USER1 now has the right to connect to

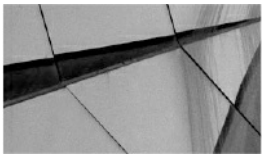
CDB$ROOT, in addition to the right to connect to PDB, which was just granted. We can also equip C##USER1 with the ability to connect to any PDB, whether currently existing or to be created later (which we show for theoretical rather than practical purposes) in the CDB:

```
SQL> grant create session to C##USER1 container=all;
Grant succeeded.
```

So let's see the current grants from CDB$ROOT:

```
SQL> select * from dba_sys_privs where grantee like 'C##%';
GRANTEE                        PRIVILEGE                ADM COMMON
------------------------------ ------------------------ --- ------
C##USER1                       CREATE SESSION           NO  NO
C##USER1                       CREATE SESSION           NO  YES
```

Both grants are there, even if the local ones are redundant, as long as the common ones exist.



NOTE

Be careful with the default value for CONTAINER in a grant statement. The default is CURRENT, even when in CDB$ROOT, which means the privilege will be granted only locally. This is different from the CREATE USER default. Our recommendation is always to specify the `CONTAINER` *clause.*

Let's use the CDB_SYS_PRIVS that shows the result from each container's DBA_SYS_PRIVS (Chapter 9 will detail these cross-PDB views):

```
SQL> select * from cdb_sys_privs where grantee like 'C##%';

GRANTEE                        PRIVILEGE                ADM COMMON     CON_ID
------------------------------ ------------------------ --- ------ ----------
C##USER1                       CREATE SESSION           NO  NO              1
C##USER1                       CREATE SESSION           NO  YES             1
C##USER1                       CREATE SESSION           NO  NO              3
C##USER1                       CREATE SESSION           NO  YES             3
C##USER1                       CREATE SESSION           NO  YES             4
C##USER1                       CREATE SESSION           NO  YES             5
```

These results mirror the output from our previous examples, in that we see a local grant for CDB$ROOT (`CON_ID=1`) and PDB (`CON_ID=3`), and common grants made from CDB$ROOT, which are visible in all containers. From an administration perspective, this lack of clarity does not make sense, and it is better not to mix common and local privileges for the same users.

Conflicts Resolution

Data movement and database plug-in will be addressed in Chapter 9, but you are already aware that multitenant and PDBs bring agility in data movement and cloning. However, you can only imagine the kinds of conflicts you may encounter when plugging in a PDB with a local user that shares the same name as a common one, or vice versa. Oracle will attempt to merge them, but you may have to resort to resolving conflicts manually.

Let's take an example here with the C##USER1. We unplug the PDB and drop the C##USER1:

```
SQL> alter pluggable database PDB close immediate;
Pluggable database altered.
SQL> alter pluggable database PDB unplug into '/tmp/PDB.xml';
Pluggable database altered.
SQL> drop pluggable database PDB;
Pluggable database dropped.
SQL> drop user C##USER1;
User dropped.
```

Then we plug it back in—that is, a PDB that had a common user then plugged into a PDB without one:

```
SQL> create pluggable database PDB using '/tmp/PDB.xml' nocopy;
Pluggable database created.
SQL> alter pluggable database PDB open;
Pluggable database altered.
SQL> select * from cdb_sys_privs where grantee like 'C##%';
GRANTEE                        PRIVILEGE              ADM COMMON     CON_ID
------------------------------ ---------------------- --- ------ ----------C##USER1
CREATE SESSION         NO  NO              3
C##USER1                       CREATE SESSION         NO  YES             3
```

This shows a common user in the PDB, although it is actually unknown from the CDB$ROOT:

```
SQL> select username,common,con_id from cdb_users where username like 'C##%';
USERNAME     COMMON     CON_ID
------------ ------ ----------
C##USER1     YES             3
```

Here you see the process of *inheritance* that we have described as working differently. Common users are not shared; instead, they are propagated to containers. In our example, we dropped the common user but it remained in the unplugged database, with local and common grants all intact.

But no user at CDB$ROOT means we cannot connect:

```
SQL> connect C##USER1/oracle@//localhost/CDB
ERROR:
ORA-01017: invalid username/password; logon denied
```

So are we able to connect to the PDB, because the user is there, with the CREATE SESSION privilege?

```
SQL> connect C##USER1/oracle@//localhost/PDB
ERROR:
ORA-28000: the account is locked
```

In fact, because Oracle was not able to merge the common user automatically with CDB$ROOT, as that user did not exist in CDB$ROOT, the user has been locked until we resolve this issue manually.

If we want to keep this user as the common user, we have to create it from CDB$ROOT:

```
SQL> connect / as sysdba
Connected.
SQL> create user C##USER1 identified by oracle;
create user C##USER1 identified by oracle
*
ERROR at line 1:
ORA-65048: error encountered when processing the current DDL statement in
pluggable database PDB
ORA-01920: user name 'C##USER1' conflicts with another user or role name
```

To avoid such conflicts, we need to close the PDB first:

```
SQL> alter pluggable database PDB close;
Pluggable database altered.
SQL> create user C##USER1 identified by oracle;
User created.
```

All conflicts are then resolved at open, because the common user now matches in both containers:

```
SQL> alter pluggable database PDB open;
Pluggable database altered.
SQL> select username,common,con_id from cdb_users where username like 'C##USER1';
USERNAME     COMMON     CON_ID
------------ ------ ----------
C##USER1     YES             3
C##USER1     YES             1
```

There is no need to unlock the account—everything is OK, and we can successfully connect to the PDB:

```
SQL> connect C##USER1/oracle@//localhost/PDB
Connected.
```

Of course, if we want to connect to CDB$ROOT with this user, we need to grant CREATE SESSION from the root, so it's best to grant it with `CONTAINER=ALL` and revoke the CREATE SESSION privilege that was granted locally.

Keep It Clear and Simple

Be assured that there is nothing to be afraid of here, because it is all very logical if you understand that, physically speaking, the commonality is neither a link nor a logical inheritance, but only the propagation of privileges when DDL is issued. Second, any conflicts that may appear when plugging in a PDB coming from another CDB are resolved when the PDB is opened. And don't forget to check `PDB_PLUG_IN_VIOLATIONS` for more detail.

We can't include all the conflicts that may appear, but let's imagine a common user with a local function to validate the password. You must ensure that the function exists in all PDBs. Our recommendation is to keep it simple, and use the prefix, which enforces a name convention to make it clear about what is common or local. In general terms, common users are mainly for administrators, while local users are for application schemas. Note that if you want to use external authentication with common users, you can match `COMMON_USER_PREFIX` with `OS_AUTHENT_PREFIX`.

With regard to the common user prefix, you should be aware of two additional points. First, the comparison of the prefix is case-insensitive, and

second, even if you change it from its default, the C## is still forbidden for local users, so you will have two prefixes that can lead to ORA-65094: invalid local user or role name.

Note that in 12.2 it is possible to have your own root for your application, which is called an *application container*, where you can manage application user commonality in the same way. There is the `APPLICATION_USER_PREFIX` for this, which is empty by default, and it cannot be set to C##.

CONTAINER_DATA

Common users can see information from the whole CDB, so they can query the V$ views because they show information about the instance, and the instance is common. They can also query the CDB_ views, which collate information from the DBA_ views, from each of the containers. However, the `CONTAINER_DATA` parameter option is a means of implementing fine-grained control, and it enables the administrator to restrict common user access to a subset of containers. Here is an example in which we allow C##USER1 to see V$SESSION common data only from CDB (`CON_ID=0`), CDB$ROOT (`CON_ID=1`), and PDB1 (`CON_ID=4`):

```
SQL> show con_name
CON_NAME
------------------------------
CDB$ROOT
SQL> show user
USER is "SYSTEM"
```

The `ALTER USER` statement to authorize container access from a query on V$SESSION at root level is as follows:

```
SQL> alter user C##USER1 set container_data=(CDB$ROOT,PDB1) for
v$session container=current;
User altered.
```

We can then check what is now permitted, from the `DBA_CONTAINER_DATA` dictionary view:

```
SQL> select * from dba_container_data where username='C##USER1';
USERNAME   DEFAULT_AT OWNER      OBJECT_NAM ALL_CONTAI CONTAINER_
---------- ---------- ---------- ---------- ---------- ----------
C##USER1   N          SYS        V_$SESSION N          CDB$ROOT
C##USER1   N          SYS        V_$SESSION N          PDB1
```

Note that the object to which the restrictions apply is V_$SESSION here, which is the dictionary view on the V$SESSION fixed view. However, the V$SESSION in our statement is actually the public synonym that has been resolved to that view. This is interesting to know, because if you attempt to do the same as SYS you will get an error (ORA-02030: can only select from fixed tables/views): from SYS the V$SESSION is the fixed view itself. So if you want to do something similar from SYS, you have to name the dictionary view directly:

```
SQL> alter user C##USER1 set container_data=(CDB$ROOT,PDB1) for
v_$session container=current;
```

In the following, we are connected as SYSTEM and can, therefore, count sessions from all containers:

```
SQL> select con_id,type,count(*) from v$session group by con_id,type;

    CON_ID TYPE         COUNT(*)
---------- ---------- ----------
         4 USER                1
         1 USER                2
         0 BACKGROUND         51
         5 USER                1
```

But if we attempt the same with the C##USER1 user, we don't see information from PDB2 (`CON_ID=5`), because it was not included in the authorization list:

```
SQL> connect C##USER1/oracle@//localhost/CDB
Connected.

SQL> show con_name

CON_NAME
------------------------------
CDB$ROOT

SQL> show user
USER is "C##USER1"
SQL> select con_id,type,count(*) from v$session group by con_id,type;

    CON_ID TYPE         COUNT(*)
---------- ---------- ----------
         4 USER                1
         1 USER                2
         0 BACKGROUND         51
```

Remember that this restriction is only for queries executed from CDB$ROOT. And, as we have granted the SET CONTAINER to C##USER1 on container PDB2, this user can always switch to it and view the sessions there:

```
SQL> alter session set container=PDB2;
Session altered.

SQL> select con_id,type,count(*) from v$session group by con_id,type;

    CON_ID TYPE         COUNT(*)
---------- ---------- ----------
         0 BACKGROUND         52
         5 USER                2
```

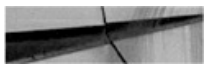
The same result is also possible by connecting directly to PDB2 as C##USER1, which has the CREATE SESSION privilege there. So don't forget to restrict access from all possible avenues: CONNECT_DATA for query from CDB$ROOT, and GRANT/REVOKE within the PDB itself.

Roles

As with users, you can create roles that are either local or common. However, keep in mind that a big advantage of the multitenant architecture is the ability to separate user metadata from system metadata, so don't mix system roles with user roles. If you want to create roles with a subset of the DBA privileges for your administrators, you can create them common, and perhaps grant them to individual local users in PDBs. But as far as user roles go, these should pertain to a specific PDB.

Proxy Users

Proxy users enable you to connect to a user without knowing the user password. This is useful for a DBA who needs to create an object, which can be created only by its owner, such as a database link. It is also a good way to audit who logs in by name and still behaves as if logged in by the schema user. This option is still possible in multitenant, which means that local users can actually proxy through a common user. For example, in PDB1, the DBA enables the common user C##USER1 to be a proxy user for the local user APPOWNER:



```
SQL> alter user APPOWNER grant connect through C##USER1;
User altered.
```

The C##USER1 can then connect to APPOWNER by providing his own password:

```
SQL> connect C##USER1[APPOWNER]/oracle@//localhost/PDB1
Connected.
SQL> show user
USER is "APPOWNER"
```

And from then on, everything functions as if connected with APPOWNER directly.

It's a good practice to disallow direct logon so that you are sure to audit who was connected by their actual usernames. You cannot lock the account, because proxy connections will be blocked as well, but you can set a password that nobody knows. There's also a better option, but this was not yet documented at the time of writing:

```
SQL> alter user APPOWNER proxy only connect;
User altered.
SQL> connect APPOWNER/oracle@//localhost/PDB1
ERROR:
ORA-28058: login is allowed only through a proxy
```

With this clause, the connection can be performed via a proxy user. This behavior can be canceled with the following:

```
SQL> alter user APPOWNER cancel proxy only connect;
User altered.
```

This `proxy only connect` option is interesting, but our recommendation is to wait until it is officially documented before using it.

One final note about proxy users—for security reasons, if you connect as

a common user through a local proxy user, you are locked in the container of the proxy user. Here is an example in which we allowed the common user C##USER1 in PDB1 to connect through the local user ADMIN:

```
alter user C##USER1 grant connect through ADMIN;
```

When the common user connects directly to the PDB, the user can change to another container later, as long as the user has the SET CONTAINER privilege:

```
SQL> connect C##USER1/oracle@//localhost/PDB1
Connected.
alter session set container=CDB$ROOT;
Session altered.
```

However, this operation is not allowed when connected through a local proxy user:

```
SQL> connect ADMIN[C##USER1]/oracle@//localhost/PDB1
Connected.
SQL> alter session set container=CDB$ROOT;
ERROR:
ORA-01031: insufficient privileges
```

This is a security lockdown hard-coded since 12.1, and 12.2 has brought more possibilities to control this through lockdown profiles.

Lockdown Profiles

PDBs bring a new separation of database administrator roles. The DBA administers the CDB but can delegate the administration of individual PDBs. Let’s take an example of a CDB that is a dedicated development environment. The fast and thin provisioning features we will see with snapshot clones make it possible to give a PDB to each developer. Because it is their database, the CDB administrator can grant developers the DBA role

for the PDB, so that developers can do whatever they want there, as long as their privileges are limited to this PDB.

In 12.1, this strategy is almost impossible to implement. Even if the DBA role is granted locally only to a local PDB user, this privilege enables the user to do things that can potentially break the CDB or the server. For example, a local DBA can create files wherever he wishes, can execute any program on the host (which will run as the oracle user), and can generate massive trace files. If we want to limit what a local DBA can do, we need better control over these privileges, and this is why 12.2 introduced lockdown profiles.

Here, connected to CDB$ROOT, we create a profile for our application DBAs:

```
SQL> create lockdown profile APP_DBA_PROFILE;
Lockdown Profile created.

SQL> select * from DBA_LOCKDOWN_PROFILES;
PROFILE_NAME  RULE_TYPE RULE         CLAUSE CLAUSE_OPT OPTION_VAL STATUS
------------- --------- ------------ ------ ---------- ---------- ------
APPDBA_PROF                                                      DISABLE
```

Disable Database Options

With profiles, we can disable access to some features available only with licensed options. For example, if we don’t have the partitioning option, we must be sure that nobody will create a partitioned table, so let’s disable it from our application DBA profile:

```
SQL> alter lockdown profile APPDBA_PROF disable option = ('Partitioning');
Lockdown Profile altered.

SQL> select * from DBA_LOCKDOWN_PROFILES;
PROFILE_NAME  RULE_TYPE RULE         CLAUSE CLAUSE_OPT OPTION_VAL STATUS
------------- --------- ------------ ------ ---------- ---------- ------
APPDBA_PROF   OPTION    PARTITIONING                              DISABLE
```

We can now apply the lockdown profile to PDB1 simply by setting the `pdb_lockdown` parameter for that container:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> alter system set pdb_lockdown=APPDBA_PROF;
System altered.
```

So now let's try to create a partitioned table in that PDB1:

```
SQL> connect admin/oracle@//localhost/PDB1
Connected.
SQL> create table DEMO(id number) partition by hash(id) partitions 4;
create table DEMO(id number) partition by hash(id) partitions 4
             *
ERROR at line 1:
ORA-00439: feature not enabled: Partitioning
```

As you can see, this is impossible because the feature has been disabled.

The `ENABLE` and `DISABLE` clauses of `ALTER LOCKDOWN PROFILE` can also be specified with an `ALL` option:

```
SQL> alter lockdown profile APPDBA_PROF disable option all;
SQL> alter lockdown profile APPDBA_PROF disable option all except =
('Oracle Data Guard');
```

Disable ALTER SYSTEM

The ALTER SYSTEM privilege is very powerful, but with the `GRANT` syntax you can only allow or disallow it. However, with lockdown profiles you have fine-grained control, because you can enable or disable specific clauses of the statement. Let's say, for example, that you want to allow your developers to

kill sessions in their PDB, but no other `ALTER SYSTEM` activities. From CDB$ROOT you can add the following rule:

```
SQL> alter lockdown profile APPDBA_PROF disable statement = ('ALTER SYSTEM')
clause all except = ('KILL SESSION');
SQL> select * from DBA_LOCKDOWN_PROFILES;
PROFILE_NAME  RULE_TYPE RULE         CLAUSE       CLAUSE_OPT OPTION_VAL STATUS
------------- --------- ------------ ------------ ---------- ---------- -------
APPDBA_PROF   OPTION    PARTITIONING                                    DISABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM KILL SESSION                       ENABLE
```

With this, a user in the PDB who has the lockdown profile assigned will get an “ORA-01031: insufficient privileges” message for any `ALTER SYSTEM` command, except an `ALTER SYSTEM KILL SESSION`.

The scope of control can be defined further with the `ALTER SYSTEM SET` command, because you can even control which parameters are allowed. For example, the following will allow only some parameters to be set at the PDB level:

```
SQL> alter lockdown profile APPDBA_PROF disable statement = ('ALTER SYSTEM')
clause = ('SET');
Lockdown Profile altered.
SQL> alter lockdown profile APPDBA_PROF  enable statement = ('ALTER SYSTEM')
clause = ('SET') option = ('undo_retention', 'temp_undo_enabled', 'resumable_
timeout', 'cursor_sharing', 'session_cached_cursors', 'heat_map', 'resource_
manager_plan', 'optimizer_dynamic_sampling');
Lockdown Profile altered.
```

We can query the dictionary to see these defined:

```
SQL> select * from DBA_LOCKDOWN_PROFILES where profile_name='APPDBA_PROF';
PROFILE_NAME  RULE_TYPE RULE         CLAUSE  CLAUSE_OPTION              STATUS
------------- --------- ------------ ------- -------------------------- -------
APPDBA_PROF   STATEMENT ALTER SYSTEM SET                                DISABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     CURSOR_SHARING             ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     HEAT_MAP                   ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     OPTIMIZER_DYNAMIC_SAMPLING ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     RESOURCE_MANAGER_PLAN      ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     RESUMABLE_TIMEOUT          ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     SESSION_CACHED_CURSORS     ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     TEMP_UNDO_ENABLED          ENABLE
APPDBA_PROF   STATEMENT ALTER SYSTEM SET     UNDO_RETENTION             ENABLE
```

From a PDB where this lockdown profile is set, we can set one of these allowed parameters:

```
SQL> alter system set optimizer_dynamic_sampling=4;
System altered.
```

But we will receive a privilege error when trying to set one that is not in the permitted list:

```
SQL> alter system set optimizer_index_cost_adj=1;
alter system set optimizer_index_cost_adj=1
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

When disabling the change of a parameter, we can also define a value to be set at the same time, when the `PDB_LOCKDOWN` parameter is set:

```
alter lockdown profile APPDBA_PROF disable statement=('ALTER SYSTEM')
clause=('SET') option=('cursor_sharing') value=('EXACT');
```

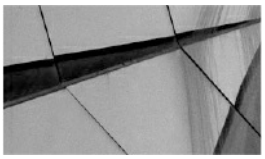
This is an effective means of creating a lockdown profile with several

parameters set to values that cannot be changed later.

Disable Features

We will not go into detail on disable features here, but in the same way that you can disable database options, you can also disable features. For example, the following command disables the specified PL/SQL package usage:

```
SQL> alter lockdown profile APPDBA_PROF disable feature = ('UTL_
HTTP','UTL_SMTP','UTL_TCP');
```



NOTE

You can disable all networking packages with the `NETWORK_ACCESS` *feature name*.

PDB Isolation

In 12.2, in addition to the `PDB_LOCKDOWN` parameter that can be used to set a lockdown profile to limit network access, you can also limit the interaction with the OS file system and processes.

PDB_OS_CREDENTIALS

From a dbms_scheduler job, or through an external procedure, it is possible to run a program on the host server. But, more than likely, you will probably not want to let the PDB administrator run anything with the oracle user privileges. In this case, you can create a credential, from the root, defining the OS user and password, and also including a domain if you are on Windows:

```
exec dbms_credential.create_credential( credential_name=>'PDB1_OS_
USER', username=>'limitedUser', password=>'secret');
```

You can limit a PDB to this user when running jobs or external procedures:

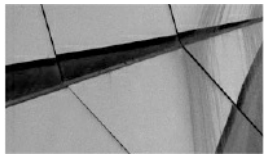
```
alter session set container=PDB1;
alter system set pdb_os_credential=CDB_PDB_OS_USER scope=spfile;
```

PATH_PREFIX

In a similar vein, a PDB administrator can create a directory and write files anywhere on the system. This was not a problem before multitenant, because the DBA controls both the database and the host, but in multitenant you can delegate some administration tasks to the PDB administrator and then need more control on how the PDB admin can interact with the host. Since 12.1, it has been possible to define a `PATH_PREFIX` as the root of all directories created in a PDB, which is then defined with a relative path from there. Note that you cannot change the `PATH_PREFIX` after creation.

CREATE_FILE_DEST

Another way to write files onto a server is to create tablespaces and add datafiles, and this is also an operation the CDB administrator needs to restrict. Starting with 12.1.0.2, you can now set the `CREATE_FILE_DEST` to a directory specific for the PDB, so that datafiles are written there. However, a user with CREATE or ALTER TABLESPACE privileges can still specify a fully qualified filename, and then write everywhere the oracle user can write. OS credentials are not used here.



NOTE

We have opened an enhancement request regarding this gap, and we

hope to have the option to lockdown a PDB administrator to use OMF only, without specifying an absolute file path or a disk group, in the near future.

Transparent Data Encryption

Encrypting data on disk is a key part of a sound security strategy; however, authorized users can still access the data unencrypted. This means that the database has the decryption key onboard, and such encryption does not prevent an attack that accesses the data through the database software.

There are still other paths to compromise the system, such as getting access at OS level or directly accessing the disks, and data encryption does protect against this.

There are essentially two ways to implement a sound security strategy: programming access procedures and encrypting data ourselves, or using Transparent Data Encryption (TDE).

In the first case, it's the application that does the encryption and decryption—that is, the database stores and presents the encrypted data, oblivious to the fact that it is encrypted. The disadvantages of this method are that it is more difficult to implement, and we have to be very careful to implement key management correctly; it's no use encrypting the data if an attacker can compromise the keys himself.

Second, Oracle Database provides, as part of the Advanced Security Option, another solution: TDE. This is part of the Oracle Database, and it's a trusted, proven, and supported solution that is well documented; many DBAs are familiar with it. If we are really serious about security, TDE also supports hardware security modules, which are dedicated pieces of hardware that securely store encryption keys.

Setting Up TDE

Each encrypted database has a master key that is used to generate all additional keys, whether for separate tables or tablespaces. This master key is the only one stored outside of the database itself, and it's the only one that the database needs to open.

There are several reasons why the Oracle Database uses multiple keys

internally, and not just the master. First of all, encrypting data with the master key directly can be slow and expensive, especially hardware-stored keys, which have very limited throughput and can be licensed by capacity. In addition, we can change the master key when we decide to do so—for example, if the key is compromised, or simply to alter master keys on a schedule. Changing the master key re-encrypts the subordinate (table) keys, but not the data in the database itself.

Let's go through an example of setting up TDE for a multitenant database. In this example we will use a software keystore, not a hardware one.

Setting Up the Keystore Location

A *software keystore* is essentially a file in a specified directory. Oracle Database also supports wallets for storing secure information, which enables scripts, for example, to log in without hard-coding passwords. Conceptually, wallets are very similar, and the way to administer them is also similar to the TDE keystore. In fact, in 11*g*, both were called "wallets," and some syntax and documentation still refer to both by this name; even the location for each can default to the same value. Nevertheless, we recommend keeping these separate, as they do contain different data with different purposes.

The location of the keystore is defined in sqlnet.ora. It is important that you know that each CDB or non-CDB has its own keystore, so if we have multiple databases running on the same host, we must configure a path that is different for each database.

We could create multiple sqlnet.ora files and make sure each database is started with the correct one, but the easiest method—and the least error-prone—is to include the ORACLE_SID in the path.

Sqlnet.ora is in the TNS_ADMIN path, which is $ORACLE_HOME/network/admin by default. The entry for database encryption can be set as follows:

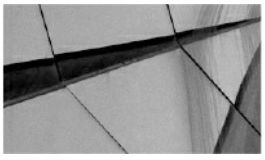
```
ENCRYPTION_WALLET_LOCATION=
  (SOURCE=
    (METHOD=FILE)
    (METHOD_DATA=
      (DIRECTORY=$ORACLE_BASE/WALLET/$ORACLE_SID)))
```

Creating the Keystore

Creating the keystore involves one simple command in the root container. All the commands that work with the keystore need SYSKM or ADMINISTER KEY MANAGEMENT privilege.

```
SQL> administer key management create keystore '/u01/app/oracle/WALLET/
CDBSRC' identified by "AVeryLongPassword";

keystore altered.
```



NOTE

This command requires a full path to the directory where the keystore will be created. This must be the same as we specified (or will specify) in sqlnet.ora, and be aware that the command does not check that these two paths match. This command is new in Oracle Database 12c, replacing `alter system set encryption` *commands from earlier versions.*

A keystore created in this way will require the same password to open. Oracle Database also supports autologin wallets, which alleviate the need to specify the password. (Refer to the documentation for more details.)

Setting Up Using Cloud Control

Most of these operations can also be performed using Enterprise Manager Cloud Control. Figure 6-3 shows the TDE home screen, from which we can

set up the keystore and manage the keys.

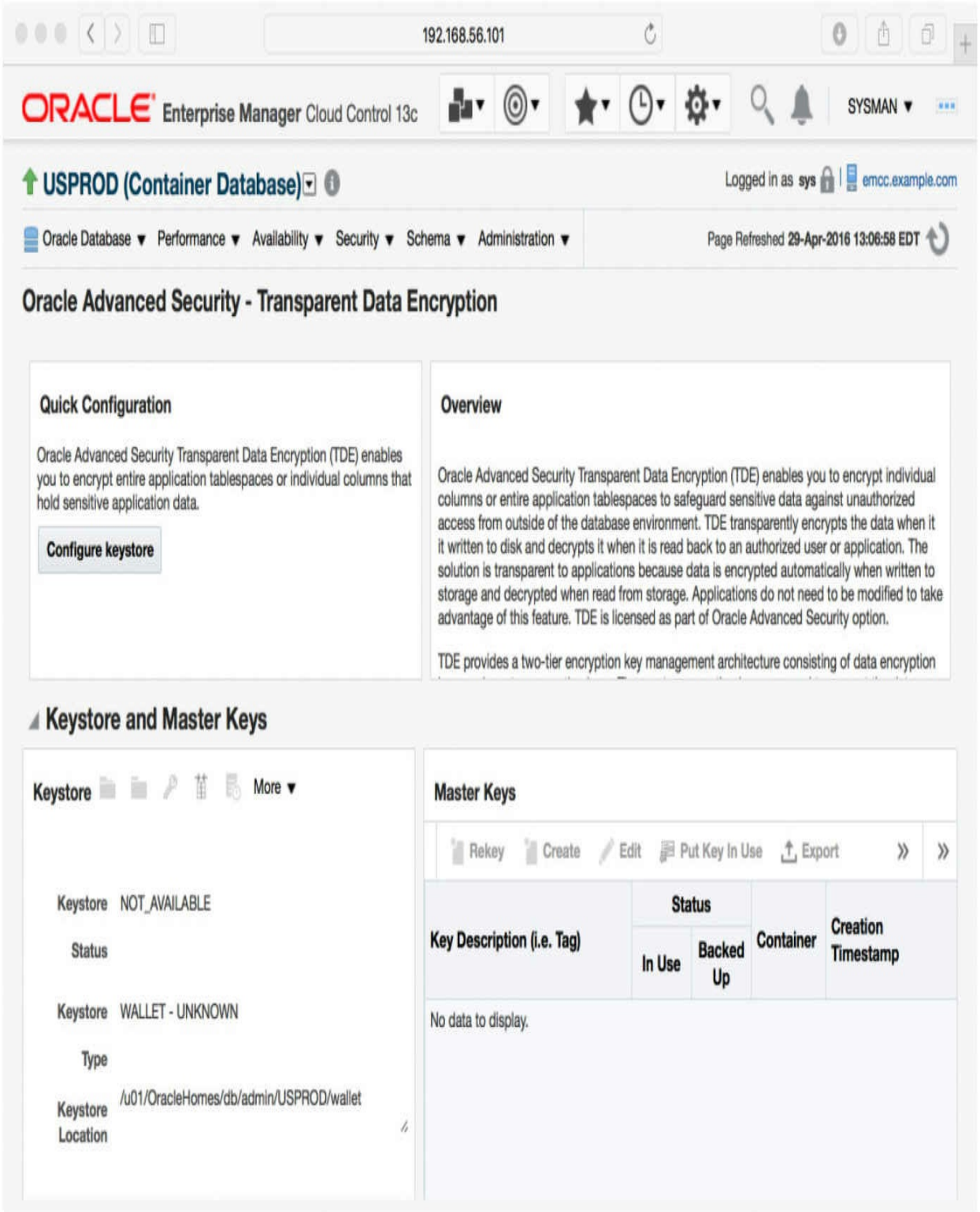

192.168.56.101
ORACLE Enterprise Manager Cloud Control 13c
SYSMAN
USPROD (Container Database)
Logged in as sys
emcc.example.com
Oracle Database
Performance
Availability
Security
Schema
Administration
Page Refreshed 29-Apr-2016 13:06:58 EDT
Oracle Advanced Security - Transparent Data Encryption
Quick Configuration
Oracle Advanced Security Transparent Data Encryption (TDE) enables you to encrypt entire application tablespaces or individual columns that hold sensitive application data.
Configure keystore
Overview
Oracle Advanced Security Transparent Data Encryption (TDE) enables you to encrypt individual columns or entire application tablespaces to safeguard sensitive data against unauthorized access from outside of the database environment. TDE transparently encrypts the data when it it written to disk and decrypts it when it is read back to an authorized user or application. The solution is transparent to applications because data is encrypted automatically when written to storage and decrypted when read from storage. Applications do not need to be modified to take advantage of this feature. TDE is licensed as part of Oracle Advanced Security option.
TDE provides a two-tier encryption key management architecture consisting of data encryption
Keystore and Master Keys
Keystore
More
Keystore NOT_AVAILABLE
Status
Keystore WALLET - UNKNOWN
Type
Keystore Location /u01/OracleHomes/db/admin/USPROD/wallet
Master Keys
Rekey
Create
Edit
Put Key In Use
Export
Key Description (i.e. Tag)
Status
In Use
Backed Up
Container
Creation Timestamp
No data to display.

FIGURE 6-3. *TDE in Enterprise Manager Cloud Control*

Opening the Keystore

Before anyone can use the keystore, we must open it in the root container and then in the PDBs. Only then we can set up the master key and read or modify the encrypted data.

```
SQL> administer key management set keystore open identified by
"AVeryLongPassword";

keystore altered.
```

We can also use the `CONTAINER` clause syntax introduced by multitenant, and then in the root container, open the keystore in all PDBs:

```
SQL> administer key management set keystore open identified by
"AVeryLongPassword" container=all;

keystore altered.
```

It is in this step that we will find out whether the configuration of the wallet location has been done correctly—if not, we will receive an error like this:

```
SQL> administer key management set keystore open identified by identified
by  "AVeryLongPassword";
ENCRYPTION_WALLET_LOCATION=
administer key management set keystore open identified by identified by
"AVeryLongPassword"
*
ERROR at line 1:
ORA-28367: wallet does not exist
```

Creating the Master Key

Now that we created an *empty* keystore, the obvious next step is to generate a master key and store it in the keystore; this is where things differ for a multitenant database. As noted earlier, each CDB has one keystore; however, in that keystore, each PDB has its own master key. In Oracle Database 12*c* Release 2, we should be able to specify keystores specific for each PDB, too, meaning that plugging and cloning would require just copy of the keystore, not an export.

This means a bit more work is required during the setup, but it also means that PDBs can be unplugged and cloned and moved into another CDB (see Chapter 9), with their own key that we move along. This was always an issue with transportable tablespaces, in that they are encrypted by their database key, and the target database, if also encrypted, can't accommodate two different master keys.

To create the new key, we run the following command:

```
SQL> administer key management set key using tag 'our key 1' identified
by "AVeryLongPassword" with backup using 'backup1';

keystore altered.
```

This can be run in the container for which we want to create the key—either CDB$ROOT or a PDB. And, again, we can add the `CONTAINER=ALL` clause when running the command in CDB$ROOT to create keys in all PDBs. However, this also sets the same descriptive comment (the tag) for all keys, which may not be desired.

It is also mandatory to specify a backup of the wallet, so that, should the operation go wrong, we would still have the previous copy and would not lose the keys. So, after adding the first key, we now have two files in the wallet directory, as in the following example:

```
-rw-r--r-- oracle oinstall 4022 Mar  3 19:26 ewallet.p12
-rw-r--r-- oracle oinstall 2400 Mar  3 19:26 ewallet_2016030319264839_
backup1.p12
```

Verifying the Created Keys

We can now simply check which keys have been created so far:

```
SQL> select key_id, tag, user, con_id from V$ENCRYPTION_KEYS;

KEY_ID                                                   TAG        USER       CON_ID
-------------------------------------------------------- ---------- ---------- -------
ATuQBA32eU+bv5beU2m0vNwAAAAAAAAAAAAAAAAAAAAAAAAAAAA PDB1 key   SYS            3
AXlCULHXvU/2v83pq36VH7UAAAAAAAAAAAAAAAAAAAAAAAAAAAA our key 1  SYS            1
```

As you can see, the key ID is a long, generated base64 encoded value, while the tag value is a human-readable comment.

Encrypting the Data

Encryption of the data is defined at the PDB level, using the same syntax and rules as in a non-CDB database.

TDE can encrypt column(s) in a table, which we can request by simply adding the `ENCRYPT` keyword:

```
create table scott.emp_enc(
  empno number(4) primary key,
  ename varchar2(10),
  job varchar2(9),
  mgr number(4),
  hiredate date,
  sal number(7,2) ENCRYPT,
  comm number(7,2) ENCRYPT,
  deptno number(2));

Table created.
```

The second way TDE can work is at the tablespace level, encrypting all data in this entity:

```
SQL> create tablespace tbsenc
  datafile '/u01/app/oracle/data/CDBSRC/PDB1/tbsenc01.dbf'
  size 100m
  encryption using 'AES256'
  default storage (ENCRYPT);

Tablespace created.
```

We can see a list of all TDE-encrypted columns in DBA_ENCRYPTED_COLUMNS:

```
SQL> select * from dba_encrypted_columns;

OWNER      TABLE_NAME COLUMN_NAM ENCRYPTION_ALG                SAL INTEGRITY_AL
---------- ---------- ---------- ------------------------------ --- ------------
SCOTT      EMP_ENC    SAL        AES 192 bits key               YES SHA-1
SCOTT      EMP_ENC    COMM       AES 192 bits key               YES SHA-1
```

And the list of encrypted tablespaces is displayed in v$encrypted_tablespaces:

```
SQL> select ts#, encryptionalg, encryptedts, status from  v$encrypted_tablespaces;

       TS# ENCRYPT ENC STATUS
---------- ------- --- ----------
         5 AES256  YES NORMAL
```

Plug and Clone with TDE

When a PDB is copied/moved to a new CDB, the target CDB needs to know its encryption key. If the operation is a clone, Oracle will do this for us

automatically. But if we plug in/unplug a PDB, we must ship the master key along with the XML file and the datafiles. This is achieved by exporting the key on the source into a password-protected file:

```
SQL> administer key management export encryption keys with secret
"exportPassword" to '/home/oracle/exportPDB1.p12' identified by
"AVeryLongPassword";

keystore altered.
```

We then proceed with the plug-in as per normal (see Chapter 9). The PDB will refuse to open and will instead remain open in restricted mode. But now that the target database knows about the PDB, we can connect or switch to it and import the keystore export file, as follows:

```
SQL> administer key management import encryption keys with secret "exportPassword"
from '/home/oracle/exportPDB1.p12' identified by oracle with backup;

keystore altered.
```

From here we can close and then open the PDB again, and access the encrypted data.

TDE Summary

TDE is a feature that is, on paper at least, easy to use, but it comes with its own limitations, and it is a key management process that is complicated at times.

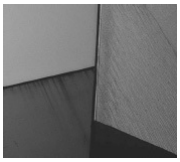
It’s important that we build a thorough understanding as to how this feature works if we are going to use it. After all, this is data security we are talking about—an area where it is often difficult to assess whether we have done something wrong, until it’s too late.

TDE is a topic that warrants a book on its own, and we have only lightly scratched the surface here. The message is clear, however: TDE still functions the same as before, with the only change in multitenant being one

key for each PDB, and thus multiple keys in a single CDB.

Summary

In a multitenant environment, particularly in the cloud, all the features covered in this chapter are must knows—and must use. Some of these can be distilled into simple directives, such as: don’t have all your database administrators connecting as sysdba, and the system administrators should have their common usernames to administer the CDB. It is probable that all other users will be local to PDBs, so their actions are appropriately isolated. Moving beyond this type of access protection, encryption is a powerful means of preventing illegitimate access to data, although this is not sufficient protection alone. Even with the best security policy, an error may occur that results in some data being lost or corrupted. This brings us to the most important facets of data protection, backup and recovery, which are covered in the next chapter.



PART III

Backup, Recovery, and Database Movement

CHAPTER 7

Backup and Recovery

Backup and recovery is an exciting topic. Creating backups is straightforward, but many DBAs do not spend much time digging into this area until an actual restore and recovery is required. Then all the books and notes are dusted off in search of the correct commands or processes to follow. Although this might not be optimal, it does highlight something of utmost importance when talking about backup and recovery: documentation. And this brings us to the goal of this chapter—to document the key concepts and areas relative to backup and recovery when using the Oracle Database 12*c* Multitenant option.

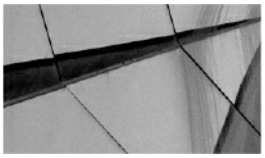
In this chapter we will first review the basics to highlight a number of key aspects that will help you quickly establish effective backup and recovery procedures in your multitenant environment. What you will notice as we progress is that, in many cases, it does not make a difference whether you are using a container database (CDB) or a non-CDB; overall, the principles are similar, if not identical, for both.

Back to Basics

Our encouragement is this: Do not be afraid of backup and recovery. In fact, the more time you spend on planning and testing your backups, the easier the second part—restore and recovery—will become. But before we dive into the detail, let’s consider two key areas:

- Hot versus cold backups

- ARCHIVELOG mode versus NOARCHIVELOG mode



NOTE

In this chapter a number of examples will make use of the RMAN `TAG` *option. Strictly speaking, this is not necessary, but it is a recommended option in certain scenarios because it makes it easier for you to identify specific backups. For more detail on the use of the* `TAG` *option, refer to the Oracle online documentation.*

Hot vs. Cold Backups

Nowadays, there are few references to cold backups. So what is this and why would you use it? In short, cold backups (also called consistent backups) are created when a database is not open for transactions—that is, it has been shut down with the `IMMEDIATE`, `TRANSACTIONAL`, or `NORMAL` option. The effect of one of these clean shutdowns is that when the database is restored following this type of backup, no additional recovery is required to bring it to a consistent state, because it was in this state when the backup was performed.

In this day and age, most companies simply cannot afford downtime on their primary systems, so the cold backup is not the ideal backup method, because it requires an outage (planned downtime) on your primary database. You may ask, why have downtime when creating backups at all, if this is not required? The answer depends on an organization's requirements, but a cold backup is still an option for the modern DBA, even when using the multitenant option.

Performing a cold backup is simple: the database is closed when backups are being performed. If RMAN *is not* used to perform cold backups, the database must be completely shut down. When RMAN *is* used, the database must be in a *mounted* state. Here's an example of creating a cold backup of a CDB:

```
RMAN> connect target /
RMAN> shutdown immediate;
RMAN> startup force dba;
RMAN> shutdown immediate;
RMAN> startup mount;
RMAN> backup format '/backups/%U' database;
RMAN> backup format '/backups/cfc-%U' current controlfile;
RMAN> alter database open;
RMAN> alter pluggable database all open;
```

In this example, a consistent backup of the CDB is performed and the backup is stored on disk in the /backups folder. By default, RMAN backup sets are used with the *backup* command. If required, you can also perform a cold backup using image copies; to do so, you would replace the fifth line in the preceding code with this:

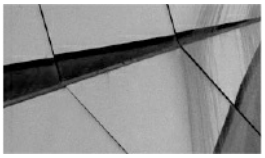
```
RMAN> backup as copy format '/backups/%U' database;
```

As you can see, creating consistent cold backups is easy. The database can be in NOARCHIVELOG mode when you performing them. But the downtime required that renders these types of backups is not acceptable for most.

This brings us to hot, or online/inconsistent, backups. In contrast, when creating these backups, the database may be online—open read/write. However, there is one key requirement when you are performing hot backups: the database must be in ARCHIVELOG mode. For most DBAs, this is likely to be the default option when creating a new database in any case. Enabling ARCHIVELOG mode is easy and brings with it the advantage of enabling you to perform backups while the database is in full use. Yes, backups created in this fashion are considered inconsistent, but by using the archive logs that are generated during the process, the backup can be restored and recovered to a consistent state so that the database can be opened again.

Archive logs assist in resolving Oracle split blocks, which may occur during hot backups. Oracle data blocks, the smallest units of data used by a database (which is made up of multiple operating system blocks), include identifying start and end markers. The start and end markers of blocks are compared during recovery, and if they do not match, the block is considered

inconsistent and the redo copy of the block is required to recover (reconstruct) the block to a consistent state.



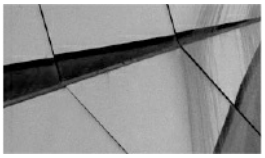
NOTE

For more detail on split blocks, refer to Oracle Support Note 1048310.6 and the Oracle Database 12c documentation regarding the `LOG_BLOCKS_DURING_BACKUP` *initialization parameter.*

Enabling ARCHIVELOG mode in a CDB is no different from doing it in a non-CDB configuration:

```
SQL> connect / as sysdba
SQL> shutdown immediate;
SQL> startup mount;
SQL> alter database archivelog;
SQL> alter database open;
```

You can use the SQL command `archive log list` to confirm whether a database has ARCHIVELOG mode enabled, or by reviewing the `V$DATABASE.LOG_MODE` value.



NOTE

When you enable ARCHIVELOG mode in a CDB, an outage is required, because the database needs to be restarted in a mount state. Also note that ARCHIVELOG mode can be set only at the CDB level.

To perform a basic backup on a CDB database that is open and has ARCHIVELOG mode enabled, run the following commands:

```
RMAN> connect target /
RMAN> backup format '/backups/db-%U' database;
RMAN> backup format '/backups/arc-%U' archivelog all;
RMAN> backup format '/backups/cfc-%U' current controlfile;
```

With the fundamentals of hot and cold backups now covered, let's review some of the default RMAN configuration options before we get into the details of backup and recovery concepts in a container database environment.

RMAN: The Default Configuration

By default, to back up a database using RMAN, the simplest command is `backup database`. It doesn't get any easier than that! For most purposes, however, this simple command is not really adequate; to get the most from your backups and to provide more options, you have to go a bit further than this. But before we launch into this, we need to discuss the default RMAN configuration options.

When you run the `backup database` command, a number of default options are invoked behind the scenes. Many of them are perfectly acceptable and may never need changing, but by adjusting a key few, you can make backup and recovery of your Oracle 12*c* database even easier.

Use NLS_DATE_FORMAT

The `NLS_DATE_FORMAT` environment variable is highly recommended. In case you hadn't noticed, when running RMAN commands, the default date format is limited to displaying only the day, month, and year:

```
BS Key  Type LV Size       Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ ---------------
10      Full    1.32G      DISK        00:00:08     07-MAR-16
```

Notice that there is no time option in the output. For many, this might not be an issue, but having additional date/time output in the log files and on the

screen when running RMAN commands is extremely useful:

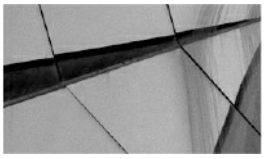
```
BS Key  Type LV Size       Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ -------------------
10      Full    1.32G      DISK        00:00:08     07/03/2016:22:38:02
```

To enable better display of date and time when using RMAN, simply set the `NLS_DATE_FORMAT` environment variable prior to starting your RMAN session:

```
oracle@linux3[/home/oracle]: export NLS_DATE_FORMAT=dd/mm/yyyy:hh24:mi:ss
```

List and Adjust Default Configuration

The default RMAN configuration can be reviewed in two ways: you can use the RMAN `show all` command, or you can review the details in `V$RMAN_CONFIGURATION`.



NOTE

`V$RMAN_CONFIGURATION` will show only the nondefault values. If you have not changed the default configuration, no rows will be returned.

For example, to adjust the default configuration to allow the autobackup of the control file (a recommended setting) to a specified location, including the enabling of compressed backup sets for the disk type backups, you can execute the following at the RMAN prompt:

```
RMAN> configure controlfile autobackup on;
RMAN> configure controlfile autobackup format for device type disk to '/backups/cfa-%F';
RMAN> configure channel device type disk format '/backups/%U';
RMAN> configure device type disk backup type to compressed backupset;
```

Once this has been executed, you can review the settings with the following:

```
RMAN> show all;
```

In this state, you can run the `backup database` command and RMAN will take your default settings into account.

RMAN Backup Redundancy

Backups are important, and to help you protect this resource, RMAN enables you to duplicate your backups—up to four of them in fact. You can take this additional measure to ensure that your backups are protected from media failure or human error. The `COPIES` clause can be used with the *backup* command to specify how many backup copies should be created, or, if you prefer, you can update the default configuration and specify the `COPIES` clause as part of your default configuration. The command that follows demonstrates how this can be done with the RMAN `backup` command:

```
RMAN> backup copies 2 pluggable database XPDB1
        format '/backups/CDB1-%U', '/backups2/CDB1-%U';
```

In this example, while connected to CDB$ROOT, the pluggable database (PDB) XPDB1 is backed up to /backups, and the backup is then duplicated to /backups2 as well. To enable the duplicate option in the default configuration, using two copies for both the datafile and archive log backups, for example, you can use the following commands while connected to the CDB$ROOT:

```
RMAN> configure channel device type disk
       format '/backups/CDB1-%U', '/backups2/CDB1-%U';
RMAN> configure datafile backup copies for device type disk to 2;
RMAN> configure archivelog backup copies for device type disk to 2;
```

The SYSBACKUP Privilege

In Oracle Database 12*c*, separation of administration duties has been extended from the few basic options in earlier versions. One of the new system privileges introduced is SYSBACKUP, which can be granted to users who need to perform backup and recovery operations. A user with the SYSBACKUP privilege will be restricted to allow only backup and recovery operations. This permission can also be granted to a local user account in a PDB, equipping the user with backup and recovery permissions on a specific PDB. Let's review the different connection options available with this role.

Connecting as the SYS user with SYSDBA permission:

```
RMAN> connect target /
```

Connecting as a common user with SYSBACKUP permission:

```
SQL> connect c##backup as sysbackup;
```

Using RMAN to connect to the CDB root:

```
RMAN> connect target '"c##backup@cdb1 as sysbackup"';
```

Using a local PDB account with SYSBACKUP permission:

```
RMAN> connect target 'localadmin@pdb1 as sysbackup'
```

For more detail on the new administrative privileges and user security when using Oracle Database 12*c* Multitenant, see Chapter 6.

CDB and PDB Backups

There are two key aspects to backups in an Oracle Database 12*c* Multitenant environment: backups at the CDB level, and then those at the PDB level. You can connect to a PDB and perform backup and recovery operations, albeit with a few restrictions, which we will get to later in this chapter. You can use RMAN to perform backups on a CDB or PDB, and later we will cover two additional options: Oracle Cloud Control and Oracle SQL Developer. But first let's focus on using RMAN via the command line interface.

Before jumping in, note an important change introduced in Oracle Database 12*c*. When connected to CDB$ROOT, if we specify only the key word `DATABASE` in our *backup*, *restore*, or *recover* command, it applies to the whole database (CDB root and all its PDBs). Here's an example:

```
RMAN> backup database;
```

However, if you are connected to a PDB, the commands apply specifically to the PDB you are connected to. In Oracle Database 12*c*, Oracle introduced a new RMAN clause, `PLUGGABLE DATABASE`, which enables you to perform tasks on specific PDBs, as follows, while connected to the CDB$ROOT:

```
RMAN> backup pluggable database PDB1;
```

In the next sections, we will outline additional syntax changes that have been introduced.

CDB Backups

In most cases you will find that backups will be scheduled and performed at

the CDB level, so that CDB$ROOT and PDB$SEED, as well as all the other PDBs associated with a CDB, will be backed up. But this does not mean you cannot be more specific, and in this section we will show you, by way of example, how easy it is to perform backups while connected to the CDB root as target.

Full CDB Backups

When you perform a full (whole) CDB backup, the following files should be included:

- The control file
- All datafiles (CDB$ROOT, PDB$SEED, and all PDBs)
- All archived logs

Backing up the SPFILE is recommended, but in most cases it can easily be rebuilt, so this is not mandatory. If you do have autobackup of the control file enabled (which is highly recommended), the SPFILE will automatically be backed up together with the control file when any structural changes are made in the database.

Multiple options are available to you when backing up the whole CDB database. Backup sets are common and the default, but using image copies can be useful, and if kept locally—in the fast recovery area (FRA), for example—they can be switched rapidly for fast recovery. This is especially the case if the image copies are kept up-to-date with incremental backups applied regularly. Backing up the entire CDB is perhaps the most common method and is demonstrated in the following examples, which assume that the RMAN environment is configured with the settings outlined earlier, and the autobackup of the control file is enabled.

This first example uses the most basic form for backing up the whole CDB. The command is executed while connected to CDB$ROOT:

```
RMAN> backup database plus archivelog;
```

In the next example, while connected to CDB$ROOT, we take this a little

further and explicitly specify the use of compressed backup set output, along with the location for this output, specified in the `FORMAT` option. In addition, we include a `TAG`, which can be extremely useful to identify particular backups:

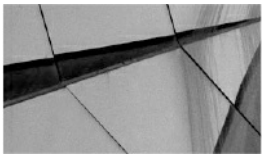
```
RMAN>  run {
  backup as compressed backupset
         format '/backups/%U' database tag='FULLCDB';
  backup as compressed backupset
         format '/backups/a-%U' archivelog all tag='FULLCDB';
  backup format '/backups/c-%U' current controlfile tag='FULLCDB';
}
```

Next, we make use of image copies. Here we assume that sufficient redundant storage is available, and a disk group called +DBBACKUP exists, which will be used to store the image copies. Archived logs not yet backed up will be written to the /backups folder on the file system. The database can then be backed up as follows (while connected to the root container):

```
RMAN> run {
    backup as copy
          format '+DBBACKUP ' database tag='CDBIMGCOPY';
    backup as compressed backupset
          format '/backups/a-%U' archivelog all not backed up;
 }
```

The end result is that a backup copy of the CDB database, including all PDBs, can be located in the +DBBACKUP disk group. If something were to happen to any of the primary files in this example on the +DATA disk group, we could switch to an image copy quickly, followed by recovery of the image copy, which in some cases can be much faster than restoring from a backup set. This method of backup can be extremely useful, but note that sufficient storage is required to keep the copy of the database. It is also possible to adjust the `FORMAT` specification and write the image copies to a file system

location.



TIP

To identify your RMAN backup sessions easily in V$SESSION, you can use the RMAN command to set the command ID for the session: `RMAN> set command id to "FULLCDBBKP";`*. In the end, you will be able to use the* `CLIENT_INFO` *and look for the value* `id=FULLCDBBKP`*.*

At this stage, you can use the `LIST` command to view the backup details, including listing the image copies that have been created; by using the `PLUGGABLE DATABASE` flag, you can specify a specific PDB to provide listings for. So to detail the image copies for PDB1, created in the third example in the preceding examples, use the `LIST COPY` command as follows:

```
RMAN> list copy of pluggable database PDB1 tag='CDBIMGCOPY';
List of Datafile Copies
========================
Key     File S Completion Time     Ckp SCN    Ckp Time             Sparse
------- ---- - -------------------- ---------- -------------------- ------
22      8    A 08/03/2016:21:46:59 4719331    08/03/2016:21:46:57 NO
        Name: +DBBACKUP/CDB1/2D5B1DE419B57F1CE053E902000A271D/DATAFILE/
system.278.905982417
        Tag: IMGCOPY
        Container ID: 3, PDB Name: PDB1

19      9    A 08/03/2016:21:46:39 4719275    08/03/2016:21:46:36 NO
        Name: +DBBACKUP/CDB1/2D5B1DE419B57F1CE053E902000A271D/DATAFILE/
sysaux.270.905982397
        Tag: IMGCOPY
        Container ID: 3, PDB Name: PDB1

27      10   A 08/03/2016:21:47:11 4719345    08/03/2016:21:47:11 NO
        Name: +DBBACKUP/CDB1/2D5B1DE419B57F1CE053E902000A271D/DATAFILE/
users.299.905982431
        Tag: IMGCOPY
        Container ID: 3, PDB Name: PDB1
```

These examples demonstrate how easy it can be to back up an entire CDB.

Partial CDB Backups

In some cases, you might not want to back up the full CDB, but only a subset. This is where the `PLUGGABLE DATABASE` keywords are invoked to get the task done. The examples in this section are executed while connected to the root container. In this first example, we will back up only CDB$ROOT, PDB$SEED, and PDB1:

```
RMAN> backup pluggable database "CDB$ROOT","PDB$SEED", PDB1;
```

As illustrated, you can selectively specify the PDBs to include in the backup. Note that when performing a partial CDB backup, CDB$ROOT should be included.

In the second example, only the CDB root is backed up. Both possible command options for this are listed:

```
RMAN> backup pluggable database "CDB$ROOT";
```

or

```
RMAN> backup database root;
```

It is also possible to take a more fine-grained approach and back up only a specific tablespace. In this example the command will back up the USERS tablespace located in PDB1; it is executed while RMAN is connected to CDB root:

```
RMAN> backup tablespace PDB1:USERS;
```

Note that the USERS tablespace needs to be prefixed with its PDB name of origin; if the name is omitted, RMAN will attempt to back up this tablespace from CDB$ROOT since we are connected to CDB$ROOT.

The final example demonstrates how we can back up the USERS tablespaces in CDB$ROOT, as well as in PDB1:

```
RMAN> backup tablespace USERS, PDB1:USERS;
```

CDB Reporting Using RMAN

When connected to the CDB as target, the `report schema` command will list details for the CDB$ROOT, PDB$SEED, and all the PDBs associated with this CDB. The next example shows the RMAN `report schema` command output on a CDB using Automatic Storage Management (ASM) and Oracle Managed Files (OMF), with one PDB called PDB1. A closer look at the Tablespace column also reveals that the CDB$ROOT tablespaces do not have a prefix, whereas all other PDBs, including those in PDB$SEED's tablespaces, have this defined.

```
RMAN> report schema;
using target database control file instead of recovery catalog
Report of database schema for database with db_unique_name CDB1
List of Permanent Datafiles
===========================
File Size(MB) Tablespace           RB segs Datafile Name
---- -------- -------------------- ------- ------------------------
1    900      SYSTEM               YES     +DATA/CDB1/DATAFILE/system.278.905794969
3    770      SYSAUX               NO      +DATA/CDB1/DATAFILE/sysaux.276.905795015
4    175      UNDOTBS1             YES     +DATA/CDB1/DATAFILE/undotbs1.270.905795049
5    270      PDB$SEED:SYSTEM      NO      +DATA/CDB1/28E530CCFE9C1B52E0534940E40A
7A88/DATAFILE/system.266.905795111
6    5        USERS                NO      +DATA/CDB1/DATAFILE/users.268.905795051
7    510      PDB$SEED:SYSAUX      NO      +DATA/CDB1/28E530CCFE9C1B52E0534940E40A
7A88/DATAFILE/sysaux.277.905795111
8    280      PDB1:SYSTEM          NO      +DATA/CDB1/2D5B1DE419B57F1CE053E902000A27
1D/DATAFILE/system.273.905795557
9    610      PDB1:SYSAUX          NO      +DATA/CDB1/2D5B1DE419B57F1CE053E902000A27
1D/DATAFILE/sysaux.285.905795557
10   5        PDB1:USERS           NO      +DATA/CDB1/2D5B1DE419B57F1CE053E902000A27
1D/DATAFILE/users.265.905795573

List of Temporary Files
=======================
File Size(MB) Tablespace           Maxsize(MB) Tempfile Name
---- -------- -------------------- ----------- --------------------
1    137      TEMP                 32767       +DATA/CDB1/TEMPFILE/temp.274.905795107
2    58       PDB$SEED:TEMP        32767       +DATA/CDB1/28E530CCFE9C1B52E0534940E40A
7A88/DATAFILE/temp012016-03-06_17-45-40-432-pm.dbf
3    20       PDB1:TEMP            32767       +DATA/CDB1/2D5B1DE419B57F1CE053E902000A
271D/TEMPFILE/temp.297.905795561
```

To display additional details of backups and image copies that have been created, you can run the `LIST` command. It is easy to use and can help you quickly identify backup sets and image copies. The commands detailed next are a small subset of those that you may find helpful:

```
RMAN> list backup;
RMAN> list backup tag=FULLCDB;
RMAN> list backup of datafile 1;
RMAN> list backup of pluggable database PDB1;
RMAN> list copy of database;
RMAN> list copy of datafile 1;
RMAN> list copy of pluggable database PDB1;
```

PDB Backups

Now that you have seen how you can back up an entire CDB, let's focus on PDBs, which can be backed up while connected to the CDB root as the target in RMAN, or you can connect directly to a PDB to perform a full or partial backup. In this section we will review both of these methods.

Full PDB Backups

You can perform full backups of a PDB in a number of ways. You can back up the PDB while connected to the CDB$ROOT as target, as shown in the following example. First, the entire PDB, PDB1, is backed up with a single command:

```
RMAN> backup pluggable database PDB1;
```

If you are looking at using image copies, you can also specify this as follows:

```
RMAN> backup as copy pluggable database PDB1;
```

This command can be extended to include more than just one PDB in the backup. Simply specify the PDBs in a comma-delimited list, provided that you are connected to the CDB root as the target. Here's an example:

```
RMAN> backup as copy pluggable database PDB1, PDB2, PDB5;
```

The next option is to connect directly to a specific PDB as the target, and perform a full backup of the connected PDB. Here's an example:

```
RMAN> connect target sys@pdb1
target database Password:
connected to target database: CDB2:PDB1 (DBID=1739880102)
```

Or, here's an example using a local account with SYSBACKUP permission:

```
RMAN> connect target 'localadmin@pdb1 as sysbackup';
target database Password:
connected to target database: CDB1:PDB1 (DBID=1543305986)
```

Once you are connected to the specified PDB, you can execute the `backup database` commands without the `PLUGGABLE` keyword, because all `backup` commands in this context will apply only to the specified PDB. If you try to use the `PLUGGABLE DATABASE` syntax instead of only `DATABASE`, the following error will be generated:

```
RMAN-07538: Pluggable Database qualifier not allowed when connected to
a Pluggable Database
```

Several key options are available when creating a backup of a specific PDB while connected directly to it, as shown with PDB1 here:

```
RMAN> backup database;
RMAN> backup database tag='PDB1';
RMAN> backup as copy database tag='PDB1_IMCOPY';
RMAN> backup as compressed backupset format '/backups/%U' database TAG='PDB1';
```

When connected to a PDB as a target, you can use the `LIST` or `REPORT` command to display information specific to this PDB. In terms of the archived logs, you can show them with the `LIST` command, but other operations, such as backup, restore, or delete, are not permitted while connected to the PDB as a target.

Partial PDB Backups

While connected to a specific PDB as a target, you are allowed to perform only operations that are specific to that PDB—so you cannot, for example, perform backups of other PDBs in this context.

Once connected to a PDB, you can perform backups of datafiles or tablespaces as per normal:

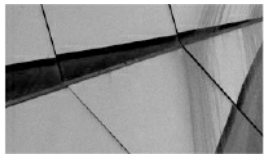
```
RMAN> connect target sys@pdb1
target database Password:
connected to target database: CDB1:PDB1 (DBID=1543305986)
RMAN> backup tablespace users tag='PDB1_USERSTS';
```

If you then connect to the CDB root and list the backups with `TAG=PDB1_USERSTS`, you will see the following output, noting the highlighted line showing the container ID and PDB name.

```
RMAN> list backup tag='PDB1_USERSTS';
using target database control file instead of recovery catalog
List of Backup Sets
===================
BS Key  Type LV Size       Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ -------------------
78      Full    1.02M      DISK        00:00:00     09/03/2016:11:40:08
        BP Key: 78   Status: AVAILABLE  Compressed: YES  Tag: PDB1_USERSTS
        Piece Name: /backups/3lr01t8o_1_1
  List of Datafiles in backup set 78
  Container ID: 3, PDB Name: PDB1
  File LV Type Ckp SCN    Ckp Time             Abs Fuz SCN Sparse Name
  ---- -- ---- ---------- ------------------- ----------- ------ ----
  10      Full 5020501    09/03/2016:11:40:08              NO    +DATA/CDB1/2D5B1DE419
B57F1CE053E902000A271D/DATAFILE/users.265.905795573
```

It is possible to perform partial PDB backups from the CDB root. This enables you to back up specific PDBs or specific tablespaces from them. Following are two such examples, run while connected to the CDB root as target. In the first, we back up the USERS tablespace in the CDB$ROOT, as well as the USERS tablespace from PDB1 and PDB2:

```
RMAN> backup tablespace USERS, PDB1:USERS, PDB2:USERS tag='ALL_USERSTS';
```



NOTE

The datafile number within a CDB is unique.

The second example illustrates backing up specific datafiles from various PDBs. Note that you do not have to specify the PDB names, but you must know its datafile number within the CDB.

```
RMAN> backup datafile 6,10 tag='USERDF';
…
RMAN> list backup tag='USERDF';
List of Backup Sets
===================
BS Key  Type LV Size       Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ -------------------
86      Full    1.02M      DISK        00:00:00     09/03/2016:11:49:57
        BP Key: 86   Status: AVAILABLE  Compressed: YES  Tag: USERDF
        Piece Name: /backups/3tr01tr5_1_1
  List of Datafiles in backup set 86
  File LV Type Ckp SCN    Ckp Time             Abs Fuz SCN Sparse Name
  ---- -- ---- ---------- ------------------- ----------- ------ ----
  6       Full 5021431    09/03/2016:11:49:57              NO    +DATA/CDB1/DATAFILE/
users.268.905795051

BS Key  Type LV Size       Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ -------------------
87      Full    1.02M      DISK        00:00:00     09/03/2016:11:49:58
        BP Key: 87   Status: AVAILABLE  Compressed: YES  Tag: USERDF
        Piece Name: /backups/3ur01tr6_1_1
  List of Datafiles in backup set 87
  Container ID: 3, PDB Name: PDB1
  File LV Type Ckp SCN    Ckp Time             Abs Fuz SCN Sparse Name
  ---- -- ---- ---------- ------------------- ----------- ------ ----
  10      Full 5021432    09/03/2016:11:49:58              NO    +DATA/CDB1/2D5B1DE419
B57F1CE053E902000A271D/DATAFILE/users.265.905795573
```

PDB Reporting

As noted earlier, when connected to a PDB as the target, you can view only details relating to that particular PDB—for example, using the `report schema` command.

Restrictions

When you are connected to a PDB as your target, some restrictions are placed on backups, including the following:

- You are not permitted to back up, restore, or delete archived logs while connected to a PDB as target. Tasks related to archive logs must be managed from the CDB$ROOT. Note that during the recovery process (if connected to a PDB as target), if required, RMAN will

restore any archived logs needed.

- You cannot update the default RMAN configuration using the `CONFIGURE` command, because this is managed from the CDB level.

Do Not Forget Archive Logs!

As a rule, when performing backups, you should always be sure to include the archive logs in the backup schedule. In Oracle Database 12*c*, you back up the archive logs from CDB$ROOT. Following are a number of basic variations that can be used for this.

Here's how to back up all available archive logs:

```
RMAN> backup archivelog all;
```

Adding an additional step, you can purge the archive logs once they are backed up:

```
RMAN> backup archivelog all delete input;
```

Here's how to back up all archived logs not backed up at least twice:

```
RMAN> backup archivelog all not backed up 2 times;
```

It is also possible to update the archive log deletion policy for the default RMAN configuration. For more detail on managing archive logs with RMAN, see the Oracle Database 12*c* online documentation.

Recovery Scenarios

Several levels of recovery are possible in a CDB environment. For example, media recovery can be performed for the entire CDB, or for just one or

multiple PDBs. As with non-CDB configurations, you can perform media recovery on database files, tablespaces, and even at the block level.

Instance Recovery

Instance recovery is specific to a CDB as a whole. There is only one instance for the entire CDB, rather than instances allocated on a per-PDB basis. This means that there is a single redo stream, and during crash recovery the redo information is used to recover the instance when the CDB root is opened. This process requires that the datafiles be consistent with the control file, so the redo information is used to roll back any uncommitted transactions at the time of the instance failure. And once the CDB root is opened, all PDBs will be in a mount state.

When reviewing the alert log during system startup, you will notice messages similar to the following, indicating instance crash recovery:

```
Beginning crash recovery of 1 threads
 parallel recovery started with 3 processes
…
 read 168075 KB redo, 9027 data blocks need recovery
2016-03-09T17:05:23.221280+13:00
Started redo application at
 Thread 1: logseq 58, block 88657, offset 0
2016-03-09T17:05:23.232114+13:00
Recovery of Online Redo Log: Thread 1 Group 1 Seq 58 Reading mem 0
  Mem# 0: +DATA/CDB1/ONLINELOG/group_1.279.905795099
  Mem# 1: +DATA/CDB1/ONLINELOG/group_1.282.905795101
2016-03-09T17:05:23.391522+13:00
…
Completed redo application of 15.28MB
2016-03-09T17:05:24.589190+13:00
Completed crash recovery at
 Thread 1: RBA 60.68923.16, nab 68923, scn 0x0000000000518eb5
 9027 data blocks read, 8527 data blocks written, 168075 redo k-bytes read
…
Thread 1 advanced to log sequence 61 (thread open)
Thread 1 opened at log sequence 61
  Current log# 1 seq# 61 mem# 0: +DATA/CDB1/ONLINELOG/group_1.279.905795099
  Current log# 1 seq# 61 mem# 1: +DATA/CDB1/ONLINELOG/group_1.282.905795101
Successful open of redo thread 1
..
Completed: alter database open
```

The next section will focus on restore and recovery of an entire (whole) CDB, and will also explore the full restore of a PDB. Point-in-time recovery (PITR), including the use of Flashback Database, will be covered in more detail in Chapter 8.

Restore and Recover a CDB

The restore and recovery of the CDB database includes all contained PDBs, assuming that you followed the steps outlined in the previous section to perform the backups. If backups are performed correctly, executing restore and recovery procedures becomes a much easier task.

Restore a CDB Using a Cold Backup

As mentioned, you can back up a full CDB using a cold backup, in which it is

possible for the database to run in NOARCHIVELOG mode.

The following steps can be used to perform a full restore. In this example, the autobackup of the control file is used (in this case the file was called cfc-CDB3-c-603345334-20160309-01') to restore the SPFILE and control file. However, if you already have the SPFILE, you can skip the step of restoring the SPFILE and continue with the next:

```
RMAN> startup nomount;
RMAN> restore spfile from '/backups/cfc-CDB3-c-603345334-20160309-01';
RMAN> shutdown immediate;
RMAN> startup nomount;
RMAN> restore controlfile from '/backups/cfc-CDB3-c-603345334-20160309-01';
RMAN> alter database mount;
RMAN> restore database;
RMAN> alter database open resetlogs;
RMAN> alter pluggable database all open;
```

Perform a Complete Recovery of a CDB

Performing a full restore and complete recovery of a CDB is almost as easy as performing a full backup. Again, the assumption is that you have followed the steps from earlier in this chapter to perform the full backup of the CDB, and that these backups are available. In this scenario, note that all required archive logs are available to perform the recovery. The archive logs may still be available on disk, or perhaps they are part of the backup as well.

The RMAN connection is initiated to the CDB$ROOT as target:

```
RMAN> startup mount;
RMAN> restore database;
RMAN> recover database;
RMAN> alter database open;
RMAN> alter pluggable database all open;
```

Perform a Complete Recovery of CDB$ROOT

The process required to restore the CDB$ROOT container alone, when all other PDBs are intact without any issues, is similar to the previous steps:

```
RMAN> startup mount;
RMAN> restore database "CDB$ROOT";
RMAN> recover database "CDB$ROOT";
RMAN> alter database open;
RMAN> alter pluggable database all open;
```

Recover from a Lost CDB$ROOT Tablespace

The steps required to perform a full restore and recovery of a CDB$ROOT tablespace should be executed while connected to the CDB$ROOT as target. It is not necessary to have the CDB in a mounted state, or even to have the PDBs closed, as long as the tablespace in question is not the CDB$ROOT SYSTEM or UNDO tablespace. The tablespace must be offline when the restore is performed.

The steps to follow to restore and recover the USERS tablespace in this context are shown here:

```
RMAN> alter tablespace USERS offline;
RMAN> restore tablespace USERS;
RMAN> recover tablespace USERS;
RMAN> alter tablspace USERS online;
```

If a restore or recovery is required for the SYSTEM or UNDO tablespace, the CDB root must be in a mounted state before the `restore` and `recover` commands are executed.

Recover from a Lost CDB$ROOT Datafile

Recovery from a lost datafile in the root container can be accomplished without restarting the CDB in a mounted state, as long as the datafiles do not belong to a SYSTEM or UNDO tablespace (which otherwise would necessitate a restart into a mounted state, prior to executing restore or

recovery). The following commands demonstrate restore and recovery of a lost datafile from the USERS tablespace (datafile 6 in this case) located in the CDB$ROOT. The connection here is with the CDB root as the target:

```
RMAN> alter database datafile 6 offline;
RMAN> restore datafile 6;
RMAN> recover datafile 6;
RMAN> alter database datafile 6 online;
```

Recover from Loss of Tempfiles

If you have lost a tempfile due to media failure, you have two possible options to resolve this. The first is simply to add a new tempfile to the temporary tablespace and drop the old file. The second option will take effect on the next restart of the CDB. The tempfile will be re-created on the next restart of the CDB root, and if it belongs to a PDB, on the next open of the PDB the tempfile will be created. Following is an extract from the alert log showing this:

```
…
Errors in file /u01/app/oracle/diag/rdbms/cdb3/CDB3/trace/CDB3_dbw0_28682.trc:
ORA-01186: file 201 failed verification tests
ORA-01157: cannot identify/lock data file 201 - see DBWR trace file
ORA-01110: data file 201: '/u01/app/oracle/oradata/CDB3/temp01.dbf'
2016-03-09T10:55:17.459152+13:00
File 201 not verified due to error ORA-01157
2016-03-09T10:55:17.465595+13:00
Re-creating tempfile /u01/app/oracle/oradata/CDB3/temp01.dbf
…
```

Restore and Recover a PDB

In this section we will cover the recovery steps with specific focus at the PDB level, and the key scenarios will be addressed.

Restore and Recover a PDB

If a PDB is lost, you can restore and recover it without having the root container in a mounted state, although the restore and recovery process is managed via the root container as the target connection. If the PDB is still open, you must close it before executing the restore and recovery commands:

```
RMAN> alter pluggable database PDB1 close;
RMAN> restore pluggable database PDB1;
RMAN> recover pluggable database PDB1;
RMAN> alter pluggable database PDB1 open;
```

What if your backups included image copies of the PDB? Using image copies may be very effective in reducing any downtime, rather than waiting for a restore of the full PDB from backup. When performing restore and recovery operations, time is of the essence, and the faster a database can be restored and brought back online for users the better. So if image copies are available for a PDB, why not use them? You can switch to use the image copies, create a new backup of the PDB as an image copy in the original location, and then schedule a switch back during a quiet period.

Here are the steps to follow:

```
RMAN> alter pluggable database PDB1 close;
RMAN> list copy of pluggable database PDB1;
RMAN> switch pluggable database PDB1 to copy;
RMAN> recover pluggable database;
RMAN> alter pluggable database PDB1 open;
```

Recover a Lost PDB System Datafile

If a datafile from a PDB's SYSTEM or UNDO tablespace (when using local undo) is lost, the recovery process must be performed from the root container. The root container, including all other PDBs, can be open read-write, but the affected PDB must be in a mounted state.

The steps to perform a restore of datafile 8, which is the SYSTEM

datafile for PDB1 in this particular example, can be restored from the CDB$ROOT as follows:

```
RMAN> alter pluggable database PDB1 close;
RMAN> restore datafile 8;
RMAN> recover datafile 8;
RMAN> alter pluggable database PDB1 open;
```

Recover a Lost PDB Nonsystem Datafile

If the lost datafile in a PDB is not part of the SYSTEM or UNDO (if using local undo) tablespace, the restore and recovery process can occur from either the CDB$ROOT or the PDB. The PDB does not have to be in a mounted state, but the datafile must be taken offline, if it's not already down, prior to performing the restore and recovery.

In the following example, the restore and recovery is performed while connected to the PDB. The `report schema` command is used to confirm the datafile number, which, in this case, is datafile 10 (one of the USERS tablespace datafiles):

```
RMAN> connect target sys@pdb1
RMAN> report schema;
RMAN> alter database datafile 10 offline;
RMAN> restore datafile 10;
RMAN> recover datafile 10;
RMAN> alter database datafile 10 online;
```

As with our earlier example, if image copies are available, the option also exists to switch to using this. Note that if you do so, the location of the image copy will most likely differ from the current file location. You can replace the `restore datafile 10` command with the following if a copy is available (use the `list copy` command to identify datafile copies):

```
RMAN> switch datafile 10 to copy;
```

Recover a Lost PDB Tablespace

If a tablespace needs to be restored and recovered in a PDB, two options are available. This can be done within the PDB (as long as this is not a SYSTEM or UNDO tablespace), or it can be done from the CDB$ROOT. The tablespace must be taken offline prior to starting the restore, and this should be done from within the PDB:

```
RMAN> connect target sys@pdb1
RMAN> alter tablespace users offline;
```

Then you can restore the tablespace from within the PDB with the following commands:

```
RMAN> restore tablespace USERS;
RMAN> recover tablespace USERS;
RMAN> alter tablespace USERS online;
```

Alternatively, you can perform the restore and recover commands from the CDB$ROOT:

```
RMAN> restore tablespace PDB1:USERS;
RMAN> recover tablespace PDB1:USERS;
```

We've covered a number of the key areas you should be familiar with when performing complete recovery operations in an Oracle Database 12*c* Multitenant environment. In the next chapter we will look at using PITR and the Flashback Database options.

RMAN Optimization Considerations

Working with large databases is a prevalent trend today, and it is less common to find the need to purge old data. Data is retained for longer periods, and often it will never be removed. This increase in overall database sizes affects backup and recovery, and optimizing backup and recovery operations is becoming more and more important. Furthermore, backup windows are shrinking, while additional load on production systems needs to be kept to a minimum. To assist with this issue, Oracle introduced a number of options, including the following:

- Incremental backups
- Block change tracking
- Multiple channel backups (parallel backup and recovery)
- Multisection backups
- RMAN backup optimization

In this section we will review the first four of these five options, which are highly recommended when using multitenant.

Incremental Backups

Two key options are available for incremental backups: differential and cumulative backups. By default, the differential option is selected, and when using this method all database blocks changed since the previous backup will be included. Table 7-1 illustrates a schedule that employs differential backups; first an incremental level 0—a full backup—is performed on Monday, followed by a differential backup each day thereafter.

Mon	Tue	Wed	Thu	Fri
FULL				
	|----->			
		|----->		
			|----->	
				|----->

TABLE 7-1. *Differential Incremental Backup*

If you need to perform a restore on Thursday, the full backup from Monday, as well as the incremental backups from Tuesday and Wednesday, will be required. One of the potential risks when relying on incremental differential backups is that if you were to lose one of the incremental backups, you might not be able to restore and recover to the required point in time. This risk can be reduced by ensuring that archive logs are backed up at least twice, so that if one day's worth of backups is lost, it may still be possible to use archive logs to recover past this point. For example, to back up PDB1 using differential incremental backups, we'd follow these steps.

1. Create the base incremental level 0 backup:

```
RMAN> backup incremental level 0 pluggable database PDB1;
```

2. Create the differential incremental level 1 backup:

```
RMAN> backup incremental level 1 pluggable database PDB1;
```

With the cumulative incremental backup, as with the differential incremental backup, only changed blocks are backed up; in this instance, however, the backup includes the data since the last base incremental level 0 backup. Table 7-2 illustrates this.

Mon	Tue	Wed	Thu	Fri
FULL				
--------	|----->			
--------	--------	|----->		
--------	--------	--------	|----->	
--------	--------	--------	--------	|----->

TABLE 7-2. *Cumulative Incremental Backup*

First an incremental level 0 backup is created, and every cumulative incremental backup following this will back up all the changed blocks since the last full level 0 backup. This method does extra work in backing up blocks more than once and requires additional storage, but the overall risk is less. When using this approach, the keyword `CUMULATIVE` must be used—so, for example, if we want to back up PDB1 using this method, the following commands can be run.

1. Create the base incremental level 0 backup:

```
RMAN> backup incremental level 0 pluggable database PDB1;
```

2. Create the differential incremental level 1 backup:

```
RMAN> backup incremental level 1 cumulative pluggable database PDB1;
```

Incremental backups can be extremely useful, especially in large database environments. But this brings us to our next point, which is that during an incremental backup, the datafile blocks are scanned to identify changed blocks in need of backup. This process can take time, perhaps even as long as a full backup itself. To make this faster, Oracle introduced block change tracking.

Block Change Tracking

When you're working with incremental backups, the use of block change tracking is highly recommended. Note, however, that this is an Enterprise Edition feature and cannot be used in Standard Edition. This feature cannot

be enabled within a PDB, but instead should be enabled while connected to the CDB$ROOT. If you do attempt to enable block change tracking while connected to a PDB, an ORA-65040 error will be generated, as shown in the following example:

```
ORA-65040: operation not allowed from within a pluggable database
```

Enabling or disabling this option is easy, especially when using OMF. The block change-tracking file is created and will grow in 10MB chunks as needed; the default location when using OMF is DB_CREATE_FILE_DEST. If you are not using OMF, the filename and location should be specified manually. The file will track all the changed blocks in the database and can be enabled with the following command:

```
SQL> alter database enable block change tracking;  --- if using OMF
SQL> alter database enable block change tracking using file '/u01/oracle/oradata/DEV/
bc-track.dbf';   --- if not using OMF
```

Block change tracking can be disabled with the following command:

```
SQL> alter database disable block change tracking;
```

Using this option together with incremental backups is recommended for multitenant environments, especially for larger configurations, because it will assist in creating faster backups and reducing resource consumption.

Multiple Channel Backup

Using multiple channels may also lead to faster backup and recovery times. However, having sufficient CPU and I/O capability to accommodate this is equally important; otherwise, you may slow down operations when using multiple channels. If you want to use this option by default, you can update the default configuration and specify the parallelism parameter as shown

here:

```
RMAN> configure device type disk parallelism 4;
```

You may also allocate multiple channels as part of your backup and recovery commands:

```
RMAN> run {
 allocate channel ch1 type disk;
 allocate channel ch2 type disk;
 backup database plus archivelog;
 }
```

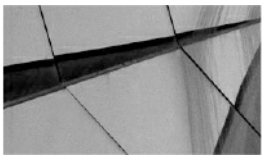
Multisection Backups

Combining incremental backups with multiple channels can help speed backup and recovery operations. Prior to 12*c*, with large files, this did not always provide significant benefit, but since 12*c*, parallel incremental backups can be taken one step further. As of Oracle Database 12*c* Release 1 (12.1), RMAN now also supports multisection incremental backups, as well as the use of multisection with image copy backups. The `COMPATIBLE` parameter must be set to 12.0.0 or higher to allow for this. Note that using this option forces the `FILESPERSET` option to be set to 1 for backup sets. The following syntax can be used to back up a PDB database called XPDB2 with large datafiles using multisection backups, and the command is executed from the CDB$ROOT:

```
RMAN> configure device type disk parallelism 4;
RMAN> backup incremental level 1 section size 200M pluggable database XPDB2;
```

As mentioned, the use of multisection can now also be used with image copy backups. In the following example, an image copy backup is taken of the XPDB PDB with the `SECTION SIZE` clause specified as follows:

```
RMAN> configure device type disk parallelism 4;
RMAN> backup as copy section size 200M pluggable database XPDB2;
```



NOTE

If the section size specified is larger than the file to be backed up, multisection backups will be ignored for the file and not be used.

The Data Recovery Advisor

Backup and recovery can become incredibly complex, especially when diagnosing and deciding on the correct repair options. To assist with this, Oracle introduced the Data Recovery Advisor in 11*g*, and it has been extended and improved in Oracle Database 12*c*. The Data Recovery Advisor can help list potential corrective actions, and, if you want, even perform these tasks for you. You can use either RMAN or Oracle Enterprise Manager Cloud Control to obtain information provided by the Data Recovery Advisor and execute the required tasks.

At the time of writing, the Data Recovery Advisor could be used in a non-CDB as well as in a single-instance (non-Oracle RAC) CDB. The Data Recovery Advisor is not supported in Oracle RAC configurations. It may be run only from the CDB$ROOT, not from within a PDB, and if this is attempted an RMAN-07536 error will be displayed.

The standard Data Recovery Advisor commands to be invoked from within the CDB$ROOT include these:

```
RMAN> list failure;
RMAN> advise failure;
RMAN> repair failure;
RMAN> change failure;
```

Block Corruption

Block corruption is a nightmare for DBAs, and only the most fortunate avoid coming across it in their careers. Fortunately, Oracle provides tools that can be used to ensure your databases are valid and that block corruption is not hiding under the covers. Running these health checks on a regular basis is, therefore, highly recommended.

The `VALIDATE` command is very easy to use to perform these checks. When the command is run from RMAN, a detected problem will trigger a failure assessment. This will then be recorded in the Automatic Diagnostic Repository (ADR), where it can be accessed by the previously discussed Data Recovery Advisor.

The `VALIDATE` command can be executed against a running database (CDB, non-CDB, and PDBs) and also on backups, including RMAN backup sets and image copies.

The following is a short listing of some of the options available with the `VALIDATE` command; for more detail see the Oracle 12c Database documentation.

```
RMAN> validate database;
RMAN> validate pluggable database PDB1;
RMAN> validate backupset 13;
RMAN> validate copy of pluggable database PDB1;
RMAN> backup validate pluggable database PDB1;
RMAN> restore pluggable database PDB1 validate;
```

Using Cloud Control for Backups

Backup and recovery operations can also be performed from within Oracle Enterprise Manager Cloud Control. Figure 7-1 and Figure 7-2 illustrate some of the options available.

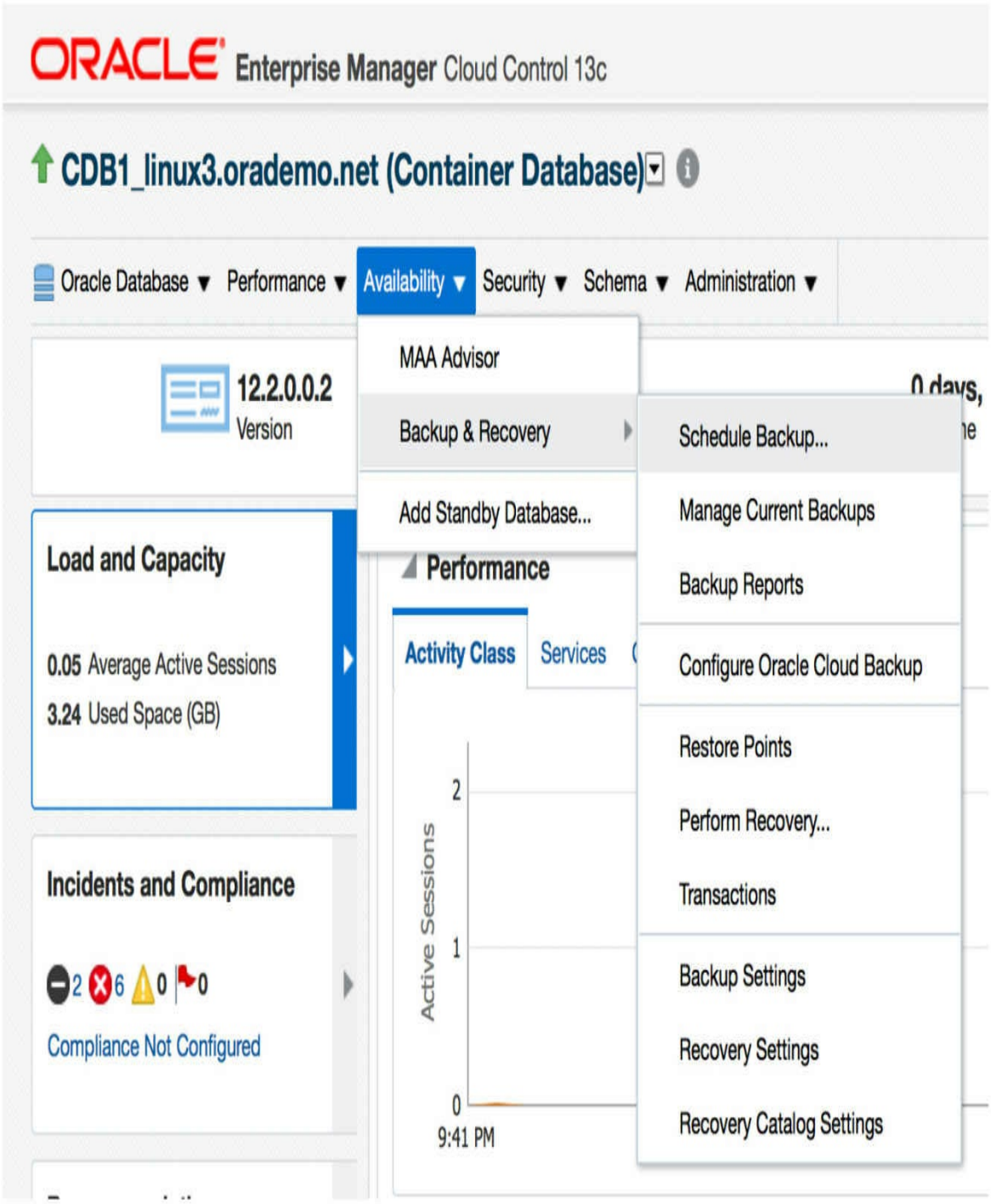


Figure 7-1. *Backup and recovery options in Cloud Control 13*c

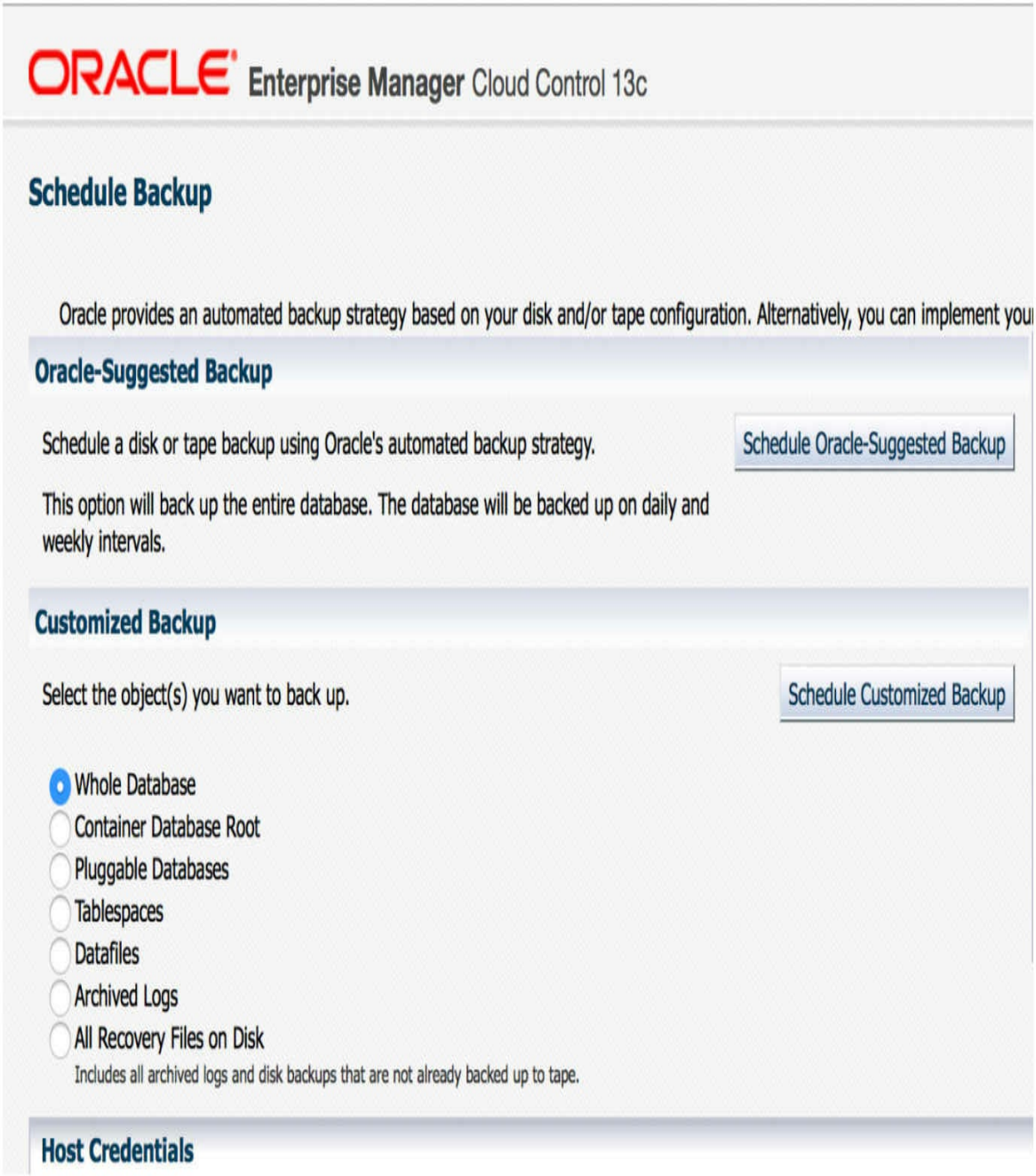


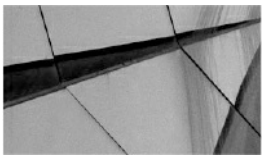
Figure 7-2. *Backup options in Cloud Control 13*c

For example, if Availability | Backup & Recovery (Figure 7-1) is selected, you will see a number of options to help guide you through creating

and scheduling backups, along with performing restores and recoveries. Selecting the Schedule Backup option presents the options shown in Figure 7-2, including the ability to back up a specific PDB.

Back Up to the Cloud

With the growing interest in using cloud-based solutions, a number of options are available for backing up Oracle Databases to the cloud. Two key players present such offerings: Amazon Web Services (AWS) and the Oracle Cloud Services.



NOTE

If you sign up for Oracle Cloud Services to perform RMAN backups to the Oracle Cloud, you can use the required RMAN encryption free of charge. This is even the case when using Oracle Standard Edition, although a specific patch will need to be applied to a Standard Edition environment to allow this.

To configure the Oracle Database Backup Service, you first need to create an Oracle Cloud account, and then sign up for both the Backup Service and the Oracle Storage Cloud Service. Here is a summary, at a high level, of the steps required to back up to the Oracle Cloud:

1. Create an Oracle Cloud account and sign up for the backup and storage services.
2. Download the Oracle Database Backup Cloud Module.
3. Install the module. See readme.txt file for parameter details (run as oracle software owner):

```
# java -jar opc_install.jar -serviceName Storage \
-identityDomain yourDomain -opcId 'your-account@example.com'  \
-opcPass '<yourpassword>' -walletDir /u01/app/oracle/cloud/wallet  \
-configFile /u01/app/oracle/cloud/conf/ocb.ora -libDir /u01/app/oracle/cloud/lib
```

   Once the module is configured, three files are created: the wallet with

your cloud account details loaded, a parameter file, and the module library.

4. When allocating the SBT_TAPE channel, options need to be supplied to indicate the use of the library and your configuration:

```
allocate channel ch1 type SBT_TAPE PARMS 'SBT_LIBRARY=/u01/app/oracle/
cloud/lib/libopc.so, ENV=(OPC_PFILE=/u01/app/oracle/cloud/conf/ocb.ora)';
```

5. You must use encryption when using the Oracle Database Backup Service with the cloud module configured here. The backup encryption can be enabled using three possible options: password encryption, Transparent Data Encryption (TDE), or a combination of both. The quick and easy method is to use the password option, which is enabled by running the following command:

```
RMAN> SET ENCRYPTION ON IDENTIFIED BY 'YourPasswordHere' only;
```

6. Run your RMAN backup commands:

```
RMAN> SET ENCRYPTION ON IDENTIFIED BY 'YourPasswordHere' only;
RMAN> run {
  allocate channel ch1
  type SBT_TAPE PARMS   'SBT_LIBRARY=/u01/app/oracle/cloud/lib/libopc.so,
  ENV=(OPC_PFILE=/u01/app/oracle/cloud/conf/ocb.conf)';
  backup as compressed backupset database plus archivelog;
backup current controlfile;
}
```

For more detail on using the Oracle Cloud, and how to work with advanced configurations in the Cloud backup module, see the Oracle Database Backup Service documentation.

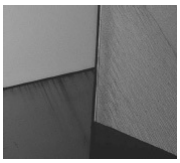
Summary

Oracle Database backup and recovery can be one of those areas that busy DBAs may not spend a lot of time on yet, but, as we have shown, it is one of the most critical areas in managing a multitenant environment. The emergence of this new multitenant paradigm (CDBs and PDBs) has generated a number of explanatory analogies, and one that seems apt is having all your eggs in one basket. It is clear that you need to look after this

basket, because if something goes wrong, you could end up with a difficult cleanup situation.

The same can be said of using Oracle Database 12*c* Multitenant: you are creating a container that houses multiple PDBs, and if you do not have adequate backups (both at the container and pluggable database levels), you may end up with a very difficult scenario should disaster strike.

This chapter demonstrated that performing backup and recovery in a multitenant environment is not particularly complex or onerous, and getting a handle on the basics means you can perform complete recovery with just a few easy commands. The next chapter will take this one step further to discuss point-in-time recovery and Flashback Database.



CHAPTER 8

Flashback and Point-in-time Recovery

In previous chapters you have learned that most of the backup and recovery tasks are performed at the container database (CDB) level, and this is perfect in terms of protecting all your pluggable databases (PDBs) from media failure. Simply create a new PDB in the CDB, and the PDB will be automatically protected in the same way.

However, a PDB administrator may have different requirements. Rarely in production, but frequently in test environments, we may need to do point-in-time recovery (PITR), so we appreciate a smart alternative: Flashback Database. Release 12.1 brought us multitenant, and although Flashback Database was not available in this release, in 12.2 this has changed thanks to the introduction of the local UNDO mode.

Pluggable Database Point-in-Time

Recall that we said that the ancestor of PDBs was the transportable tablespace. Here (whether you are using PDBs or not), we'll take a look at how you restore a tablespace to a previous state.

A tablespace has its own datafiles, so the first step is easy: Restore the datafiles from the previous backup. Then you have to apply REDO to bring the files forward, up to the desired point-in-time you are working to; but that's not enough. Datafiles by themselves are just a bunch of bits without the metadata that details what is stored within. This information is contained in the dictionary, so this is why you cannot do a tablespace point-in-time recovery (TSPITR) in place. Instead, you must also recover the SYSTEM and

SYSAUX tablespaces to the same point-in-time, which has to be done into an auxiliary database.

With PDBs, you don't expect to encounter the same problem, because they have their own SYSTEM and SYSAUX tablespaces that can be recovered to the same point-in-time. This means that you can bring your PDB to a specific point-in-time. But that's not enough, because you still can't open it in this state. As a refresher, look back at the section "Accessing Database Files at the CDB Level" in Chapter 1 and you will see what is missing.

When you recover a database, the final step is to roll back all the ongoing transactions that did not complete at that point-in-time. This is the "A" from the ACID property: atomicity of transactions. The principle holds that you need to apply the UNDO for the transactions that were there at that time, which is why you require a PITR of the UNDO tablespace as well.

In Figure 8-1, you can see the restore, roll-forward, and rollback phases of recovery illustrated. All files are brought up to the required point-in-time state by applying REDO to them. Uncommitted transactions are cleaned out by applying UNDO to roll them back.

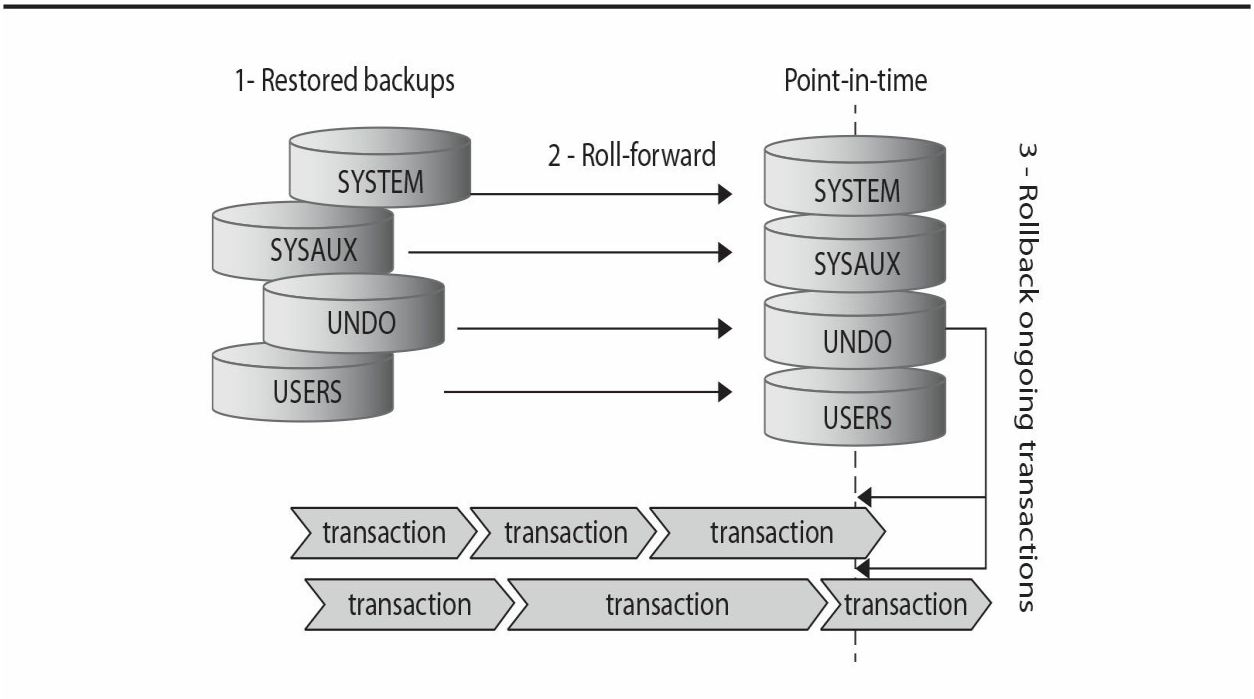


FIGURE 8-1. *The roll-forward and rollback phases of recovery over the transaction timeline*

Let’s see how it works in 12.1 Multitenant. With PDBs, you don’t need to restore *everything* into an auxiliary instance as you do for tablespace PITR. It is now possible to restore the complete PDB tablespaces in place because they contain their system tablespaces. However, you still need to restore UNDO, and for this you need an auxiliary instance. You cannot overwrite the common UNDO which is also used by the current transactions running on the other containers.

Recover PDB Until Time

Here is an example of the Recovery Manager (RMAN) commands you can run to restore and recover a PDB, specifying the point-in-time with a system change number, as of SCN 1610254, and using /var/tmp for the auxiliary instance:

```
alter pluggable database PDB close;
run {
 set until SCN = 1610254 ;
 restore pluggable database PDB;
 recover pluggable database PDB auxiliary destination='/var/tmp';
 alter pluggable database PDB open resetlogs;
}
```

In the RMAN output you can see the operations that it is performing. RMAN is verbose here, so we only show those sections of output that help you understand how it works.

First, all the PDB datafiles, those of SYSTEM, SYSAUX, and USERS tablespaces, are restored in place:

```
Starting restore at 29-JAN-16
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=136 device type=DISK
channel ORA_DISK_1: starting datafile backup set restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
channel ORA_DISK_1: restoring datafile 00143 to /u02/app/oracle/oradata/PDB/system01.dbf
channel ORA_DISK_1: restoring datafile 00144 to /u02/app/oracle/oradata/PDB/sysaux01.dbf
…
channel ORA_DISK_1: restored backup piece 1
channel ORA_DISK_1: restore complete, elapsed time: 00:00:03
channel ORA_DISK_1: starting datafile backup set restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
channel ORA_DISK_1: restoring datafile 00146 to /u02/app/oracle/oradata/PDB/usertbs01.dbf
…
channel ORA_DISK_1: restored backup piece 1
channel ORA_DISK_1: restore complete, elapsed time: 00:00:01
Finished restore at 29-JAN-16
```

Then recovery must start, and RMAN needs to determine the tablespaces that may contain UNDO:

```
Starting recover at 29-JAN-16
current log archived
using channel ORA_DISK_1
RMAN-05026: WARNING: presuming following set of tablespaces applies to specified
Point-in-time
List of tablespaces expected to have UNDO segments
Tablespace SYSTEM
Tablespace UNDOTBS1
```

Notice the warning, which we will explain later. Here RMAN is making a reasonable guess that those tablespaces that contain UNDO currently are the same as those that contained UNDO at the point-in-time you want to recover to, and RMAN lists them out.

Those tablespaces that contain UNDO cannot be restored in place, because they would override the CDB tablespaces that are used by the other containers, so for this purpose we need an auxiliary instance:

```
Creating automatic instance, with SID='ggzB'
initialization parameters used for automatic instance:
db_name=CDB
db_unique_name=ggzB_pitr_pdb_CDB
compatible=12.0.0
db_block_size=8192
db_files=200
diagnostic_dest=/u01/app/oracle
_system_trig_enabled=FALSE
sga_target=1024M
processes=150
db_create_file_dest=/var/tmp
log_archive_dest_1='location=/var/tmp'
enable_pluggable_database=true
_clone_one_pdb_recovery=true

starting up automatic instance CDB
…
Automatic instance created
```

As you can see, RMAN takes responsibility for creating an instance, using Oracle Managed Files (OMF) file naming with destination set to /var/tmp, which we defined earlier as an auxiliary destination. So let's review the parameters that are used. It is possible to define additional options, but the following are mandatory:

- `db_name` and `compatible` must be the same.
- `SID` and `db_unique_name` must be unique.
- `_clone_one_pdb_recovery` specifies that only CDB$ROOT and one PDB will be recovered.
- `_system_trig_enabled` disables the system triggers to be sure that the auxiliary instance does not have any unintended impact outside of itself.

Then this instance must use the files from the PDB that have been restored in place:

```
# switch to valid datafilecopies
switch clone datafile  143 to datafilecopy
 "/u02/app/oracle/oradata/PDB/system01.dbf";
switch clone datafile  144 to datafilecopy
 "/u02/app/oracle/oradata/PDB/sysaux01.dbf";
switch clone datafile  146 to datafilecopy
 "/u02/app/oracle/oradata/PDB/usertbs01.dbf";
```

And the CDB datafiles for SYSTEM, SYSAUX, and UNDO will be restored to the OMF destination:

```
# set destinations for recovery set and auxiliary set datafiles
set newname for clone datafile  1 to new;
set newname for clone datafile  5 to new;
set newname for clone datafile  3 to new;
# restore the tablespaces in the recovery set and the auxiliary set
restore clone datafile  1, 5, 3;
…
channel ORA_AUX_DISK_1: restoring datafile 00001 to /var/tmp/CDB/
datafile/o1_mf_system_%u_.dbf
channel ORA_AUX_DISK_1: restoring datafile 00005 to /var/tmp/CDB/
datafile/o1_mf_undotbs1_%u_.dbf
channel ORA_AUX_DISK_1: restoring datafile 00003 to /var/tmp/CDB/
datafile/o1_mf_sysaux_%u_.dbf
```

At this point, we have an auxiliary instance with access to all the necessary CDB$ROOT and PDB files, which have been restored to the specific point-in-time of interest. Next, it’s time to recover them:

```
# recover pdb
recover clone database tablespace  "SYSTEM", "UNDOTBS1", "SYSAUX"
pluggable database 'PDB'   delete archivelog;
sql clone 'alter database open read only';
```

The rollback phase of the recovery, which rolls back the transactions that were opened at the system change number (SCN) 1610254, reads the temporarily restored UNDO, but it actually updates the datafiles that were restored in place. At the end, the auxiliary instance and temporarily restored datafiles are automatically removed.

Where Is the UNDO?

In the previous example, the following warning was generated:

```
RMAN-05026: WARNING: presuming following set of tablespaces applies to
specified Point-in-time
List of tablespaces expected to have UNDO segments
Tablespace SYSTEM
Tablespace UNDOTBS1
```

RMAN needs to restore all tablespaces that may contain UNDO segments or the recovery will fail, so it lists the current tablespaces which contain UNDO. But we need the UNDO from the point-in-time we want to recover. And this may differ if we have changed the UNDO tablespace in the meantime.

In the following example, we alter the UNDO tablespace to UNDO2, dropping UNDO1, which existed as the prior default. The “recover PDB to a point-in-time” process then presents the following warning:

```
RMAN-05026: warning: presuming following set of tablespaces applies to
specified point-in-time
List of tablespaces expected to have UNDO segments
Tablespace SYSTEM
Tablespace UNDO2
```

And finally it fails, because that tablespace did not exist at the point-in-time we want to recover to:

```
RMAN-00571: ===========================================================
RMAN-00569: =============== ERROR MESSAGE STACK FOLLOWS ===============
RMAN-00571: ===========================================================
RMAN-03002: failure of recover command at 02/07/2016 16:18:58
RMAN-03015: error occurred in stored script Memory Script
RMAN-06026: some targets not found - aborting restore
RMAN-06023: no backup or copy of datafile 17 found to restore
```

Datafile 17 was from the UNDO2 tablespace, but we can't restore it, and we don't need it in any case; but we do need the datafile from UNDO1.

Here the syntax of the `RECOVER` command is useful because it allows us to specify the specific tablespaces to restore when we know which one holds UNDO:

```
recover pluggable database PDB auxiliary destination='/var/tmp'
   undo tablespace 'UNDO1';
```

We have seen cases in which RMAN does not know how to restore these datafiles, but it's probably not a good idea to rely solely on the fact that you know the name of the tablespace that was present at a previous point-in-time either.

But no need to worry, because the solution is to have the PDBPITR operation automatically restore the correct UNDO tablespace using an RMAN catalog. Then there will be no warnings or errors, and RMAN will restore the correct UNDO tablespace itself.

Summary of 12.1 PDBPITR

From what we have discussed so far, two recommendations follow: First, to be fully automated, even in those instances when the UNDO tablespace has changed, you need an RMAN catalog. If you anticipate running PITR, you should have an RMAN catalog to facilitate a large retention period. In fact, there are no irrefutable reasons not to use an RMAN catalog! Remember that you can put this anywhere, even on a virtual machine, because you don't

have to license it separately. Second, even if the auxiliary instance is supposed to be removed automatically by RMAN, if a failure occurs, you may have to clean it up manually. You can find its ORACLE_SID in the RMAN log and use this to connect and `shutdown abort` if needed.

Figure 8-2 shows the CDB that is at the current SCN, where PDB2 has been restored to a point-in-time SCN, and an auxiliary instance is able to undo the uncommitted transactions because it has restored the CDB level UNDO at that SCN.

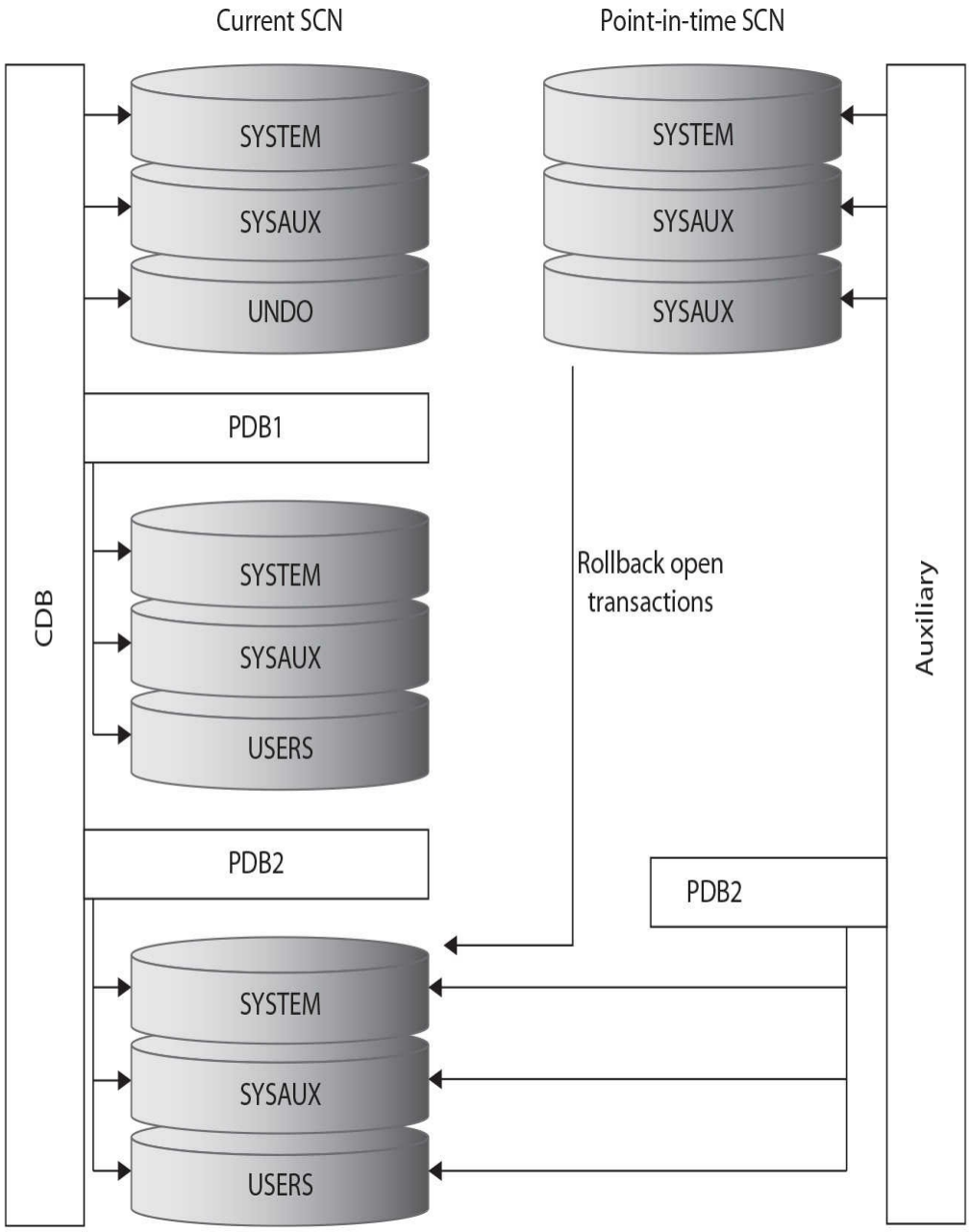


FIGURE 8-2. *Point-in-time recovery using auxiliary instance*

The creation and cleaning of the auxiliary instance is automated by RMAN, but this still takes time. You need to restore and recover all tablespaces that may contain UNDO, even though you may need only a few UNDO records—specifically those that cover the open transactions for your

PDB. Perhaps there is a better way, so let’s imagine that UNDO was stored at the PDB level.

Local UNDO in 12.2

In 12.1 the UNDO tablespace is common, meaning that it is shared among the entire CDB, to store information about all transactions. Furthermore, only the CDB administrator can create the UNDO tablespace. So in working with a PDB, the statement to create the UNDO tablespace is simply ignored.

The problem with shared UNDO is that it stores data from all other containers. As mentioned, UNDO contains information relating to transactions, which is necessary to clean up any uncommitted transactions as required. This means that PDBs on a shared UNDO CDB are essentially not self-contained, except for a closed PDB that has been shut down cleanly.

This is why it was impossible to flashback a PDB in 12.1. And in older versions, the only way to affect a PITR of part of the database was to use TSPITR, which could not be done in place. It was, in fact, necessary to restore the SYSTEM, SYSAUX, and UNDO tablespaces, in addition to the tablespace that you specifically wanted to restore, because the metadata from the system tablespace is required, along with UNDO record information, to roll back the transactions.

When 12*c* was introduced, each PDB had its own system tablespaces but UNDO was shared, which again meant that PDBs were not isolated enough for particular types of operations, such as those that shift tablespaces to another time or location.

However, beginning with 12.2, you can define the UNDO mode to local, which means that each PDB stores its own UNDO records in a local UNDO tablespace pertaining solely to that PDB. This enables many operations on an open PDB that were previously impossible, such as relocating, plugging, and flashback of a PDB. Furthermore, it improves the PITR of PDBs because the need for an auxiliary instance has been removed.

In 12.2, you can still use the shared UNDO mode if you prefer, as in 12.1, but certain operations will not be available or will require the PDB to be cleanly closed so that there are no active transactions, and no need for any UNDO records.

In short, having the UNDO local to the tablespace improves the

efficiency of unplugging and PITR of PDBs, and it is mandatory for online relocate or clone. (Note: in shared mode these operations require the source to be open read-only to ensure that there are no ongoing, open transactions.) Local UNDO is also mandatory for the referenced PDB when creating a proxy database.

Database Properties

If you have a CDB already created, you can check the UNDO mode from the database properties:

```
SQL> select * from database_properties where property_name like '%UNDO%';
PROPERTY_NAME        PROPERTY_VALUE       DESCRIPTION
-------------------- -------------------- ----------------------------------
LOCAL_UNDO_ENABLED   TRUE                 true if local undo is enabled
```

The `LOCAL_UNDO_ENABLED` property is set to TRUE when you are in local UNDO mode, and it is set to FALSE in shared UNDO mode. Be careful if you don't see anything, because this means that the database was created with shared UNDO, either when UNDO mode was not specified in the `CREATE DATABASE` statement or if it was upgraded from 12.1

Create Database

You choose the local UNDO mode when you create the database. You can check the option on the Database Configuration Assistant (DBCA), or add `LOCAL UNDO ON` to the `ENABLE PLUGGABLE DATABASE` clause of the `CREATE DATABASE` statement.

When you create the database in local UNDO mode, in addition to the UNDO tablespace of the CDB$ROOT container, you have an UNDO tablespace in PDB$SEED and in *any* PDB that you create. Here is an example, noting again that in the `report schema` the tablespace names are prefixed with the PDB name:

```
RMAN> report schema;
Report of database schema for database with db_unique_name CDB
List of Permanent Datafiles
===========================
File Size(MB) Tablespace           RB segs Datafile Name
---- -------- -------------------- ------- ------------------------
1    700      SYSTEM               YES     /u02/oradata/CDB/system01.dbf
2    210      PDB$SEED:SYSTEM      NO      /u02/oradata/CDB/pdbseed/system01.dbf
3    550      SYSAUX               NO      /u02/oradata/CDB/sysaux01.dbf
4    165      PDB$SEED:SYSAUX      NO      /u02/oradata/CDB/pdbseed/sysaux01.dbf
5    270      UNDOTBS1             YES     /u02/oradata/CDB/undotbs01.dbf
6    225      PDB$SEED:UNDOTBS1    NO      /u02/oradata/CDB/pdbseed/undotbs01.dbf
7    5        USERS                NO      /u02/oradata/CDB/users01.dbf
8    210      PDB:SYSTEM           NO      /u02/oradata/CDB/pdb/system01.dbf
9    185      PDB:SYSAUX           NO      /u02/oradata/CDB/pdb/sysaux01.dbf
10   225      PDB:UNDOTBS1         NO      /u02/oradata/CDB/pdb/undotbs01.dbf
11   5        PDB:USERS            NO      /u02/oradata/CDB/pdb/users01.dbf
List of Temporary Files
=======================
File Size(MB) Tablespace           Maxsize(MB) Tempfile Name
---- -------- -------------------- ----------- --------------------
1    20       TEMP                 32767       /u02/oradata/CDB/temp01.dbf
2    20       PDB$SEED:TEMP        32767       /u02/oradata/CDB/pdbseed/temp01.dbf
3    20       PDB:TEMP             32767       /u02/oradata/CDB/pdb/temp01.dbf
```

Changing UNDO Tablespace

The default UNDO tablespace that is created with a PDB may not suit your needs. It is generated based on PDB$SEED, or from the CDB$ROOT if PDB$SEED has no UNDO tablespace. If you want to change the UNDO tablespace, you can drop it and re-create as you want. Let’s look at an example:

```
SQL> alter session set container=PDB;
Session altered.
SQL> select tablespace_name,contents from dba_tablespaces;

TABLESPACE_NAME                CONTENTS
------------------------------ ---------------------
SYSTEM                         PERMANENT
SYSAUX                         PERMANENT
UNDOTBS1                       UNDO
TEMP                           TEMPORARY
USERS                          PERMANENT
```

We have the UNDOTBS1 UNDO tablespace that has been created with our PDB. As we are in local UNDO mode, we can't drop it:

```
SQL> drop tablespace UNDOTBS1 including contents and datafiles;
drop tablespace UNDOTBS1 including contents and datafiles
*
ERROR at line 1:
ORA-30013: undo tablespace 'UNDOTBS1' is currently in use
```

This tablespace is the one defined for our PDB:

```
SQL> show parameter undo
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
temp_undo_enabled                    boolean     FALSE
undo_management                      string      AUTO
undo_retention                       integer     900
undo_tablespace                      string      UNDOTBS1
```

So, first, we have to create a new UNDO tablespace, then switch to it, and then drop the old one when there are no transactions in it. As an example, we want to define an UNDO tablespace as a bigfile tablespace, with guaranteed retention.

```
SQL> create bigfile undo tablespace UNDOTBS2 datafile '/u02/oradata/
CDB/pdb/undotbs02.dbf' size 100M autoextend on next 100M maxsize 5G
retention guarantee;
Tablespace created.
```

Once the UNDO tablespace is created, we can then switch to it:

```
SQL> alter system set undo_tablespace='UNDOTBS2';
System altered.
```

And then we can drop the old one:

```
SQL> drop tablespace UNDOTBS1 including contents and datafiles;
Tablespace dropped.
```

Of course, you may want to wait for the UNDO retention time to elapse before dropping the old tablespace; otherwise, you may risk having some queries fail with the infamous ORA-1555 error.

Changing UNDO Mode

If you have created the database in shared UNDO mode, you can change it later, but this requires downtime on the CDB because you need to be in upgrade mode to do so.

In the following example, we have no rows in our database properties relating to UNDO mode, which means that `LOCAL_UNDO_ENABLED` is set to off.

```
SQL> select * from database_properties where property_name like '%UNDO%';
no rows selected
```

If you try to change this property you get the following error:

```
SQL> alter database local undo on;
alter database local undo on
*
ERROR at line 1:
ORA-65192: database must be in UPGRADE mode for this operation
```

However, once you have a maintenance window, you can work this through with the following commands:

```
SQL> shutdown immediate
SQL> startup upgrade
SQL> alter database local undo on;
SQL> shutdown immediate
SQL> startup
```

And we now see this change reflected in the UNDO mode property setting:

```
SQL> select * from database_properties where property_name like '%UNDO%';
PROPERTY_NAME        PROPERTY_VALUE       DESCRIPTION
-------------------- -------------------- ----------------------------------
LOCAL_UNDO_ENABLED   TRUE                 true if local undo is enabled
```

At that point, the local UNDO tablespaces will be created when you open the PDBs.

If you want to create the UNDO tablespace yourself, you have to open the PDB, then create your UNDO tablespace, and then drop the UNDO tablespace that was created at open.

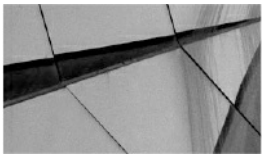
If you want to come back to shared UNDO mode, the opposite operation can be done: start upgrade, `ALTER DATABASE LOCAL UNDO OFF`, and then drop the UNDO tablespaces in PDBs because they are no longer used.

PDB$SEED

If you changed to local UNDO and you want to have the same behavior as though the database was created in local UNDO from the get-go, you have to create an UNDO tablespace in the PDB$SEED. Then new PDBs created from SEED will have it as a template.

For that, you need to open the SEED read/write:

```
SQL> alter pluggable database PDB$SEED open read write force;
SQL> alter session set container=PDB$SEED;
SQL> create undo tablespace LOCALUNDO datafile size 100M autoextend on next 100M;
SQL> alter pluggable database PDB$SEED close;
SQL> alter pluggable database PDB$SEED open read only;
```



NOTE

No error results when you open the PDB$SEED read/write because customization of PDB$SEED is allowed when you are in local UNDO for the goal of UNDO tablespace creation. If you try to open the seed in read/write mode when you are in shared UNDO, you get an error (ORA-65017: seed pluggable database may not be dropped or altered). Only sessions with `"_oracle_script"=true` can open the PDB$SEED when in shared UNDO mode.

When using OMF, the UNDO tablespace of PDB$SEED is automatically created, based on the CDB$ROOT tablespace attributes.

Shared or Local UNDO?

We see no reason to use shared UNDO mode in 12.2, and our recommendation is to set `LOCAL UNDO ON`. You want multitenant for PDB isolation and easy operations, so you probably want local UNDO.

Note that even in local UNDO, some UNDO records can be generated in the CDB$ROOT if they are done by internal transactions that switched

temporarily to CDB$ROOT.

PDB Point-in-Time Recovery in 12.2

We have explained the complex operations that have to be done when recovering a PDB at a different time than the CDB. The reason was to get the UNDO records necessary to clean the transactions that were not completed at the point-in-time we restore to. Now let's see what is different in 12.2.

PDBPITR in Shared UNDO Mode

If you are in shared UNDO mode, the `LOCAL UNDO OFF` state, you are in exactly the same situation you were in with 12.1. The UNDO for all transactions is stored in the CDB$ROOT tablespace and shared by all PDBs. As a consequence, we cannot recover it to another point-in-time there; we need an auxiliary instance for this, along with a place for it.

When a fast recovery area (FRA) is defined, you can run `RECOVER PLUGGABLE DATABASE` without the `AUXILIARY DESTINATION` and the auxiliary instance will be created in the FRA. While it runs, you can see those files listed as `AUXILIARY DATAFILE COPY` in `V$RECOVERY_AREA_USAGE`:

```
FILE_TYPE               PCT_SPACE_USED PCT_SPACE_RECLAIMABLE NUMBER_OF_FILES
----------------------- -------------- --------------------- ---------------
CONTROL FILE                       .3                     0               1
REDO LOG                          2.5                     0               3
ARCHIVED LOG                     3.37                     0              12
BACKUP PIECE                    36.81                     0               3
IMAGE COPY                          0                     0               0
FLASHBACK LOG                    5.21                     0               7
FOREIGN ARCHIVED LOG                0                     0               0
AUXILIARY DATAFILE COPY         17.42                     0               3
```

If no FRA is defined and you try the same operation, you will get the following error:

```
RMAN-00571: ===========================================================
RMAN-00569: =============== ERROR MESSAGE STACK FOLLOWS ===============
RMAN-00571: ===========================================================
RMAN-03002: failure of recover command at 02/08/2016 21:44:59
RMAN-05107: AUXILIARY DESTINATION option is not specified
```

Without an FRA, you need to specify a location as per the following:

```
recover pluggable database PDB
until restore point PIT auxiliary destination='/var/tmp';
```

So you have a choice: put it in FRA or choose a destination.

PDBPITR in Local UNDO Mode

With the introduction of local UNDO mode in 12.2, you have all required UNDO in the local UNDO tablespace that is restored in place, so there is no need for an auxiliary instance, and the PDB PITR is simpler and faster.

You run the same command, `RECOVER PLUGGABLE DATABASE`, without specifying an auxiliary location since the auxiliary instance is no longer needed. The best practice is to put the `restore` and `recover` commands in a `RUN` block with a `SET UNTIL`:

```
run {
 set until restore point 'T0' ;
 restore pluggable database PDB;
 recover pluggable database PDB;
 alter pluggable database PDB open resetlogs;
}
```

The point-in-time can be specified with a restore point as above, a timestamp, an SCN, or a log sequence and thread number. This is no different from the database PITR you know from versions prior to multitenant. A restore point here is used only to associate a convenient name to an SCN and

can be created at CDB or PDB level, but we will discuss that later when we describe other functionalities of restore points.

Flashback PDB

When you want to revert a database to a recent point-in-time, you can use a smart alternative that does not require restoring any datafiles. Flashback PDB was one of the most important features missing in 12.1, but it is now possible in 12.2, enabled by the addition of local UNDO. Be aware that flashback requires additional logging. Conventional recovery starts from a previous state of the datafiles and uses the REDO stream to roll them forward. In contrast, Flashback Database begins from the current state of the datafiles and applies flashback logs to bring them back to a previous state.

Flashback Logging

By default, a database does not generate the flashback logs, which means that you cannot run Flashback Database operations. There are two ways to generate a flashback log: set `FLASHBACK ON` and guaranteed restore point.

FLASHBACK ON

You can set `FLASHBACK ON` even when the database is opened. It is performed at the CDB level:

```
SQL> alter database flashback on;
Database altered.
```

From this time on, the database will store flashback logs in the FRA:

```
SQL> select file_type,percent_space_used,number_of_files
     from v$recovery_area_usage;

FILE_TYPE                 PERCENT_SPACE_USED NUMBER_OF_FILES
------------------------- ------------------ ---------------
CONTROL FILE                               0               0
REDO LOG                                   0               0
ARCHIVED LOG                             .91               1
BACKUP PIECE                           90.35               6
IMAGE COPY                                 0               0
FLASHBACK LOG                           2.08               2
FOREIGN ARCHIVED LOG                       0               0
AUXILIARY DATAFILE COPY                    0               0
```

Flashback logging has a very small overhead by itself, but there are some side effects on sessions that format new blocks, such as direct-path inserts or any UNDO generation. Usually there is no need to read those blocks before writing them, because they are new. However, with `FLASHBACK ON`, these sessions have to read the blocks in order to write the previous image to the flashback logs. So with `FLASHBACK ON`, you may see more reads from disks (`physical reads for flashback new`) in session statistics.

You don't want to keep the flashback logs forever, because Flashback Database is best used to go to a recent point-in-time only. Going weeks or months prior is more efficient with PITR. So when you set the database in `FLASHBACK ON` mode you also define a flashback log retention period:

```
SQL> show parameter flashback
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
db_flashback_retention_target        integer     1440
```

This value is in minutes and the default is 1440, which translates to 24 hours. Note that this is a target only, so in the case of space pressure, the database will give priority to generating the ARCHIVELOG rather than guaranteeing the flashback retention.

Even if the flashback logs contain all the information to bring datafiles back to a past image, it still requires REDO to bring them to a consistent

point-in-time, so enabling `FLASHBACK ON` can be done only on an ARCHIVELOG mode database.

Guaranteed Restore Point

With `FLASHBACK ON`, you can flashback to any point-in-time in the past that fits within the retention period. That point-in-time is defined by timestamp, an SCN, or a restore point. Even when you are not in flashback mode, you can enable the possibility to flashback to a restore point when you declare that restore point with `guarantee flashback database`:

```
SQL> create restore point BEFORE_RELEASE guarantee flashback database;
Restore point created.
```

Besides the `V$RESTORE_POINT` view, the easiest way to list these restore points is from RMAN:

```
RMAN> list restore point all;
using target database control file instead of recovery catalog
SCN              RSP Time  Type       Time      Name
---------------- --------- ---------- --------- --------------
4702377                              31-JAN-16 END_OF_MONTH
4770509                    GUARANTEED 13-FEB-16 BEFORE_RELEASE
```

A guaranteed restore point ensures that you can flashback to that specific point. The flashback logs and archived logs required for that point are kept in the FRA, but only a minimal set specifically for that point-in-time alone.

For example, here we've deleted all backups and will try to delete all archived logs:

```
RMAN> delete archivelog all;
using target database control file instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=18 device type=DISK
RMAN-08139: warning: archived redo log not deleted, needed for guaranteed
restore point
archived log file name=/u02/fast_recovery_area/CDB/CDB/archivelog/2016_02_13/
o1_mf_1_133_cczbo99h_.arc thread=1 sequence=133
```

And you can see that the ARCHIVELOG sequence that covers a guaranteed restore point is protected, because it is needed to make the datafiles consistent in case of flashback to that point. We also keep the flashback logs that contain the image of all blocks that have changed since that restore point:

```
SQL> select file_type,percent_space_used,number_of_files from
v$recovery_area_usage;

FILE_TYPE               PERCENT_SPACE_USED NUMBER_OF_FILES
----------------------- ------------------ ---------------
CONTROL FILE                            .6               1
REDO LOG                                 5               3
ARCHIVED LOG                            .8               1
BACKUP PIECE                             0               0
IMAGE COPY                               0               0
FLASHBACK LOG                        12.08               8
FOREIGN ARCHIVED LOG                     0               0
```

This means that you don’t want to keep old guaranteed restore points for too long, because you can’t get rid of the flashback log generated since your oldest restore point.

If you want to keep a snapshot of the database for a long time and go back to it frequently, our recommendation is to drop and re-create the snapshot once you have flashed back to it. It’s logically the same snapshot, and earlier flashback logs can be reclaimed.

In multitenant, flashback logging is done at the CDB level, but starting from 12.2 you can flashback individual PDBs as an alternative to PITR.

Flashback with Local UNDO

When your CDB runs with `LOCAL UNDO ON`, you can flashback a PDB alone without any side effects on the other PDBs, in the same way that you can do a PITR in place. Of course, this is possible if you have `FLASHBACK ON` and the point-in-time is within the flashback log retention target, but it is also possible with `FLASHBACK OFF` if you have a guaranteed restore point. This is achievable because

- flashback logs have all previous images of blocks that have changed
- UNDO tablespace is flashed back to the same point-in-time and then can be used to clean the transactions that were ongoing at that point-in-time

Flashback in Shared UNDO

When your CDB runs with `LOCAL UNDO OFF`, the UNDO cannot be flashed back in the UNDO tablespace because it's shared at CDB level. You are in exactly the same place as with PITR and the solution is the same: an auxiliary instance. You can add the `AUXILIARY DESTINATION` to the `FLASHBACK PLUGGABLE DATABASE` clause, or it will be created, implicitly, in the FRA.

Restore Points at the CDB and PDB Levels

You can create restore points at the CDB level, and then use them to flashback PDBs. In addition, it is also possible to create a restore point at the PDB level.

Here is an example in which we create a restore point at the CDB level:

```
SQL> show con_name
CON_NAME
------------------------------
CDB$ROOT

SQL> create restore point END_OF_MONTH;
Restore point created.
```

And here's one at the PDB level, with guaranteed Flashback Database:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> create restore point BEFORE_RELEASE guarantee flashback database;
Restore point created.
```

While still in the same PDB, PDB1, we can see and use both these restore points:

```
SQL> select scn, name, con_id, pdb_restore_point, guarantee_flashback_
database,  clean_pdb_restore_point from v$restore_point;
       SCN NAME                CON_ID PDB GUARANTEE CLEAN_PDB
---------- --------------- ---------- --- --------- ---------
   1514402 BEFORE_RELEASE           4 YES YES       NO
   1514393 END_OF_MONTH             0 NO  NO        NO
```

But when we use the same query from another PDB, we can't see the restore points defined under PDB1. We can see only the restore points defined under the root container:

```
SQL> select scn, name, con_id, pdb_restore_point, guarantee_flashback_
database, clean_pdb_restore_point from v$restore_point;
       SCN NAME                CON_ID PDB GUARANTEE CLEAN_PDB
---------- --------------- ---------- --- --------- ---------
   1514393 END_OF_MONTH             0 NO  NO        NO
```

NOTE

In multitenant, `CON_ID=0` displays information that is at the CDB level and not related to any specific container, neither CDB$ROOT nor any PDB.

We can also create restore points with the same name in different containers. In the following we are still in PDB2 (`CON_ID=5`):

```
SQL> show con_id
CON_NAME
------------------------------
5
SQL> create restore point BEFORE_RELEASE;
Restore point created.
SQL> create restore point END_OF_MONTH;
Restore point created.
SQL> select scn, name, con_id, pdb_restore_point, guarantee_flashback_
database, clean_pdb_restore_point from v$restore_point;
       SCN NAME                CON_ID PDB GUARANTEE CLEAN_PDB
---------- --------------- ---------- --- --------- ---------
   1514393 END_OF_MONTH             0 NO  NO        NO
   1514410 BEFORE_RELEASE           5 YES NO        NO
   1514415 END_OF_MONTH             5 YES NO        NO
```

Those restore points have the same name. However, the CON_ID and PDB_RESTORE_POINT columns let you know which one is at the PDB level.

Here is the result when running the same query from CDB$ROOT:

```
       SCN NAME                 CON_ID PDB GUARANTEE CLEAN_PDB
---------- --------------- ---------- --- --------- ---------
   1514402 BEFORE_RELEASE            4 YES YES       NO
   1514393 END_OF_MONTH              0 NO  NO        NO
   1514410 BEFORE_RELEASE            5 YES NO        NO
   1514415 END_OF_MONTH              5 YES NO        NO
```

For those of you who would like to see this information listed in RMAN, unfortunately, at the time of writing, this functionality is missing, but we have filed an enhancement request for it:

```
RMAN> list restore point all;
using target database control file instead of recovery catalog
SCN              RSP Time  Type       Time      Name
---------------- --------- ---------- --------- ----
1514393                              14-FEB-16 END_OF_MONTH
1514402                    GUARANTEED 14-FEB-16 BEFORE_RELEASE
1514410                              14-FEB-16 BEFORE_RELEASE
1514415                              14-FEB-16 END_OF_MONTH
```

Another piece of information lacking in RMAN is the `CLEAN_PDB_RESTORE_POINT`. It is another addition in 12.2, and we will get to this shortly.

PDB Level and Flashback Logging

Related to this topic, you should be aware of an important point. Using our example, with `FLASHBACK OFF` and a guaranteed restore point only at the PDB level for PDB1 (`CON_ID=4`), it seems that flashback logging is actually enabled for all CDB datafiles.

From the CDB$ROOT, we check the flashback mode:

```
SQL> select con_id,flashback_on from v$database;
    CON_ID FLASHBACK_ON
---------- ------------------
         0 RESTORE POINT ONLY
```

As expected, this is what we have when flashback logging is off, but we have a guaranteed restore point.

So let's connect to PDB2 and check that we see no guaranteed restore points from there:

```
SQL> connect demo/demo@//localhost/PDB2
Connected.
SQL> select scn, name, con_id, pdb_restore_point, guarantee_flashback_
database, clean_pdb_restore_point from v$restore_point;
       SCN NAME                CON_ID PDB GUARANTEE CLEAN_PDB
---------- -------------- ---------- --- --------- ---------
   1514393 END_OF_MONTH             0 NO  NO        NO
   1514410 BEFORE_RELEASE           5 YES NO        NO
   1514415 END_OF_MONTH             5 YES NO        NO
```

Then insert 1000 rows into an existing table:

```
SQL> insert into DEMO select lpad('x',4000,'x') from dual connect by level<=1000;
1000 rows created.
```

Now we check the session's statistics related to flashback:

```
SQL> select name,value from v$mystat join v$statname using(statistic#)
where name like '%flashback%' and value>0;
NAME                                               VALUE
---------------------------------------- ----------
physical reads for flashback new                 1008
```

This output reveals that these blocks have to be written into the flashback

logs—proof that flashback logging occurs for changes in all containers, as long as one container has a guaranteed restore point.

In the current version, 12.2, you can enable flashback logging only at the CDB level, but the error message when trying it at the PDB level gives the impression that flashback logging will be a possibility at the PDB level in the future:

```
SQL> alter session set container=PDB;
Session altered.
SQL> alter database flashback on;
alter database flashback on
*
ERROR at line 1:
ORA-03001: unimplemented feature
```

Clean Restore Point

We have seen that in a shared undo CDB, the flashback PDB needs an auxiliary instance to restore the undo tablespace to clean ongoing transactions. This makes the flashback less efficient, in both time and required space. The preferable solution is to run in local UNDO mode.

However, even when in shared UNDO mode, you don’t need to restore the UNDO when you know that you have no ongoing transactions at all. If you want to create a restore point in production before applying a patch or an application release, or in a test before a run of regression tests, you can close the database. And when it’s closed cleanly, there are no outgoing transactions. In this scenario, we can create a clean restore point that can be used to flashback efficiently, even in shared UNDO mode.

The following example affects a database with shared UNDO, created without a `LOCAL UNDO` clause:

```
SQL> select * from database_properties where property_name like '%UNDO%';
no rows selected
```

As you have seen before, no property for LOCAL_UNDO_ENABLED means that it is false. We connect to PDB1, which is currently opened, and try to create a clean restore point:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> show pdbs
    CON_ID CON_NAME                       OPEN MODE  RESTRICTED
---------- ------------------------------ ---------- ----------
         4 PDB1                           READ WRITE NO
SQL> create clean restore point REGTEST1;
create clean restore point REGTEST1
*
ERROR at line 1:
ORA-65025: Pluggable database is not closed on all instances.
```

But to be sure it's a clean restore point, we need to close the PDB:

```
SQL> alter pluggable database PDB1 close;
Pluggable database altered.
SQL> create clean restore point REGTEST1 guarantee flashback database;
Restore point created.
```

Note that we have created a guaranteed restore point here because we do not have FLASHBACK ON, and we want to be able to flashback to it.

At this point, we can open back the PDB. Any blocks written to the datafiles will have their previous image written to flashback logs, so you can quickly flashback to the initial state, without the need of an auxiliary instance thanks to the clean restore point:

```
SQL> alter pluggable database PDB1 close;
Pluggable database altered.
SQL> flashback pluggable database PDB1 to restore point REGTEST1;
Flashback complete.
SQL> alter pluggable database PDB1 open resetlogs;
Pluggable database altered.
```

The concept of a clean restore point applies to shared UNDO mode only. It makes no sense in local UNDO mode, and if you try to use the same syntax, you will get the following error:

```
ORA-39891: Clean PDB restore point cannot be created when CDB is in
local undo mode.
```

Resetlogs

This chapter is all about bringing a PDB to a point-in-time in the past. When you do that on a CDB or a non-CDB, you have to `OPEN RESETLOGS`. This operation resets the REDO stream because it is interrupted: the REDO that was generated before the recovery or the flashback cannot be used on that new incarnation of the database. When you open a non-CDB or a CDB with resetlogs, the online redo logs are re-created and the old ones are discarded.

The REDO stream is at the CDB level, so the “resetlogs” term may be misleading when dealing with a PDB. The redo logs are not re-created, and in this case, they just continue to log all changes for all modifications that occur in the instance. But it’s the same idea: mark the REDO stream so that the REDO from that point on is known to protect a new incarnation of the PDB.

In the following example, we flashback PDB1 and try to open it without the resetlogs clause:

```
SQL> flashback pluggable database PDB1 to restore point REGTEST1;
Flashback complete.
SQL> alter pluggable database PDB1 open;
alter pluggable database PDB1 open
*
ERROR at line 1:
ORA-01113: file 11 needs media recovery
ORA-01110: data file 11:
'/u02/oradata/CDB/2BBE9EBBB1B57588E053754EA8C075FE/datafile/o1_mf_sysaux_cd1cr7y
g_.dbf'
```

The message is not explicit here, because we have two possibilities with the current state of the datafiles: we can choose to revert our flashback and apply recovery to bring it up to the latest state, or we can open it in that state, because this is what we wanted to do with the flashback operation:

```
SQL> alter pluggable database PDB1 open resetlogs;
Pluggable database altered.
```

Each PDB's open resetlogs operation creates a new incarnation of the PDB, which is a subset of the CDB incarnation. You can list the history of all incarnations by querying the V$PDB_INCARNATION view. Here is an example, where we have flashed back the PDB to the same restore point several times:

```
SQL> select db_incarnation#, pdb_incarnation#, status,incarnation_time from
v$pdb_incarnation;

DB_INCARNATION# PDB_INCARNATION# STATUS  INCARNATION_TIME BEGIN_RE END_RESE
--------------- ---------------- ------- ---------------- -------- --------
…
              1               22 ORPHAN                   20:21:17 20:21:17
              1               23 ORPHAN                   20:21:21 20:21:21
              1               24 ORPHAN                   20:21:24 20:21:24
              1               25 ORPHAN                   20:21:27 20:21:28
              1               26 ORPHAN                   20:21:31 20:21:31
              1               27 ORPHAN                   20:21:34 20:21:35
              1               28 ORPHAN                   20:21:38 20:21:38
              1               29 ORPHAN                   20:21:42 20:21:42
              1               30 ORPHAN                   20:21:45 20:21:45
              1               31 ORPHAN                   20:21:49 20:21:49
              1               32 ORPHAN                   20:21:52 20:21:52
```

This is a massive advantage of the multitenant architecture: you don’t need to restart the instance when you flashback a PDB. This makes the flashback operation very fast—in fact, just a few seconds. It can be used for test databases where, for example, a number of tests must run on the same data. This is much faster than reimporting from a dump file, and it’s even faster than reattaching a transportable tablespace.

Flashback and PITR

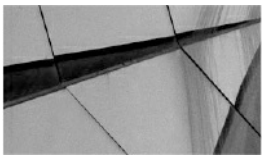
This chapter covers two ways to bring a PDB back to a previous state. They have some common effects, but are used for different situations.

When Do You Need PITR or Flashback?

Taking a production database back to a point in the past is rare, because you will lose all transactions that have happened since that point. If you encounter a logical or physical corruption, Flashback Database is considered only on those occasions where the corruption is database-wide and occurred in the last few seconds. But in other cases, you can benefit a lot from restore points

and flashback. For example, before any maintenance operation, such as an application release or database upgrade, you can take a guaranteed restore point, and if anything goes wrong, your fallback scenario takes only few minutes to enact. And because you flashback before reopening the service, you don't lose any transactions. Just don't forget to drop the restore point when maintenance is completed and validated.

The benefit of flashback shows up every day in test environments, especially in continuous integration environments. If you have several runs of tests that need the same data set, without flashback, rerunning all DDL and DML simply takes too long, and a dump import is also probably not quick enough. A solution is potentially found in the transportable tablespace functionality, but even that takes too long to import all metadata. Flashback is the best solution to revert back to a previous state, but before multitenant this required an instance restart. In multitenant, the flashback PDB takes only a few seconds, so it can be run hundreds of times during a batch of nonregression tests.



TIP

A PDB administrator can flashback a PDB as long as the admin has the SYSDBA privilege granted on that PDB. There's no need for a common user for that. So it's possible to give that right to a trusted application DBA so that he or she can interact with this PDB only.

Impact on the Standby Database

We will detail the Data Guard configuration in multitenant in Chapter 11, but you already know that the REDO is at CDB level, and that what you do in a primary database, especially when changing the structure or in case of `OPEN RESETLOGS`, can have consequences on the physical standby.

Changing UNDO Mode

When changing from shared UNDO to local UNDO, you have to startup upgrade. If you are using real-time query (Active Data Guard), you need to

stop it while the primary is in startup upgrade: the REDO from upgrade mode cannot be applied when a physical standby is open read-only. The second point is that new UNDO tablespaces will be created, so be sure that `StandbyFileManagement = 'AUTO'`.

Flashback or PITR on Primary

After a PDBPITR or a flashback PDB, you have to open the PDB with `RESETLOGS`. The managed recovery process (MRP) on the physical standby stops when it encounters the `RESETLOGS` marker because the datafiles are at the same state as the primary before the flashback, and current REDO cannot be applied on the previous incarnation. Here is what you can see in the alert.log:

```
2016-02-19T22:31:38.142324+01:00
(4):Recovery of pluggable database PDB1 aborted due to pluggable database open
resetlog marker.
(4):To continue recovery, restore all data files for this PDB to checkpoint SCN lower
than 4345934, or timestamp before , and restart recovery
MRP0: Background Media Recovery terminated with error 39874
2016-02-19T22:31:38.143019+01:00
Errors in file /u01/app/oracle/diag/rdbms/nzprod/USPROD/trace/USPROD_pr00_8201.trc:
ORA-39874: Pluggable Database PDB1 recovery halted
ORA-39873: Restore all data files to a checkpoint SCN lower than 4345934.
Managed Standby Recovery not using Real Time Apply
Recovery interrupted!
Recovery stopped due to failure in applying recovery marker (opcode 17.46).
Datafiles are recovered to a consistent state at change 4347096 but controlfile could
be ahead of datafiles.
```

The message is clear: You must do a PITR on the physical standby as well, to the same point-in-time. If the physical standby is in `FLASHBACK ON` mode, it's easy. Stop `APPLY`, flashback to the SCN given in the alert.log, and restart `APPLY`:

```
DGMGRL> edit database nzprod set state=apply-off;
SQL> flashback pluggable database PDB1 to scn 4345934;
DGMGRL> edit database nzprod set state=apply-on;
```

An alternative if you are not in flashback mode is to recover from service, which is a 12*c* feature:

```
DGMGRL> edit database nzprod set state=apply-off;
RMAN> restore pluggable database PDB1 from service 'usprod_dgmgrl';
DGMGRL> edit database nzprod set state=apply-on;
```

Disable Recovery on Standby

In multitenant, you don't want the standby to stop the `APPLY`, because one PDB has been flashed back. So instead of waiting for the message shown previously, you should disable recovery for that PDB before the `OPEN RESETLOGS` on the primary. In the standby database, you suspend APPLY just for the time it takes to disable recovery. Here are the commands from the Data Guard command line interface (DGMGRL) and SQL*Plus:

```
DGMGRL> edit database nzprod set state=apply-off;
SQL> alter pluggable database pdb1 close;
SQL> alter session set container=PDB1;
SQL> alter pluggable database PDB1 disable recovery;
DGMGRL> edit database nzprod set state=apply-on;
```

Then recovery occurs for the CDB except for this PDB. You can flashback or point-in-time restore the PDB1 on the primary and open it with resetlogs. Then re-enable its synchronization on standby:

```
DGMGRL> edit database nzprod set state=apply-off;
RMAN> recover database;
SQL> alter session set container=PDB1;
SQL> alter pluggable database PDB1 enable recovery;
SQL> alter pluggable database PDB1 open;
DGMGRL> edit database nzprod set state=apply-on;
```

From there, the PDB1 is synchronized again with the primary.

Auxiliary Instance Cleanup

When an auxiliary instance is created automatically, it is supposed to be cleaned up at the end of the operation. But our recommendation, after lots of testing, is to check that nothing is left over, especially when something has failed in the process. Check `AUXILIARY DATAFILE COPY` in `V$RECOVERY_AREA_USAGE` that no unreclaimable file is left over. It's also a good idea to have a look at `FLASHBACK LOG` to be sure that you removed the unnecessary guaranteed restore points.

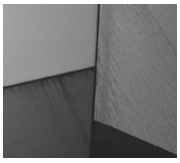
You may see other traces from the auxiliary instance that remain, such as in the DIAG directory. Here is ours after a few PITRs in shared UNDO:

```
adrci> show alert
Choose the home from which to view the alert log:
1: diag/rdbms/uced_pitr_pdb_cdb/uCEd
2: diag/rdbms/cobc_pitr_pdb_cdb/CoBc
3: diag/rdbms/kqwu_pitr_pdb_cdb/kqwu
4: diag/rdbms/dwkf_pitr_pdb_cdb/dwkf
5: diag/rdbms/zcdx_pitr_pdb_cdb/zCDx
6: diag/rdbms/ccfq_pitr_pdb_cdb/ccFq
7: diag/rdbms/azfb_pitr_pdb_cdb/azFB
…
```

With this in mind, if you have automated PITR in shared UNDO, you should adapt your housekeeping scripts to clean up appropriately.

Summary

The PITR and flashback features that were missing in 12.1 have been implemented in 12.2, thanks to the introduction of local UNDO mode. On production databases, it provides a safety net for your application releases or maintenance operations, giving you an instantaneous fallback plan. But it’s in development environments that the feature will bring more agility. How often do you have to refresh environments, restore the previous state of a test database, or revert a change made by a test that touched more data than required? PITR and flashback, plus the moving and cloning features we will cover in the next chapter, truly make multitenant the agile environment for modern development.



CHAPTER 9

Moving Data

There are probably very few (likely no) Oracle administrators charged with the care of a solitary database who don't need to share data among other databases. Equally implausible is an administrator who doesn't need to upgrade or move a database to a different platform at some point. In most situations, at one point or another, the administrator is required to move data at the physical level—that is, the admin will be required to move the datafiles or entire databases from one place to another.

Multitenant architecture brings new scenarios and challenges into play that must be addressed by the DBA, because it is the DBA's job to manage the database at this level. This includes new features such as the ability to move complete pluggable databases (PDBs), along with extensions to existing features, and it presents new opportunities regarding how to approach some core DBA tasks.

The multitenancy architecture encompasses multiple PDBs. The first feature we will explore is disassociating a PDB from the container database (CDB)—that is, moving the physical files, and then making the PDB part of another CDB. This *unplugging/plugging in* ability with a PDB basically involves a database move.

We also want to be able to copy a database—and even better, we want to let Oracle perform the file copy for us. Database *cloning* extends past a simple local machine copy, because in Oracle Database 12*c* we can clone from a remote database or harness storage features to make "cheap" copies. One facet of this feature is that the source can be an Oracle Database 12*c* non-CDB, thus enabling a simple way of converting a non-CDB into a PDB.

Oracle Database 12*c* also introduces many new data movement features

that are not PDB-specific, and one of the most interesting is the ability to move databases between previously incompatible platforms.

And last, but not least, we can move PDBs to and from the cloud.

Grappling with PDB File Locations

Moving all the PDB files involves, of course, altering the file locations. The easiest approach is to work with Oracle Managed Files (OMF) and let Oracle generate the file paths and names, using the CDB name and PDB GUID—for example, /u01/app/oracle/data/SRC/29E63B5BE26B3ABEE053050011ACA3F3/datafi In other words, you can specify the `DB_CREATE_FILE_DEST` initialization parameter and let Oracle take care of this for you.

However, if you want to manage the locations yourself, you can use the `FILE_NAME_CONVERT` parameter to map the files to new locations in the `create pluggable database` statements.

If the datafiles for the target database are already in place, you can specify `NOCOPY`, so that Oracle uses the files already present. And if the new location is different than the source location, you can use the `SOURCE_FILE_NAME_CONVERT` clause to map the files to the new location.

The examples in this chapter show how these various options are used.

Plugging In and Unplugging

As you know, a PDB is an independent subset of a CDB. As such, it enjoys a kind of independence, which you have already learned about in the chapters on management and security. Therefore, it makes sense to be able to move data from a single PDB, or indeed the whole PDB, from one database to another.

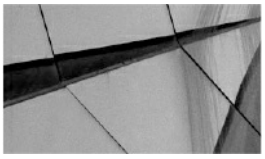
Moving an entire non-CDB (or the CDB as whole) has always been possible with Oracle. You simply move the datafiles to a different place; set up the configuration files for networking; set up the password file, init.ora, and so on; and then start the database from the new location. The Recovery Manager (RMAN) `duplicate` command further simplifies this process, but we won’t go into detail on this here. However, you should review Chapter 11, where we explain how to create a standby database, because the two

processes are similar.

With multitenancy, moving databases has become easier, and the expectation is that these moves will now occur more frequently as a result. First of all, you don't need to move the entire CDB, which means that much of the tedious configuration work disappears. There is no need to handle relocating the password file, init.ora, /etc/oratab, for example, and this makes the move simpler, and thus a potentially viable option in many more scenarios.

Second, PDBs contain significantly less Oracle internal data dictionary information. So, as discussed in Chapter 4, you can create a new CDB with a new patch version, and then move the PDBs to this new CDB. From there, you can patch the PDB, which is actually much faster, because there is less dictionary data to update.

Furthermore, multitenancy is often thought of as being synonymous with consolidation, which means having many databases in a CDB. This can inevitably lead to situations in which the number of PDBs outgrows the server capacity. The easy solution now is to move some of the PDBs someplace else.

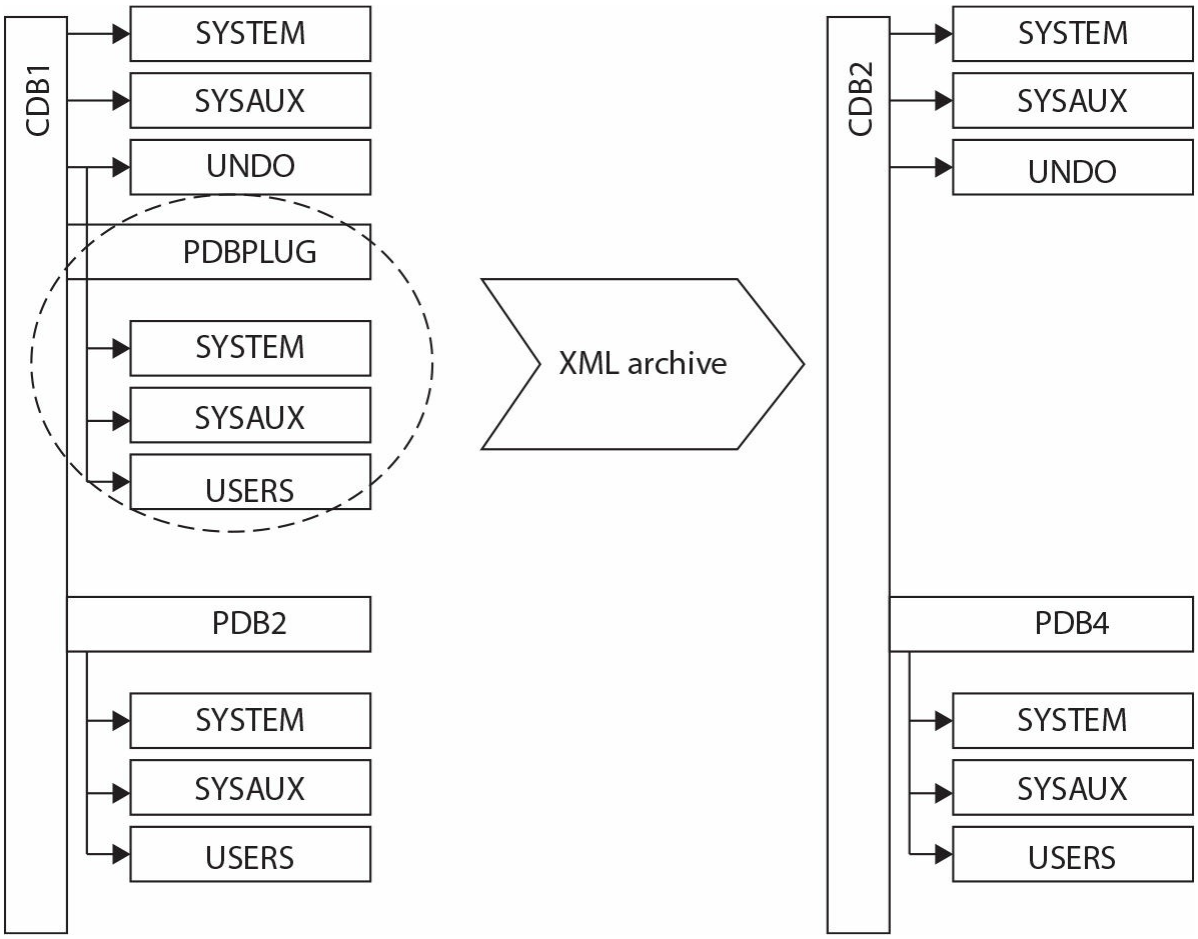


NOTE

Unlike other cloning options, the plug-in operation does not need a source database up and running; only its datafiles and an XML parameter file must be available. This opens up new scenarios for provisioning, such as making the files available on various media or for download.

Unplug and Plug In a PDB

Let's start with the simplest scenario—unplugging a PDB and then plugging it into a different CDB, as depicted in the following diagram. Note that the PDB must be closed during this operation.




```
SQL> alter pluggable database PDBPLUG close;

Pluggable database altered.

SQL> alter pluggable database PDBPLUG unplug into '/home/oracle/
pdbplug.xml';

Pluggable database altered.
```

After this command has completed, the unplugged database (PDB) consists of its datafiles, which are still in place as they were, along with a new XML file that describes the database (version, DBID), tablespaces, datafiles, installed options, and database parameters set at the PDB level.

Now we copy the files to the target database location. A simple `scp` to the target server will do:

```
$ mkdir /u01/app/oracle/data/CDBTRG/PDBPLUG
$ scp oracle@source:/u01/app/oracle/data/CDBSRC/PDBPLUG/* /u01/app/
oracle/data/CDBTRG/PDBPLUG
```

The third step is to plug in the database. Note that the files will probably be in a different location than they were on the source, as recorded in the XML parameter file, so remapping may be required.

```
SQL> create pluggable database PDBPLUG using '/home/oracle/pdbplug.xml'
source_file_name_convert=('/u01/app/oracle/data/CDBSRC/PDBPLUG','/u01/
app/oracle/data/CDBTRG/PDBPLUG') nocopy tempfile reuse;

Pluggable database created.

SQL> alter pluggable database PDBPLUG open;

Pluggable database altered.
```

An Unplugged Database Stays in the Source

Even after we unplug a PDB and produce the related XML file, the PDB continues to be part of the source database, as shown in v$pdbs and dba_pdbs:

```
SQL> select pdb_id, pdb_name, status from dba_pdbs order by 1;

    PDB_ID PDB_NAME                       STATUS
---------- ------------------------------ ---------
         2 PDB$SEED                       NORMAL
         3 PDBPLUG                        UNPLUGGED
```

This also means that the PDB is still part of any RMAN backups, as you can see in the following:

```
RMAN> backup database;
…
channel ORA_DISK_1: starting full datafile backup set
channel ORA_DISK_1: specifying datafile(s) in backup set
input datafile file number=00013 name=/u01/app/oracle/data/CDBSRC/PDBPLUG/sysaux01.dbf
input datafile file number=00012 name=/u01/app/oracle/data/CDBSRC/PDBPLUG/system01.dbf
channel ORA_DISK_1: starting piece 1 at 09-JAN-16
channel ORA_DISK_1: finished piece 1 at 09-JAN-16
piece handle=/u01/app/oracle/data/FRA/CDBSRC/28F00F4199F504BFE053030011A
C9CBD/backupset/2016_01_09/o1_mf_nnndf_TAG20160109T235506_c937nd7c_.bkp
tag=TAG20160109T235506 comment=NONE
channel ORA_DISK_1: backup set complete, elapsed time: 00:01:05
…
```

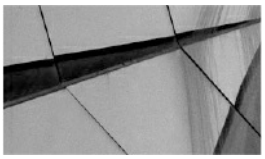
This ensures that the PDB's data is still protected, even before we plug it into the target database and begin backing it up there. It also means that all the original datafiles are still in place, and we have ample time to copy them to the desired location for the plug-in. However, there is little else that can be done with this unplugged PDB for now:

```
SQL> alter pluggable database PDBPLUG open;
alter pluggable database PDBPLUG open
*
ERROR at line 1:
ORA-65086: cannot open/close the pluggable database
```

Once the PDB is plugged in and backed up in its new location, we can drop the PDB from the original source CDB to clean up:

```
SQL> drop pluggable database PDBPLUG;

Pluggable database dropped.
```



NOTE

The default setting is `KEEP DATAFILES`*. This prevents us from accidentally dropping the data files before we copy or move them to the target location.*

What Exactly Is in the XML File?

It's worth reviewing the XML file that describes the unplugged PDB. In doing so, we can see exactly what information is retained when we move the PDBs from one CDB to another, and it also shows what Oracle can actually check when considering whether the PDB is compatible to be plugged in (as we will discuss in the next section).

The following are some parameters for a PDB named PDBXML:

```
<pdbname>PDBXML</pdbname>
<cid>4</cid>
<byteorder>1</byteorder>
<vsn>203424000</vsn>
<vsns>
  <vsnnum>12.2.0.1.0</vsnnum>
  <cdbcompt>12.2.0.0.0</cdbcompt>
  <pdbcompt>12.2.0.0.0</pdbcompt>
  <vsnlibnum>0.0.0.0.24</vsnlibnum>
  <vsnsql>24</vsnsql>
  <vsnbsv>8.0.0.0.0</vsnbsv>
</vsns>
<dbid>2150329683</dbid>
<ncdb2pdb>0</ncdb2pdb>
<cdbid>1348575968</cdbid>
<guid>2A8090EFCD800637E053030011ACB487</guid>
```

In this section we see various version and compatibility-level listings.

The `vsn` refers to the version (203424000 is 0xC200100 in hex, which is an internal representation for 12.2.0.1.0—convert the number digit-by-digit to decimal, 0xC is 12). This ensures that the versions match or, if required, whether an upgrade is needed upon plug-in. Note also the byte order; 1 is little endian.

We also see the different PDB IDs: `cid` (CON_ID), DBID, `cdbid` (CON_UID), and `guid`. This is also not a non-CDB (hence `ncdb2pdb` is 0), but later in this chapter we will discuss the details for converting such a database.

For each tablespace, we see the following section:

```
<tablespace>
  <name>SYSTEM</name>
  <type>0</type>
  <tsn>0</tsn>
  <status>1</status>
  <issft>0</issft>
  <isnft>0</isnft>
  <encts>0</encts>
  <bmunitsize>8</bmunitsize>
  <file>
    <path>/u01/app/oracle/data/CDBSRC/PDBXML/system01.dbf</path>
    <afn>12</afn>
    <rfn>1</rfn>
    <createscnbas>3081073</createscnbas>
    <createscnwrp>0</createscnwrp>
    <status>1</status>
    <fileblocks>34560</fileblocks>
    <blocksize>8192</blocksize>
    <vsn>203423744</vsn>
    <fdbid>2150329683</fdbid>
    <fcpsb>3083026</fcpsb>
    <fcpsw>0</fcpsw>
    <frlsb>2716779</frlsb>
    <frlsw>0</frlsw>
    <frlt>902125218</frlt>
    <autoext>1</autoext>
    <maxsize>4194302</maxsize>
    <incsize>1280</incsize>
  </file>
</tablespace>
```

This covers name, type (temporary tablespace has 1), encryption, list of files—file name, absolute file number, relative file number, create SCN, size in blocks, block size, version, DBID, checkpoint SCN, and autoextend settings.

```
<csid>873</csid>
<ncsid>2000</ncsid>
<options>
  <option>APS=12.2.0.1.0</option>
      <option>CATALOG=12.2.0.1.0</option>
…
      <option>XML=12.2.0.1.0</option>
      <option>XOQ=12.2.0.1.0</option>
    </options>
```

The `csid` is the database character set (873 is AL32UTF8), and `ncsid` is the national character set (2000 is AL16UTF16; we can verify this with SQL function `NLS_CHARSET_ID`). The remainder lists all the installed options and their various versions:

```
<dv>0</dv>
<APEX>5.0.3.00.02:1</APEX>
<parameters>
  <parameter>processes=300</parameter>
  <parameter>memory_target=0</parameter>
  <parameter>db_block_size=8192</parameter>
  <parameter>db_2k_cache_size=16777216</parameter>
  <parameter>compatible='12.2.0'</parameter>
  <parameter>open_cursors=300</parameter>
  <parameter>pga_aggregate_target=209715200</parameter>
  <parameter>enable_pluggable_database=TRUE</parameter>
</parameters>
<sqlpatches/>
<tzvers>
  <tzver>primary version:25</tzver>
  <tzver>secondary version:0</tzver>
</tzvers>
<walletkey>0</walletkey>
<services/>
<opatches/>
```

This penultimate section specifies the following:

- Whether Data Vault is enabled
- The APEX version
- Parameters set at the PDB level
- Installed SQL patches
- Versions of time zone file
- Wallet
- Defined services
- Patches installed by OPatch

```
<awr>
  <loadprofile>CPU used by this session=55.038961</loadprofile>
  <loadprofile>DB time=62.853896</loadprofile>
  <loadprofile>db block changes=1944.274084</loadprofile>
  <loadprofile>execute count=1517.727127</loadprofile>
  <loadprofile>logons cumulative=0.036344</loadprofile>
…
  <loadprofile>user rollbacks=0.024011</loadprofile>
</awr>
<hardvsnchk>0</hardvsnchk>
<localundo>0</localundo>
```

Finally, the XML file includes AWR load profile information, and whether local undo has been enabled.

Check Compatibility for Plug-In

In the examples so far we have taken a few shortcuts, ignoring the fact that Oracle imposes some limitations on PDBs with respect to installed options, character sets, and versions. You can either read in detail the limitations or, perhaps more effectively, ask Oracle to perform these checks, using the supplied DBMS_PDB package.

Get the XML Describing the Database

First, we need to get the XML file that describes the unplugged database. If we have unplugged the PDB, then we already have it on hand and can skip to

the next step.

However, we can ask Oracle to generate the XML file even if we haven't yet actually unplugged the PDB. Although this requires an extra step, it is definitely better to have this information ahead of time, knowing the results of the compatibility check well before we perform the unplug. This will give us ample time to rectify the issues or come up with an alternate solution—such as creating a new target CDB—and all this can occur well before we incur any downtime on the database.

```
-- NB: unlike unplugging a PDB, in this case the database must be
opened (or read only)
SQL> alter pluggable database PDBPLUG open;

Pluggable database altered.
SQL> BEGIN
  DBMS_PDB.DESCRIBE(
    pdb_descr_file => '/home/oracle/pdbplug.xml',
    pdb_name       => 'PDBPLUG');
END;
/

PL/SQL procedure successfully completed.
```

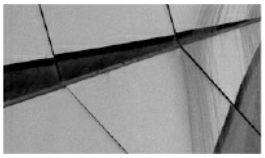
Run the Compatibility Check

Running the compatibility check is simple and is achieved by executing one PL/SQL function in the target database root or application root:

```
SET SERVEROUTPUT ON
DECLARE
  compatible CONSTANT VARCHAR2(3):=
    CASE DBMS_PDB.CHECK_PLUG_COMPATIBILITY(
           pdb_descr_file => '/home/oracle/pdbplug.xml',
           pdb_name       => 'PLUGDB')
    WHEN TRUE THEN 'YES'
    ELSE 'NO'
END;
BEGIN
  DBMS_OUTPUT.PUT_LINE(compatible);
END;
/
YES

PL/SQL procedure successfully completed.
```

Of course, if the answer is NO, we want more detail, and this is readily available in the PDB_PLUG_IN_VIOLATIONS view. The view can contain data even if the answer is YES, so it is a good idea to review this regardless of the answer.



NOTE

If you are plugging in databases as part of an upgrade, the compatibility check will return NO, because the versions are different. This is to be expected, and you can still plug in that database, although you won't be able to open it until you have first finished the upgrade. This is covered in more detail in Chapter 4

Read the Requirements

The list of compatibility requirements is changing with each version, so it is advisable always to review the list for the particular version in use. In short, however, there are four main requirements:

- **Platform** The databases need to be of the same endianness.
- **Options** The databases must have the same options installed (partitioning, data mining, and so on).
- **Versions** The source PDB must be the same version as the target CDB before the PDB can be opened. However, it is possible to plug in a PDB of an older version and upgrade it, and then it can be opened.
- **Character sets** The character sets must match. Version 12.2 eases this condition: if the CDB uses AL32UTF8, the PDB can use any character set.

Plug In a PDB as Clone

To this point we have been thinking of the plug-in/unplug process as a move operation, with a single PDB at the start and end, even if the PDB at the end runs somewhere else. Sometimes, however, we need to create a copy of the PDB. Although the other options described in the upcoming "Cloning" section are usually used for this, it is possible to use pluggable functionality to create a copy, too. The only principal difference is the number of copies that will reside in the environment, and, as with non-CDB databases, every copy needs to be uniquely identifiable; otherwise, we end up with collisions when `DBID`, `GUID`, or `CON_UID` is used. Note that the RMAN catalog mandates that `DBID` be unique, and a single database won't allow multiple identical PDBs, either. Fortunately, the solution is easier than the description of the problem itself: just add the `AS CLONE` clause when you're performing the plug-in operation.

Let's start with a simple PDB, created at the source database:

```
SQL> alter session set container = PDBCLONE;

Session altered.
SQL> select pdb_name, dbid, con_uid, guid from dba_pdbs;
PDB_NAME         DBID    CON_UID GUID
---------- ---------- ---------- --------------------------------
PDBCLONE   1307782881 1307782881 28F3AE16F69E151AE053030011ACF9A3
```

Now let’s unplug it:

```
SQL> alter pluggable database close;

Pluggable database altered.

SQL> alter session set container=cdb$root;

Session altered.

SQL> alter pluggable database PDBCLONE unplug into '/home/oracle/
pdbclone.xml';

Pluggable database altered.
```

Now plug it into the target database. This will be a move operation, so it will keep its existing IDs.

```
SQL> create pluggable database PDBCLONE using '/home/oracle/pdbclone.
xml' FILE_NAME_CONVERT=('/u01/app/oracle/data/CDBSRC/PDBCLONE', '/u01/app/
oracle/data/CDBTRG/PDBCLONE');
```

The PDB has been created:

```
SQL> select pdb_name, dbid, con_uid, guid from dba_pdbs where PDB_NAME like 'CLONE%';
PDB_NAME         DBID    CON_UID GUID
---------- ---------- ---------- --------------------------------
PDBCLONE   1307782881  304242047 28F3AE16F69E151AE053030011ACF9A3
```

You can see that Oracle changed the `CON_UID`, but the `DBID` and `GUID` values are still the same.

As mentioned, Oracle won’t allow the same database to be plugged in again:

```
SQL> create pluggable database PDBCLONE2 using '/home/oracle/pdbclone.xml' FILE_
NAME_CONVERT=('/u01/app/oracle/data/CDBSRC/PDBCLONE', '/u01/app/oracle/data/CDBTRG/
CLONE2PDB')
*
ERROR at line 1:
ORA-65122: Pluggable database GUID conflicts with the GUID of an existing container.
```

The `AS CLONE` prompts Oracle to generate new identifiers, thus allowing us to have multiple copies of the same source PDB in the CDB:

```
SQL> create pluggable database PDBCLONE2 as clone using '/home/oracle/pdbclone.xml'
file_name_convert=('/u01/app/oracle/data/CDBSRC/PDBCLONE', '/u01/app/oracle/data/
CDBTRG/CLONE2PDB');

Pluggable database created.
SQL> select pdb_name, dbid, con_uid, guid from dba_pdbs where pdb_name like 'CLONE%';

PDB_NAME    DBID       CON_UID    GUID
---------- ---------- ---------- --------------------------------
PDBCLONE   1307782881  304242047 28F3AE16F69E151AE053030011ACF9A3
PDBCLONE2  2901289706 2901289706 28F3F521825118DAE053030011AC0A65
```

PDB Archive File

Starting with Oracle Database version 12.2, the plug-in/unplug operations can also work with PDB archive files. A PDB archive is a compressed file containing the XML parameter file and its data files, as well as any other necessary auxiliary files (wallet). This is really just a convenience option, because it’s easier to copy one compressed file than multiple files.

The basic syntax is identical to that used to unplug to XML file, with the only difference being the specified extension of the target file:

```
SQL> alter pluggable database PDBCLONE unplug into '/home/oracle/pdbclone.pdb';

Pluggable database altered.
```

And if we look at the output file more closely, we can see that it's just a plain ZIP file:

```
$ unzip -t pdbclone.pdb
Archive:  pdbclone.pdb
    testing: system01.dbf             OK
    testing: sysaux01.dbf             OK
    testing: soe.dbf                  OK
    testing: users.dbf                OK
    testing: /home/oracle/pdbclone.xml   OK
No errors detected in compressed data of pdbclone.pdb.
```

Note that the XML file inside the archive still refers to the same paths for the PDB on the source, so it's the same result that we would get from unplugging into XML; however, these paths are ignored. Instead, Oracle unpacks all the datafiles to the directory where the PDB archive is located. This is only a temporary location, Oracle then moves the files to the proper place. `NOCOPY` is not a valid option here.

```
create pluggable database PDBPLUG using '/home/oracle/pdbclone.pdb' nocopy;

ORA-65314: cannot use NOCOPY when plugging in a PDB using an archive file

create pluggable database PDBPLUG2 as clone using '/home/oracle/pdbclone.pdb'
file_name_convert=('/home/oracle','/u01/app/oracle/data/TRG/PDBPLUG2');

Pluggable database created.
```

Cloning

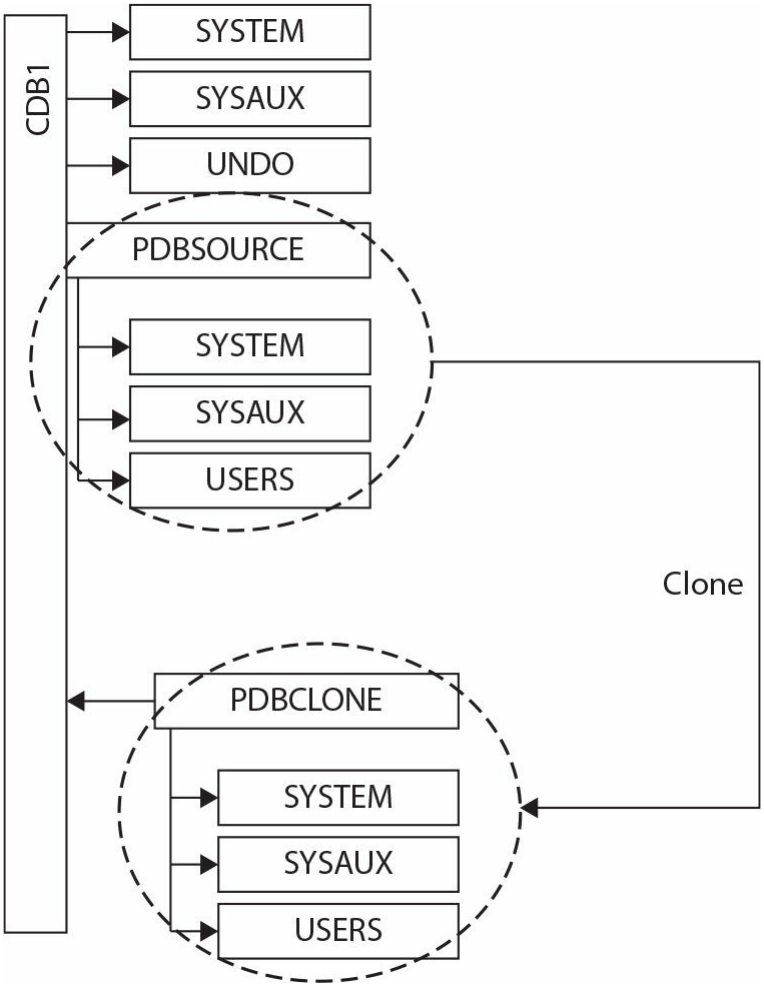
In contrast to unplugging operations, cloning does not go through the intermediate step of working with a staged set of files (datafiles and XML file or a PDB archive). Instead it, more directly, reads the files of a running CDB and copies them to a new PDB. This copy can either be local—inside a single CDB—or remote—to a different database, copying the datafiles over a

database link.

A clone operation always assigns a new GUID and UID, and there is no concept of moving, as in plug-in/unplug. The relocate feature is no exception to this, assigning new IDs in the process.

Cloning a Local PDB

The easiest use case is a clone inside a single CDB—a *local clone*, in which no other CDB is involved. The following diagram illustrates this operation:



With Active Data Guard, the standby database will also do the same copy. Therefore, no manual intervention on the standby side is needed; this is unlike in virtually all other cases described in this chapter.

Although a local clone is the simplest use case, there are still multiple options to consider. Let’s start with the most basic one:

```
create pluggable database PDBCLONE from PDBSOURCE;

Pluggable database created.
```

In Oracle Database 12.1, the source PDB must be open read-only (there are even bugs when Oracle doesn't check the open mode; clone in read/write mode then fails with various internal errors). From Oracle Database 12.2 on, the source PDB can be open read/write, provided that the CDB is in ARCHIVELOG mode and has local UNDO enabled. Either way, we must be logged in as a common user in the CDB root or in the application root.

After the clone finishes, we must open the database in read/write mode, because Oracle won't allow any other operation on it until the database has been opened at least once:

```
SQL> alter pluggable database PDBCLONE open;

Pluggable database altered.
```

Snapshot Copy

Given that a clone is a one-to-one copy of the source PDB, you might wonder whether the underlying file system can help in performing such copies more efficiently. After all, snapshot and copy-on-write functionality are, nowadays, widely used and proven storage features.

The answer is a "limited yes." If the database is on a supported file system, or Oracle Direct NFS is in use, a clone can be created as follows:

```
create pluggable database PDBCLONE from PDBSOURCE snapshot copy;

Pluggable database created.
```

This is one of the features that we expect will evolve rapidly, changing with every patch set, so it's advisable that you regularly review the latest

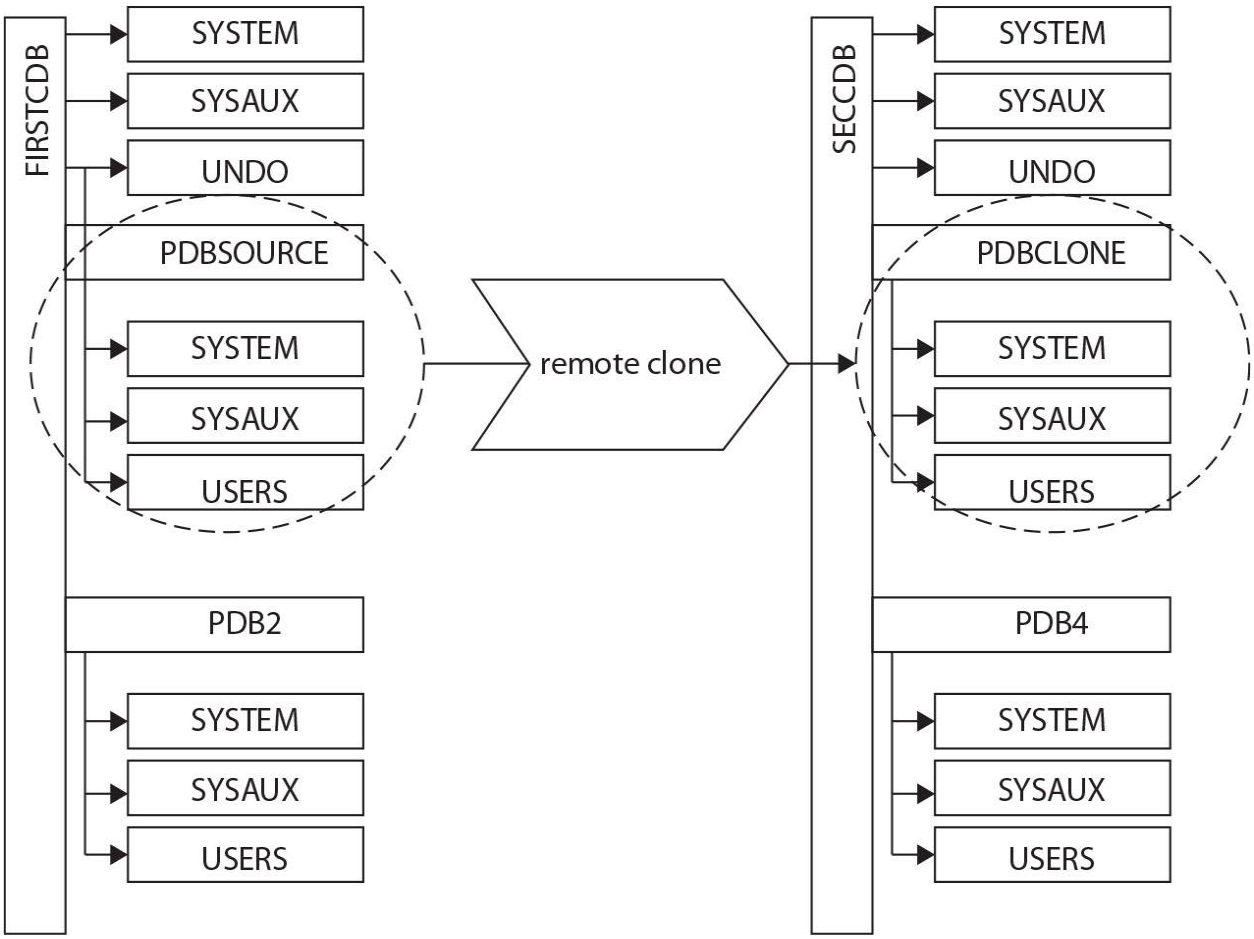
status in the Oracle documentation and in My Oracle Support note 1597027.1.

Generally, Oracle favors and promotes ASM Clustered File System (ACFS) and Direct NFS. In the former, the file system provides read/write snapshots, which means that a source PDB can be open and written to. By contrast, the Direct NFS approach mandates a read-only source PDB, but on the other hand it supports many more file systems, essentially needing only sparse file support for the target files. In both cases, the source PDB cannot be unplugged or dropped.

The specification as to which snapshot technology to use is regulated by the `CLONEDB` initialization parameter; TRUE implies use of Direct NFS clones. Also note that Direct NFS might require setting up credentials in the keystore; refer to the documentation for more information and specific syntax.

Cloning a Remote PDB

The remote clone process extends the local clone concept, with the differentiator being that the source and target database are distinct, and the communication between the two happens over a database link, as shown in the following diagram:



You must be logged in as a common user, either in the CDB root or in an application root. In addition, a database link must be available, because this link will be used to transfer all the metadata and datafiles. This link should be able to connect either to the target CDB or to the PDB.

With plug-in/unplug operations, the database must be compatible, as described previously in this chapter. In a local clone, these conditions are trivial to fulfill, but in a remote clone we must check them. We can use the same steps you used with the DBMS_PDB package, as outlined in the section "Run the Compatibility Check," earlier in the chapter.

As with a local clone, if we want to keep the source PDB open read/write, we need to use version 12.2+, with ARCHIVELOG mode and local UNDO enabled. In case of a remote clone, it's sufficient if just one of the sides involved in the clone has local UNDO and ARCHIVELOG mode.

```
SQL> create pluggable database PDBCLONE from PDBSOURCE@firstcdb;

Pluggable database created.
```

Splitting the PDB

In all the clone operations, we can specify the `USER_TABLESPACES` clause. This is a list of all the tablespaces that we want actually cloned; Oracle will add the necessary system tablespaces, but all other tablespaces will be omitted.

In respect to this, two use cases come to mind: First, we may want to provide our developers a subset of the production database, omitting some archival data and keeping the overall size of the clones smaller. Or, second, we may want to split a large, monolith PDB into more manageable, smaller PDBs. A good example of this would be if the source PDB was created by importing a non-CDB database that consolidated multiple applications, each in its own schemas and tablespaces.

```
create pluggable database PDBCLONE from PDBSOURCE user_tablespaces =
('APPTBS1','APPTBS2');

Pluggable database created.
```

Nodata Clone

A *nodata clone* is a special type of clone. It does not contain any user data, and all the tables listed in user tablespaces are imported empty, thus effectively creating a metadata-only copy. This being the case, its value is for specific use cases only.

This function is possible only from 12.1.0.2. Furthermore, it cannot be used in conjunction with index-organized tables, table clusters, or Advanced Queuing tables, additionally constraining its usefulness.

```
create pluggable database PDBCLONE from PDBSOURCE no data;

Pluggable database created.
```

With this function, the new database has the same tablespaces as the source; however, the datafiles are not copied, but instead are created empty. The size of the datafiles matches those on the source, and in the case of datafiles set to autoextend, the new files are created with the initial size.

Refreshable Copy

A *hot clone*, or a clone from a read/write source PDB, works by recovering all the changes that occur during the copy itself, by use of the ARCHIVELOGs and UNDO data.

Extending on this idea, Oracle can also use this data to recover the clone repeatedly, regularly bringing it up-to-date with the source PDB. This type of clone in Oracle Database 12.2 is a *refreshable copy*, and it can be refreshed automatically (`EVERY` *nn* `MINUTES`) or manually on demand.

Note, however, that the PDB must be closed for the refresh to occur, both in on-demand and automatic modes. The PDB cannot ever be open read/write, as any local change would prevent Oracle from applying undo from the source. Thus this is the only case where a newly cloned PDB is not first opened read/write in order to be usable.

```
SQL> create pluggable database PDBREFR from PDBSOURCE@firstcdb refresh mode manual;
Pluggable database created.

SQL> alter pluggable database PDBREFR open read only;

Pluggable database altered.
```

As an example, let’s request a manual refresh. Again, note that this must be done from within the PDB and, as mentioned, the PDB must be closed:

```
SQL> alter session set container=PDBREFR;

Session altered.

SQL> alter pluggable database PDBREFR refresh;
alter pluggable database PDBREFR refresh
*
ERROR at line 1:
ORA-65025: Pluggable database PDBREFR is not closed on all instances.

SQL> alter pluggable database close;

Pluggable database altered.

SQL> alter pluggable database PDBREFR refresh;

Pluggable database altered.
```

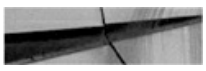
A manual refresh is possible for PDBs configured for both manual and automatic refresh.

A PDB configured for refresh has the status `REFRESHING` in the CDB_PDBS view.

Relocate

A common use case in database administration is the move of a database, not just a plain copy. To this end, version 12.2 introduces the relocate feature, which makes this explicit. Working in conjunction with the hot-cloning method introduced in the same version, this allows much shorter downtime, instead of doing a clone and then dropping the source.

When we issue the `RELOCATE` statement, the source database is still open, so it needs to be open read/write and requires local UNDO and ARCHIVELOGs, as expected. The first stage of this process is to clone the PDB while the users are still connected to the source:



```
SQL> create pluggable database PDBRELO from PDBSOURCE@sourcecdb
relocate;

Pluggable database created.
```

After this, the old PDB remains open and read/write, and it can still be accessed and used. At this point, the new PDB is mounted and has the status of `RELOCATING`:

```
SQL> select pdb_id, pdb_name, status from cdb_pdbs where pdb_name = 'PDBSOURCE';
-- source
PDB_ID     PDB_NAME   STATUS
---------- ---------- ----------
         3 PDBSOURCE  NORMAL
-- target
SQL> select pdb_id, pdb_name, status from cdb_pdbs where pdb_name = 'PDBRELO';

PDB_ID     PDB_NAME   STATUS
---------- ---------- ----------
         6 PDBRELO    RELOCATING
```

It’s the opening of the new, relocated PDB that finishes the relocation and drops the old database.

```
SQL> alter pluggable database PDBRELO open;

Pluggable database altered.
```

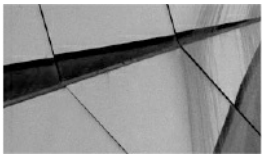
Note that this feature requires more privileges on the source side than all the other options—namely, that the user that the database link connects to needs the sysoper and create pluggable database privileges.

```
SQL> create user c##reluser identified by oracle;

User created.

SQL> grant sysoper, create pluggable database, create session to c##reluser
container=all;

Grant succeeded.
```



NOTE

We can also keep the existing PDB around (in RELOCATED state), to redirect the connections, if the listeners can't do it for us. This is enabled by specifying AVAILABILITY MAX in the CREATE PLUGGABLE DATABASE … RELOCATE command, and in this case, it's up to us to drop the source PDB, when it's no longer needed.

Proxy PDB

Sometimes, with PDBs moving around, it can be difficult for you to keep track of all the various PDBs and updating clients to connect to the correct location for each. Perhaps even more critically, features such as container map and queries using the `CONTAINERS()` clause (both described in more detail in Chapter 12) require all the participating PDBs to be in the same container database.

To address this, Oracle Database 12.2 brings to the table the concept of a *proxy PDB*. These PDBs act as a façade, redirecting all requests to the referenced underlying PDB location via an internal database link. So any user commands executed while connected to the proxy PDB affect the remote PDB, not the proxy itself, with these exceptions: `ALTER PLUGGABLE DATABASE` and `ALTER DATABASE` statements.

A proxy database has the IS_VIEW_PDB column set to YES in the CDB_PDBS view, so this is a way of identifying one:

```
SQL> create pluggable database PDBPX as proxy from PDBSOURCE@cdbtrg;

Pluggable database created.
SQL> alter pluggable database PDBPX open;

Pluggable database altered.
SQL> select pdb_id, pdb_name, IS_VIEW_PDB from cdb_pdbs where pdb_name='PDBPX';

    PDB_ID PDB_NAME   IS_VIEW_PDB
---------- ---------- -----------
         4 PDBPX      YES
```

Note that a proxy PDB actually clones the SYSTEM, SYSAUX, and temporary tablespaces of its referent, so it's not a completely empty shell.

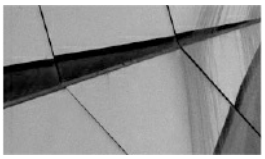
A proxy requires a database link to be defined, and this is one that connects to the target CDB root or the PDB. Although a proxy can point back to the same source database, a database link is always required. This database link is used only during the setup, as the proxy PDB actually creates a new internal database link for passing the user requests. However, this internal database link is not a simple copy of the link that was manually created; instead, Oracle generates this using the host name of the target CDB and port 1521. Note that it is possible to specify the PORT and HOST if these defaults do not match your environment.

```
SQL> create pluggable database PDBPX as proxy from PDBSOURCE@cdbtrg
PORT=1522 HOST='trghost';

Pluggable database created.
```

Application Container Considerations

An application container, introduced in Oracle Database 12.2, does not significantly extend cloning functionality. At this stage, only individual PDBs —not the whole application—can be unplugged/plugged in and cloned, and an application root can be unplugged only when empty.



NOTE

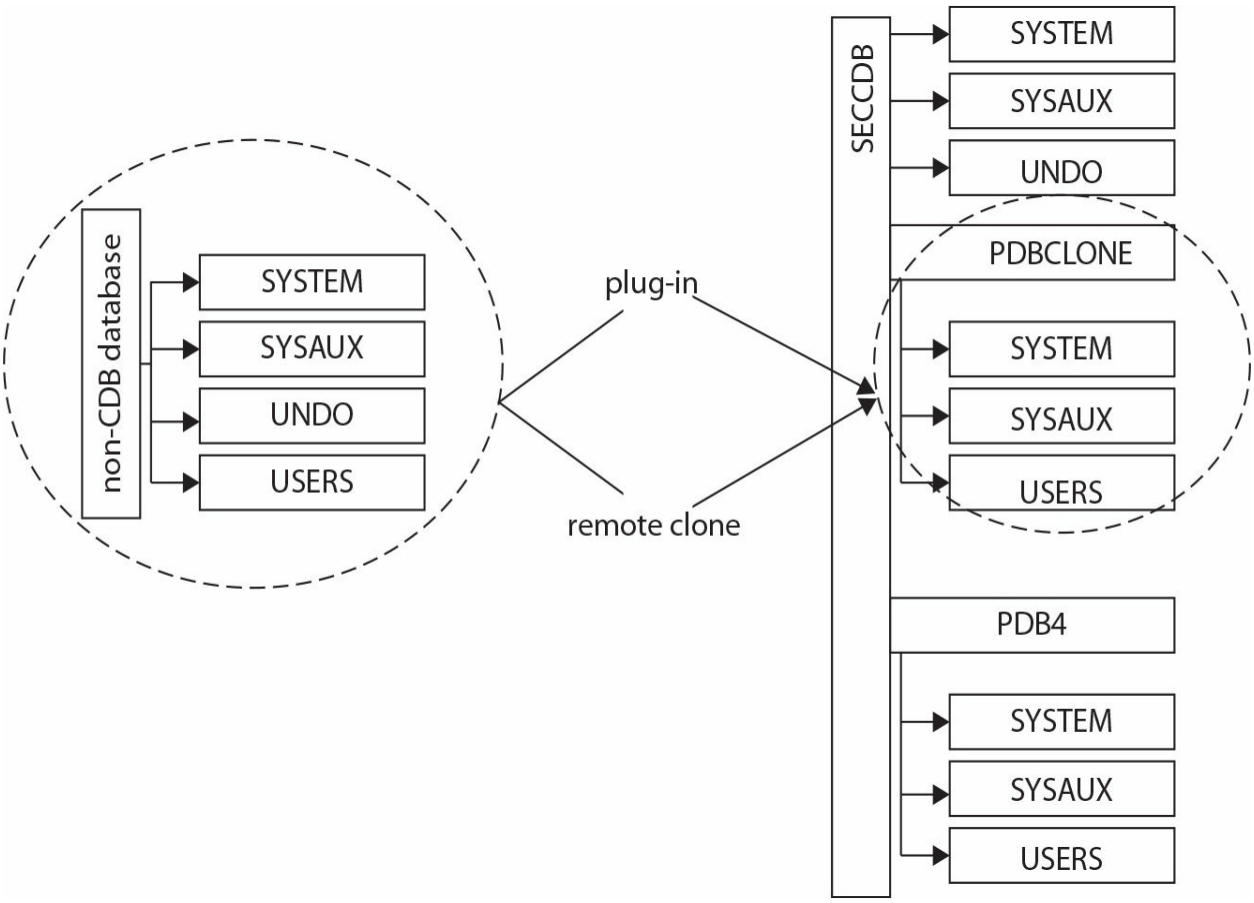
The location of a new PDB, created with clone or plug-in, is determined by the current container used when executing the `create pluggable database` *command, and this can be run in the CDB root or in an application root.*

Converting Non-CDB Database

As a special case, Oracle allows a database that does not use the multitenant architecture to be plugged in. Such an operation is more of a migration path from the legacy architecture to the new container-based one, not merely a move of data.

For this feature to work, both the source non-CDB and the target CDB versions must be 12.1.0.2 or later. The endian must also match, as the datafiles cannot be converted this way and, as usual, the installed options must also cohere.

There are multiple ways to approach this procedure using PDB operations. The non-CDB can be plugged in, or it can be cloned. Both options are shown in the following illustration.



Plug In a Non-CDB

For a non-CDB plug-in, we need to run the compatibility check, and the source database must be open or open read-only. Also, as usual, we should back it up before we attempt the plug-in, in case anything goes wrong.

We create the XML file describing an unplugged database:

```
BEGIN
  DBMS_PDB.DESCRIBE(
    pdb_descr_file => '/home/oracle/noncdb.xml');
END;
/

PL/SQL procedure successfully completed.
```

Next, we can check for any issues or violations, running a simple script in the target CDB:

```
SET SERVEROUTPUT ON
DECLARE
  compatible CONSTANT VARCHAR2(3):=
    CASE DBMS_PDB.CHECK_PLUG_COMPATIBILITY(
           pdb_descr_file => '/home/oracle/noncdb.xml',
           pdb_name       => 'NONCDBPDB')
    WHEN TRUE THEN 'YES'
    ELSE 'NO'
END;
BEGIN
  DBMS_OUTPUT.PUT_LINE(compatible);
END;
/
YES

select message, action from PDB_PLUG_IN_VIOLATIONS where name='NONCDBPDB';
MESSAGE                                          ACTION
------------------------------------------------ ---------------------------------
PDB plugged in is a non-CDB, requires            Run noncdb_to_pdb.sql.
noncdb_to_pdb.sql be run.
CDB parameter memory_target mismatch:            Please check the parameter in the
Previous 764M Current 0                          current CDB
```

In this output, the first message is obvious and expected, because we are going to plug in a non-CDB. However, the second message is an example of a parameter mismatch. Some of the parameters can be set at the CDB level only, so upon plugging in the PDB, the value of the CDB will override it. It is therefore up to us to determine whether we set the CDB parameter to match the original non-CDB one or not. This is the same issue we face when putting multiple different PDBs into the same CDB, as many parameters are global across all PDBs.

When we are satisfied with these settings, we are ready to perform the actual plug-in operation.

The XML parameter file must be created from a read-only non-CDB in order for the SCNs to be consistent. Thus, if it was generated from a

read/write database, we'd set that non-CDB to read-only and create the XML file again. Following on from there, we can shut down the source database, because it won't be needed anymore. Note, however, that after the plug-in completes, we could continue to use this non-CDB source database, because unlike an unplug of a PDB, it is not marked for drop.

Then we can plug in the database:

```
SQL> create pluggable database NONCDBPDB using '/home/oracle/noncdb.xml'
file_name_convert=('/u01/app/oracle/data/NONCDB','/u01/app/oracle/data/
CDBSRC/NONCDBPDB');

Pluggable database created.
```

Alternatively, we can plug it in, keeping the files in place. In that case, we should be aware of the TEMP file trap that may be encountered in many scenarios: unless we specify `TEMPFILE REUSE`, Oracle will try to create new tempfile(s) and fail when it finds the file already exists:

```
SQL> create pluggable database NONCDBPDB using '/home/oracle/noncdb.xml' nocopy;
create pluggable database NONCDBPDB using '/home/oracle/noncdb.xml' nocopy
*
ERROR at line 1:
ORA-27038: created file already exists
ORA-01119: error in creating database file '/u01/app/oracle/data/NONCDB/temp01.dbf'

SQL> create pluggable database NONCDBPDB using '/home/oracle/noncdb.xml' nocopy
tempfile reuse;

Pluggable database created.
```

As indicated in our earlier violation check, we need to run the conversion script, noncdb_to_pdb.sql, at this point. This step is mandatory. Although we might be able to open the database without it, we would face various issues later, such as having a corrupted dictionary, and similar. In version 12.1, it's not even possible to rerun the script if it fails, because that also corrupts the dictionary. For more on this, refer to My Oracle Support note 2039530.1.

In our tests, the longest running section of this script was in fact utl_recomp, which compiles all invalid objects.

```
SQL> alter session set container=NONCDBPDB;

Session altered.

SQL> @$ORACLE_HOME/rdbms/admin/noncdb_to_pdb.sql
…a lot of output omitted…
```

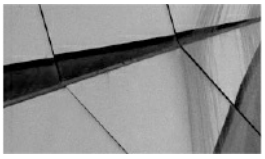
Now we can open the new PDB and begin to use it:

```
SQL> alter pluggable database open;

Pluggable database altered.
```

Cloning a Non-CDB

Another approach for non-CDB conversion is to use cloning. The difference between this and the previous approach is the same as those with unplug/plug-in versus cloning methods for PDBs. This means that a clone requires fewer steps, can be done remotely, and does not require the physical copying and transmission of files.



NOTE

The clone of a non-CDB is always remote; the source database is different from the target.

In the following command, we specify the special NON$CDB name for the PDB:

```
SQL> create database link OLDDB connect to system identified by oracle using 'OLDDB';

Database link created.

SQL> create pluggable database OLDDBPDB from NON$CDB@OLDDB file_name_convert=('/u01/
app/oracle/data/OLDDB','/u01/app/oracle/data/CDBSRC/OLDDBPDB');

Pluggable database created.
```

Once these commands are completed, we need to run the noncdb_to_pdb.sql script, and then we can open the database:

```
SQL> alter session set container=olddbpdb;

Session altered.

SQL> @$ORACLE_HOME/rdbms/admin/noncdb_to_pdb.sql
…a lot of output omitted…

SQL> alter pluggable database open;

Pluggable database altered.
```

Moving PDBs to the Cloud

Another use case of PDB operations is to move the database to the cloud—or back from cloud to on-premise. In essence, this does not differ from moving the PDB around our on-premise servers. The cloud (Oracle or other) database is just another Oracle database and all the operations described in this chapter are valid there as well, whether using the command line or Enterprise Manager.

Thus, for the move, we can use ordinary unplug and plug-in, remote clone, or the full range of options, such as relocate or proxy PDB. For some of the basic scenarios, note that Enterprise Manager Cloud Control has a wizard we can use, shown in Figure 9-1.

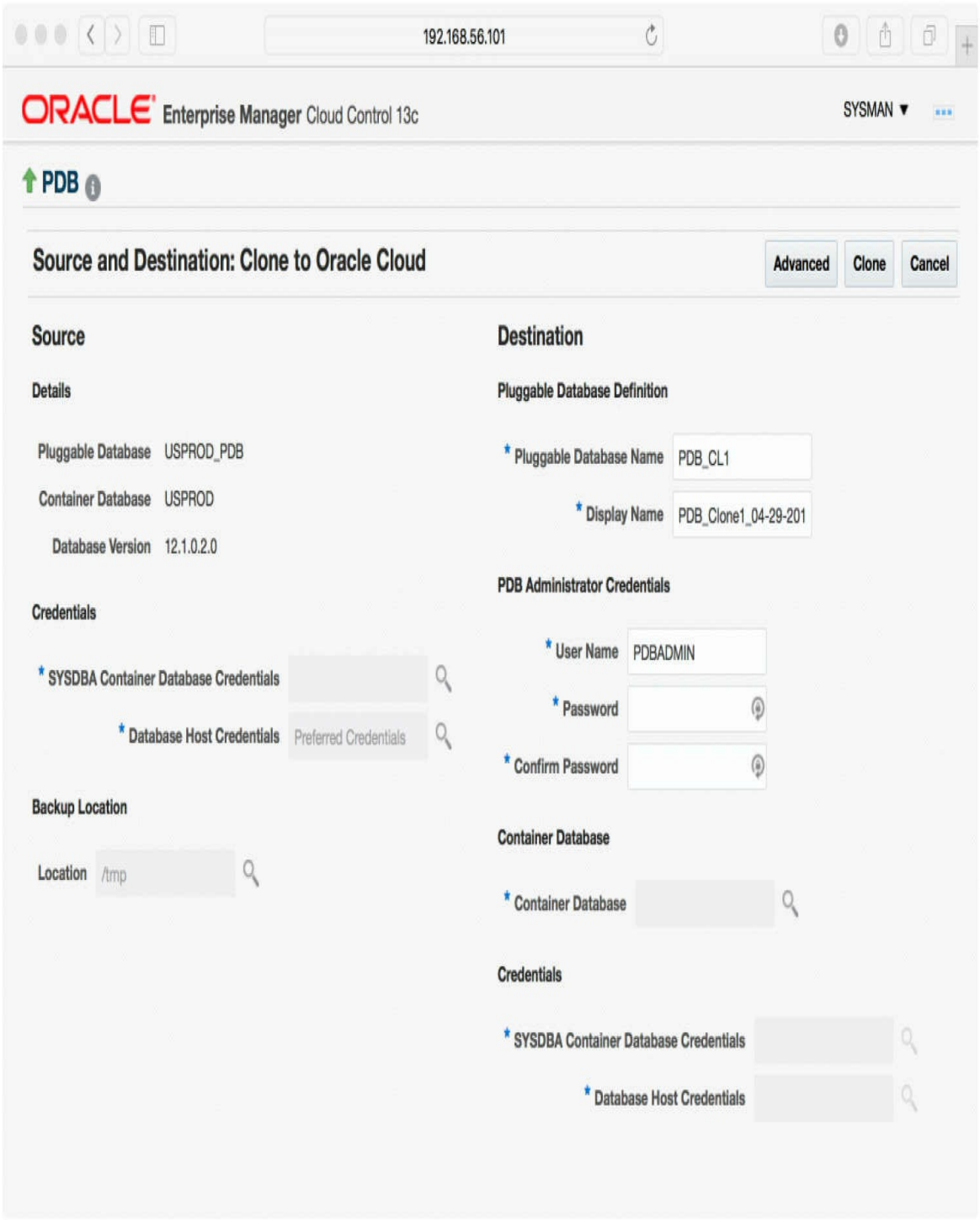


FIGURE 9-1. *Clone to Oracle Cloud in Oracle Enterprise Manager*

Triggers on PDB Operations

Like CDBs, PDBs support database event triggers, including opening and closing the database, server error, logon/logoff, and so on. The only trigger that is CDB-only is `AFTER DB_ROLE_CHANGE`, as switching primary/standby can occur on CDB only (we discuss Data Guard in detail in Chapter 11).

A new trigger introduced in 12*c* is the `BEFORE SET CONTAINER` or `AFTER SET CONTAINER`, fired when a session changes the current container using `ALTER SESSION SET CONTAINER`. The use of this trigger is very similar to a logon/logoff trigger, especially for connection pools and the like, where one session switches between containers; the single logon/logoff trigger is not enough to cover different requirements for the various PDBs.

Finally, and perhaps most importantly in the context of this chapter, as they apply to the types of operations described herein, two new trigger types were added that work at the PDB level only: `AFTER CLONE` and `BEFORE UNPLUG`.

As the name suggests, the `AFTER CLONE` trigger fires at the new database, after it is cloned. If the trigger fails, the new database is left in an UNUSABLE state and the only operation permitted is to drop it.

The `BEFORE UNPLUG` trigger fires when the unplug operation starts. If the trigger fails, the operation does not happen, so no XML file is created and the PDB remains part of the CDB.

In both cases, when the trigger succeeds, it is deleted.

There are several use cases for triggers—for example, if we want to delete any stored passwords and keys to external systems and drop database links before we unplug the database and distribute it. Another use case would be to employ it to mask data after we have cloned a PDB to a test environment.

Full Transportable Export/Import

The transportable database feature is a logical step-up from the trusted and proven transportable tablespace feature, first introduced way back in 8*i*.

Transportable tablespace functionality was originally introduced to move just one or a few tablespaces, such as from an OLTP system to a data warehouse. But over the years, its usage developed to moving large amounts

of data, such as all of the user datafiles during a migration or upgrade.

Additionally, transportable tablespace was also enhanced to handle cross-endian data movement, although one gap still existed, in that it could not move non-table objects such as views or PL/SQL.

The new transportable database feature now addresses this. It handles user tablespaces like transportable tablespace always did and, in addition, it can also export the other objects as a Data Pump export would do, in a single step. And being based on the proven foundation, it includes cross-endian conversion possibilities.

Like transportable tablespaces, the transported tablespaces must be read-only during the import process. RMAN allows a workaround for this for transportable tablespaces, in that it can create an auxiliary instance and run the export there, but this is not available for full transportable export.

This full transportable export feature has been available since version 11.2.0.3.

The multitenant twist on this feature is that the result can be imported into an existing CDB. In other words, it can be used to plug in a non-CDB into a CDB. The advantage of this, as opposed to the simpler way described earlier, is the cross-endian support. It also does not automatically copy the files, so if the conversion happens on the same machine, we do not need to copy the datafiles at all, speeding up the conversion and saving disk space.

Another scenario is plugging in a PDB that would otherwise not meet the requirements: so we can move a database cross-endian, or get around an unmatched list of installed features. (Of course, the target database needs to have installed any features the new PDB and its users and applications will use.)

Unlike the simple and standard plug-in, full transportable import requires that an already existing (empty) database be created first. The import process then imports only user data.

Note also that only one PDB is exported at a time; and when we specify a connection to the CDB, only the CDB user data is exported, not the PDB's.

As an example, let's export a non-CDB database and import it into a fresh new PDB.

First, all of the user tablespaces must be made read-only. However, the database itself must be read/write, as Data Pump creates a job table in the SYS schema:

```
SQL> alter tablespace example read only;

Tablespace altered.

SQL> alter tablespace users read only;

Tablespace altered.
```

Then we run the Data Pump export, creating a directory beforehand, if necessary. Because this exports all the object definitions, it does take some time, although this is minimal in comparison to a data export:

```
expdp system/oracle@noncdb full=y dumpfile=noncdb.dmp directory=dp
transportable=always  logfile=noncdb.log
Starting "SYSTEM"."SYS_EXPORT_FULL_01":  system/********@noncdb full=y
dumpfile=noncdb.dmp directory=dp transportable=always logfile=noncdb.log
Processing object type DATABASE_EXPORT/PRE_SYSTEM_IMPCALLOUT/MARKER
Processing object type DATABASE_EXPORT/PRE_INSTANCE_IMPCALLOUT/MARKER
…a lot of output omitted…
. . exported "WMSYS"."WM$EXP_MAP"                          7.718 KB       3 rows
Master table "SYSTEM"."SYS_EXPORT_FULL_01" successfully loaded/unloaded
******************************************************************************
Dump file set for SYSTEM.SYS_EXPORT_FULL_01 is:
  /home/oracle/dp/noncdb.dmp
******************************************************************************
Datafiles required for transportable tablespace EXAMPLE:
  /u01/app/oracle/data/SRC/example01.dbf
Datafiles required for transportable tablespace USERS:
  /u01/app/oracle/data/SRC/users01.dbf
Job "SYSTEM"."SYS_EXPORT_FULL_01" successfully completed at Fri Jan 22 03:01:35
2016 elapsed 0 00:09:38
```

As you can see, the export conveniently prints the list of necessary files —the dump file and all the datafiles—at the footer of the screen output and the log.

Now we create a new PDB:

```
SQL> create pluggable database PDBIMPORT admin user ads identified by
oracle file_name_convert=('/u01/app/oracle/data/CDBSRC/pdbseed','/u01/
app/oracle/data/CDBSRC/PDBIMPORT);
Pluggable database created.

SQL> alter session set container=PDBIMPORT;

Session altered.

SQL> alter database open;

Database altered.

SQL> create directory dp as '/home/oracle/dp';

Directory created.
SQL> grant read, write on directory dp to public;

Grant succeeded.
```

The next step is to copy the files to the destination directories and change the endian using RMAN if necessary:

```
RMAN> convert datafile '/tmp/example01.dbf', '/tmp/users01.dbf'
  from platform 'Linux x86 64-bit'
  db_file_name_convert '/tmp', '/u01/app/oracle/data/TRG/PDBIMPORT';
```

Now we import the dump:

```
$ impdp system/oracle@PDBIMPORT dumpfile=noncdb.dmp directory=dp transport_
datafiles=/u01/app/oracle/data/TRG/PDBIMPORT/example01.dbf,/u01/app/oracle/data/
TRG/PDBIMPORT/users01.dbf logfile=noncdb.log
Master table "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01" successfully loaded/unloaded
Starting "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01":  system/********@PDBIMPORT
dumpfile=noncdb.dmp directory=dp transport_datafiles=/u01/app/oracle/data/
TRG/PDBIMPORT/example01.dbf,/u01/app/oracle/data/TRG/PDBIMPORT/users01.dbf
logfile=noncdb.log
Processing object type DATABASE_EXPORT/PRE_SYSTEM_IMPCALLOUT/MARKER
Processing object type DATABASE_EXPORT/PRE_INSTANCE_IMPCALLOUT/MARKER
…a lot of output omitted…
Processing object type DATABASE_EXPORT/AUDIT_UNIFIED/AUDIT_POLICY_ENABLE
Processing object type DATABASE_EXPORT/POST_SYSTEM_IMPCALLOUT/MARKER
Job "SYSTEM"."SYS_IMPORT_TRANSPORTABLE_01" completed at Fri Jan 22 04:59:56 2016
elapsed 0 00:11:02
```

Transportable Tablespaces

Not much has been altered with this functionality since 11*g*. The transportable tablespace feature is still present in multitenant and can be used to move data from one database to another, but it's more complicated than unplug/plug-in and limited when compared to a full transportable export, although it still has its use cases.

One notable unique feature is the ability to use RMAN to obtain the datafiles without setting them to read-only, thanks to using an auxiliary instance. In 12*c*, RMAN has been further enhanced with the addition of syntax to specify a tablespace in a particular PDB:

```
RMAN> transport tablespace PDBCOPY:users tablespace destination '/home/oracle/tts';
```

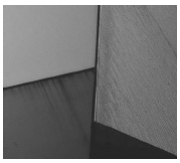
See Chapter 12 for more examples of how transportable databases can be used.

Summary

In this chapter we covered one of the most interesting topics which the multitenant feature has given rise to: separation of the PDBs and their

movement from a CDB to another one. As we have seen, there are a multitude of options, with many ways leading to Rome, and each has its own pros and cons.

As you gain more real-world experience with multitenant, you'll find this functionality more and more useful for solving problems in ways not possible before. This creative leverage comes with real-world experience, lots of experimentation and testing, and a fresh mind when approaching problems.



PART IV

Advanced Multitenant

CHAPTER 10

Oracle Database Resource Manager

Up until now we have focused on creating and configuring a multitenant database environment, including key aspects of the day-to-day administration tasks. In this part of the book we move toward some of the more advanced configuration options. We will begin with a look at Oracle Database Resource Manager, before moving on to disaster recovery (DR) implementations using Oracle Data Guard, followed by a focus on the movement and sharing of data in the final two chapters.

In this chapter, the focus is Resource Manager, which may be familiar territory for some, albeit with new considerations to factor in with Oracle 12*c*. One of the key advantages of multitenant is consolidation, but it also introduces new questions in relation to Resource Manager, such as these:

- How do you manage resources such as CPU, memory, and I/O available to your database?
- Can resource management be micromanaged in a way, enabling distribution of resources to pluggable databases (PDBs) depending on priorities or even time schedules?
- What about resource allocations inside a PDB itself?
- Can resources be managed at a more fine-grained level in a PDB?

In this chapter, we will address these questions and show how you can easily get started with Resource Manager in a multitenant database environment.

Resource Manager Basics

We have already spoken about some of the key advantages of multitenant, such as the ability to consolidate many databases easily into one, as well as the flexibility to provision new databases quickly. In conjunction with these advantages, one core area of the database functionality needs consideration, and that is database resource allocation. Imagine having a server with an abundance of resources such as CPU, memory, and fast storage, such that you would expect everything to be well-equipped for general operational purposes, only to realize later that one of the PDBs consumes virtually all of these resources during busy periods. The flow-on effect is that other PDBs are starved of resources during these times, with the end result being non-optimal performance and disgruntled end users.

By default, operating systems will attempt to distribute resources as requested and do not prioritize among different processes, because they are not aware of which processes should have higher priorities than others. There are some exceptions, but these operating systems require manual configuration or the use of additional software to effect resource allocation and prioritization in some shape or form.

Oracle Resource Manager, which resides within the database itself, has full access to all the runtime information and performance statistics. All the information that describes what is happening inside the database is available, and if Resource Manager is configured correctly, the database can draw upon this information to make decisions on resource distribution; we can ensure, then, that if one area of the database is busy, other PDBs are not starved of resources.

In using Resource Manager in a multitenant environment, the following options are available to you:

- Distribute resources among PDBs based on their priority, ensuring that PDBs requiring higher priority and more resources have the appropriate amount of resources allocated
- Limiting CPU usage of PDBs
- Limiting number of parallel execution servers of PDBs
- Limiting memory usage of PDBs, including ensuring that the minimum memory requirements of a PDB are met

- Limiting resource usage within a PDB for particular sessions
- Limiting PDB I/O generation

Key Resource Manager Terminologies

Before diving into the details of Resource Manager, let's review some key terminology.

Resource consumer group

Resource Manager will allocate resources to resource consumer groups, not to individual sessions or processes. Resource consumer groups can be thought of as sessions, grouped together, based on specific resource usage requirements. Sessions are mapped to a consumer group based on rules configured by the DBA and can be switched between different groups automatically or manually.

Resource plan directive

Resource plan directives are used to associate resource consumer groups with particular resource plans and to specify how resources are to be allocated to the associated resource consumer group. A plan directive in a current active resource plan may be associated with only one consumer group.

Resource plan

The resource plan is the top-level container for the directives. It is the resource plan itself that is activated, which then enables the underlying resource plan directives that specify how resources are allocated to the individual consumer groups. There can be only one active resource plan at any time in the database, but you can create many different resource plans and switch between them as needed. This can be done via the scheduler or manually using the `ALTER SYSTEM` commands.

This hierarchy of components is depicted in Figure 10-1.

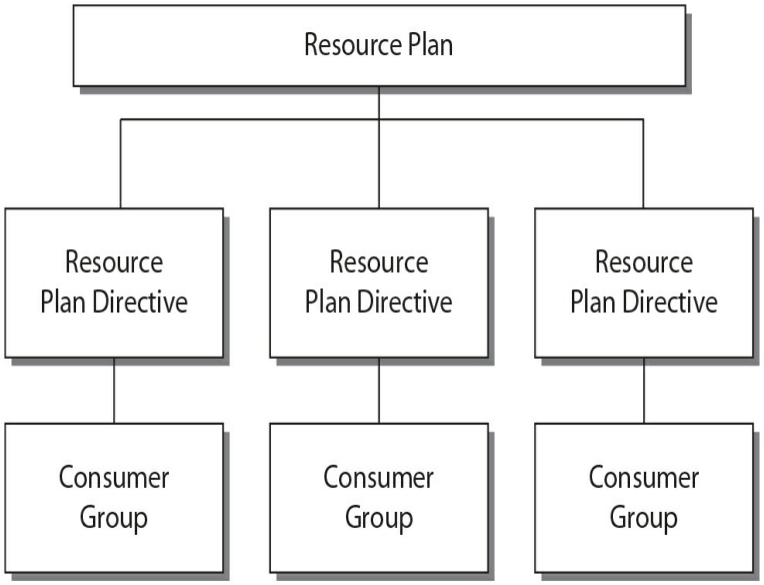


FIGURE 10-1. *A basic Resource Manager plan, directives, and consumer groups*

CDB resource plan

Think of this as the master or top-level plan created at the container database (CDB) level, which specifies how resources (via the use of resource plan directives) are allocated among PDBs within the CDB. A CDB resource plan can have many directives, but each directive in an active plan may reference only one PDB or PDB profile. Note also that two directives cannot both reference the same PDB or PDB profile.

PDB resource plan

A PDB resource plan is the next step in terms of granularity, taking the resources allocated by the CDB resource plan (to specific PDBs) and determining how these resources are then distributed within the PDB among its configured consumer groups.

Figure 10-2 provides a high-level representation of the relationship between a CDB resource plan and PDB resource plans. It also indicates the plan directives and consumer groups, which will be discussed in more detail in the next section.

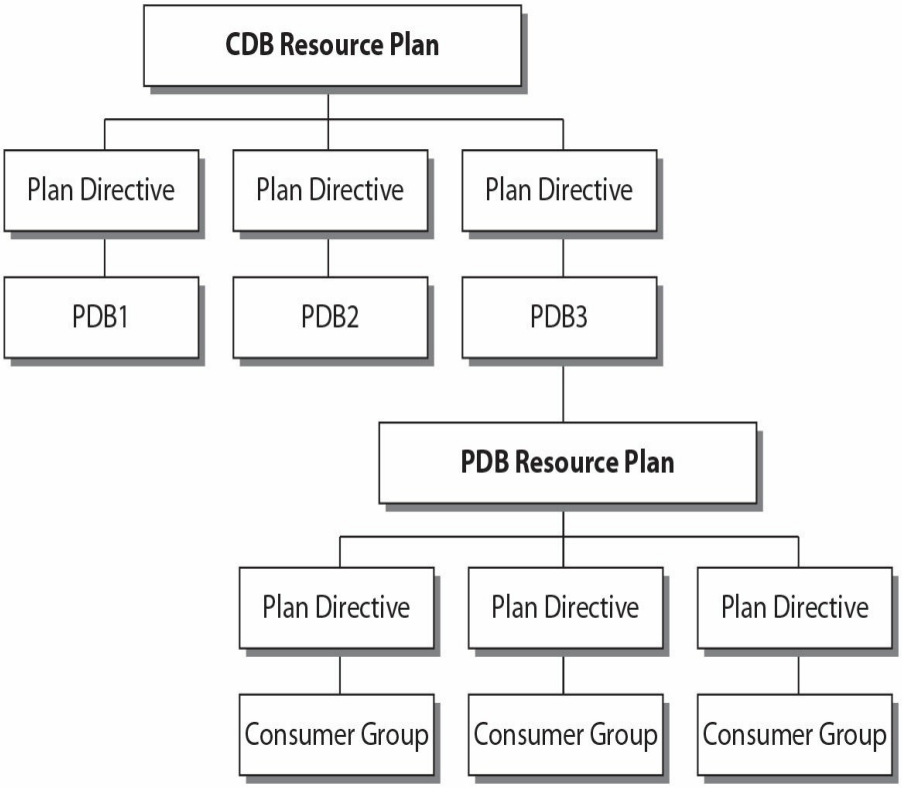


FIGURE 10-2. *A CDB and PDB resource plan relationship*

Subplan

A resource plan directive may reference another resource plan instead of a resource consumer group, and this "stacked" component, or *subplan*, provides additional flexibility to outwork fine-grained resource management as required.

Shares

Resources on a system can be allocated proportionately using *shares*. For example, a CDB containing four PDBs can allocate one share to each of three PDBs and allocate two shares for the remaining PDB, denoting it has higher priority.

PDB profiles

When working with large numbers of PDBs, you can also make use of PDB profiles, which determine the share of system resources allocated to the PDBs to which the profile applies. This includes CPU, memory, and total parallel execution server allocations.

Resource Manager Requirements

Before Resource Manager can be used in a CDB, the CDB must have at least one PDB. Resource Manager is configured via the DBMS_RESOURCE_MANAGER package, and the system privilege ADMINISTER_RESOURCE_MANAGER is required to administer it. The ADMINISTER_RESOURCE_MANAGER system privilege is granted by default to the DBA role with the ADMIN option.

This system privilege cannot be granted via regular SQL grant or revoke statements, but must be done via the DBMS_RESOURCE_MANAGER_PRIVS package using the following two procedures:

- `GRANT_SYSTEM_PRIVILEGE`
- `REVOKE_SYSTEM_PRIVILEGE`

So, for example, if you want to grant user C##XADMIN this system privilege, you would run the following command:

```
BEGIN
   dbms_resource_manager_privs.grant_system_privilege (
     grantee_name => 'C##XADMIN',
     privilege_name => 'ADMINISTER_RESOURCE_MANAGER',
     admin_option=> FALSE);
END;
```

In this example, the parameter `PRIVILEGE_NAME` is specified even though this is not strictly required, because the default value is ADMINISTER_RESOURCE_MANAGER.

Resource Manager is configured and managed via the

DBMS_RESOURCE_MANAGER package, but to allow users to switch consumer groups, they must be granted a specific system privilege using procedures available in the DBMS_RESOURCE_MANAGER_PRIVS package:

- `GRANT_SWITCH_CONSUMER_GROUP`
- `REVOKE_SWITCH_CONSUMER_GROUP`

Resource Manager Levels

Resource management in a CDB can quickly ramp up in complexity when compared to managing a non-CDB. In a CDB, you have to take into account multiple PDBs that may have differing workloads, and competition for resources both within a PDB, as well as at the CDB level. When running Resource Manager in a multitenant environment, resources can be managed at two levels:

- **CDB level** You can manage resources within the CDB, catering for the different PDB workloads and specifying how resources are distributed among them. PDBs may have different priorities, and resultant use limits can be imposed to distribute the total resources available to the CDB accordingly. In most cases in which a CDB contains multiple PDBs, it would be reasonable to assume that some PDBs will have a higher priority than others. Resource Manager at a CDB level helps to enforce and manage these priorities and limitations.
- **PDB level** Drilling down, we can manage workloads and resource usage within a given PDB. For example, let's assume from a high-level point of view that 50 percent of the CDB resources are allocated to PDB1. Within PDB1, this 50 percent portion of the total CDB resources can then be further divided up and portioned out between the different consumer groups.

In the following sections, we will focus on CDB and PDB resource plans in more detail, with examples of how these can be configured.

The CDB Resource Plan

A CDB resource plan is the top-level resource plan, configured for the CDB itself. It directs the management of resources within the CDB, catering to different PDB workloads and resource distribution between the PDBs. When running a CDB with multiple PDBs, it is very likely that you will have to allocate more resources to a specific PDB or PDB group. Or you may need to distribute the system and CDB resources evenly to ensure that all PDBs get sufficient resources, and that no PDBs are being starved of resources. In either case, the CDB resource plan is used to map such allocations appropriately.

Resource Allocation and Utilization Limits

Using Resource Manager, you can prioritize resource usage among PDBs with *share values*. The higher the share value allocated, the higher the priority, which means there is a greater likelihood of obtaining resources when resource contention is encountered. In the same way that resource share values are implemented for individual PDBs, PDB profiles can be applied to a set of PDBs. Figure 10-3 illustrates the basic concept of using shares. In this example, we have a CDB resource plan with two resource plan directives specified, with a total of four shares; one share is assigned to PDB1 and three shares to PDB2. In the event of resource contention, PDB1 will be guaranteed at least one quarter of the resources available, and PDB2 will get three quarters.

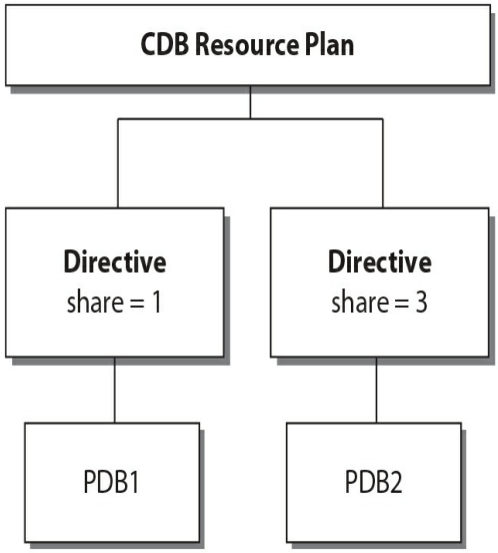
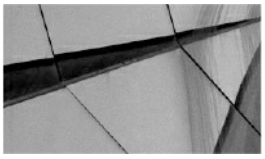


FIGURE 10-3. *Using shares in a CDB resource plan directive*



NOTE

If there is no current resource contention, either PDB1 or PDB2 can have a larger share of the resources than what they have been assigned in the plan directive.

In addition to using shares as a method of prioritizing resources, you can also set utilization limits for PDBs and PDB profiles, which are specified as a percentage. If not specified, the value is 100, which indicates that the associated PDB or PDBs included in a PDB profile can potentially use 100 percent of the CPU resources available in a CDB. The utilization limits can be specified for CPU, memory, and parallel execution servers. The following parameters can be set in a resource plan directive with respect to CPU and parallel execution servers:

- `utilization_limit`
- `parallel_server_limit`

The `utilization_limit` parameter differs from using shares, because it is specific to how much of the CPU resource may be used. For example, if the `utilization_limit` for a particular PDB is specified as 100, it indicates that the PDB may use up to 100 percent of the system CPU resources. If the value is set to 50, it indicates that the PDB, if the system is under load, can use up to 50 percent of the CPU resource. Shares are used to indicate which PDBs have higher priority with regard to all resources, not just CPU resource. Specifying a combination of options, such as shares, `utilization_limit`, and `parallel_server_limit`, provides you with more fine-grained control over how system resources are distributed.

In addition to `utilization_limit` and `parallel_server_limit`, two new memory limits are introduced in Oracle Database 12*c* Release 2. Their values are expressed as percentages in relation to the Program Global Area (PGA), buffer cache, and shared pool sizes. Even though these two parameters may be set, remember that shares are also considered to maintain the fairness of resource allocation. The following two parameters can be specified with regard to memory limits:

- **`memory_min`** The `memory_min` limit takes a default value of 0 if not set explicitly. The goal is that each PDB should be allocated at least this minimum if requested, for the PGA, buffer cache, and the shared pool. If a PDB has reached the minimum, it will be prioritized for releasing memory if needed. If it has not yet reached its minimum, it will be preferred when requesting memory.
- **`memory_limit`** The `memory_limit` default value is 100. This is a hard limit on the maximum memory a PDB can consume. If a PDB reaches the maximum, it can allocate only memory that it has released itself, while other PDBs that have not yet reached their limit (maximum) may allocate memory that was freed from any PDB.

Figure 10-4 shows a high-level overview of how resource utilization limits can be specified as part of the resource plan directives. In the example, PDB2 does not have any utilization limits imposed (limits are specified as 100), but PDB1 limitations are specified up to only 40 percent of resources (CPU, parallel execution servers, and memory).

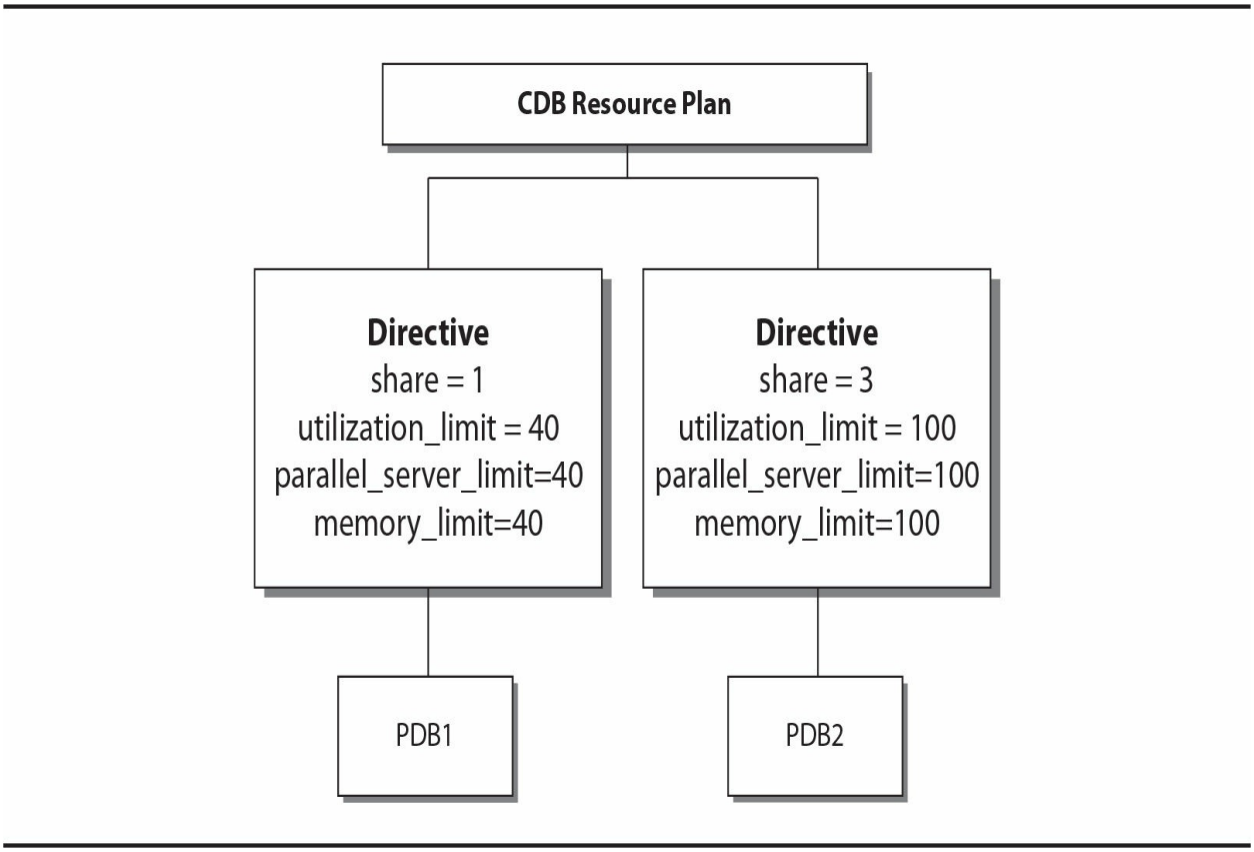


FIGURE 10-4. *High-level CDB resource plan directives with utilization limits*

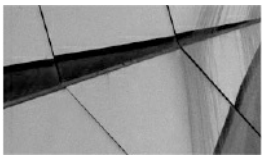
Memory- and I/O-related parameters can also be set at the PDB level. For more information, see the section “Manage PDB Memory and I/O via Initialization Parameters” later in the chapter.

But what happens to PDBs that are not specified in any of the created resource plan directives? The answer brings us to our next topic: default directives.

Default and Autotask Directives

Up to this point, we have discussed CDB resource plans and the resource plan directives that have been applied to specific PDBs. But what if you create a new PDB, plug a new PDB into the CDB, or have already configured PDBs that were not explicitly defined when creating directives? Default resource plan directives are used for such scenarios, and they will have one

share assigned and resource limit parameters set to default values. Alternatively, you can generate new directives for freshly created PDBs or adjust the default PDB directives if you prefer.



NOTE

The directive for a PDB will be retained if the PDB is unplugged from a CDB. If a directive is no longer needed, you will need to drop the directive manually.

Should it be required, you can adjust the default directive using a procedure inside the DMBS_RESOURCE_MANAGER package called `UPDATE_CDB_DEFAULT_DIRECTIVE`. The following code block illustrates how the default directive may be updated:

```
BEGIN
   dbms_resource_manager.create_pending_area();
   dbms_resource_manager.update_cdb_default_directive(
           plan => 'CDB_RPLAN'
         , new_shares => 2
         , new_utilization_limit => 50);
   dbms_resource_manager.validate_pending_area();
   dbms_resource_manager.submit_pending_area();
 END;
```

From this code block, we can see that the default directive for PDBs was updated to two shares and a new utilization limit of 50 percent was specified.

Notice in the preceding code block that a pending area is created and validated before being submitted. The pending area is a staging area where a resource plan is created, updated, or deleted, without affecting currently running applications. After changes are made to a pending area, it is validated using the `VALIDATE_PENDING_AREA` procedure. Once the pending area is validated, the `SUBMIT_PENDING_AREA` procedure is used to apply all pending changes to the data dictionary. Once the submission is complete, the pending

area will be cleared.

The second default directive, the autotask directive, applies to automatic maintenance tasks during the maintenance windows. The default allocation is no shares (actually, –1), which means that the automated maintenance tasks gets 20 percent of the system resources. The utilization limit is set to 90 percent and the parallel server limit is set to 100 percent. As with the default directive for PDBs, you may also update the autotask directive, by using the `UPDATE_CDB_AUTOTASK_DIRECTIVE` procedure in the DBMS_RESOURCE_MANAGER package. In the following example, we update the autotask directive‘s share value to 2:

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.update_cdb_autotask_directive(
          plan => 'CDB_RPLAN'
        , new_shares => 2);
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

Creating a CDB Resource Plan

Now that you understand CDB resource plans and their elements, we can launch into a few examples to show you how to create these plans. There are multiple ways to achieve this, but the most common method is to use SQL commands via SQL*Plus and execute the required DBMS_RESOURCE_MANAGER procedures. It is also possible to perform some of these tasks via Enterprise Manager Cloud Control, Enterprise Manager Database Express, or Oracle SQL Developer. In this section, we will show you how to outwork these tasks using SQL*Plus and the DBMS_RESOURCE_MANAGER package.

Example: Creating a CDB Resource Plan for Individual PDBs