Before jumping into the commands, we first need to plan out what we want to implement. A simple way of doing this is to create a table with all the allocation options listed, or perhaps even a basic flow diagram. In the first example, we will make use of the former to illustrate the CDB resource plan, which will focus specifically on individual PDBs, without the use of PDB profiles.

In this example, we will create a CDB resource plan and directives based on the information listed in the following table. This example also includes the Default and Autotask directives, and they will be modified from the default values to those specified in the table.

PDB/Directive Name	Shares	Utilization Limit	Parallel Server Limit	Memory Minimum	Memory Limit
PDB1	1	70	50	20	40
PDB2	4	100	100	50	80
Default	1	90	50		100
Autotask	2	80	80		100

The goal of this example is to show the commands needed to execute the CDB resource plan, as per the values in the table. The steps follow:

1. Create the pending area:
```
dbms_resource_manager.create_pending_area();
```
2. Create the resource plan:
```
dbms_resource_manager.create_cdb_plan(
        plan =>'CDB_RPLAN'
        ,comment => 'DEMO CDB Resource Plan');
```

At this stage, even though we are going to perform the rest of the configuration using SQL commands, it is also possible to do this via a graphical interface. For example, Figure 10-5 illustrates how this can be done using EM Database Express 12.2 when the CDB resource plan is created. (Note also that the default directive can be updated at the CDB resource plan creation time.)

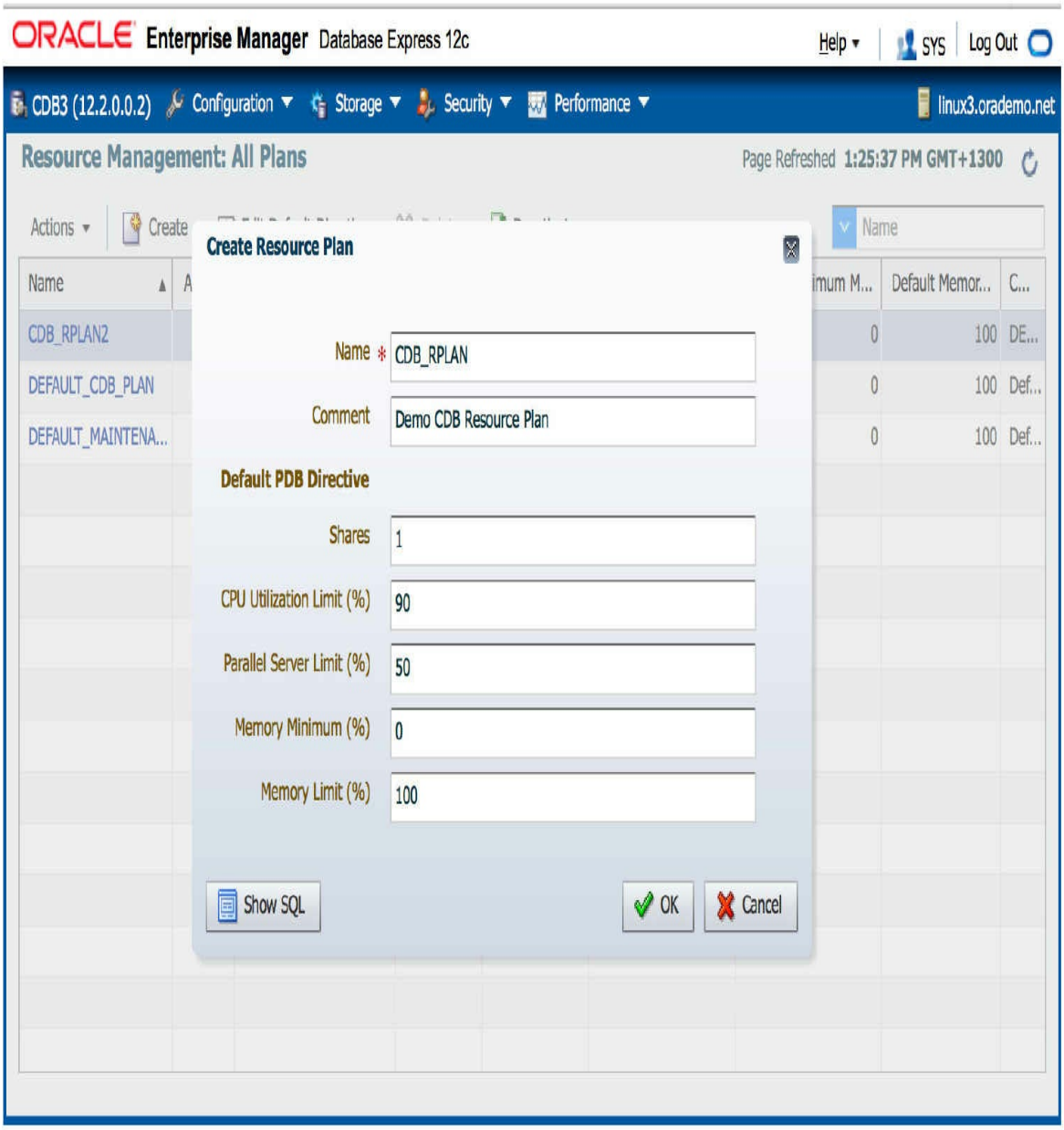


FIGURE 10-5. *Using EM Database Express 12.2 to create a CDB resource plan*

Now, returning to our command line example, the following steps will be performed using SQL commands.

3. Create two plan directives for PDB1 and PDB2:

```
dbms_resource_manager.create_cdb_plan_directive(plan => 'CDB_RPLAN'
      , comment => 'PDB1 Plan Directive'
      , pluggable_database => 'PDB1'
      , shares => 1
      , utilization_limit => 70
      , parallel_server_limit => 50
      , memory_min => 20
      , memory_limit => 40);
dbms_resource_manager.create_cdb_plan_directive(plan => 'CDB_RPLAN'
      , comment => 'PDB2 Plan Directive'
      , pluggable_database => 'PDB2'
      , shares => 4
      , utilization_limit => 100
      , parallel_server_limit => 100
      , memory_min => 50
      , memory_limit => 80);
```

4. Update the default PDB directive:

```
dbms_resource_manager.update_cdb_default_directive(
      plan => 'CDB_RPLAN'
   , new_shares => 2
   , new_utilization_limit => 90
   , new_parallel_server_limit => 50);
```

5. Update the default autotask directive:

```
dbms_resource_manager.update_cdb_autotask_directive(
    plan => 'CDB_RPLAN'
   , new_shares => 2
   , new_utilization_limit => 80
   , new_parallel_server_limit => 80);
```

6. Validate the pending area:

```
dbms_resource_manager.validate_pending_area();
```

7. Submit the pending area:

```
dbms_resource_manager.submit_pending_area();
```

At this stage, the CDB resource plan and its directives have been created, but we must keep in mind that it has not yet been enabled. We can review the CDB resource plan directives by listing the information in

DBA_CDB_RSRC_PLAN_DIRECTIVES, as shown in Figure 10-6.

```
SQL> col pluggable_database     heading "Pluggable Database|Directive" for a25
SQL> col directive_type         heading "Directive|Type" for a20
SQL> col shares                 heading "Shares" for 99999
SQL> col utilization_limit      heading "Util|Lmit" for 99999
SQL> col parallel_server_limit  heading "Par|Limit" for 99999
SQL> col memory_min             heading "Memory|Min" for 99999
SQL> col memory_limit           heading "Memory|Limit" for 99999
SQL> select pluggable_database
  2  , directive_type
  3  , shares
  4  , utilization_limit
  5  , parallel_server_limit
  6  , memory_min
  7  , memory_limit
  8  from dba_cdb_rsrc_plan_directives
  9  where plan='CDB_RPLAN';

Pluggable Database       Directive                      Util    Par Memory Memory
Directive                Type               Shares   Lmit  Limit    Min  Limit
------------------------ ------------------ ------ ------ ------ ------ ------
ORA$DEFAULT_PDB_DIRECTIVE DEFAULT_DIRECTIVE      2     90     50
ORA$AUTOTASK             AUTOTASK                2     80     80
PDB1                     PDB                     1     70     50     20     40
PDB2                     PDB                     4    100    100     50     80

SQL>
```

FIGURE 10-6. *Review CDB resource plan directives*

8. Enable the CDB resource plan:

Now that we have created a CDB resource plan called CDB_RPLAN, we can enable it for use. This is achieved by executing the following command:

```
SQL>  alter system set resource_manager_plan = CDB_RPLAN;
```

Reviewing the database alert log and initialization parameters, we can see that the change has taken effect. When reviewing EM Cloud Control, as shown in Figure 10-7, the CDB resource plan displays with the status of Active. (Note that EM Cloud Control 13*c* [13.1] does not show the memory limit parameters.)

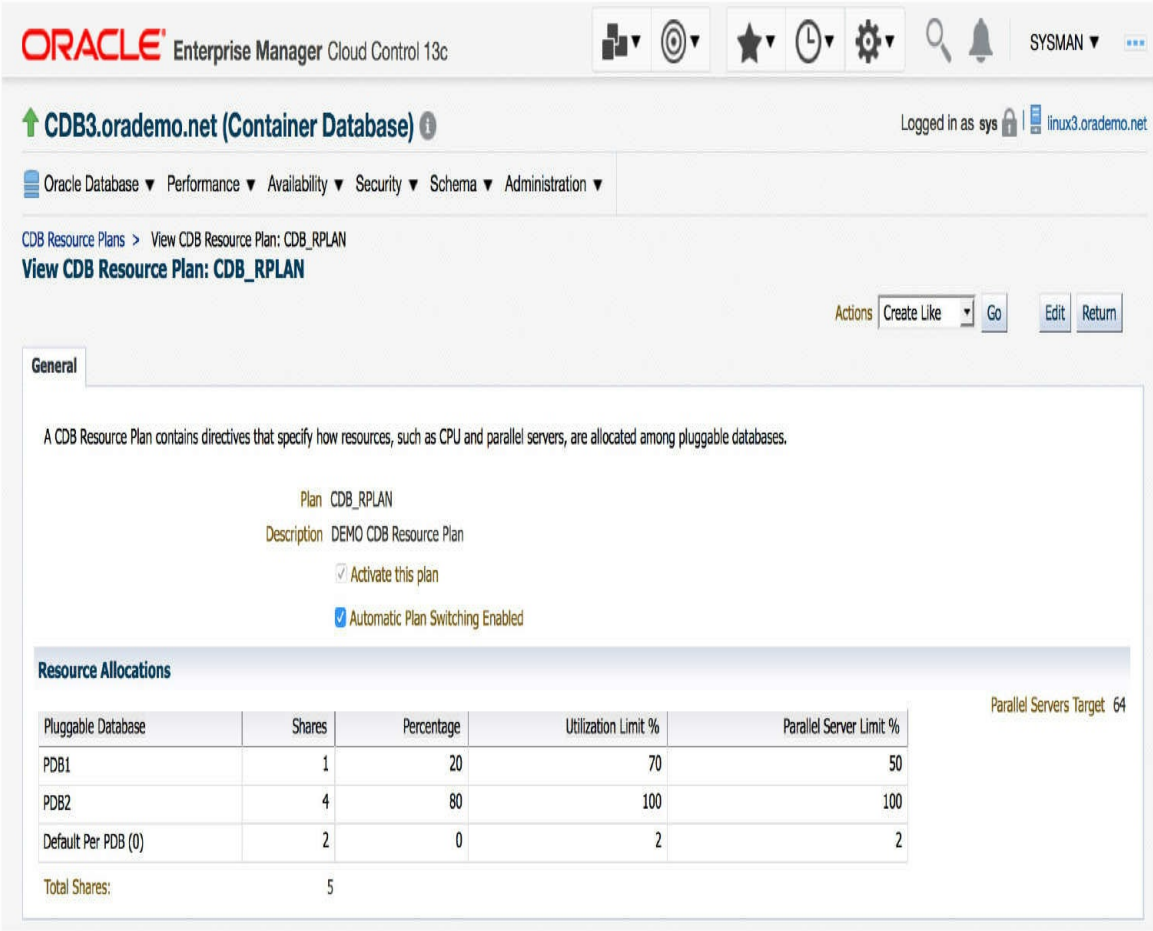
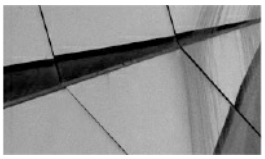


FIGURE 10-7. *EM Cloud Control showing current active CDB resource plan*



NOTE

As mentioned, it is also possible to configure the Resource Manager from Oracle SQL Developer.

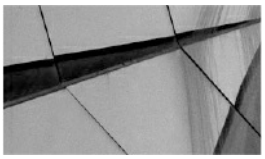
In the preceding example, the different commands to be executed were listed as discrete steps. However, instead of running each of these individually, you can group them together in one code block and execute it as a single unit. The following code block shows this, grouping all the commands described in steps 1–7; once executed, you will have created the CDB resource plan, ready to be activated when needed.

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.create_cdb_plan(
         plan =>'CDB_RPLAN'
        , comment => 'DEMO CDB Resource Plan');
  dbms_resource_manager.create_cdb_plan_directive(plan => 'CDB_RPLAN'
        , comment => 'PDB1 Plan Directive'
        , pluggable_database => 'PDB1'
        , shares => 1
        , utilization_limit => 70
        , parallel_server_limit => 50
        , memory_min => 20
        , memory_limit => 40);
  dbms_resource_manager.create_cdb_plan_directive(plan => 'CDB_RPLAN'
        , comment => 'PDB2 Plan Directive'
        , pluggable_database => 'PDB2'
        , shares => 4
        , utilization_limit => 100
        , parallel_server_limit => 100
        , memory_min => 50
        , memory_limit => 80);
  dbms_resource_manager.update_cdb_default_directive(
         plan => 'CDB_RPLAN'
        , new_shares => 2
        , new_utilization_limit => 90
        , new_parallel_server_limit => 50);
  dbms_resource_manager.update_cdb_autotask_directive(
         plan => 'CDB_RPLAN'
        , new_shares => 2
        , new_utilization_limit => 80
        , new_parallel_server_limit => 80);
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

Modifying a CDB Resource Plan

The CDB resource plan can be updated using the following DBMS_RESOURCE_MANAGER procedures:

- UPDATE_CDB_PLAN
- UPDATE_CDB_PLAN_DIRECTIVE



NOTE
The UPDATE_CDB_PLAN enables the update only of the comments associated with the plan.

Updating a plan directive is similar to creating a new one, in that you have to specify PLAN and PLUGGABLE_DATABASE (directive) values to indicate which directive you would like to update. To modify the shares, comments, or limits, your parameters are all prefixed with NEW. As such, there is no need to specify the old and new values—only the new need to be specified. For example, to update the directive for PDB1 in the previously created CDB resource plan (CDB_RPLAN) and increase the utilization_limit from 70 to 80 percent, we can execute the following:

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.update_cdb_plan_directive(
          plan => 'CDB_RPLAN'
        , pluggable_database => 'PDB1'
        , new_utilization_limit => 80);
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

Executing this update will take effect immediately, even on an active plan. It is also possible to add CDB resource plan directives at any time for a PDB using the CREATE_CDB_PLAN_DIRECTIVE procedure.

Enabling or Removing a CDB Resource Plan

It is possible to have more than one CDB resource plan in a container

database, but only one can be active at any given point in time. The following commands can be used to enable or, when no longer required, remove a CDB resource plan.

Enable a CDB Resource Plan To enable a CDB resource plan, use the following:

```
SQL>  alter system set resource_manager_plan = CDB_RPLAN;
```

Or set `resource_manager_plan` to an empty value to disable it:

```
SQL>  alter system set resource_manager_plan = '';
```

It is also possible to enable the CDB resource plan using a scheduler window, because it is commonplace for the resource usage profile of an environment to differ between day and night. The following example demonstrates how to enable a resource plan based on a scheduler window. The CDB resource plan name in this instance is DAYTIME_RPLAN:

```
BEGIN
  dbms_scheduler.create_window (
          window_name => 'DAYTIME_WINDOW',
          resource_plan => 'DAYTIME_RPLAN',
          start_date => '28-MAR-15 7:30:00AM',
          repeat_interval => 'freq=daily',
          duration => interval '12' hour);
END;
```

Remove a CDB Resource Plan To remove a CDB resource plan, use the following:

```
BEGIN
   dbms_resource_manager.create_pending_area();
   dbms_resource_manager.DELETE_CDB_PLAN('CDB_RPLAN');
   dbms_resource_manager.submit_pending_area();
END;
```

Removing a CDB Resource Plan Directive

When performing plug-in/unplug operations on PDBs, you may encounter a requirement to remove CDB resource plan directives, although in some cases you will want to keep directives, particularly when you know you are going to plug in the PDB again. To remove a directive, the `DELETE_CDB_PLAN_DIRECTIVE` procedure can be used. Following is a basic example of this, in which we remove the CDB resource plan directive for the PDB1 database:

```
BEGIN
   dbms_resource_manager.create_pending_area();
   dbms_resource_manager.DELETE_CDB_PLAN_DIRECTIVE(
        plan => 'CDB_RPLAN'
        pluggable_database => 'PDB1');
   dbms_resource_manager.submit_pending_area();
END;
```

Creating a CDB Resource Plan Using PDB Profiles

We've covered how to create CDB resource plans and directives associated with individual PDBs, but what if you have a group of PDBs?

The steps to create a CDB resource plan using PDB profiles are very similar to those that we have already detailed. In short, this method is appropriate when you want to set specific directives for a resource plan that applies to a number of PDBs—in this case, you can think of the PDBs as being grouped together. One key step is required when following this method: you have to set the PDB initialization parameter `DB_PERFORMANCE_PROFILE`, which requires you to close and reopen the PDB once it has been set. The following commands illustrate how this can be done:

```
SQL> connect / as sysdba
SQL> alter session set container=PDB1;
SQL> alter system set db_performance_profile='IMPORTANT_PDBS'
scope=spfile;
SQL> alter pluggable database close;
SQL> alter pluggable database open;
SQL> show parameter db_performance_profile
```

You can create a CDB resource plan using a PDB profile using the `CREATE_CDB_PROFILE_DIRECTIVE` procedure of the DBMS_RESOURCE_MANAGER package. The principles are the same as in the earlier example, but instead of specifying the `PLUGGABLE_DATABASES` parameter, you specify the `PROFILE`. Working from the earlier example, instead of using the individual PDB1 and PDB2 when the directives are created, we create two profiles: IMPORTANT_PDBS and LOWPRIORITY_PDBS. Note that in this example, the default and autotask directives are left as-is, using the default values, although you may update them if required.

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.create_cdb_plan(
          plan =>'CDB_RPLAN2'
          ,comment => 'DEMO CDB Resource Plan using PDB Profiles');
  dbms_resource_manager.create_cdb_profile_directive(
           plan => 'CDB_RPLAN2'
         , comment => 'Low Priority PDBs Directive'
         , profile => 'LOWPRIORITY_PDBS'
         , shares => 1
         , utilization_limit => 70
         , parallel_server_limit => 50
         , memory_min => 20
         , memory_limit => 40);
  dbms_resource_manager.create_cdb_profile_directive(
           plan => 'CDB_RPLAN2'
         , comment => 'High Priority PDBs Directive'
         , profile => 'IMPORTANT_PDBS'
         , shares => 4
         , utilization_limit => 100
         , parallel_server_limit => 100
         , memory_min => 50
         , memory_limit => 80);
  dbms_resource_manager.validate_pending_area();
  dbms_resource_manager.submit_pending_area();
END;
```

Once the CDB resource plan using PDB profiles has been created, you can review the directive details by selecting from DBA_CDB_RSRC_PLAN_DIRECTIVES. The following SQL query can be used for this purpose; make sure you include the `pluggable_database` and `profile` columns:

```
SQL> select plan, pluggable_database, profile, directive_type, shares,
     utilization_limit, parallel_server_limit, memory_min, memory_limit
     from dba_cdb_rsrc_plan_directives
     where plan='CDB_RPLAN2';
```

Removing a CDB Resource Plan Directive for a PDB Profile The same procedure, DELETE_CDB_PLAN_DIRECTIVE, is used to remove a directive for a specific PDB or a PDB profile. But instead of specifying the `pluggable_database` parameter, the `profile` parameter should be used, with the profile name being the assigned value. In this example we remove the `LOWPRIORITY_PDBS` directive:

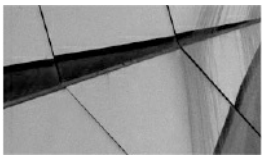
```
BEGIN
   dbms_resource_manager.create_pending_area();
   dbms_resource_manager.DELETE_CDB_PLAN_DIRECTIVE(
      plan => 'CDB_RPLAN'
      profile => 'LOWPRIORITY_PDBS');
   dbms_resource_manager.submit_pending_area();
END;
```

The PDB Resource Plan

When a CDB resource plan allocates resources to a PDB, the PDB resource plan then goes one step further and distributes them based on its own directives. Using this method gives you a more fine-grained level of resource management control and flexibility.

Creating a PDB resource plan is similar to the method used for Resource Manager with non-CDB configurations. The DBMS_RESOURCE_MANAGER package is invoked for this purpose, and a requirement for the following high-level steps to be performed:

- Creation of consumer groups (using `CREATE_CONSUMER_GROUP`)
- Setting the consumer group mapping (using `SET_CONSUMER_GROUP_MAPPING`)
- Creation of a PDB resource plan (using `CREATE_PLAN`)
- Creation of a PDB resource plan directives (using `CREATE_PLAN_DIRECTIVE`)



NOTE

When performing these tasks, you must be connected to the PDB for which you are creating the PDB resource plan. To be clear, a PDB resource plan will manage the workload and resource allocations within a single PDB only.

You should be aware of a number of PDB resource plan restrictions, including these:

- It cannot contain subplans.
- It can include a maximum of eight consumer groups.
- It cannot have a multiple-level scheduling policy.

In the next section, we will show you the steps to create a PDB resource plan. The concepts are the same as for non-CDB environments, but for more detail on consumer groups, plan directives, and mappings refer to the Oracle Database 12*c* documentation.

Creating a PDB Resource Plan

To bring it all together, we'll now work through how we can create a PDB resource plan. Extending the CDB resource plan, CDB_RPLAN, created earlier, in this example we will focus on a PDB called PDB2 and create a basic PDB resource plan called PDB_RPLAN. A high-level view of this configuration is shown in Figure 10-8.

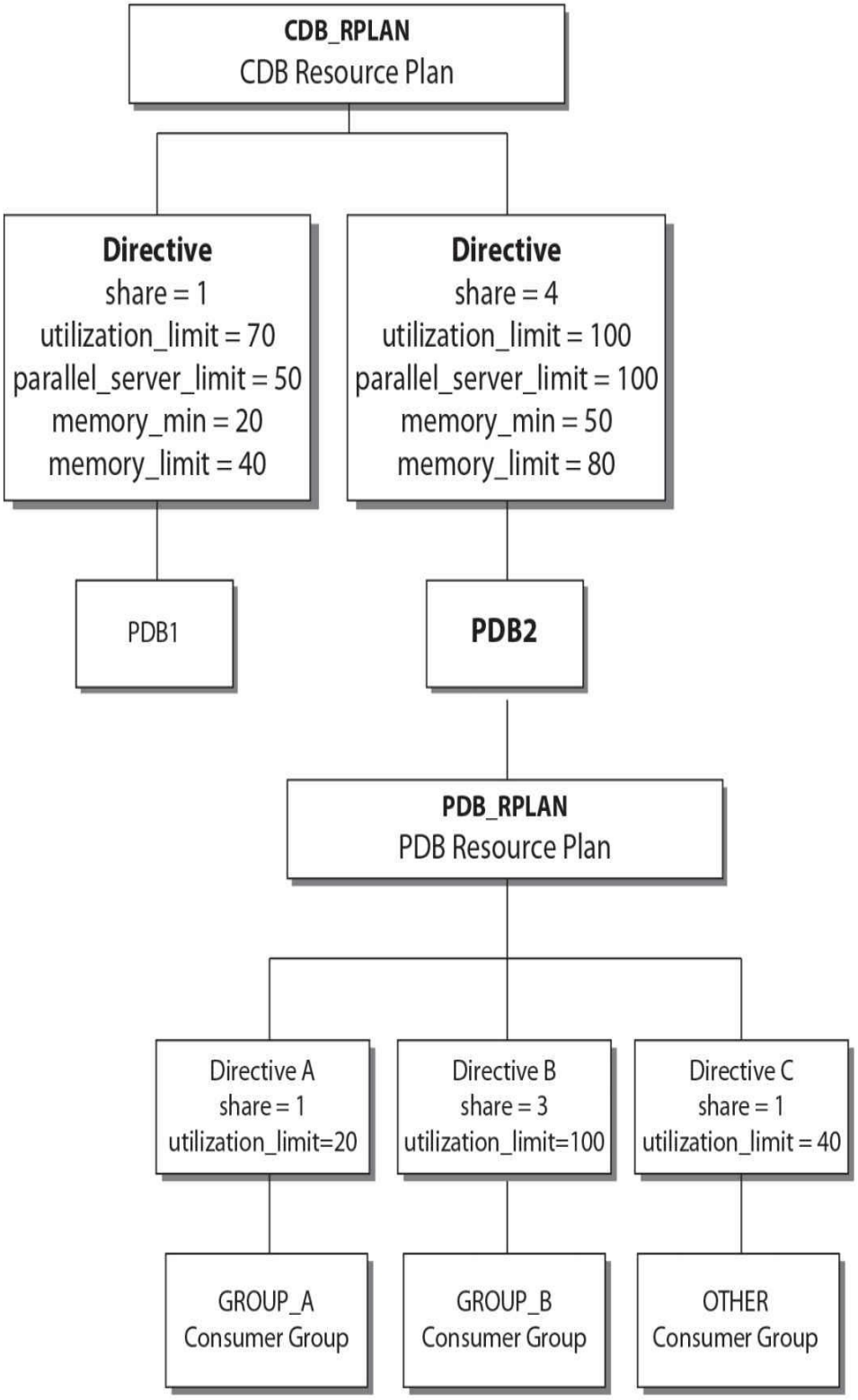


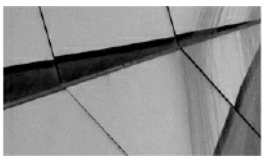
FIGURE 10-8. *Resource Manager: CDB and PDB resource plan overview*

We can now map out the guaranteed and estimated resource allocations:

- For consumer group GROUP_A:
  - Will be guaranteed 16 percent of the total resources: (4/5 shares) 80% × (1/5 shares) 20%
  - Limited to 20 percent of the total resources: (80% × 20%)
- For consumer group GROUP_B:
  - Will be guaranteed 48 percent of the total resources: (4/5 shares) 80% × (3/5 shares) 60%
  - Limited to 80 percent of the total resources: (80% × 100%)
- For consumer group OTHER (OTHER_GROUPS):
  - Will be guaranteed 16 percent of the total resources: (4/5 shares) 80% × (1/5 shares) 20%
  - Limited to 32 percent of the total resources: (80% × 40%)

The code example that follows documents how this PDB resource plan can be created. Note that it does not show how users are assigned to the consumer groups; that is achieved with the DBMS_RESOURCE_MANAGER_PRIVS package, and specifically the `GRANT_SWITCH_CONSUMER_GROUP` procedure.

```
BEGIN
 dbms_resource_manager.clear_pending_area();
 dbms_resource_manager.create_pending_area();
 dbms_resource_manager.create_plan(
           plan =>'PDB_RPLAN'
         , comment => 'DEMO PDB Resource Plan');
 dbms_resource_manager.create_consumer_group(
          consumer_group =>  'GROUP_A'
         , comment => 'Demo Consumer Group A - Low Priority');
 dbms_resource_manager.create_consumer_group(
          consumer_group =>  'GROUP_B'
         , comment => 'Demo Consumer Group B - High Priority');
 dbms_resource_manager.create_plan_directive(
           plan => 'PDB_RPLAN'
         , group_or_subplan => 'GROUP_A'
         , comment => 'For Low Priority Users'
         , shares => 1
         , utilization_limit => 70);
 dbms_resource_manager.create_plan_directive(
           plan => 'PDB_RPLAN'
         , group_or_subplan => 'GROUP_B'
         , comment => 'For High Priority Users'
         , shares => 3
         , utilization_limit => 100);
 dbms_resource_manager.create_plan_directive(
           plan => 'PDB_RPLAN'
         , group_or_subplan => 'OTHER_GROUPS'
         , comment => 'For Other Users'
         , shares => 1
         , utilization_limit => 40);
 dbms_resource_manager.validate_pending_area();
 dbms_resource_manager.submit_pending_area();
END;
```



NOTE

If a non-CDB is plugged into a CDB as a PDB, it will function the

same as before, as long as there are no subplans, all resource allocations are at one level (level 1), and the consumer groups total does not exceed eight. If there are any violations, or any of these three mentioned restrictions exist, the plan will be converted and its status will be updated to LEGACY. It is recommended that converted plans with the status of LEGACY be reviewed thoroughly, because they may behave differently than expected.

Enable or Disable a PDB Resource Plan

To enable or disable a PDB resource plan, you update the initialization parameter `RESOURCE_MANAGER_PLAN` for the specified PDB. This can be done by executing the `alter system` command while the current container is set to the PDB whose plan you want to update (enable or disable). As with the CDB resource plan, to disable the PDB resource plan, you set the `RESOURCE_MANAGER_PLAN` parameter to an empty value. To enable or disable the PDB resource plan—PDB_RPLAN for PDB1 in the following example—you use the following commands:

```
SQL> connect / as sysdba
SQL> alter session set container=PDB1;
SQL> alter system set RESOURCE_MANAGER_PLAN='PDB_RPLAN';
```

Use this to disable the plan:

```
SQL> alter system set RESOURCE_MANAGER_PLAN='';
```

Removing a PDB Resource Plan

To remove a PDB resource plan, use the `DELETE_PLAN` procedure in the DBMS_RESOURCE_MANAGER package. So, to remove the PDB resource

plan created in the preceding example, PDB_RPLAN, execute the following from PDB1:

```
BEGIN
   dbms_resource_manager.create_pending_area();
   dbms_resource_manager.DELETE_PLAN('PDB_RPLAN');
   dbms_resource_manager.submit_pending_area();
END;
```

Manage PDB Memory and I/O via Initialization Parameters

You might have noticed that in Oracle Database 12*c*, Automatic Shared Memory Management (ASMM) is preferred over the use of Automatic Memory Management (AMM). There are a number of reasons for this, which are beyond the scope of this book, but it is recommended that you review the method you are using. If you are unfamiliar with HugePages, consult Oracle Support Note 361468.1. This functionality can have a significant impact on an environment and may assist in more effective memory management and usage.

As mentioned earlier in the chapter, one of the new features introduced in Oracle Database 12*c* Release 2 is the option to specify memory limits as part of the CDB resource plan directives. The two parameters mentioned previously are `MEMORY_MIN` and `MEMORY_LIMIT`, but it does not stop there; you can also control memory allocations for PDBs by using initialization parameters.

PDB Memory Allocations

You can set a number of initialization parameters at the PDB level, including some specifically related to memory. Do take caution when setting these values, however, because you need to ensure that you have sufficient memory allocated for the CDB and the rest of the PDBs contained within it. The following parameters can be configured:

- **DB_CACHE_SIZE** Minimum guaranteed buffer cache for the PDB.
- **SHARED_POOL_SIZE** Minimum shared pool size for the PDB. If not set at the PDB level, there is no limit to the amount of shared pool it can use (though it is limited to the CDB's shared pool size).
- **PGA_AGGREGATE_TARGET** Maximum PGA size for the PDB.
- **SGA_MIN_SIZE** Minimum SGA size for the PDB. This new parameter introduced in 12.2 applies to PDBs only. If this is set on a CDB level it will be ignored for the CDB but will be inherited by all PDBs in the CDB. It is not recommended to set the total sum of `SGA_MIN_SIZE` for all PDBs higher than 50 percent of the CDB SGA size.
- **SGA_TARGET** Maximum SGA size for the PDB.

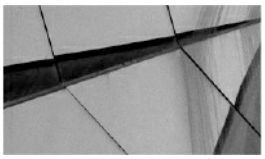
These parameters can be set for a specific PDB by having that PDB set as the current container, prior to running the following `ALTER SYSTEM` commands:

```
SQL> connect / as sysdba
SQL> alter session set container=PDB1;
SQL> alter system set SGA_MIN_SIZE=1G scope=both;
```

Limit PDB I/O

I/O can also be limited for PDBs in a similar way to how memory is controlled—at the PDB level using initialization parameters. There are two key parameters to be aware of. Both have a value of 0 by default, which actually disables any I/O limits being imposed on the PDB, and they are specific to PDBs, meaning they cannot be set in a CDB:

- **MAX_IOPS** Specifies a limit of I/O operations per second in the PDB
- **MAX_MBPS** Limits the megabytes per second operations in a PDB



NOTE

The memory and I/O limit parameters can be set using the `ALTER SYSTEM` *command while connected to a PDB as the current container.*

Instance Caging

Resource Manager can be configured only for one instance on a server, because it is not aware of what other instances on the same machine are doing. To overcome this limitation, *instance caging* was introduced in Oracle Database 11*g* (11.2). Prior to Oracle Database 12*c*, when multiple databases (non-CDB) were consolidated onto one server, instance caging was used to manage and distribute CPU resources among database instances. However, with the introduction of multitenant, consolidation can be taken a step further, in that these non-CDBs can be converted to PDBs. This opens the possibility of administering CPU resource allocation inside the database instance using Resource Manager.

Instance caging, together with Resource Manager, is an effective way to manage multiple database instances on a single server.

Instance Caging to Resource Manager

As mentioned earlier in the chapter, prior to 12*c*, instance caging could be used when consolidating non-CDB environments. With multitenant, this can be further enhanced by converting the non-CDBs to PDBs and making use of Resource Manager to distribute the resource between the PDBs. In this example, we will show you how this can be done.

In the next example, two non-CDB databases are located on the same server. They are CRMDEV and CRMREP. These databases use instance caging, where `CPU_COUNT` is set to 3 on the CRMREP reporting database and 1 on the CRMDEV development. The two databases are converted from non-CDB to PDBs (see Chapter 9 for more detail on how to convert a non-CDB to a PDB) in a new container database called CDB3. In this example, we will

distribute the resource as shown in the following table. Notice that the default directive is also listed for completeness: the default SHARE is 1 and the `UTILIZATION_LIMIT` is 100.

PDB	SHARE	UTILIZATION_LIMIT
CRMDEV	1	25
CRMREP	3	75
DEFAULT	1	100

We use a total of four shares, and we distribute this in the same way that the `CPU_COUNT` was distributed. Using the `UTILIZATION_LIMIT` is not strictly necessary (it enforces a hard limit). Figure 10-9 shows the resource plan to be implemented.

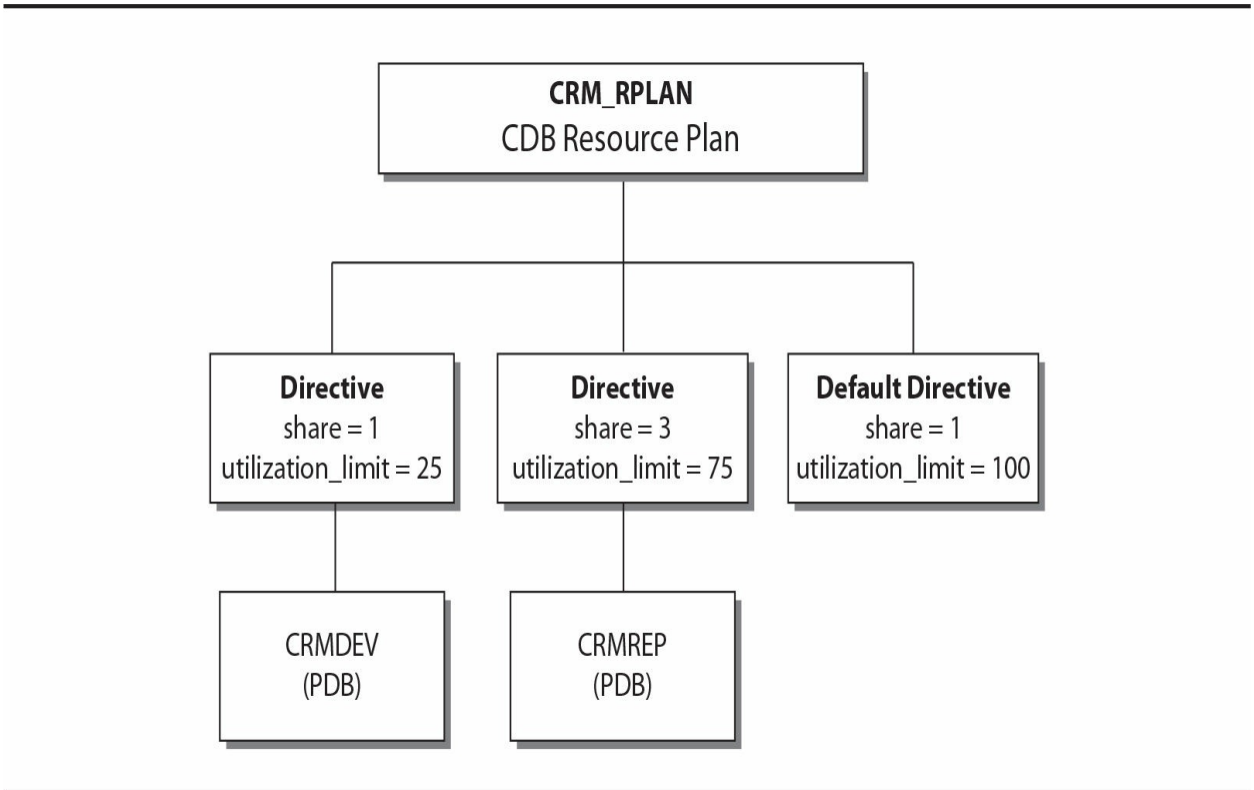


FIGURE 10-9. *Non-CDB instance caging to CDB resource plan*

The following code example can be used to create a resource plan called CRM_RPLAN for container database CDB3 running two PDBs. The code block is executed while connected to the CDB$ROOT:

```
BEGIN
   dbms_resource_manager.create_pending_area;
   dbms_resource_manager.create_cdb_plan(plan =>'CRM_RPLAN');
   dbms_resource_manager.create_cdb_plan_directive(plan => 'CRM_RPLAN'
         , comment => 'CRMDEV Plan Directive'
         , pluggable_database => 'CRMDEV'
         , shares => 1
         , utilization_limit => 25);
   dbms_resource_manager.create_cdb_plan_directive(plan => 'CRM_RPLAN'
         , comment => 'CRMREP Plan Directive'
         , pluggable_database => 'CRMREP'
         , shares => 3
         , utilization_limit => 75);
   dbms_resource_manager.submit_pending_area;
 END;
```

Monitoring Resource Manager

Monitoring Resource Manager can be challenging. Luckily, there are a number of ways to extract information with regard to how resources are used and distributed in the database. The following options are available to help you in this area:

- Oracle Enterprise Manager – Database Express 12*c* (EM Database Express 12*c*)
- Oracle Enterprise Manager Cloud Control (EM Cloud Control)
- Oracle SQL Developer
- SQL commands via SQL*Plus

Yes, SQL commands via SQL*Plus is included. At the end of the day, the other tools are running SQL statements to extract the required information

required and then display it in a nicely formatted manner. Running SQL scripts might be a bit daunting for some, but once you build up a number of monitoring scripts to review your implementation of Resource Manager, you will find it is a quick and easy way to get an overview of the current status.

Viewing the Resource Plan and Plan Directives

The fastest ways to get information about the CDB resource plan and plan directives are to use the following views while connected to the CDB$ROOT:

- `DBA_CDB_RSRC_PLANS`
- `DBA_CDB_RSRC_PLAN_DIRECTIVES`

It is possible to view the information from EM Cloud Control or even EM Database Express, as shown in Figure 10-10, for example.

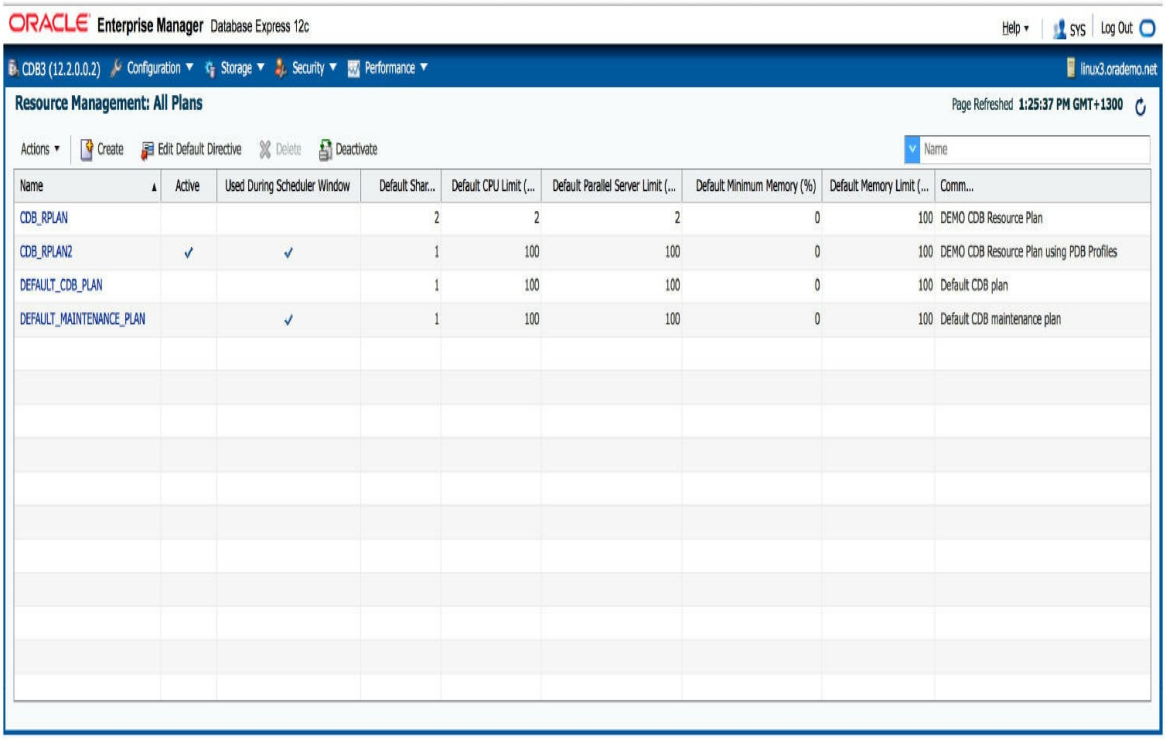


FIGURE 10-10. *Monitoring using Oracle EM Database Express*

Monitoring PDBs Managed by Resource Manager

For monitoring, nothing beats a good graphical representation of statistics. Using EM Cloud Control or EM Database Express is highly recommended. Figure 10-11 is a perfect example of how EM Database Express can be used to get an overview of resource usage quickly.

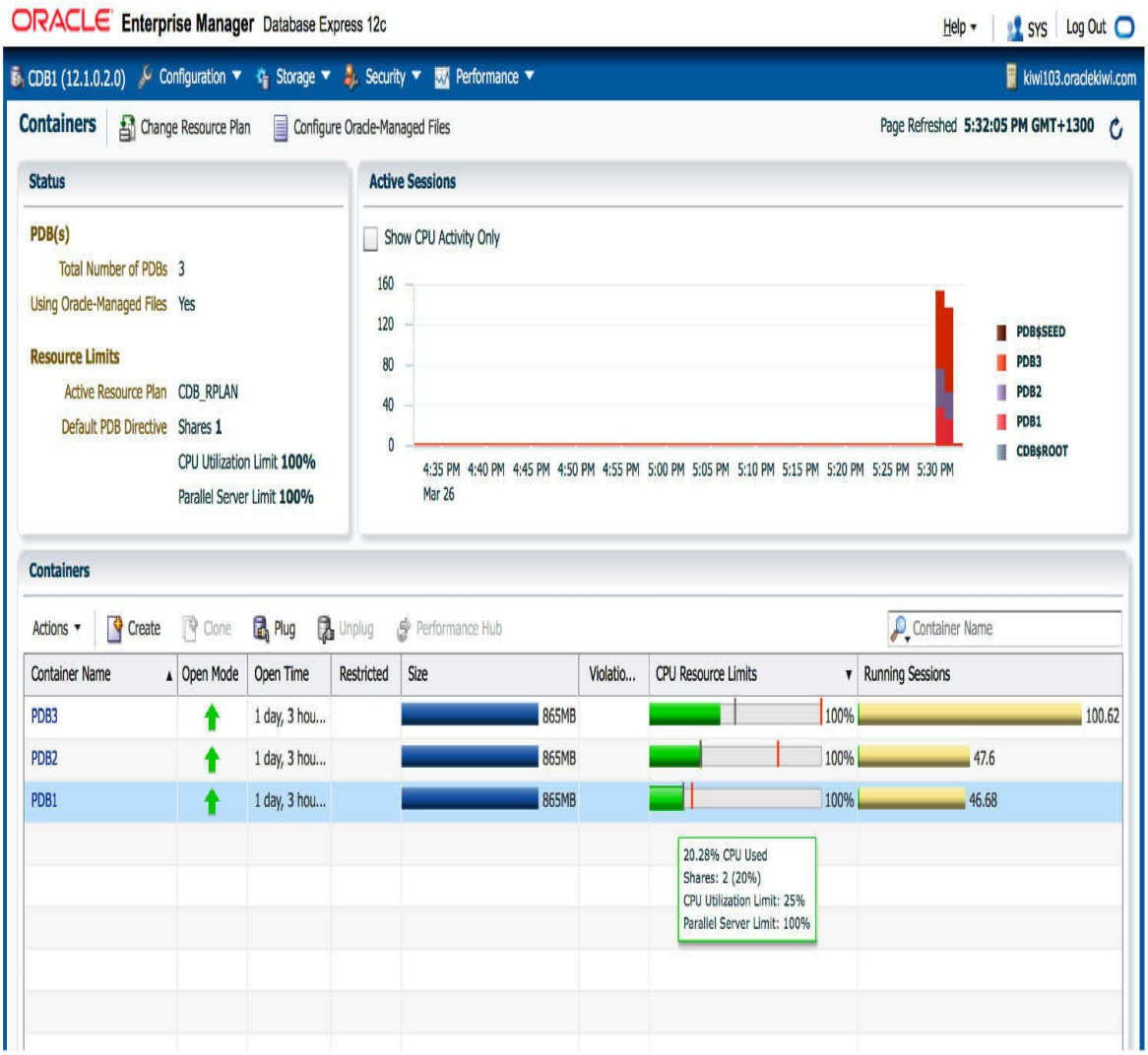


FIGURE 10-11. *Monitoring using Oracle EM Database Express 12.1*

It is also possible to make use of the dynamic performance views to monitor the results of a Resource Manager implementation. The following views can be used:

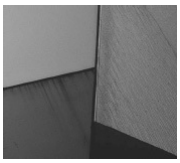
`V$RSRCPDBMETRIC` Current statistics for the last minute are shown per PDB. One record is displayed per PDB, including one row for the CDB$ROOT. Note that this view is available only from Oracle Database 12*c* Release 2.

V$RSRCPDBMETRIC_HISTORY Statistics for the last 60 minutes are displayed per PDB (including the CDB$ROOT).

Summary

At the core of multitenant are consolidation, easy provisioning, and more effective use of resources. Without Resource Manager, we might end up with an implementation where resources are not distributed evenly or available where needed, where certain applications experience bad performance because of resource starvation. It might sound easy, but careful planning and ongoing monitoring is highly recommended. Workloads change, and it is the job of the DBA to ensure Resource Manager is configured and updated when and as needed. It is recommended to start with a less complex configuration, monitor it over a period of time, and adjust it as needed. Having multiple resource plans on a CDB and PDB level is acceptable, and with the use of scheduled windows, you can enable and disable these plans as required.

This brings us to other interesting subjects: Data Guard, the sharing of data between PDBs, and then, last but not least, moving of data using logical replication. These might seem unrelated to resource management, but if you look a bit closer, you will realize that a standby database can be used to offload certain application operations, especially when Active Data Guard is in play. The end result may free up valuable resources on your primary system and help you get more value out of your disaster recovery site.



CHAPTER 11

Data Guard

An old adage says that a DBA can get many things wrong but must never fail in one single competency: database recovery. It holds that if there is a way to restore the database to the time before the error happened, then any error is potentially reversible. This saying was coined many years ago, and the boom in overall database sizes and increased dependence of modern organizations on databases as foundational to their business operations have made it all the more true!

Over time, one additional requirement has emerged—we need *fast* database recovery. Restoring a multi-terabyte database from tapes is not something the business wants to wait for. This does not mean, however, that the good-old tape backup is a relic of a bygone era; in fact, it is still a necessary part of a sound backup plan, and we covered it in detail in Chapter 7. It just happens to be the last line of defense, reserved for massive natural disasters. In the milieu of day-to-day operations, littered with faulty disks, human errors, server crashes, and building fires, we need a faster way to recover.

Since 8*i*, Oracle Database has supported the standby database functionality; improved and enhanced through the versions, it eventually evolved into the Data Guard feature. Conceptually speaking, however, a physical standby database is merely an exact copy of the production, or primary, database. This copy, or standby, is kept constantly in recovery mode, applying archived redo from the primary, in the same way that a media recovery would happen.

This basic level of functionality was introduced in Oracle Database 8*i*, and if the database is not running on Enterprise Edition, this is all that is

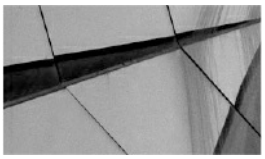
available. It is a proven and solid foundation but lacks facilities such as automatic redo transfer and adequate monitoring. Fortunately, third-party tools are available to help automate this process and manage these environments, such as Dbvisit Standby.

With an Oracle Database Enterprise Edition, the full Data Guard functionality is available, which includes, at the very least, automatic redo transfer, the ability to apply online redo as it is shipped, the availability of the Data Guard broker and Data Guard command line interface (DGMGRL), and the administration options included in Cloud Control and Enterprise Manager Express. In this chapter, we will cover the Enterprise Edition functionality; after all, only EE supports the multitenant option.

Data Guard supports multitenant databases, and most of the management is identical to that of a non-container database (non-CDB). In this chapter, we walk through a simple setup of a physical standby database and will then look at some of the differences that multitenant brings.

Active Data Guard Option

With the standard physical standby functionality, the standby database runs in recovery mode and is unavailable for any end user operations. However, the Active Data Guard option enables the database to be opened read-only while the recovery occurs, simultaneously, in the background, enabling users to utilize the hardware and licenses dedicated to the standby database to support various loads such as reporting or local caches.



CAUTION

There is no direct initialization parameter or setting that we can use to enable or disable Active Data Guard. It is always available to be used, irrespective of whether the server is licensed for it or not, and the Oracle Database is overly "enthusiastic" about enabling this option. So it falls to us, the DBA, as our responsibility to mitigate against enabling it unintentionally and never open the database when redo apply is active; this includes not starting the database

with `startup`, but always with `startup mount` instead. We might well be wise to go the extra mile and create an automated task to verify that the database is not opened with redo apply active.

We can, however, consider using the undocumented (and thus unsupported parameter) `_query_on_physical=no` to prevent from using the Active Data Guard option. Our hope is that Oracle will eventually make the parameter documented and supported.

All of this is true for any Oracle 11*g* or 12*c* database, whether multitenant is in use or not. We'll see later in this chapter the impact of Active Data Guard on multitenant when we discuss the various scenarios for creating, copying, and moving pluggable databases (PDBs) and how they affect the standby.

Creating a Physical Standby

As mentioned, a standby database is essentially an identical binary copy of the source database, with some configuration differences and perhaps only a subset of the source data. As such, it's essentially a backup, so using backups and backup tools to create it is an obvious choice. From the command line, this means Recovery Manager (RMAN), and in terms of GUI options, it means Oracle's Cloud Control, which, incidentally, also uses RMAN in the background.

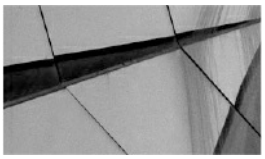
Over the course of the database versions, this process has been streamlined and enhanced, and nowadays very few steps are necessary. Oracle can help us generate the required configuration files, and it can create the standby both from an existing backup and from a running database.

After the standby is created, the next steps are to set up the Data Guard configuration, set the desired level of protection, and monitor the configuration. Again, both command line (DGMGRL) and Cloud Control options are available for this purpose.

In this chapter, we go through a basic scenario, focusing on differences brought by the multitenant database.

Duplicate with RMAN

As creating a standby database is a fundamental step in a Data Guard setup, it is obviously described in depth in the *Data Guard Concepts and Administration* part of the Oracle Database documentation. However, for whatever reason, for a long time this information has been split between two locations: one regular chapter that describes the process in quite vague terms, and an appendix (Appendix E) detailing the RMAN-focused steps. In both places, Data Guard Broker is not mentioned, but it is covered in another book in the Oracle Database documentation, *Data Guard Broker*. This is quite unfortunate, because the easiest way to create a nicely working Data Guard configuration is to combine these three pieces. Multitenant brings yet another element and an additional set of documentation to consider, but we hope this chapter will serve to orient the reader.



NOTE

For the examples listed in this chapter, the current database is called USPROD and the standby is NZPROD. The source database is already in ARCHIVELOG mode, force logging is set, and `STANDBY_FILE_MANAGEMENT` *is set to AUTO. The servers are two distinct machines, and the databases will be placed in the same locations on both of them.*

Set Up Static Network Services

The first step is to configure static service definitions for both the primary and standby container databases. There is a single purpose in this: to enable the Data Guard console to start the database instances, as necessary, during switchover and failover operations.

We edit the listener.ora on the servers and add the static service definition. Note the addition of *_DGMGRL* to the database unique name, so it becomes, for example, USPROD_DGMGRL and NZPROD_DGMGRL.

```
SID_LIST_LISTENER=(SID_LIST=(SID_DESC=(SID_NAME=sid_name) (GLOBAL_
DBNAME=db_unique_name_DGMGRL.db_domain)
(ORACLE_HOME=oracle_home)))
```

Although not strictly necessary, but beneficial for administration purposes, we also add an entry to /etc/oratab for the databases to be created:

```
NZPROD:/u01/app/oracle/product/12.2/dbhome_1:N
```

Back Up the Source Database

If we want to create the standby database from backup, we must first back up the database or ensure that a recent backup is available. Note that the backup itself must be accessible from the standby server. How this is done depends on the environment, and often the tape library will actually do this for us, or we can mount the disks/SAN volumes manually as necessary.

The last option is, of course, to copy the backup files to the target server. Although this option often doesn't make much sense in a production environment, because we can use duplication from an active primary directly, it's a good idea during testing and learning; any failures and retries during the process will, therefore, not force the transfer of the entire database again and again over the network.

Set Up the Network

During the setup and management of the configuration, the Data Guard Broker establishes a connection to the primary and standby databases. We therefore need to set up USPROD and NZPROD connection strings in the tnsnames.ora on both servers. Also, when we do the initial duplication using RMAN, it needs to connect to the not-yet-created NZPROD database, so a connection to a statically defined service is required. And because we might need such connections in the future, again for either of these possibilities, we add connection strings for USPROD_DGMGRL and NZPROD_DGMGRL services to tnsnames.ora, too, on both nodes.

Copy Password File and Create a Temporary

Parameter File

The standby needs an exact copy of the primary password file, so we need to be aware that a newly created file will not work, even if the SYS password is identical. Note also that the name is based on DB_UNIQUE_NAME; thus, the new name of this file is orapwNZPROD.

For a backup-based process, we manually copy the password file from the primary to the standby server. Although an active database–based duplication process will perform this copy automatically, it still needs a password file present to log in in the first place, so we still must provide it.

Next we need to create a temporary parameter file for NZPROD. The duplication will copy over the correct primary (binary) SPFILE, but for now, we just need the minimum of parameters to start up an instance. The only required entry is the DB_NAME:

```
DB_NAME=USPROD
```

We could also add DB_DOMAIN if applicable for our environment. If we are creating the standby on the same server as the primary—not a good use case for a production database, but perhaps convenient for testing—we need to specify DB_UNIQUE_NAME. This ensures that the new instance name does not clash with the source, before RMAN has the chance to set DB_UNIQUE_NAME in the SPFILE.

There are, in fact, two paths we can take with respect to generating the SPFILE for the new database. The first is to create the PFILE from the source database SPFILE (using the `CREATE PFILE` command) and then manually change the file as necessary. The second is to have RMAN create a copy of the SPFILE automatically, specifying the necessary changes in the duplicate command.

The first option is more manual and time-consuming; however, during the process we actually read the parameter file and have a chance to review the settings. We can also use the parameter file to start the instance before the duplication, and thus any incorrect paths or options are brought to our attention immediately and can be amended.

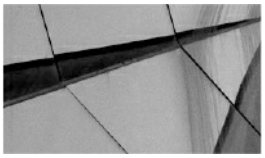
On the other hand, we can let RMAN do the magic for us, but it is very possible that some parameter will turn round and bite us, so to speak, during

the duplication process. This means that we will have to clean up any files already created and restart the duplication.

One such example is `LOCAL_LISTENER`; this parameter might be set to a value other than the `LISTENER` default, and in that case, we must either create such a listener on the target and update tnsnames.ora as well, or adjust this to the correct value in the RMAN duplicate command.

Also note that the target database will want some directories to be precreated for the instance startup and for the duplication to succeed. Usually this is the audit file directory, and, for non-OMF installations, also the paths to the datafiles and online REDO:

```
mkdir -p /u01/app/oracle/admin/USPROD/adump
mkdir -p /u01/app/oracle/data/USPROD
mkdir -p /u01/app/oracle/data/FRA/USPROD/
mkdir /u01/app/oracle/data/USPROD/pdbseed
mkdir /u01/app/oracle/data/USPROD/PDB1
```



NOTE

In a multitenant database, the datafiles are scattered among multiple subdirectories, and we have to create all of them on the target.

Run the Duplicate Process

Let’s run the duplication now. As mentioned, we are not changing the file locations, so there is no need to specify `DB_FILE_NAME_CONVERT` and `PARAMETER_VALUE_CONVERT` values, but we do have to specify `NOFILENAMECHECK`.

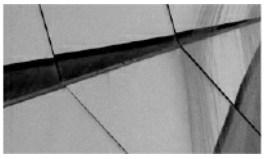
```
rman target=sys/oracle@usprod_dgmgrl
auxiliary=sys/oracle@nzprod_dgmgrl
connected to target database: USPROD (DBID=3345163260)
connected to auxiliary database: USPROD (not mounted)
RMAN> DUPLICATE TARGET DATABASE
      FOR STANDBY
      FROM ACTIVE DATABASE
      DORECOVER
      SPFILE
      SET "db_unique_name"="NZPROD"
      NOFILENAMECHECK;
```

This example uses the active database–based duplication functionality, in which RMAN essentially performs an image `COPY` of the database datafiles. Since the release of Oracle 12*c*, we can request that RMAN take a backup of the source database during the `DUPLICATE` command and make use of these backups.

And, of course, the least sophisticated means of getting started is simply to use backups available to RMAN and created earlier. The major advantage here is minimal impact on the primary database, because the new database's files are created from the backups. As for syntax, we simply omit the `FROM ACTIVE DATABASE` clause.

Choosing a Subset of the Source Database

Ever since the standby database feature was made available, there has been an option to have only a subset of the source database protected by the standby. The idea is very simple: we can offline any file we don't want to have at the target and the recovery will ignore this.



NOTE

Subsetting is a feature used for specific use cases only. With a regular standby, we want to protect the primary from disasters, which implies that we want a full copy in this secondary location. However, sometimes a PDB may only be temporary (similar to a

nologging table in a data load process), or perhaps we are just using snapshot standby for testing, so we don't need all of the PDBs for the tests. In such cases, the subsetting option makes good sense.

Oracle Database 12*c* Multitenant provides a new syntax that achieves similar results to tablespace offline, but with a superior usage. First of all, the new syntax works at the PDB level, and, second, it has dedicated syntax for recovering from such subsetting, should we later decide we actually want the PDBs.

Unfortunately, one major aspect remains unchanged: although RMAN has the `[SKIP] TABLESPACE` clause (and now with 12*c* it also has a `[SKIP] PLUGGABLE DATABASE`, too) these are not valid for `DUPLICATE FOR STANDBY`, though they are valid for the other `DUPLICATE` option that creates an independent database copy. According to Oracle (MOS note 1174944.1), this is an intentional limitation, because "a physical standby must match the primary."

In other words, Oracle forces us to create the standby as a full copy, and only then can we remove the unnecessary pieces:

```
SQL> alter pluggable database PDB3 disable recovery;

Pluggable database altered.
```

Following this command, the PDB is still known to the target database, but its recovery flag is set to disabled, meaning that no redo is applied.

```
SQL> select con_id, name, open_mode, recovery_status from v$pdbs;

CON_ID NAME            OPEN_MODE  RECOVERY
------ --------------- ---------- --------
     2 PDB$SEED         MOUNTED    ENABLED
     3 PDB1             MOUNTED    ENABLED
     4 PDB2             MOUNTED    ENABLED
     5 PDB3             MOUNTED    DISABLED
     6 PDB4             MOUNTED    ENABLED
```

Thanks to all this metadata still present at the standby, it's easy to add the database back again. Later in this chapter, in the section "Enabling the PDB Recovery," you'll learn more about this.

Start Data Guard Broker Processes and Set Up the Configuration

Let's now quickly go over the steps necessary to finish the standby creation. The first step in properly establishing a Data Guard environment is to configure the database to run the DMON processes that act as background agents for the Data Guard configuration. To do so, on each of the databases, we set the parameter as a common user:

```
alter system set dg_broker_start=TRUE;

System altered.
```

Now with the brokers running, we can actually create a configuration specifying the primary and the standby databases.

As a rule, when using DGMGRL, we always connect using a connection string, never locally. Although most operations work fine with a local connection (that is, by relying on ORACLE_SID of the CDB), switchovers and failovers don't. Note also that DGMGRL converts all identifiers to lowercase by default, but we can use double quotes to retain case. However, we would then have to double-quote them in all other places, as well.

```
dgmgrl sys/oracle@USPROD
DGMRGL> create configuration USNZ as
primary database is USPROD
connect identifier is USPROD;

Configuration "usnz" created with primary database "usprod"

DGMGRL> show configuration
Configuration - usnz

  Protection Mode: MaxPerformance
  Databases:
    usprod - Primary database

Fast-Start Failover: DISABLED

Configuration Status:
DISABLED
```

Next we add the standby database:

```
DGMGRL> add database NZPROD as
connect identifier is NZPROD;

Database "nzprod" added

DGMGRL> show configuration
Configuration - usnz
  Protection Mode: MaxPerformance
  Databases:
    usprod - Primary database
      nzprod  - Physical standby database

Fast-Start Failover: DISABLED

Configuration Status:
DISABLED
```

Now comes the moment of truth: enabling the configuration. In this process, Oracle sets the log shipping parameters and a few others, so this step is anything but trivial and can fail for many different reasons. In such cases, the error description is usually helpful, and we can also use the oerr utility or look it up in the Error Messages documentation book.

```
DGMGRL> enable configuration;

Enabled.

DGMGRL> show configuration

Configuration - usnz

  Protection Mode: MaxPerformance
  Members:
  usprod - Primary database
    Warning: ORA-16789: standby redo logs configured incorrectly

    nzprod - Physical standby database
      Warning: ORA-16809: multiple warnings detected for the member

Fast-Start Failover: DISABLED

Configuration Status:
WARNING   (status updated 8 seconds ago)
```

Here we can see that NZPROD has more than one warning, so let's review the list by using `show database`:

```
DGMGRL> show database NZPROD

Database - nzprod

  Role:               PHYSICAL STANDBY
  Intended State:     APPLY-ON
  Transport Lag:      0 seconds (computed 49 seconds ago)
  Apply Lag:          (unknown)
  Average Apply Rate: (unknown)
  Real Time Query:    OFF
  Instance(s):
    NZPROD

  Database Warning(s):
    ORA-16854: apply lag could not be determined
    ORA-16857: member disconnected from redo source for longer than
specified threshold
    ORA-16789: standby redo logs configured incorrectly

Database Status:
WARNING
```

To remove the ORA-16854 and ORA-16857 warnings, we can simply issue a log switch on the primary to force a redo log to be shipped. This will update the lag and satisfy Data Guard, and, of course, once we create the standby redo logs in the next section, the changes will be applied in real time.

```
SQL> alter system switch logfile;

System altered.

DGMGRL> show configuration

Configuration - usnz

  Protection Mode: MaxPerformance
  Members:
  usprod - Primary database
    Warning: ORA-16789: standby redo logs configured incorrectly

    nzprod - Physical standby database
      Warning: ORA-16789: standby redo logs configured incorrectly

Fast-Start Failover: DISABLED

Configuration Status:
WARNING   (status updated 8 seconds ago)
```

Verify the Configuration and Fill In the Missing Pieces

The next step is to fill in the missing pieces—enabling flashback and adding standby redo logs. We can do this immediately, but let's have a look at a new diagnostic command first, which outlines the steps we need to perform.

In Oracle 12*c*, the `VALIDATE DATABASE` command has been introduced in DGMGRL. Upon execution of the command, Oracle checks various settings, along with the status of the database, and prints a comprehensive summary. It's useful during an initial setup, to remind us of steps we still have to do, as well as during the course of normal processing.

```
DGMGRL> validate database USPROD

  Database Role:    Primary database

  Ready for Switchover:  Yes

  Flashback Database Status:
    usprod:  Off

DGMGRL> validate database NZPROD

  Database Role:     Physical standby database
  Primary Database:  usprod

  Ready for Switchover:  No
  Ready for Failover:    Yes (Primary Running)

  Flashback Database Status:
    usprod:  Off
    nzprod:  Off

…

  Log Files Cleared:
    usprod Standby Redo Log Files:  Cleared
    nzprod Online Redo Log Files:   Cleared
    nzprod Standby Redo Log Files:  Not Available

  Current Log File Groups Configuration:
    Thread #  Online Redo Log Groups  Standby Redo Log Groups Status
              (usprod)                (nzprod)
    1         3                       0
Insufficient SRLs
    Warning: standby redo logs not configured for thread 1 on nzprod

  Future Log File Groups Configuration:
    Thread #  Online Redo Log Groups  Standby Redo Log Groups Status
              (nzprod)                (usprod)
    1         3                       0
Insufficient SRLs
    Warning: standby redo logs not configured for thread 1 on usprod
```

There is still room for improvement of the VALIDATE command, and perhaps this will come in future versions; nevertheless, it's a good tool. From its output, we can see it complains about standby redo logs and Flashback Database not being enabled.

So let's now create the standby redo logs. The steps are simple: use the same size as for the redo logs and create one more than the number of online redo log groups.

```
SQL> select distinct bytes from v$log;

     BYTES
----------
  52428800
```

On the primary, add the standby logfiles and enable flashback. The naïve approach is to add the standby redo logs with the minimal syntax required on both source and target:

```
SQL> alter database add standby logfile size 52428800; -- four times
```

Unfortunately, in using the simple syntax for adding standby redo, the logs were created but unassigned to any thread. This is not such an issue for the standby itself, because the logs will be assigned to threads as required; however, the VALIDATE command ignores such unassigned redo logs and complains that there are insufficient logs for the thread(s).

The SQL command for creating the standby redo logs enables us to specify the actual thread to assign. Using this, the assignment is preset at the time of standby log creation, fulfilling the VALIDATE command criteria and, at the same time, preventing any possible surprises should the autoallocation go awry.

```
SQL> alter database add standby logfile thread 1 group 4 size 52428800;
SQL> alter database add standby logfile thread 1 group 5 size 52428800;
SQL> alter database add standby logfile thread 1 group 6 size 52428800;
SQL> alter database add standby logfile thread 1 group 7 size 52428800;
```

the `VALIDATE` command would be satisfied with the standby redo log allocation:

```
  Current Log File Groups Configuration:
    Thread #  Online Redo Log Groups  Standby Redo Log Groups Status
              (usprod)                (nzprod)
    1         3                       4
Sufficient SRLs

  Future Log File Groups Configuration:
    Thread #  Online Redo Log Groups  Standby Redo Log Groups Status
              (nzprod)                (usprod)
    1         3                       4
Sufficient SRLs
```

We should also enable Flashback Database (see Chapter 8), so we don't need to rebuild the entire previous primary database on a failover.

Finally, we can verify the validity of the configuration with the following command, although we may need to wait a minute or so for the broker configuration to be updated:

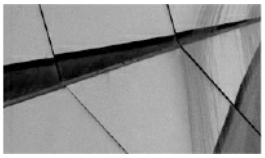
```
DGMGRL> show configuration

Configuration - usnz

  Protection Mode: MaxPerformance
  Members:
  usprod - Primary database
    nzprod - Physical standby database

Fast-Start Failover: DISABLED

Configuration Status:
SUCCESS   (status updated 2 seconds ago)
```

We can also use a new option for the `SHOW CONFIGURATION` command to understand what the configuration would look like if we were to switch the database roles:

```
DGMGRL> show configuration when primary is nzprod

Configuration when nzprod is primary - usnz

Members:
  nzprod - Primary database
    usprod - Physical standby database
```

Test the Configuration

The last, and perhaps most important, step is to verify that the standby database can be used in case of a disaster or to facilitate a planned maintenance window.



NOTE

You should always start your experimentation with and learning of

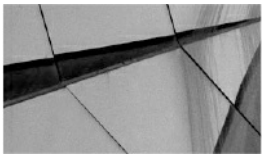
this functionality on a test database. Disaster recovery is a business-critical function and you need to become very familiar with all the tasks it entails, and the administration of these environments, before you depend on it for the protection of a production database.

A basic test simply consists of doing a switchover back and forth. This verifies that the redo logs are being shipped and applied, and that DGMGRL can actually connect to the databases even if they are down, using the _DGMGRL static connection strings we created earlier.

If this is a nonproduction database, we can perform the test immediately; if it's a production database, we should schedule a maintenance window to run the test. This is important as a core foundation of a thorough, planned backup, recovery, and DR strategy is to verify periodically that backups can be restored, and that applications and databases can be switched over and successfully run, from the backup data center.

In our example, we want to verify in both directions, keeping the primary in its original location once concluded, so we perform two switchovers:

```
DGMGRL> switchover to nzprod;
Performing switchover NOW, please wait...
Operation requires a connection to database "nzprod"
Connecting ...
Connected to "NZPROD"
Connected as SYSDBA.
New primary database "nzprod" is opening...
Operation requires start up of instance "USPROD" on database "usprod"
Starting instance "USPROD"...
ORACLE instance started.
Database mounted.
Connected to "USPROD"
Switchover succeeded, new primary is "nzprod"

DGMGRL> switchover to usprod;
Performing switchover NOW, please wait...
Operation requires a connection to database "usprod"
Connecting ...
Connected to "USPROD"
Connected as SYSDBA.
New primary database "usprod" is opening...
Operation requires start up of instance "NZPROD" on database "nzprod"
Starting instance "NZPROD"...
ORACLE instance started.
Database mounted.
Connected to "NZPROD"
Switchover succeeded, new primary is "usprod"
```



TIP

To deepen your knowledge and familiarity with Data Guard, we recommend doing more tests, although most of them are likely to be limited to test databases. Give special attention to the multitenant scenarios described in this chapter, which are also new to us and quite often require manual intervention.

Further Configuration

Now it’s time to review additional settings available in Data Guard. We might want to change the protection mode, set up an observer, and configure fast start failover. Or perhaps we want to change the RMAN retention policy to account for the standby when considering archive logs eligible for deletion. These are but a few of the many useful options available to explore. However, these are beyond the scope of this book, so we recommend again reviewing the Data Guard documentation for this material.

Create a Standby with Cloud Control

Creation of the standby using RMAN is a well-tested and proven process, but Enterprise Manager Cloud Control also has powerful capabilities, including a nice step-by-step wizard. We can get to the wizard by choosing the Availability menu (see Figure 11-1), in which both MAA Advisor and Add Standby Database options have links to the wizard.

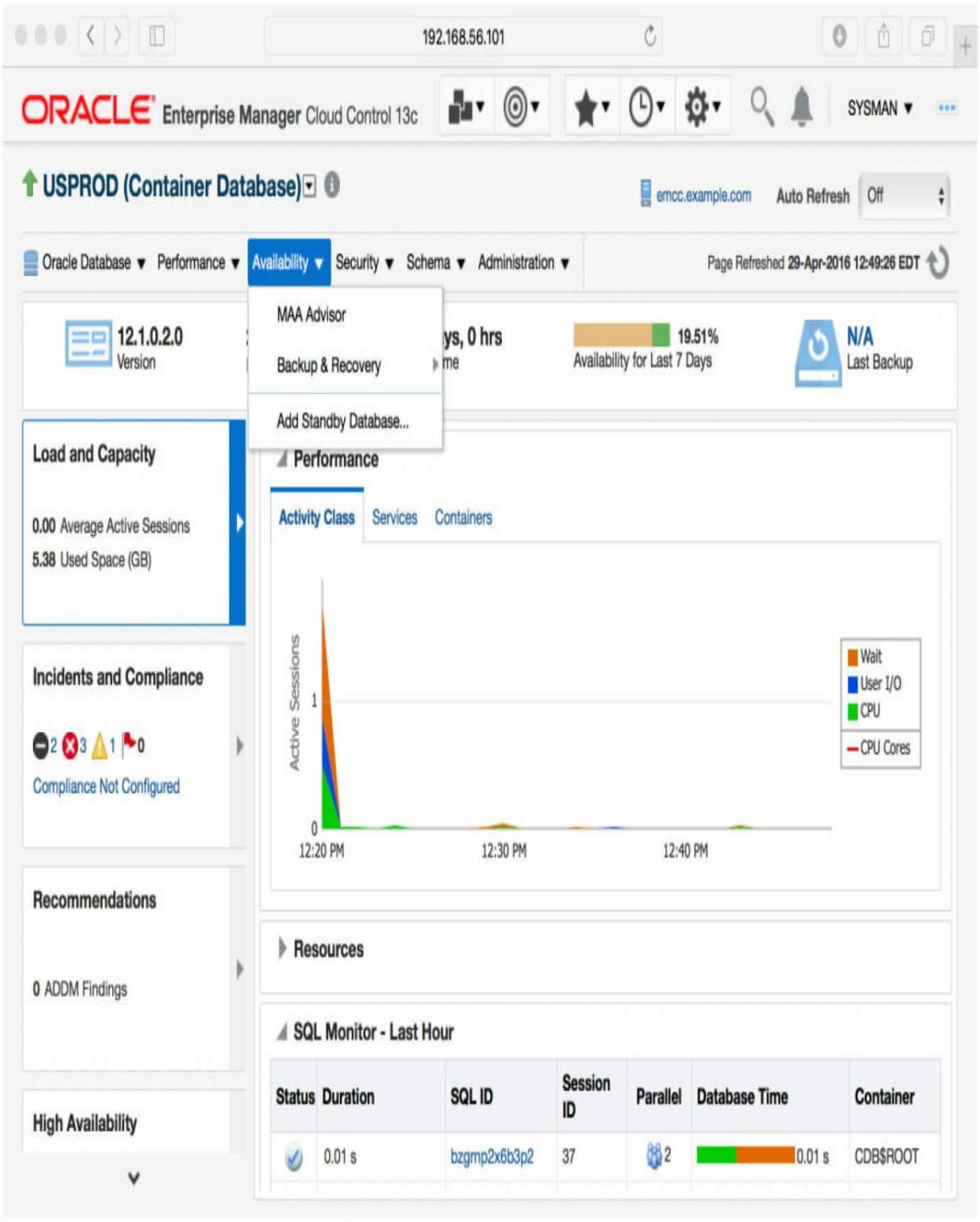


FIGURE 11-1. *The database home page*

The wizard covers both physical and logical standbys. It can also register an existing standby—for example, one that we created manually using the steps we outlined earlier, as shown in Figure 11-2.

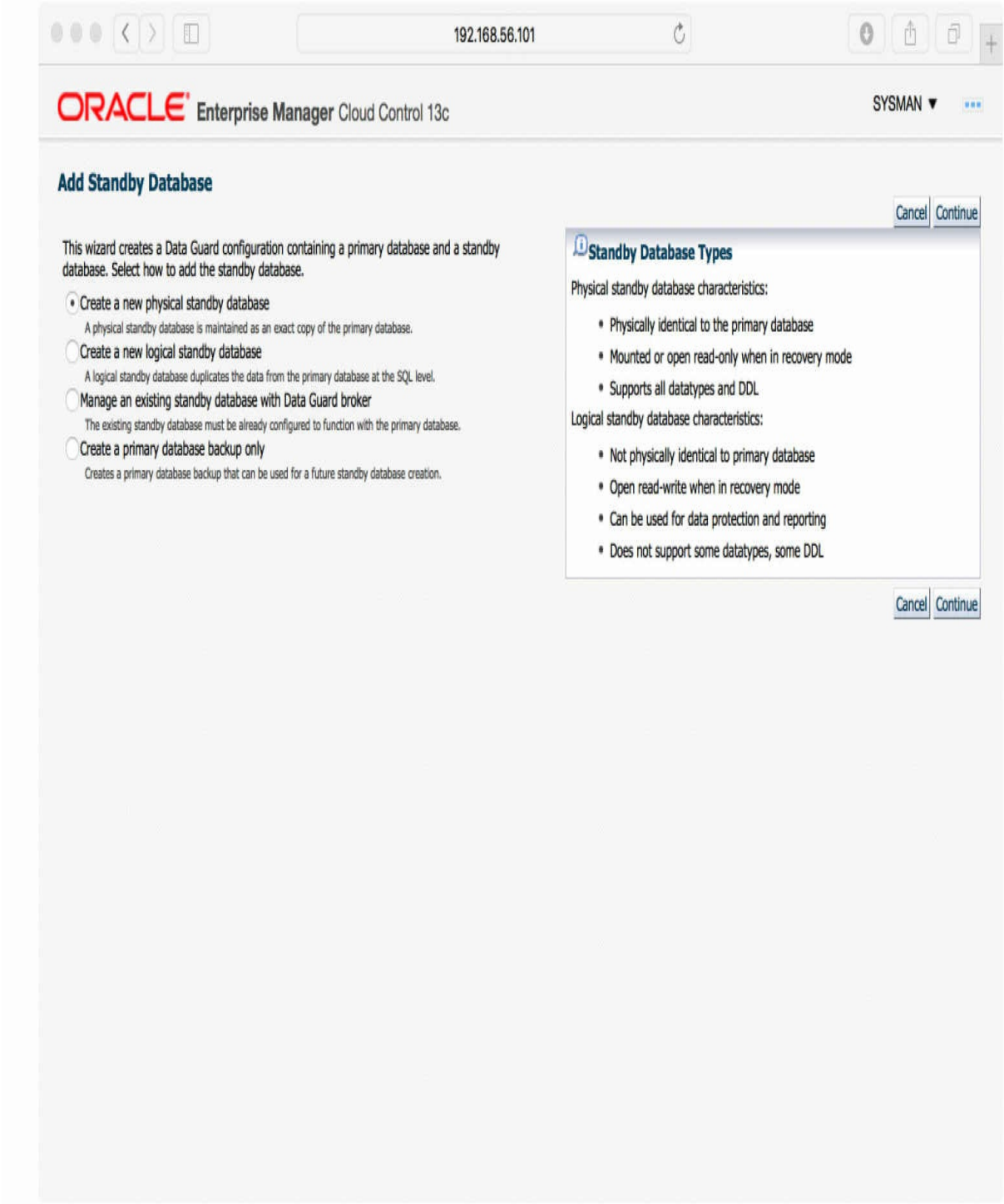


FIGURE 11-2. *The Add Standby Database wizard start page*

A general grievance with Cloud Control is the time it takes EM developers to catch up with the features that the database itself offers. One such example is shown in Figure 11-3, where there is no option for using backup set for the duplication from an active database.

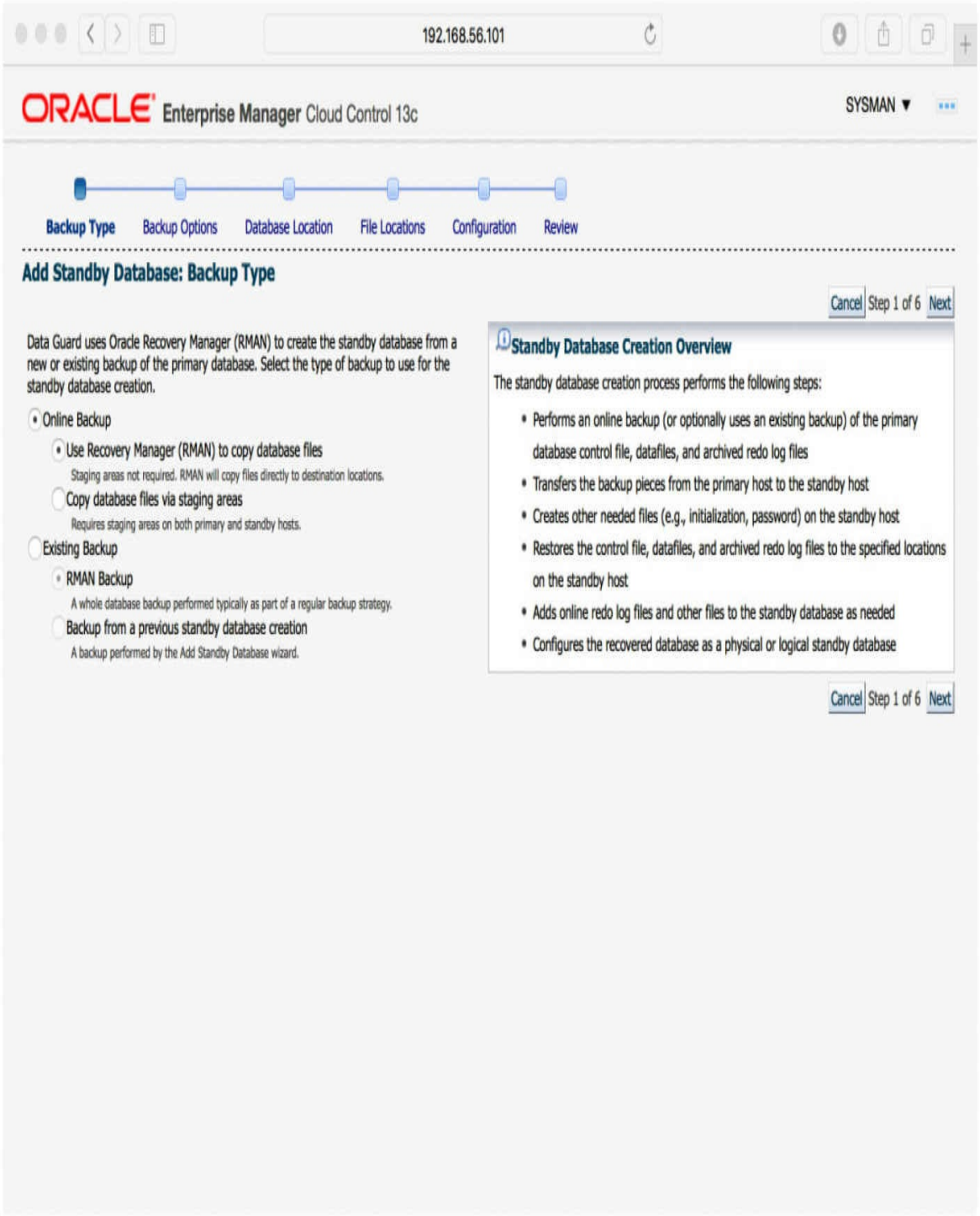


FIGURE 11-3. *Selecting a backup type page*

Figure 11-4 shows various options for the backup, and as we have selected active database duplication, there is really not much to configure. A nice touch is that EM will create the standby redo logs for us.

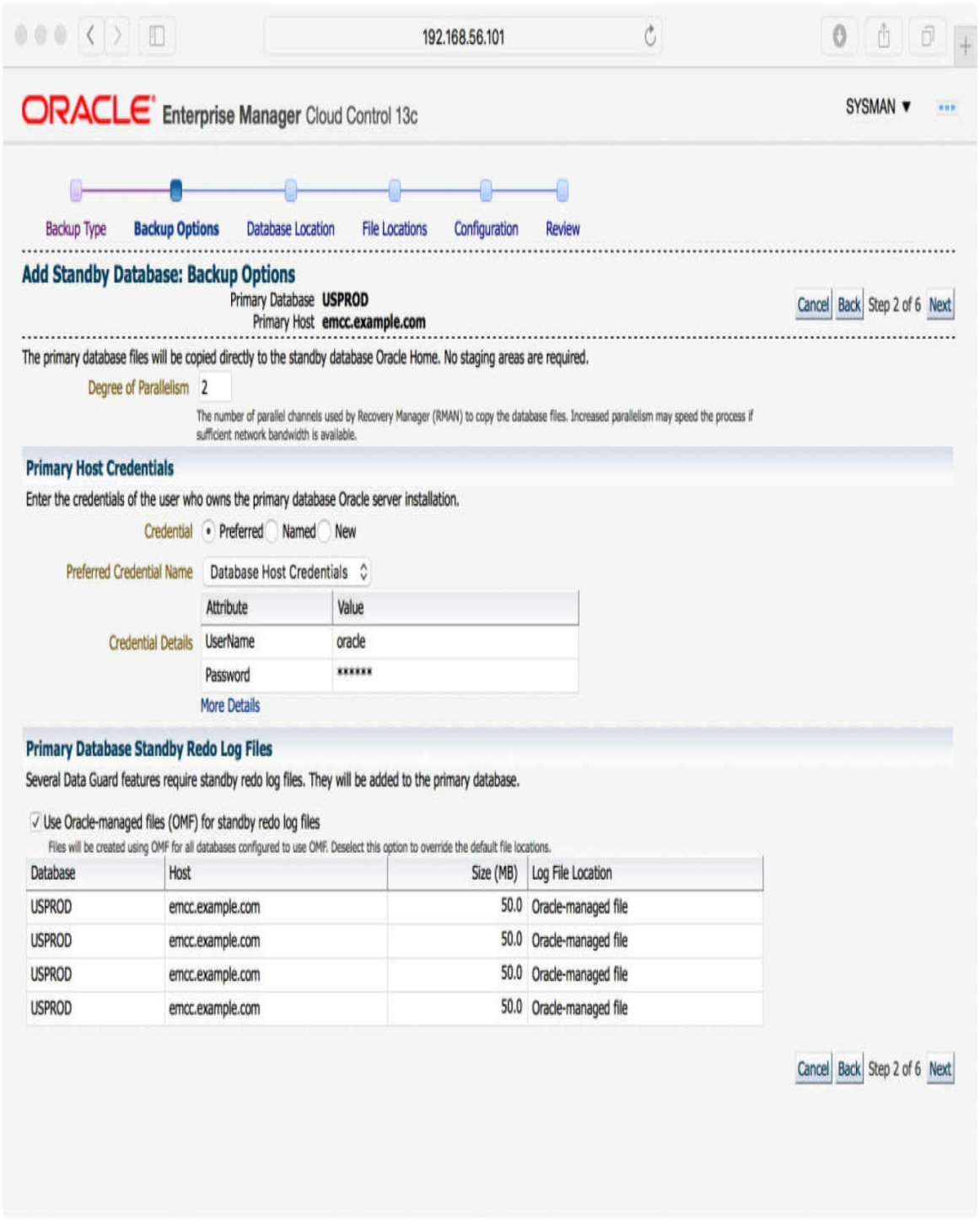


FIGURE 11-4. *Selecting backup options*

In the end (Figure 11-5), a job is created that performs the actual work of creating the standby database and setting up the Data Guard configuration, including the broker. Note that both EM and DGMGRL use the same DG Broker configuration, so it is possible to monitor and manage the configuration using either tool.

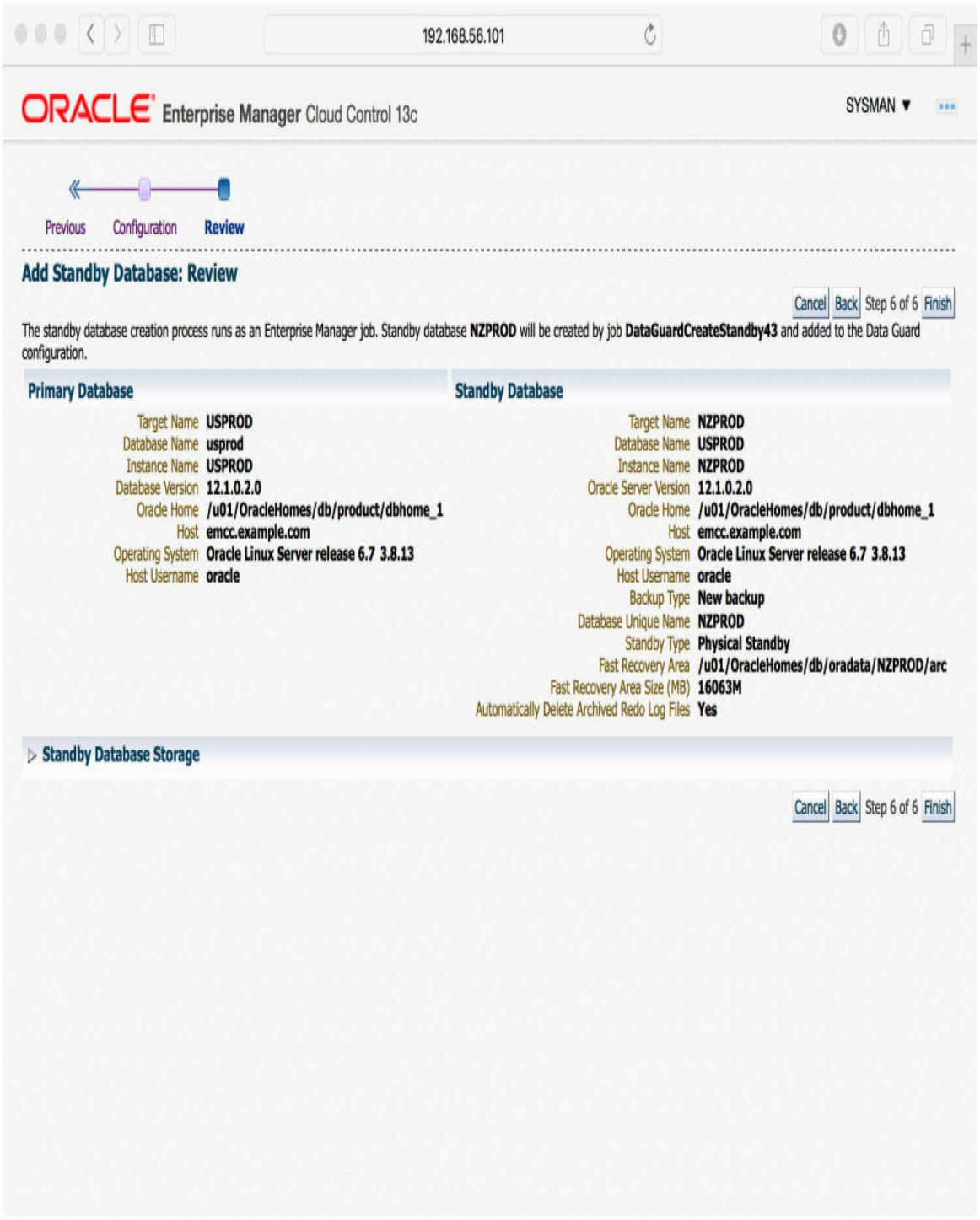


FIGURE 11-5. *Reviewing the job*

After the standby is created, the Availability menu contains new options. The Data Guard Administration page shown in Figure 11-6 displays a overview of the configuration status.

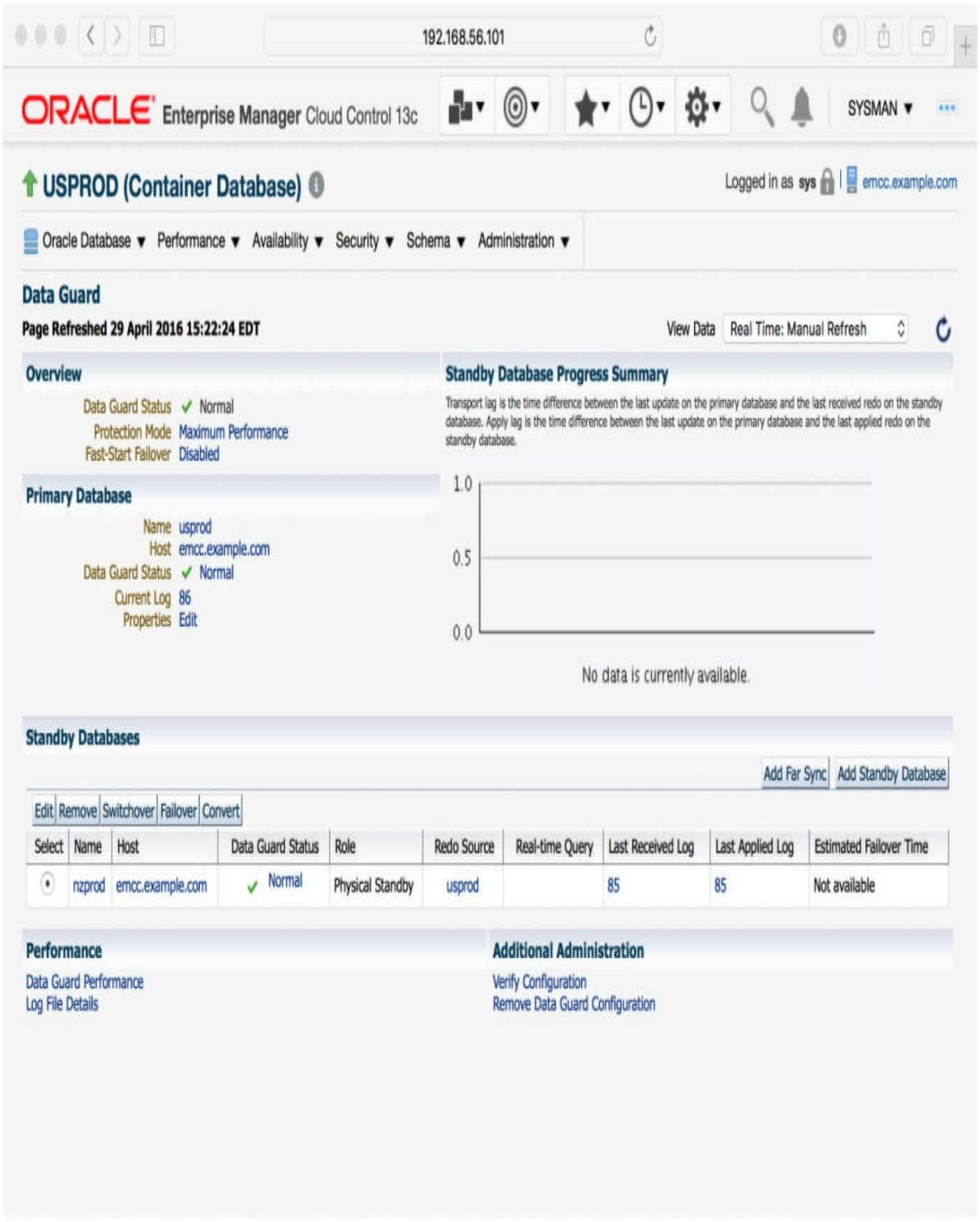


FIGURE 11-6. *The Data Guard status page*

Managing a Physical Standby in a Multitenant Environment

At the basic level, a container database is still a single database, and physical standby works at the whole CDB level. That means that all the components we were used to managing in a non-CDB environment still apply and are done at the root level.

To begin with, this means that both DGMGRL and Cloud Control need to connect to the root container and issue all the commands there. This also includes parameters and options such as protection mode, standby redo logs, transport mode, real-time apply, read-only open mode, observer, and observer thresholds—and many others. It is a similar case with monitoring; the new Oracle Database 12*c* `VALIDATE` command works at the root level, and the lag is displayed for the CDB as whole.

However, the creation, movement, and disposal of PDBs does inject new elements, and issues, into the world of the standby database. New PDBs, and related tablespaces and datafiles, should "appear" on the standby side—but how does this happen, and how can they get there?

Creating a New PDB on the Source

There are multiple ways in which we can create a new PDB on the source, including from scratch, as well as by using the different clone options discussed in Chapter 9.

Note that in all the examples, we use the (thoroughly recommended) parameter, `STANDBY_FILE_MANAGEMENT=AUTO`. Setting this to `MANUAL` would introduce extra steps in the resolutions, meaning that we would first need to set the desired names for all the files involved, and that can get tedious.

Deciding Whether the PDB Should Be on the Standby

When a new PDB is created, we can specify whether we want it to be present on the standby. In version 12.1.0.2, you didn't have much of a choice, because the PDB was available on all standbys or on none, but with Oracle Database 12.2, we can now specify the standbys by name.

```
create pluggable database pdb7 as clone using
'/home/oracle/pdbunplug.xml' standbys=none;
create pluggable database pdb7 as clone using
'/home/oracle/pdbunplug.xml' standbys=(NZPROD);
```

This `standbys` clause is valid for all the `create pluggable database` varieties—from seed, plug-in, and clone.

From Seed

The basic database creation, or creation of a fresh, empty PDB or application in an application container, is from the seed PDB or application container seed PDB. In this case, the standby database will create the PDB, too, as it has the seed readily available.

Local Clone

A local clone, or a clone from the same PDB, copies files from an existing PDB into a new one that is part of the same CDB. The standby databases can perform a similar operation; however, this feature requires Active Data Guard to be enabled at the time.

If Active Data Guard is not in use, the standby will stop applying the redo and wait for a resolution:

```
ALTER DATABASE RECOVER MANAGED STANDBY DATABASE
*
ERROR at line 1:
ORA-00283: recovery session canceled due to errors
ORA-01274: cannot add data file that was originally created as
'/u01/oracle/data/USPROD/PDB5/system01.dbf'
```

At this time, we have to decide whether or not to include the database in the standby. If not, we can simply issue the following:

```
alter pluggable database PDB5 disable recovery;

Pluggable database altered.
```

Of course, if we don't want to have the PDB at the standby, its easier to specify that directly in the statement creating the PDB, as shown in the previous section. However, if we decide that we want to include the PDB, we need to provide the missing files. Provided that we still have available a consistent version of the files (the clone was from a closed PDB and the PDB is still closed), it is sufficient to copy them to the expected path and restart the recovery (that is, `alter database recover managed standby database disconnect from session`). For other cases, see the section "Enabling the PDB Recovery" later in this chapter.

Note that if working with a clone from a closed PDB, and the filenames on standby are known in advance (for example, they don't use OMF, which includes GUID, which is unknown before the clone happens), then the easiest way is to copy the files beforehand. Doing so means that the redo apply won't even have to stop.

Remote Clone

The remote clone option is, in fact, very similar to a local one. In this case, Active Data Guard does not have access to the source files and thus it cannot perform the copy automatically. All the other options are valid here, though, including both skipping the PDB as well as providing the files to the standby.

Plug-in

For a plug-in operation, we have the files on hand before the operation, so it is easy to copy them to the standby and place them in the correct location. If we plug in a PDB archive, we must unpack the files manually on the standby. The documentation gives the impression that making such a copy beforehand is always enough.

Although this operation works for basic scenarios, in real life, we may encounter additional complexity, including various situations in which the datafiles are modified during the plug-in, such as when the database is plugged in as a clone. In such cases, the standby will reject the files, meaning that we will have to copy the files again from the source, after the plug-in has

occurred.

Proxy, Relocate

All the other clone operations are variations of the basic cloning option and must be treated as such—that is, copy the files after the operation is done, or use RMAN to add the PDB back.

Removing PDB from Source

Obviously, sometimes we want to get rid of a PDB, too. As removing does not need any new datafiles to be created, it is generally an easier task to do on with a standby database in place.

And let’s look at how a rename happens, too.

Drop

The `DROP PLUGGABLE DATABASE` command affects all configured standby databases, meaning that the specified PDB will be dropped from all of them.

For this command to succeed, the PDB must be closed on all standbys. This obviously applies only to Active Data Guard configurations, as otherwise none of the PDBs can be open. If they are not closed, the redo apply stops and must be restarted again after the PDB is closed before proceeding.

Unplug

An unplug operation on a primary is also honored by all standbys. On the standbys, there is no XML or PDB archive created; instead, the PDB is simply marked as UNPLUGGED. As with the drop operation, the PDB must be closed on all the standbys for this to succeed.

Rename

A rename of PDB is, again, honored by all standbys. The operation requires the source to be in open restricted mode and closed on the standbys.

Changing the Subset

We can also change the list of included PDBs later on the fly. Let's see how to handle these scenarios.

Remove an Existing PDB

The standby database must be open in order to be modified. Furthermore, the actual statement must be run with the selected PDB:

```
SQL> alter database recover managed standby database cancel;

Database altered.

SQL> alter session set container=pdb1;

Session altered.

SQL> alter pluggable database pdb1 disable recovery;

Pluggable database altered.

SQL> alter session set container=cdb$root;

Session altered.

SQL> alter database recover managed standby database disconnect from session;

Database altered.
```

In this example, the alert log confirms that the datafiles have just been taken offline:

```
Warning: Datafile 12 (/u01/app/oracle/data/USPROD/PDB1/system01.dbf) is
offline during full database recovery and will not be recovered
```

After this, no redo is applied to the PDB, and the PDB is no longer usable.

The pre-12*c* method would be to alter the datafiles offline. This is, of

course, more cumbersome and does not set the recovery column of v$pdbs. However, it's the only option in version 12.1.0.1, and more importantly, it's still a valid way to remove only selected tablespaces from the standby.

Once these commands have been run, we can delete the physical files because they are no longer needed if the removal is a permanent one.

Enabling the PDB Recovery

There are multiple reasons why we might want to have a PDB made available again on the standby—the most important being in those instances where the database never made it to the standby in the first place, such as during remote cloning. Other cases include various testing scenarios, human errors, and many others that only the real-world experience of a DBA will reveal.

If the files are still present on the standby, we can try enabling the recovery:

```
alter pluggable database PDB5 enable recovery;

Pluggable database altered.
```

Oracle will attempt to recover the database using the redo available on the standby. Note that this command requires redo apply to be stopped and might also necessitate a restart of the standby to mount mode. If this fails or the files are not available, we can use RMAN to restore the files to the standby.

First of all, we need to determine whether Oracle knows where the missing files should be located. In some cases, such as with the `STANDBYS` clause, Oracle generates a name such as UNNAMED00178. In others, it carries over the name from the source, modifying it according to `DB_FILE_NAME_CONVERT` settings if those are configured. These new names are listed in the alert.log as well in v$datafile.

```
ORA-01565: error in identifying file '/u01/oracle/data/NZPROD/PDB7/system01.dbf'
ORA-27037: unable to obtain file status
Linux-x86_64 Error: 2: No such file or directory
```

Or

```
ORA-01565: error in identifying file '/u01/oracle/data/NZPROD/PDB7/system01.dbf'
ORA-27037: unable to obtain file status
Linux-x86_64 Error: 2: No such file or directory
```

For the next step, we must decide whether we want RMAN to access the primary to retrieve the datafiles, or whether we want to use backups. In the latter case, these must be accessible from the standby. Recovery catalog, or simply the `catalog` command in RMAN, can be a great help here.

If the database knows the target filenames and we are happy with these, we can issue the `restore` command:

```
restore pluggable database PDB7;
```

Or copy from the primary:

```
restore pluggable database PDB7 from service USPROD;
```

If we want new filenames, we have to detail them. Note also that we can specify `NEW` to let the standby generate OMF filenames, as an alternative:

```
run {
set newname for datafile 22 to
'/u01/oracle/data/NZPROD/PDB7/system.dbf';
set newname for datafile 23 to
'/u01/oracle/data/NZPROD/PDB7/sysaux.dbf';
restore pluggable database PDB7;
}
```

This second example uses OMF:

```
run {
set newname for pluggable database PDB7 to new;
restore pluggable database PDB7;
}
```

Now we can enable the recovery and restart the redo apply.

```
alter pluggable database PDB7 enable recovery;

Pluggable database altered.

alter database recover managed standby database disconnect from
session.

Database altered.
```

Cloud Control

Unfortunately, PDB management in EM has no provision for standby databases. It does not offer to specify the `STANDBYS` clause and is not helpful in the resolution of any issues.

The Data Guard administration page shown in Figure 11-7 displays the status after a clone, and we now have to resolve the situation manually, with little or no help from EM.

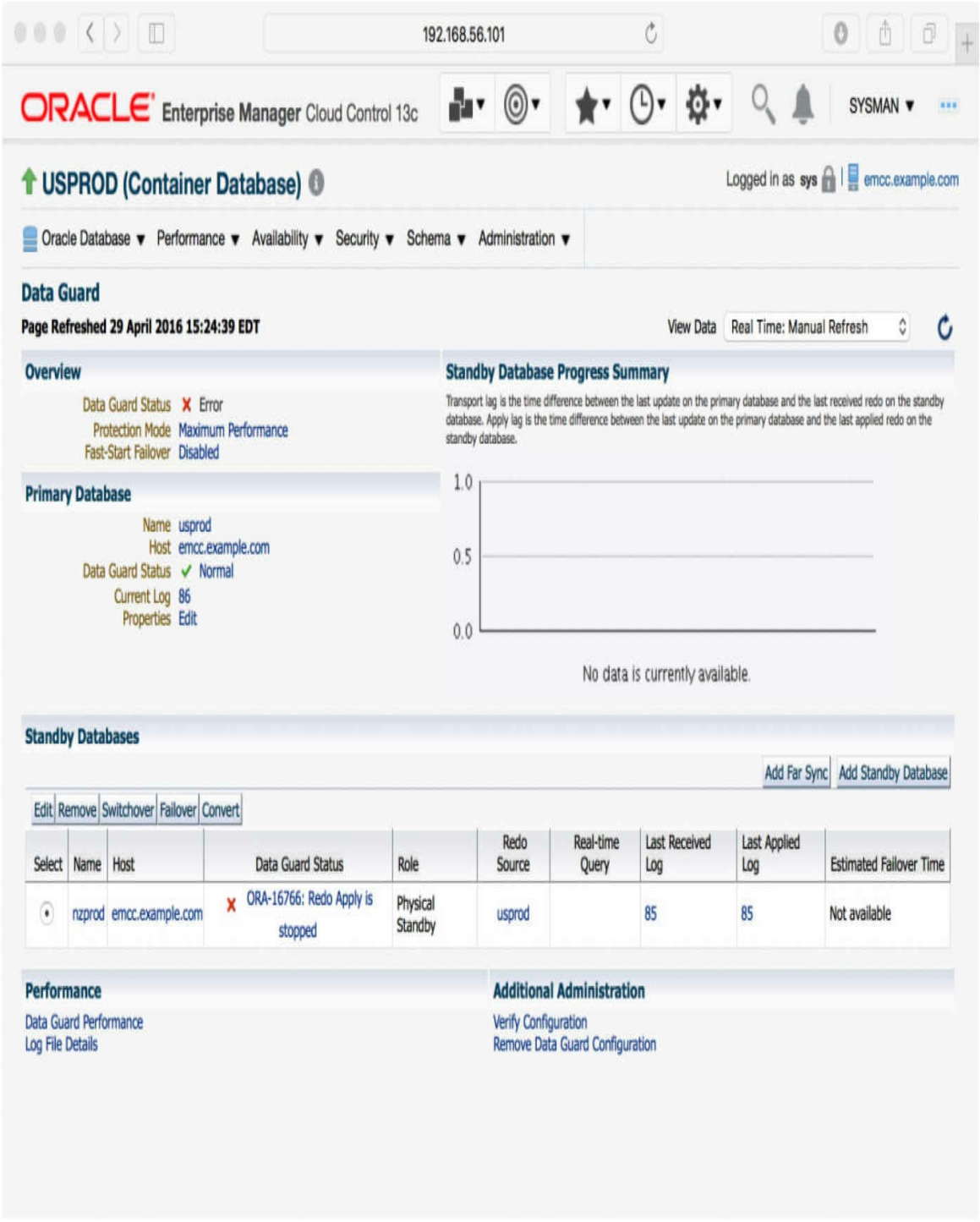


FIGURE 11-7. *Standby redo apply failure after clone operation*

Standby in the Cloud

As with RMAN backups, the cloud provides a cost-effective option for disaster recovery. The reasons for using it for this purpose are even more compelling than those for its use for primary production databases.

Two main points stand out: First, it's much cheaper and easier to run a standby in the cloud than to build a whole new data center—a backup site in case of a disaster—that might not ever be fully used. Second, the cloud usage charges are determined by allocated capacity, and a standby database, which only applies redo changes, needs much less processing power than the primary database. At the same time, if a switch to the cloud backup database is required, cloud technology means that it is easy to scale up the capacity as required.

Numerous cloud provider options are available, and many can run an Oracle database. After all, it's just an application running on a commodity operating system, usually Windows or Linux on Intel.

Oracle Public Cloud provided by the Oracle Corporation, however, promises tighter integration and added value, given that it's the same company behind this cloud as well as the database software. That is particularly true in the case of the RMAN cloud backup, as the Public Cloud provides a media management library for the cloud backup and direct integration with RMAN.

The database part of the Oracle Public Cloud is a recently introduced product and is thus evolving rapidly as an offering. As of the time of writing, Oracle started proving a one-click creation of standby database. After selecting Database as a Service (as shown in Figure 11-8), just select "Standby Database with Data Guard" and you end up with two nodes, primary and standby, instead of just one. The standby can be then managed using the Cloud Service Console or the dbaascli utility.

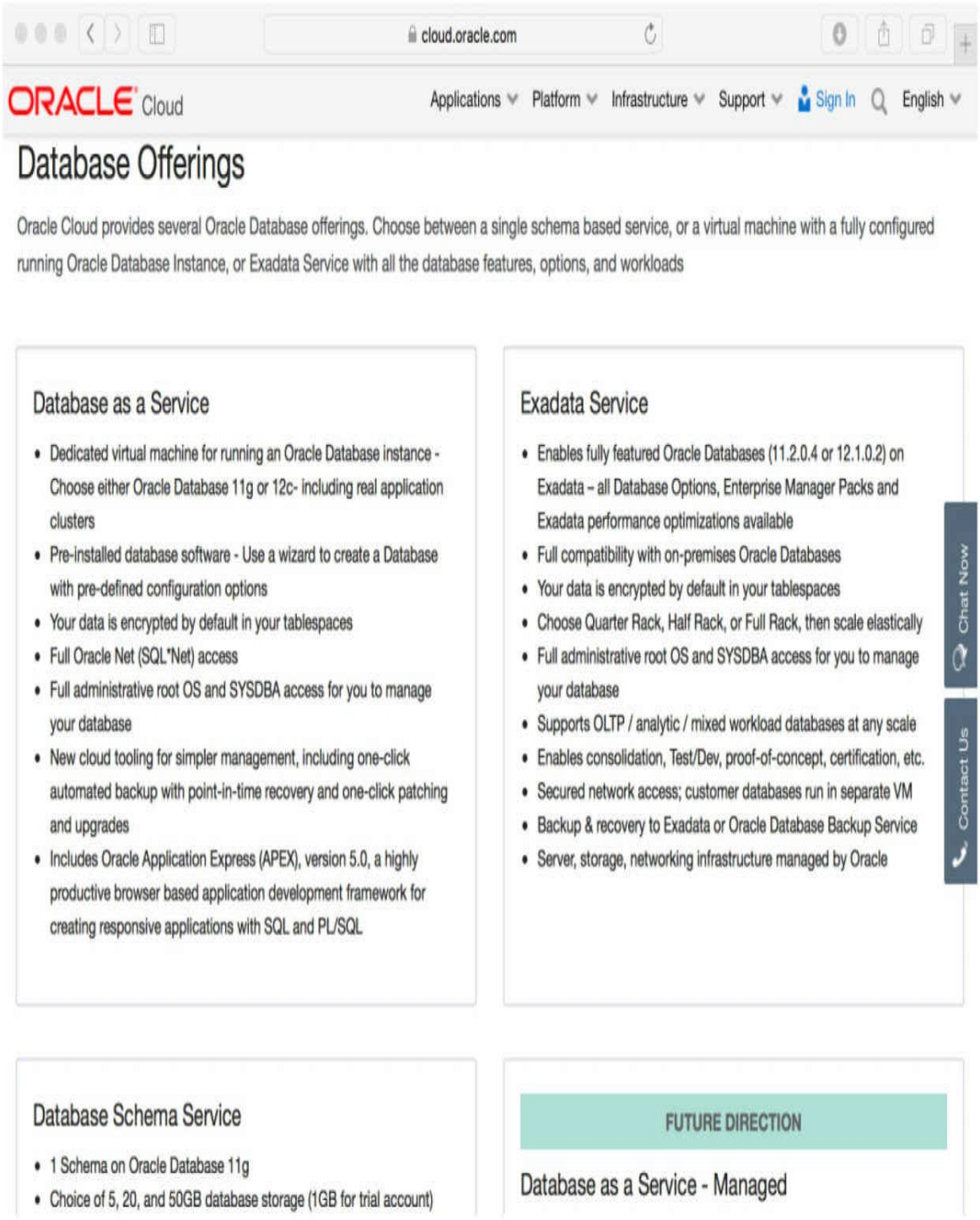


FIGURE 11-8. *The list of Database Offerings for Oracle Cloud*

Or you can use the old and proven dgmgrl because it's still just an ordinary Data Guard physical standby. And you can even skip that magic option and select a more hands-on approach, creating the standby as we described in this chapter, giving you all the flexibility and choices.

You should be aware that the cloud virtual machine comes with a license included in the price. For multitenant, we need to select at least the High Performance Service, and for Active Data Guard, only the Extreme Performance Service fits the requirement (this is true both for manually created standby and for the automatically provisioned, too).

If we already back up our on-premise database to the cloud, we can use these backups to create the standby. Again, the steps are identical to those within a local environment: install the Oracle Database Cloud Backup Module using opc_install.jar, and then instruct RMAN to use the library:

```
$ mkdir -p /u01/app/oracle/OPC/wallet
$ mkdir -p /u01/app/oracle/OPC/lib
$ export ORACLE_SID=CLSBY
$ export ORACLE_HOME=/u01/app/oracle/product/12.1.0/dbhome_1
$ java -jar opc_install.jar -serviceName <defined service name>
-identityDomain <backup service domain> -opcId '<user@company.com>' -opcPass
'<OPC password>' -walletDir /u01/app/oracle/OPC/wallet -libDir
/u01/app/oracle/OPC/lib
Oracle Database Cloud Backup Module Install Tool, build 2015-05-12
Oracle Database Cloud Backup Module credentials are valid.
Oracle Database Cloud Backup Module wallet created in directory
/u01/app/oracle/OPC/wallet.
Oracle Database Cloud Backup Module initialization file
/u01/app/oracle/product/12.1.0/dbhome_1/dbs/opcCLSBY.ora created.
Downloading Oracle Database Cloud Backup Module Software Library from file
opc_linux64.zip.
Downloaded 23169388 bytes in 20 seconds.
Download complete.
```

In RMAN, set `SBT_LIBRARY` to use this module, either in the run block, before there is a control file available,

```
run {
  allocate channel dev1 device type sbt
  parms='SBT_LIBRARY=/u01/app/oracle/OPC/lib/libopc.so';
  restore spfile TO '/tmp/spfile.ora' from autobackup;
}
```

or permanently in the control file:

```
RMAN> CONFIGURE CHANNEL DEVICE TYPE 'SBT_TAPE' PARMS 'SBT_LIBRARY=/
u01/app/oracle/OPC/lib/libopc.so, ENV=(OPC_PFILE=/u01/app/oracle/
product/12.1.0/dbhome_1/dbs/opcCLSBY.ora)';
```

Again, the process is exactly the same when using RMAN on-premise, backing up/restoring from the cloud. The notable difference is that, in this case, the data remains in the cloud, so the restore is not limited by the bandwidth of our Internet connection.

In summary, there is very little multitenant-specific functionality in terms of cloud disaster recovery. We must carefully select the machine type to have the multitenant option included in the license, and from there the further handling of PDBs is similar to that for an on-premise standby database.

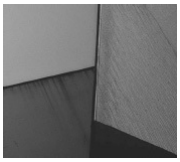
Summary

In this chapter, we covered one of the less glamorous, yet very important, features of Oracle Database and the cornerstone of the Maximum Availability Architecture: Data Guard.

You've seen that it is not difficult to create a standby database, although working with one effectively requires experience that comes from trying things over and over. We emphasized that the Oracle documentation related to this is strewn across multiple locations. On the other hand, disaster recovery is such a critical topic that digging, experimentation, and practice are beneficial in the long run.

To conclude, multitenant itself does not change how a standby database works; however, in enabling cloning and plug-in/unplug operations, such features have the potential for major impact, and many of them break that

“setup standby once and then forget it” attitude we may have once held.



CHAPTER 12

Sharing Data Across PDBs

Is a multitenant database one database, or is it many? This is a question that runs like a silver thread throughout this entire book, and, as you have seen so far, the answer is, “it depends.” For some operations, we need to think of multitenant as a single database; for others, we have to think at the pluggable database (PDB) level and treat each PDB independently.

In this chapter, we investigate how to access the data stored in one PDB from data stored in another PDB, and we will show that, again, we can address the problem from both of the angles described. Furthermore, we will discuss how Oracle introduces a completely new point of view.

At a very basic level, on one hand, we can completely ignore all the new multitenant features, treat all PDBs as completely separate databases, and, as ordinary users, log into a PDB and create a database link to another database, regardless of whether this is part of the same container database (CDB) or not. Or we can tunnel through the wall constructed by Oracle between the PDBs, asking for data from another PDB directly, although this can get very complex and elaborate, as you will see. Let’s take a look at the various options in detail.

Database Links

A database link is a tried-and-tested feature, proven over time, that has been with us since Oracle Database 5.1. That’s 30 years! This is a proven solution, one that DBAs and users are familiar with, and it is congruent with the Oracle message that nothing changes for users when the multitenant architecture is

adopted.

Of course, there are some limitations on what type of operations are possible over a database link, but even those are being addressed. For example, Oracle Database 12.2 fills in one long-standing gap, which is support for large object (LOB)–based datatypes.

In addition to the familiarity of this functionality, there is one more major advantage, and this is the opacity of the link to the user. The user does not need to know where the target PDB actually resides. So when a DBA moves the PDB to a new CDB, only the resolution of the connection string has to change, by editing tnsnames.ora, for example. Again, a database link behaves in the same way as it always did, multitenant or not.

In a non-CDB, database links can be split into two basic categories: a remote database link connecting to an external database, and a so-called "loopback" database link connecting to the same database. We can think of the loopback link as an aberrant case, but there are times when it has its place —for example, when two applications are consolidated to the same database, and a database link is required between them for use by the applications. Oracle Database recognizes this special case and introduces some optimizations, such as having only one single transaction on the database shared by both ends of the link. And Oracle Database 12*c* promises even more performance optimizations, as Oracle expects more links pointing back to the same (container) database.

Multitenant also brings a new distinction to database links, in conjunction with the remote/loopback connection, so that a database link can now connect either to the root container or to a PDB. Note that there is no way to change the current container of the remote end of a database link, because there is no *alter session set container* for it.

Because of this, a root container database link is useful for administration only. As you saw in Chapter 9, remote clones can use such a link, and we don't have to create a new one for every single PDB we want to clone. But for accessing data in PDBs, we need to create links connecting directly to the desired PDBs. Although some of the techniques described later in this chapter make user data visible even in the root container, in some cases a CDB link may make sense to access data, too.

As you can see in Figure 12-1, Enterprise Manager offers all the usual options to create a database link. There is no special option to specify whether the target is a PDB or a CDB, because this is implied by the Net

Service Name.

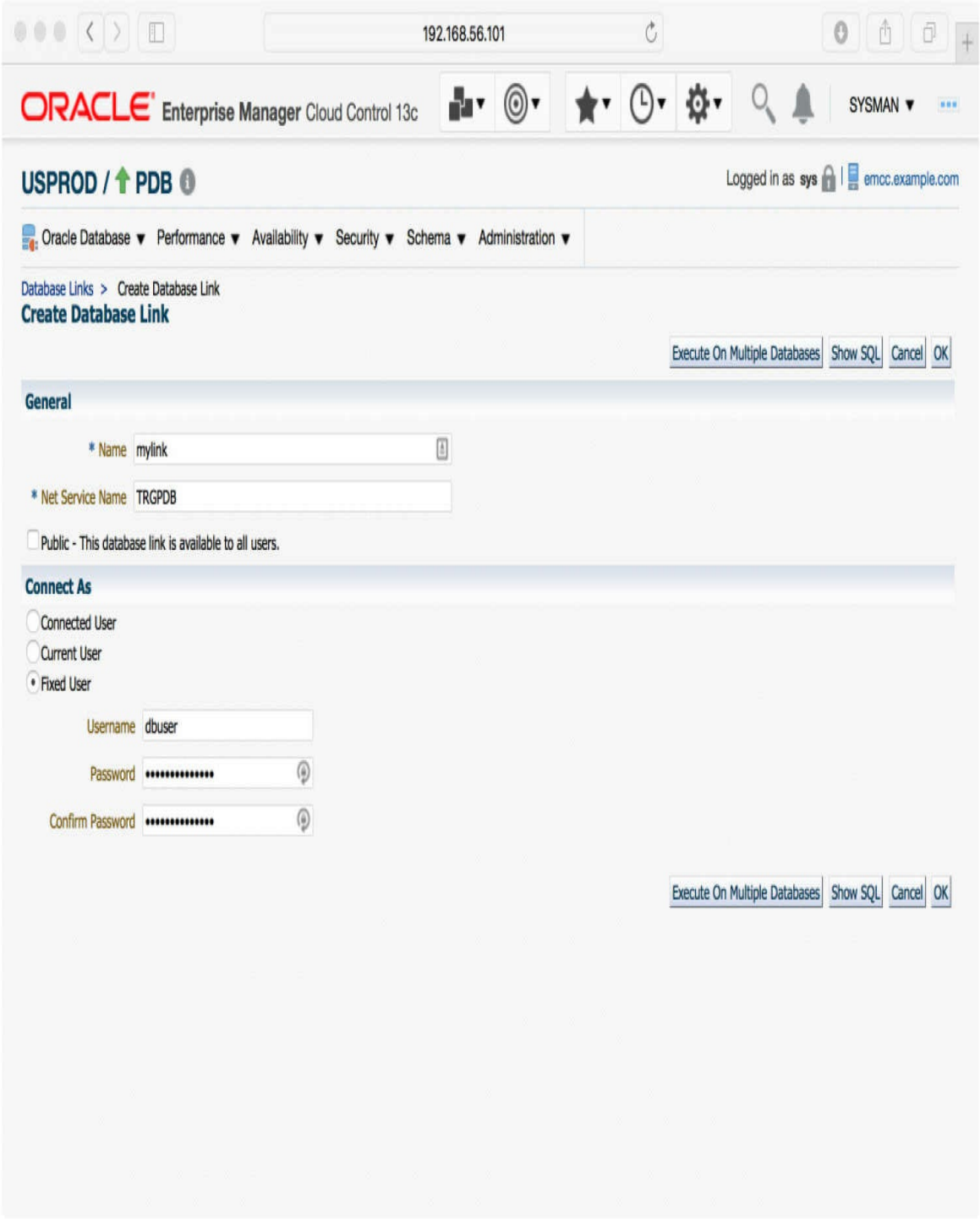
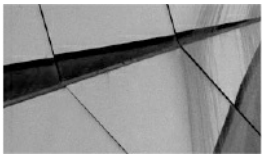


Figure 12-1. *Creating a database link in Enterprise Manager Cloud Control*

Sharing Common Read-Only Data

A special case of data sharing is sharing of the same read-only data by multiple databases. For example, if we want to provide multiple development databases and include large historical data—data that is only queried and no longer modified—it would be ineffective to copy such data multiple times, both in terms of the time taken to perform this operation and in respect to the storage consumed. In a sense, this is a poor man's solution to making database cloning less storage demanding.



NOTE

Some of the options described here will also work with read/write data, although the options' usefulness shines in cases when the data is seldom modified.

Transportable Tablespaces

In 8*i*, Oracle introduced the transportable tablespaces feature. Its prominent use case is to copy data from a production online transaction-processing database into a data warehouse, removing the need for import/export, or for an alternative data extraction and load process. Instead, entire datafiles are copied, with only the metadata imported. For the basic scenario, Enterprise Manager has a nice step-by-step wizard that starts with the screen shown in Figure 12-2.

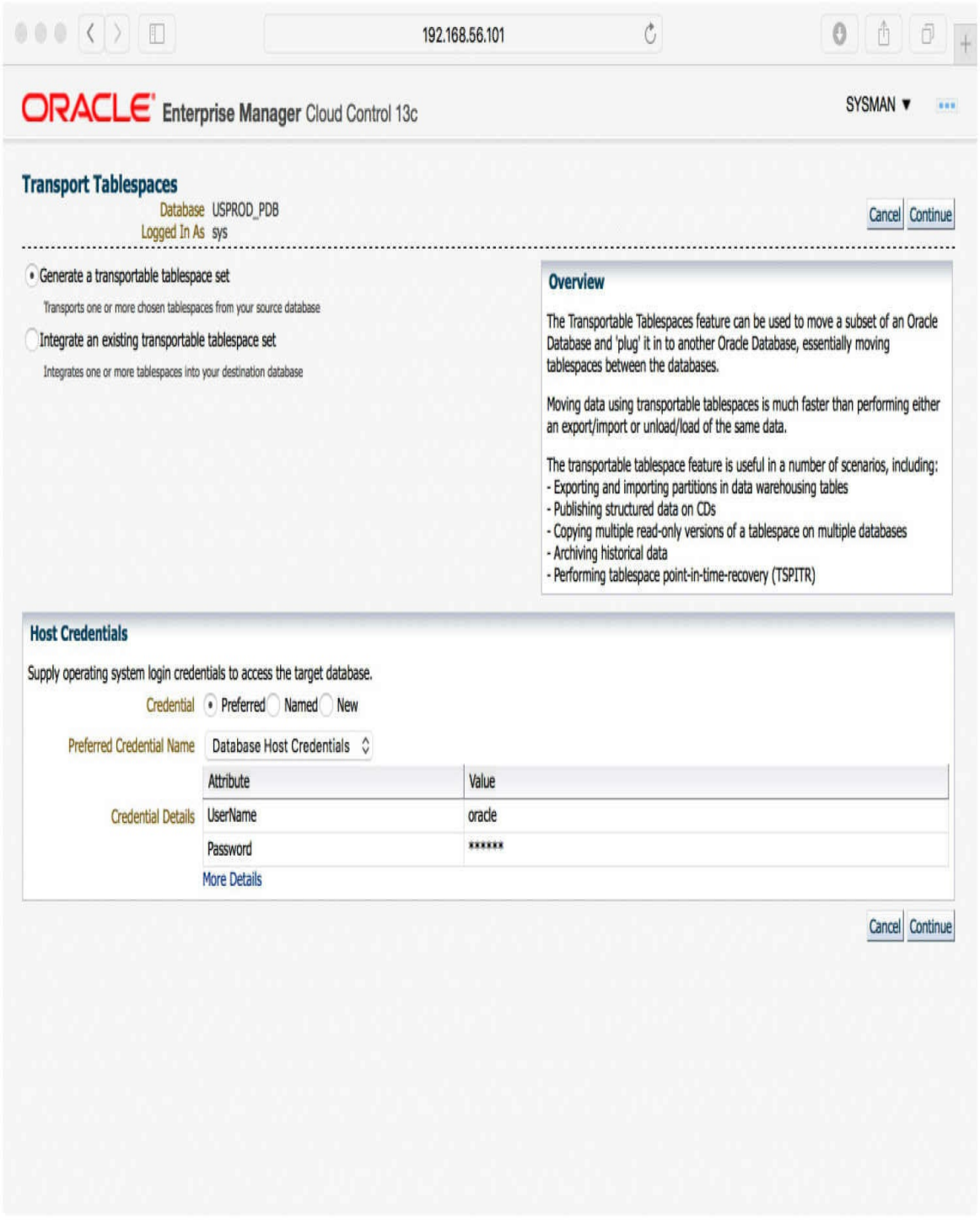


Figure 12-2. *Transportable tablespaces in Enterprise Manager Cloud Control*

First, we make the source tablespaces read-only and then export the metadata. If our users and applications preclude us from setting the tablespaces to read-only, we can use Recovery Manager (RMAN) to do this export (using the `TRANSPORT TABLESPACE` command). There is no magic technology used by RMAN; it simply does a partial restore into a new database and runs the export there.

Next, we copy the datafiles to the target server and import the metadata into the target database. This step also makes the datafiles part of the target database.

At this point, the usual next step is to set the tablespaces to read/write and let users modify them. However, this is not mandatory, as the data is already accessible and the datafiles are still not modified. This means that we can also import the same metadata and tablespaces into another database—one or many more databases, as we wish.

It's the DBA's responsibility to ensure that none of the databases will modify the tablespaces and that nobody opens them read/write. The databases are oblivious to the fact that the file is used by other databases at the same time, so it won't prevent us from enacting such changes. However, they will detect such modifications should they happen and will complain loudly.

Of course, because the files are not modified at all, it isn't necessary for us to copy them from the source database, and even the source database itself can be one of the sharing databases. Whether we want to do this depends a lot on our use case and environment. Such sharing is not appropriate on a production database, which is probably on separate storage and should not bear the penalty of I/O generated by test and development, but in other cases it might be a very useful option.

Storage Snapshots and Copy on Write

A simple extension of the same idea is to leverage storage to provide copies of the tablespaces. This enables us to have multiple copies of the same tablespace, but as of different points in time, or even read/write copies. Of course, this is then more of a "copy-provisioning" option than sharing, but it still has its use cases. After all, when using such tablespaces to provision read-only copies of a production database for testing, having the copy as of multiple points in time is a very beneficial facility.

So when and where might we find use for tablespace copies? In some cases, a test has to be done with multiple copies of the data, while in others we can just build new test environments as we go and update data over time.

To achieve the former, we would create a database that is a copy of the source database and run the transportable tablespace export from there. This copy can be a Data Guard physical standby, making the task of keeping it up-to-date very easy. Alternatively, we could make use of a simple restore and incremental backups if we want to decouple the databases more.

Then, every time we want to make a new copy, we make the tablespace read-only, run the export, and create a new file system snapshot. The clone database can continue to receive redo or incremental backups, and the file system has the copy of the tablespace for us to use. Thanks to the snapshot and copy-on-write facilities, only blocks modified by the clone database after the snapshot was created will take up any extra disk space.

Delphix

Another option is Delphix (www.delphix.com), which provides both fast and cheap cloning as well as storage de-duplication. In this case, we might do well to rethink our entire approach, because this tool excels in the provisioning of entire databases for testing and development. This means that the building of such databases is considerably faster, more flexible, and less painful than we can achieve with a simplistic solution such as sharing one or a few tablespaces.

Cross-PDB Views

As we discussed in previous chapters, Oracle introduced CDB_% dictionary views that display information collected from all open PDBs. In version 12.1.0.1, Oracle used a trick with an internal function called CDB$VIEW to achieve this. This was not documented, and some scenarios caused internal Oracle errors instead of proper results or appropriate error messages.

Fortunately, that version is now a relic of the past, and as of version 12.1.0.2, Oracle Database switched to using the *CONTAINERS()* clause, which is properly documented and supported.

Delving into this a little deeper, here's the definition of CDB_USERS in Oracle Database 12.1.0.2:

```
SELECT "USERNAME", "USER_ID", "PASSWORD", "ACCOUNT_STATUS", "LOCK_DATE",
"EXPIRY_DATE", "DEFAULT_TABLESPACE", "TEMPORARY_TABLESPACE", "CREATED",
"PROFILE", "INITIAL_RSRC_CONSUMER_GROUP", "EXTERNAL_NAME", "PASSWORD_
VERSIONS", "EDITIONS_ENABLED", "AUTHENTICATION_TYPE", "PROXY_ONLY_
CONNECT", "COMMON", "LAST_LOGIN", "ORACLE_MAINTAINED", "CON_ID" FROM
CONTAINERS("SYS"."DBA_USERS")
```

In other words, the DBA_USERS view is wrapped in the *CONTAINERS()* clause and references one new column created by this clause, *con_id*, obviously referencing the container ID of the PDB.

The *CONTAINERS()* clause causes Oracle to execute the same query on each of the open PDBs, but skipping PDB$SEED. It is valid to query such a view even from a PDB, but in that case only data for the current PDB is returned.

Let's have a look at the same view in Oracle Database 12.2.0.1:

```
SELECT k."USERNAME", k."USER_ID", k."PASSWORD", k."ACCOUNT_STATUS",
k."LOCK_DATE", k."EXPIRY_DATE", k."DEFAULT_TABLESPACE", k."TEMPORARY_
TABLESPACE", k."LOCAL_TEMP_TABLESPACE", k."CREATED", k."PROFILE",
k."INITIAL_RSRC_CONSUMER_GROUP", k."EXTERNAL_NAME", k."PASSWORD_VERSIONS",
k."EDITIONS_ENABLED", k."AUTHENTICATION_TYPE", k."PROXY_ONLY_CONNECT",
k."COMMON", k."LAST_LOGIN", k."ORACLE_MAINTAINED", k."INHERITED",
k."DEFAULT_COLLATION", k."IMPLICIT", k."CON_ID", k.CON$NAME, k.CDB$NAME
FROM CONTAINERS("SYS"."DBA_USERS") k
```

You can see that Oracle started using a table alias (*k*) in the query. For this particular view, it also added three new columns. What interests us here, in particular, are the two new columns generated by the *CONTAINERS()* clause: CON$NAME and CDB$NAME.

The first is the name of the PDB where the records come from, essentially translating the *con_id* into the PDB name. The second new column is the name of the CDB itself, and you will see later in this chapter when this might be useful.

We already noted that querying from the PDB will show us data from the PDB only, while querying from CDB$ROOT gives us data from all the PDBs

except the seed. If we run the query from an application container root, we get data from the application root and all its application PDBs, except the application seed.

Simple User Tables

Because the *CONTAINERS()* clause is documented and supported, Oracle allows us to query our own user tables as well as build views on top of them, aggregating data from multiple PDBs.

Simple Status Query

Let's start with a simple example query, supposing that we want to monitor multiple PDBs. For this, we will assume that each has a user table called ERRORLOG, and we want to query for all error messages inserted there. We are interested in only the single table, and we want to query across all the PDBs.

First of all, we need a common user to query the data, as a local user would always see data from its PDB only:

```
SQL> create user c##errorquery identified by oracle;

User created.

SQL> grant create session, create view to c##errorquery container=current;
Grant succeeded.

SQL> grant set container, create table, unlimited tablespace to
c##errorquery container=all;

Grant succeeded.
```

The ERORRLOG table must be present in all of the PDBs (actually, only in those PDBs we want to query, as you'll see later) and in the root, too, and the Data Definition Language (DDL) of all these copies must match. Because the table must also be in the root, we have to create it under a common user

schema, or we can create it under local users in the PDBs and let the common user have views or synonyms. In these examples, we don't use application containers, so the common user is created in CDB$ROOT; this mandates that the user's name is prefixed by the COMMON_USER_PREFIX, which defaults to *c##*.

We run this create DDL in the CDB, then in all the PDBs:

```
SQL> create table c##errorquery.errorlog (i number primary key, message
varchar2(4000), when date);

Table created.

SQL> alter session set container=PDB1;

Session altered.

SQL> create table c##errorquery.errorlog (i number primary key, message
varchar2(4000), when date);
…etc…
```

As simple test data, let's insert one row in each of the PDBs:

```
SQL> insert into  c##errorquery.errorlog values (0, 'root', sysdate);

1 row created.

SQL>  commit;

Commit complete.

SQL> alter session set container=PDB1;

Session altered.

SQL>  insert into  c##errorquery.errorlog values (1, 'PDB1', sysdate);

1 row created.

SQL>  commit;

Commit complete.
```

Now we can query the data:

```
SQL> select * from containers(errorlog);

 I MESSAGE         WHEN      CON_ID
-- -------------- --------- ------
 0 root           13-FEB-16      1
 1 PDB1           13-FEB-16      3
```

As noted, the DDL must match. If, for example, we generate a new PDB and create the table differently, like this (same would happen if the table is not there at all):

```
SQL> create table c##errorquery.errorlog (i number primary key, message
varchar2(4000));

Table created.
SQL> insert into table c##errorquery.errorlog values (2, 'PDB2');

1 row created.

SQL> commit;

Commit complete.
```

Oracle silently ignores this table:

```
SQL> select * from containers(errorlog);

 I MESSAGE         WHEN      CON_ID
-- -------------- --------- ------
 0 root           13-FEB-16      1
 1 PDB1           13-FEB-16      3
```

However, there is some leeway for allowed differences. Oracle takes the metadata from the table in the root, so if the PDB tables have extra columns, that's fine, because they will be ignored.

So, let's fix the missing column and add one more:

```
SQL> alter table errorlog  add (when date, who char);

Table altered.
```

Now the query from the root works again—ignoring the extra column:

```
SQL> select * from containers(errorlog);

 I MESSAGE         WHEN      CON_ID
-- -------------- --------- ------
 0 root           13-FEB-16      1
 1 PDB1           13-FEB-16      3
 2 PDB2                          4
```

Querying Only Some of the PDBs

Oracle runs the query only in the containers where it makes sense to do so. So if we specify a condition such as *con_id*, to choose only some of the PDBs, Oracle will look in these PDBs only:

```
SQL> select i, message, when, con_id, con$name, cdb$name
from containers(errorlog) where con_id in (1,3);

 I MESSAGE         WHEN      CON_ID CON$NAME   CDB$NAME
-- -------------- --------- ------ ---------- ----------
 0 root           13-FEB-16      1 CDB$ROOT   CDBSRC
 1 PDB1           13-FEB-16      3 PDB1       CDBSRC
```

And we can use CON$NAME, too:

```
SQL> select i, message, when, con_id, con$name, cdb$name
from containers(errorlog) where CON$NAME='PDB1';

 I MESSAGE         WHEN      CON_ID
-- -------------- --------- ------
 1 PDB1           13-FEB-16      3
```

For another, more sophisticated, approach to this problem, see the "Container Map" section later in this chapter.

Query Hint

In some cases, we might want or need to add a hint to a query that uses the

CONTAINERS() clause. But if we put it in the usual syntax, it will apply only to the last aggregation step, not to the queries that are run in each of the PDBs.

Fortunately, Oracle introduced a way for us to "push down" such hints into the queries. The *CONTAINERS(DEFAULT_PDB_HINT='… ')* syntax specifies these hints:

```
SQL> select /*+CONTAINERS(DEFAULT_PDB_HINT='FULL')*/ *
from containers(c##errorquery.errorlog) where con_id in (1,3);
```

Querying the Data from a PDB

As already noted, accessing the *CONTAINERS()* query from within a PDB gives us data only for that particular PDB.

In the previous examples, we had a logging table, and the query collected errors from all the PDBs. So what if one of the PDBs is a monitoring application that actually needs to access and process all these error logs?

Well, the solution is trivial: we can just create a database link pointing to the common user in the CDB, and query the data over the database link. We can't put the *CONTAINERS()* clause around a remote object, but we can work around this simply by creating a view.

```
SQL> create view errorlog_v as select * from containers(errorlog);

View created.

SQL> alter session set container = 'PDB1';

Session altered.

SQL> create database link common connect to c##errorquery identified by
oracle using 'CDBSRC';

Database link created.

SQL> select * from errorlog_v@common;
```

Consolidated Data

One of the use cases of multitenant that really fits its features and advantages, and perhaps the one that Oracle had in mind when designing this feature, is consolidation of instances of the same application. Imagine a service provider that sells an SaaS application to its customers and needs multiple copies of the same application running on the same hardware. Or perhaps a company IT department provides the same application for multiple branches or franchisees; the application is the same and the users need data separation, but the head office wants to access the data from all of the PDBs at the same time to run various reports. These types of companies can benefit from consolidated access to their data.

In the pre-multitenant world, we could either do schema consolidation and put all the users into the same database if the application supported such an approach, or we could build one consolidated staging/data warehouse and load the necessary data into it, and then run reports from there. However, as you have seen in the previous section, multitenant enables us to access all the PDBs at the same time and thus build queries that run consolidated reports on the data directly (although with imposed limitations such as having the same table names and structure). The approach described in the previous section can achieve this, but it could be a bit cumbersome should we do it for more than a handful of tables.

Oracle Database 12.2 unleashes a number of features that make this

consolidation a much more prominent citizen of the database ecosystem. Oracle promotes application containers as one of the key pieces of the solution for these scenarios, although many of the features don't actually require them. Still, if we find that working with consolidated PDB data is an important piece in our application, it might be a good time to dig deeper into the application containers and consider whether building the application using them would be a good fit.

Linking Tables Across Containers

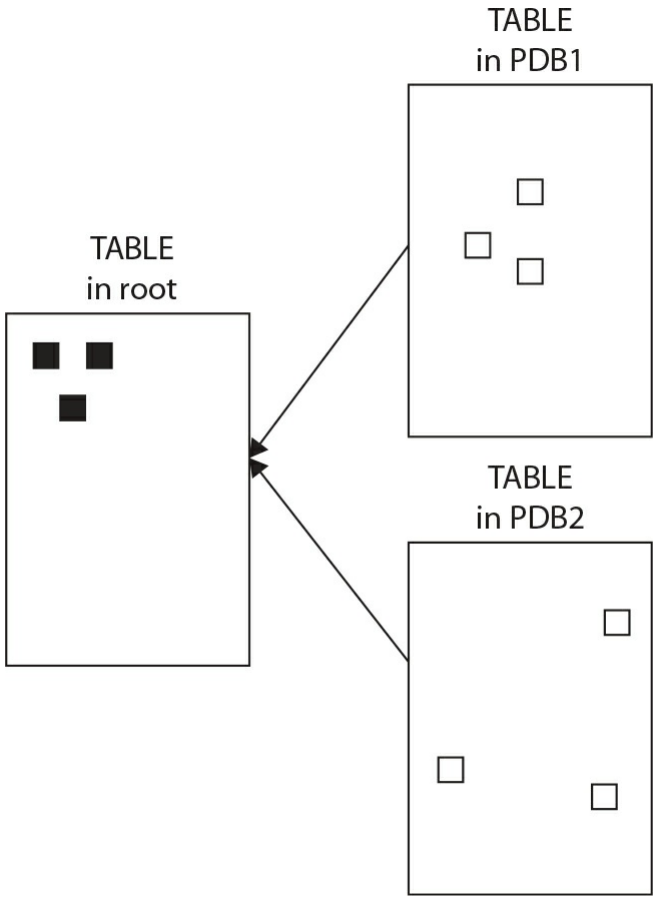
You have already seen in Chapter 1 how Oracle links the dictionary objects between the CDB and the PDBs using metadata and object links. Oracle now enables us to do the same with our objects and data—well, kind of.

This implies storing data in the root container, and Oracle is strict here in that it does not want us to enter into the bad practice of creating user objects in the CDB root. It thus mandates that we create an application container instead and store the data in its container root.

So let's create an application container with a couple of application PDBs in it, with a common user. In this simple example, we won't create an application seed in the application container.

```
SQL> create pluggable database appcont as application container admin user ac
identified by oracle;

Pluggable database created.

SQL> alter session set container=appcont;

Session altered.

SQL> alter pluggable database open;

Pluggable database altered.

SQL> create pluggable database ACPDB1 admin user ap1 identified by oracle;

Pluggable database created.

SQL> alter pluggable database acpdb1 open;

Pluggable database altered.

SQL> create pluggable database ACPDB2 admin user ap2 identified by oracle;

Pluggable database created.

SQL> alter pluggable database acpdb2 open;

Pluggable database altered.
SQL> create pluggable database ACPDB3 admin user ap3  identified by oracle;

Pluggable database created.

SQL> alter pluggable database acpdb3 open;

Pluggable database altered.

SQL> alter session set container=appcont;

Session altered.

SQL> create user shareapp identified by oracle container=all;

User created.

SQL> grant create table, create session to shareapp container=all;

Grant succeeded.
```

Metadata-Linked Objects As the application common user in the application container root, we can now create a metadata-linked table. The table definition, or metadata, is in the container root; the actual data segments are private to each application PDB. The following illustration shows such arrangement of data:



Note that we are modifying the application root, and Oracle requires us to mark any such modifications as “application install,” to be able to replay the changes in the containers.

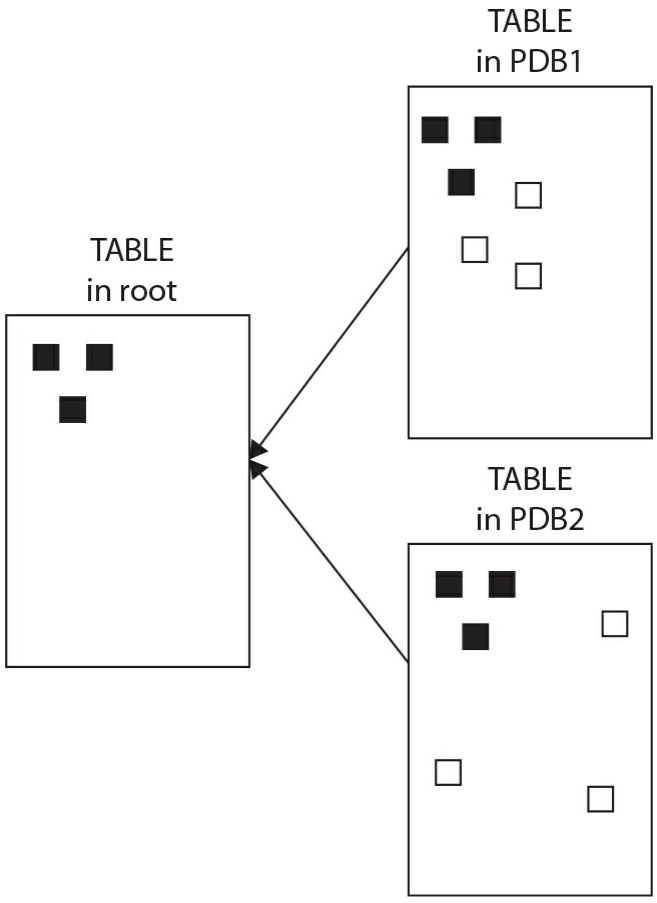
```
SQL> alter pluggable database application appcont begin install '1';

Pluggable database altered.

SQL> create table mlink sharing=metadata  (i number, message varchar2(20));

Table created.

SQL> insert into mlink values (1, 'global, install');

1 row created.

SQL> commit;

Commit complete.

SQL> alter pluggable database application appcont end install;

Pluggable database altered.

SQL> insert into mlink values (2, 'global, postinstall');

1 row created.

SQL> commit;

Commit complete.

SQL> update mlink set message='modified global' where i=1;

1 row updated.

SQL> commit;

Commit complete.

SQL> select rowid, i, message from mlink;

ROWID                         I MESSAGE
------------------ ---------- --------------------
AAAS8FAABAAAIWxAAA          1 modified global
AAAS8FAABAAAIWxAAB          2 global, postinstall

SQL> alter session set container=acpdb1;

Session altered.

SQL> alter pluggable database application appcont sync;

Pluggable database altered.
SQL> select rowid, i, message from mlink;

ROWID                         I MESSAGE
------------------ ---------- --------------------
AAAS8DAABAAAIWhAAA          1 global, install
```

As you can see, any objects and data created during the application install are synchronized to the PDB from the application root. However, no changes done after the install are reflected, and the *rowid* shows that these are really distinct rows; the sync does a one-time copy, and from then on the rows are not linked in any way.

However, there *is* some value to these oversimplified nonlinked tables. Oracle knows which containers have this particular version installed and thus contain the table; it does not complain that some of the containers don't have the table present yet, and we don't need to limit the query manually on *con_id*.

```
SQL> select * from containers(mlink);

       I MESSAGE              CON_ID
-------- -------------------- -------
       1 modified global            6
       2 global, postinstall        6
       1 global, install            7
```

Extended Data Metadata-Linked Objects A special flavor of metadata-linked objects is an *extended data object*. In this case, the data is stored in both the application root (common data, visible by all PDBs) and in the application PDBs (data private to each of the PDBs), as shown in the following illustration:



The steps are similar to those in the previous example, with the notable inclusion of the *EXTENDED DATA* clause in the DDL.

In this example, the application is already installed, thanks to the example we ran in the previous section, and we now add a new table there in a patch:

```
SQL> alter pluggable database application appcont begin patch 1 minimum version '1';

Pluggable database altered.

SQL> create table edlink sharing=extended data
(i number, message varchar2(20));

Table created.

SQL> insert into edlink values (1, 'global, install');

1 row created.

SQL> commit;

Commit complete.

SQL> alter pluggable database application appcont end patch;

Pluggable database altered.

SQL> insert into edlink values (2, 'global, postinstall');

1 row created.

SQL> commit;

Commit complete.

SQL> update edlink set message='modified global' where i=1;

1 row updated.

SQL> commit;

Commit complete.

SQL> select rowid, i, message from edlink;

ROWID                         I MESSAGE
------------------ ---------- --------------------
AAAS8RAABAAAIXxAAA          1 modified global
AAAS8RAABAAAIXxAAB          2 global, postinstall

SQL> alter session set container=acpdb1;
Session altered.

SQL> alter pluggable database application appcont sync;

Pluggable database altered.

SQL> select i, message from edlink;

         I MESSAGE
---------- --------------------
         1 modified global
         2 global, postinstall
```

We observe that the application PDB sees current values of objects in the root, including any changes done out of the application action. (Note that we can't use ROWID in the query in a PDB, however, because it will fail with an ORA error.) And it can also add new records, private to the application PDB:

```
SQL> insert into edlink values (3, 'from PDB');

1 row created.

SQL> commit;

Commit complete.

SQL> select  i, message from edlink;

         I MESSAGE
---------- --------------------
         1 modified global
         2 global, postinstall
         3 from PDB
```

However, we cannot modify records inherited from the root; an update command does not see these rows, even if a select query at the same time does.

```
SQL> update edlink set message='new' where i=1;

0 rows updated.
```

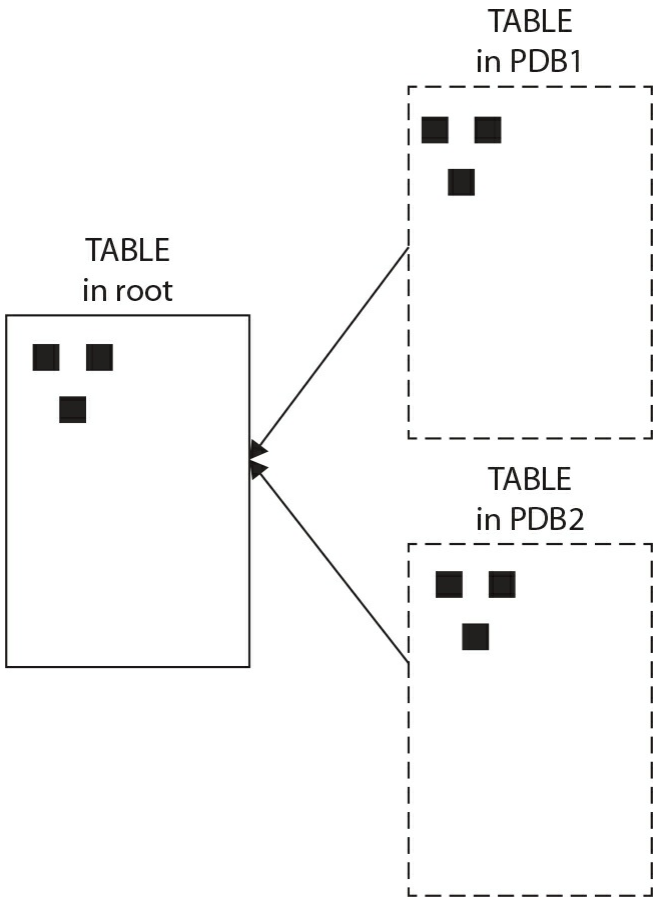
Although the root can, of course, modify the row:

```
SQL>  alter session set container=appcont;

Session altered.

SQL> update edlink set message='new' where i=1;
1 row updated.

SQL> commit;

Commit complete.

SQL> select con_id, i, message from containers(edlink);

    CON_ID          I MESSAGE
---------- ---------- --------------------
         6          1 new
         6          2 global, postinstall
         7          1 new
         7          2 global, postinstall
         7          3 from PDB
```

One caveat, as you have seen, is that each PDB shows its own copy of the common rows, and the parallel slaves executing a *CONTAINERS()* query across the PDBs do not remove these duplicates.

Data-Linked Objects An object-linked object exists in one place only: in the application container root. We are not allowed to modify the data the PDBs.

As the next illustration reveals, this is really common data shared by all PDBs, modified only during maintenance:



Or, looking at it from a completely different point of view, this could be considered read-only data shared by all PDBs—very similar to that which was achieved with transportable tablespaces.

```
SQL> alter session set container=appcont;

Session altered.

SQL> alter pluggable database application appcont begin patch 2 minimum version '1';

Pluggable database altered.

SQL> create table dlink sharing=data (i number, message varchar2(20));

Table created.

SQL> insert  into dlink values (1, 'global, install');

1 row created.

SQL> commit;

Commit complete.

SQL> alter pluggable database application appcont end patch;

Pluggable database altered.

SQL>  select rowid, i, message from dlink;

ROWID                I MESSAGE
------------------ --- --------------------
AAAS8KAABAAAIXJAAA   1 global, install
```

The root container can modify these contents even if not in the special application patch or upgrade mode:

```
SQL> insert into dlink values (2, 'global, postinstall');

1 row created.

SQL> commit;

Commit complete.
```

Now let’s switch to the application PDB:

```
SQL> alter session set container=acpdb1;

Session altered.
SQL> alter pluggable database application appcont sync;

Pluggable database altered.

SQL>  select i, message from dlink;

   I MESSAGE
 --- ------------------------------------
   1 global, install 2 global, postinstall

SQL> insert into dlink values (3, 'from PDB');

insert into dlink values (3, 'from PDB')
            *
ERROR at line 1:
ORA-65097: DML into a data link table is outside an application action
```

The application PDB cannot modify the table. There is really only one copy of the data, the PDBs can access it read-only, seeing all changes done in the CDB.

A minor limitation is that it is not possible to query rowid of the table while in a PDB - such query fails with ORA-02031: no ROWID for fixed tables or for external-organized tables.

Cross-PDB DML

In Oracle Database 12.2, and not limited to application containers, we can actually issue DML on a table wrapped in the *CONTAINERS()* clause. In other words, we can change data in multiple PDBs with a single operation, from a single transaction.

```
SQL>  update containers(nolink) set message='u'||message;

4 rows updated.
```

We can even specify the *con_id* to limit the execution to selected PDBs.

```
SQL> update containers(nolink) set message='u'||message
where con_id in (3,4);

2 rows updated.
```

Oracle introduces a new setting that we can use to specify which PDBs should be affected by this DML. This way, we don't have to specify the *con_id* condition in each DML, which means we don't have to modify the scripts every time we want to run it on a different set of PDBs.

To get the current setting, we can look at *DATABASE_PROPERTIES*:

```
SQL> select property_value from database_properties
where property_name='CONTAINERS_DEFAULT_TARGET';
```

And we can change the values with this:

```
alter [pluggable] database containers default target = ...
```

The *pluggable* keyword is to be used when we issue the statement in the application root (which is a kind of PDB itself), although we omit the keyword if we execute it in CDB$ROOT. As for the target specification, we can use <a list of PDBs>, *ALL*, *ALL EXCEPT <a list of PDBs>*, or *NONE*.

Note that the default is *NONE*, which, contrary to its name, actually contains all (application) containers, except the seed and root.

Containers Default

If we find that we are using the *CONTAINERS()* clause heavily, we might realize it's too much work, and too error-prone, to add the clause dutifully to every reference for the tables we want to query this way. Or perhaps we want to reuse existing code and queries and want to minimize the changes needed to update the code to use the clause. Fortunately, Oracle had introduced a new table setting, which, if enabled, will result in every query and DML on that table or view being automatically wrapped in the *CONTAINERS()* clause.

```
SQL>  alter table c##errorquery.errorlog ENABLE CONTAINERS_DEFAULT;

Table altered.
```

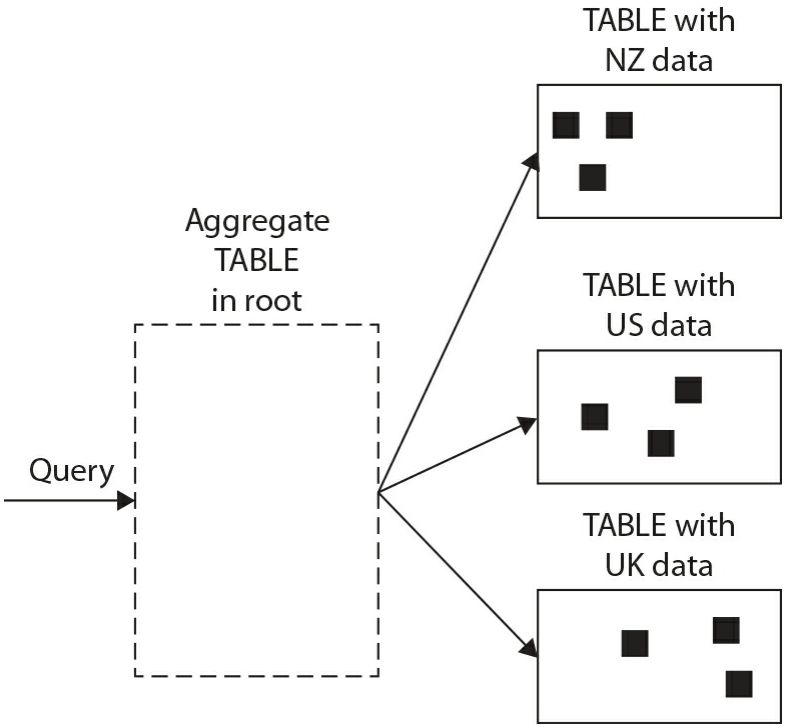
The setting can be easily checked in the data dictionary:

```
SQL> select CONTAINERS_DEFAULT from user_tables where table_name='ERRORLOG';

CON
---
YES
```

Container Map

When we have multiple PDBs with the same tables but different data (such as data for different customers or branches) and we run queries on this data, all these PDBs are interrogated, unless we specify *con_id* in the *where* condition.

In a sense, this is a partitioning schema; data for a single customer or branch resides in a single PDB only—it doesn't overlap—and if we search for a single branch, only one PDB will contain data for it, as shown in the illustration:



Therefore, it stands to reason that Oracle should have a method for selectively pruning these queries, like it does with ordinary partitions. Indeed, Oracle now enables us to declare the mapping, such that the optimizer then knows how to respond accordingly. This is called a *container map*. In it we create a dummy table with no data, which describes just the partitioning schema, with the requirement that the names of the partitions must match the names of the PDBs.

So let's create an example table in the CDB$ROOT and a few PDBs:

```
SQL> create table dwhdata(region varchar2(2), value number, descr varchar2(20));

Table created.

SQL> alter session set container=PDB1;

Session altered.

SQL> create table dwhdata(region varchar2(2), value number, descr varchar2(20));

Table created.

SQL> insert into dwhdata values ('NZ', 1, 'one NZ');

1 row created.
SQL> commit;

Commit complete.

SQL> alter session set container=PDB2;

Session altered.

SQL> create table dwhdata(region varchar2(2), value number, descr varchar2(20));

Table created.

SQL> insert into dwhdata values ('US', 1, 'one US');

1 row created.

SQL> commit;

Commit complete.

SQL>  alter session set container=PDB3;

Session altered.

SQL>  create table dwhdata(region varchar2(2), value number, descr
varchar2(20));

Table created.

SQL> insert into dwhdata values ('UK', 1, 'one UK');

1 row created.

SQL> commit;

Commit complete.
```

We can now check that there is a row in each of the containers:

```
SQL> alter session set container=cdb$root;

Session altered.

SQL> select * from containers(dwhdata)

RE      VALUE DESCR                CON_ID
-- ---------- -------------------- ------
US          1 one US                    3
UK          1 one UK                    4
CZ          1 one CZ                    5
```

Now let’s create the container map and set it. Note that the container map is global for the root or application container root.

```
SQL> create table partrule
 (region varchar2(2))
  partition by list(region)
   (partition PDB1 values ('US'),
    partition PDB2 values('UK'),
    partition PDB3 values ('CZ'));

Table created.

SQL> alter database set container_map='c##errorquery.partrule';

Database altered.
```

The last step is to enable the table(s) to declare that they should be handled in this special way:

```
SQL> alter table errorlog enable container_map;

Table altered.
```

In this example, we used a list-partitioning schema, but Oracle also allows *RANGE* and *HASH*. The inclusion of a *HASH* schema points to the idea that dividing a large application database into PDBs may be done to split the amount of data and user load, with no further attempt to determine the allocation based on any human-defined condition; *HASH* would simply split the data into PDBs of more or less the same size.

Location Independence

We have seen in Chapter 9 that it's easy to copy and move PDBs from one CDB to another. Usually we just update the Transparent Network Substrate (TNS) connection string after the move to a different server, and the clients will connect to the new location. We mentioned that Oracle can also create a proxy PDB, and you might wonder whether a proxy PDB is really such a big deal. The answer, once we begin working with consolidation queries, is a resolute yes.

The *CONTAINERS()* clause can work on one CDB only; it cannot reference remote PDBs, and there is no syntax to specify such PDBs or database links. However, there is one notable exception: proxy PDB. From the view of the queries, a proxy PDB is a pluggable database like any other, sitting locally on the CDB. The queries are not aware that their requests are actually passed on further, to a remote PDB. In other words, the *CONTAINERS()* clause will gather data from both regular, local PDBs as well as from proxy PDBs.

If we add the fact that the limit of PDBs in a single CDB has been raised to 4096, we can now run queries across many, many PDBs, even if having so many PDBs on a single machine would be practically untenable because of resource requirements.

Cross-Database Replication

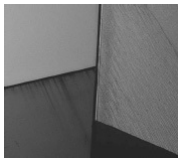
Another option for data sharing is to look at solutions that work across databases. They also usually allow the target to be the same database as source (have it loop back), or at least a different PDB than the source. We

discuss this in further detail in Chapter 13.

Summary

We have discovered that the gamut of the options for data sharing is vast. We can stay entrenched in current thinking and use proven features, such as database links or transportable tablespaces, or we can begin to rethink our approach completely and build our applications on application containers, shared objects, and container maps.

Oracle DBAs are typically conservative, so whether these ideas gain significant traction within this community (let alone among users in general) is yet to be seen apart from a few obvious types for whom these features of multitenant specifically appeal, such as cloud and SaaS providers.



CHAPTER 13

Logical Replication

A database, as the name implies, is a place to store data, but as DBAs, we are often most interested in the physical properties and implementation of the database. For example, we may want to create a backup of the database, create a standby database, or maintain a specific file of an unplugged, pluggable database (PDB). We know that if we understand these mechanisms well, when required, we can create a one-to-one copy of the database or restore the database to the exact same state as before—bit-for-bit. This is a *physical copy*, as the physical structure of the database is maintained. If we then keep this copy in sync with the source by continuously recovering the archive logs, for example, we call it a *physical replication*.

Developers—and their applications—tend to view the database as tables, rows, and columns, and should they want the data to appear somewhere else, they copy these structures, and their contents, again. They are not concerned with datafiles or archive logs, and as long as they see the same data reproduced, they are happy. This is often referred to as a `logical copy`, as only the logical structures, or data, are retained in the process. A `logical replication`, then, keeps such a target in sync with the source via a mechanism such as SQL statement replay.

Physical replication is more robust, because if our source and targets are identical at the bit level, bit-for-bit, then we can be 100-percent sure that the data is also identical. It follows then that in this context we also don’t need to be nervous about different character sets, length semantics, or cross-version feature support—or putting the data into a non-Oracle database.

The other approach, logical replication, offers a higher degree of

flexibility, however, and there are far fewer limitations as to where we can put the data. We are interested in the changes made to the "source" data, and so long as we get an initial copy of this data, followed by all subsequent changes, we will have an identical copy at the target.

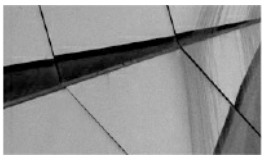
We can identify three major use cases in which the benefits of this method outweigh the disadvantages of its complexity:

- `We can enable databases with mismatched versions to be paired up.` In this way, we can keep a copy of the database in another location, running on a different version and/or platform. A common use case for this is migration, which enables us to perform near-zero-time downtime upgrades or migrations. We create a copy in a new database, keep it in sync, and then switch users over to this new database.

- `We can copy the data to a different database.` This database might be Oracle or some other RDBMS flavor, as required by the application, and the data may be complete or perhaps just a subset. This approach is commonly used to offload reporting from a source database to a secondary, less critical, one or from Oracle to a database such as MySQL, thereby saving on Oracle Database licensing costs. We can also develop distributed applications based on such replication configurations, giving each location its own copy of the data, which is then synchronized with all the others.

- `Once we work at the data level, we are not limited to keeping the data as-is.` For example, we can treat the changes as a stream of events—in other words, see it as a continuous flow of changes made to the tables. This can in turn feed stream-oriented processing flows, using frameworks such as Apache Kafka, Apache NiFi, or Apache Samza, or it can simply be written to data stores, such as Hadoop, where it can be duly processed by custom logic. This opens an additional dimension of data for analysis: evolution over time. Analysts are often interested in how data has changed chronologically. For example, they want to see how a customer has added and removed items from a shopping cart before completing her purchase. So even if a database retains only details of the final order—

the final state—the flow of changes leading up to that point is useful as it depicts how the shopping cart contents changed during the checkout process, which can give insight into customer behavior and the website's effectiveness.



NOTE

Active Data Guard also targets the simple report offloading use case; it is, however, limited to read-only queries and cannot, for example, add indexes, materialized views, or summary tables. It is also limited to an Oracle Database Enterprise Edition as a target. On the other hand, it retains the simpler and more robust physical replication 100-percent copy method.

These use cases for logical replication are not exclusive to multitenant, nor are they in fact even Oracle-specific. However, multitenant is a major architectural change and not all logical replication features and products support it.

Oracle LogMiner

Oracle LogMiner is not a replication solution per se; instead, it extracts changes from the redo logs and displays them in the v$logmnr_contents view.

Other replication products (unless they implement a trigger-based approach) work on the same basis, collecting all the changes as they happen in the redo stream. So it is interesting to play with LogMiner and observe how database changes show up there, because it allows us to see and understand the various challenges and requirements facing the logical replication products. For example, changes are recorded row-by-row, meaning that a large update changing 1000 rows shows as 1000 separate update statements, each affecting exactly one row.

A number of third-party products actually make use of LogMiner, through the exposed PL/SQL API and v$ view. This works fine for basic

operations, but it easily breaks for more complicated operations such as those dealing with large objects (LOBs).

Other products implement their own version of the log miner, completely independent of the Oracle code. These are much harder to implement than using LogMiner, but they enable more flexibility, are not limited only to the information that Oracle makes available in the v$ view, and are more efficient since they avoid calls to the database. The older “classic” version of the GoldenGate extract falls in this category.

Yet another approach is to use triggers, although this technique is less favored, because it increases load on the source database and does not scale well in terms of performance. From an administration point of view, it is also harder to maintain when DDL changes happen.

Obsolete Features

The list of logical replication features and functionality provided by Oracle is surprisingly long, although this has been extensively pruned since the introduction of Oracle Database 12*c*. For the sake of completeness, let’s quickly review features provided for logical replication in Oracle Database 11g, which were not updated to support PDBs.

Oracle CDC

Oracle Change Data Capture, or CDC, was a trigger- or redo log–based logical replication mechanism that was introduced in Oracle Database 9i. It provided a list of all changes, in the form of an event/log table stored in the database itself. But, as was already announced with the earlier Oracle Database 11gR2 release, Oracle CDC is no longer included in future versions and has been omitted from 12*c*.

A number of third-party tools used Oracle CDC in the past as a quick way to support Oracle as a source in logical replication, especially those sorts of applications for which the Oracle Database was not the primary database of interest.

Oracle Streams

Oracle Streams, introduced in Oracle 9iR2, is a redo log–based replication mechanism, which means that it reads the changes recorded in redo logs, instead of relying on changes captured by triggers defined on the table, thus putting no additional load on the database as a result. It then sends the source database changes, which have been harvested, downstream to a different Oracle Database.

Although still available in 12*c*, Streams is now deprecated. However, unlike Oracle CDC, which never gained significant popularity, Streams was adopted by many DBAs and organizations, and thus Oracle has been slower to retire this option.

Nevertheless, Oracle Streams doesn't support any of the new features introduced in Oracle Database 12*c*, which means that PDBs are not supported by it and never will be.

Oracle Advanced Replication

Another casualty of the Oracle Database 12*c* "lay offs" was Advanced Replication, which is a replication method based on updateable materialized views. This feature was formally deprecated in Oracle Database 12.1.

Oracle GoldenGate

It is clear that Oracle regards Oracle GoldenGate as the future of its replication offering. This is the product that is intended to solve all our requirements for logical replication, and that's why Oracle Streams, Oracle CDC, and Advanced Replication are no longer available.

It is true that Oracle GoldenGate is a superior product in many respects, because it is more robust, scales better, and supports databases in addition to Oracle, as both source and targets. On the other hand, it is expensive, whereas Streams and CDC were included for free with the Enterprise Edition. It also lacks support for some Oracle features, but this has rapidly improved since it was re-engineered to use elements of the Streams code for redo extraction.

Last but not least, it was created by a company that was later acquired by Oracle, so the user interface is very different from what we were used to with Streams and CDC. As a product, it is not particularly easy to learn or use, although the new GoldenGate Management Studio GUI is a huge step

forward.

Multitenant Support in Oracle GoldenGate

In GoldenGate, only the new, integrated extract supports multitenant databases. The source extract process is defined at the CDB level and connects to the database as a common user. It can extract data for one or multiple PDBs.

On the target side, each replicat process connects to a single PDB. So if we replicate multiple PDBs on the source, we have to use multiple replicat processes on the target.

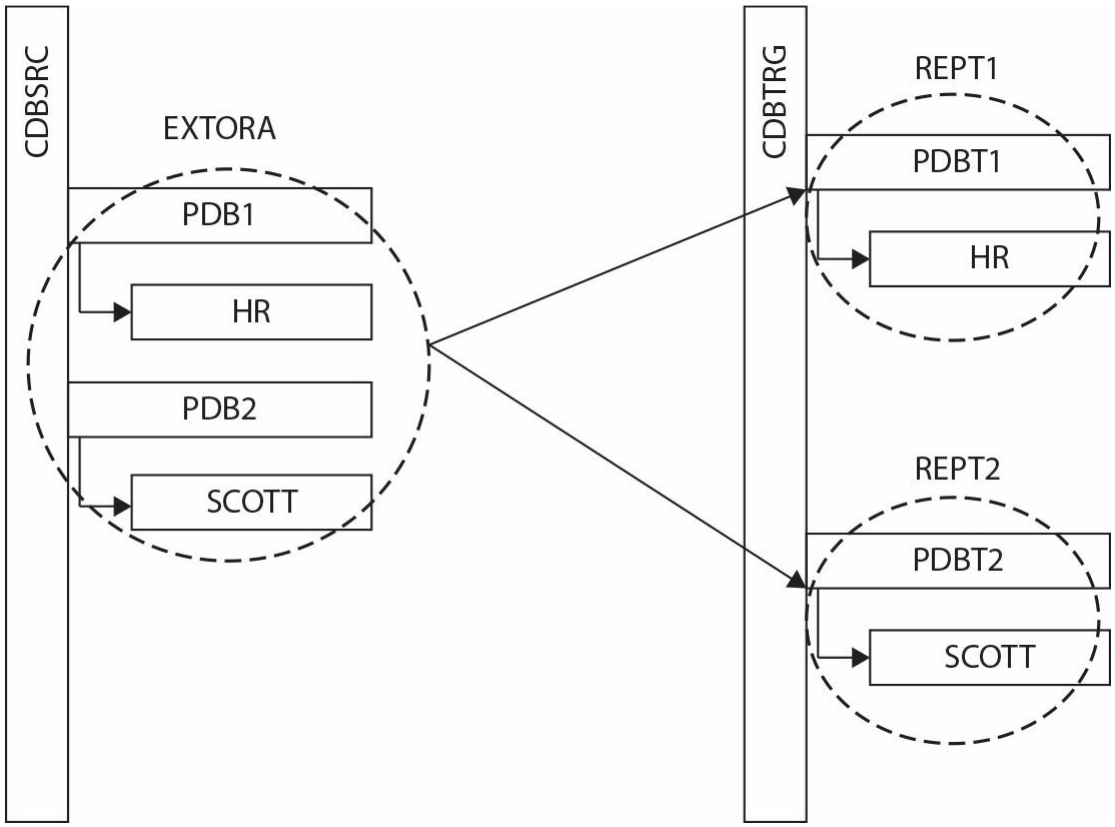
Let's set up a simple replication for a multitenant database and examine the differences that multitenant brings.

Topology in the Example

In this example configuration, the source database has multiple PDBs, but we are interested only in PDB1 and PDB2. On the target database, we have prepared PDBT1 and PDBT2 to hold the data, although there is, of course, no need to have all the PDBs in the same target database, since each has its own replicat process.

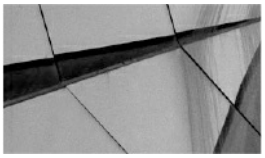
```
SQL> select name, con_id, pdb from v$services where pdb in ('PDB1','PDB2');

NAME           CON_ID PDB
---------- ---------- ------------------------------
pdb1                4 PDB1
pdb2                5 PDB2
```

On PDB1 we will replicate tables from the HR sample schema to PDBT1; on PDB2 we will replicate the SCOTT schema to PDBT2:



Source Database Setup

First of all, we have to tell the database that it's OK to use features that require a GoldenGate license, which assumes that we have actually purchased this license. A parameter, `enable_goldengate_replication`, was added in 11.2.0.4, and GoldenGate checks that this is set, or otherwise errors out, refusing to register the extract. The parameter also enables functionality such as disabling triggers on the target during replication.



NOTE

In 11.2.0.3 and earlier, these features were free to use (included in the database license), before Oracle decided to disallow other third-party tools from using them, by virtue of requiring a GoldenGate license to enable them.

Run this SQL as a common user:

```
SQL> alter system set enable_goldengate_replication=true;

System altered.
```

Every Oracle GoldenGate replication needs a connection to the source database—that is, to the CDB, to be more precise. For this we must grant a couple of additional privileges:

```
SQL> create user C##GGADMIN identified by ggadmin;

User created.

SQL> exec dbms_goldengate_auth.grant_admin_privilege('C##GGADMIN', container=>'ALL');

PL/SQL procedure successfully completed.

SQL> grant dba to c##ggadmin container=all;

Grant succeeded.
```

Every logical replication also needs to have basic supplemental logging enabled, because a row can be chained or migrated—that is, the database can store a row in multiple pieces. Minimal supplemental logging adds information that allows LogMiner, or the logical replication tools, to stitch these pieces together, and this is set at the PDB level:

```
SQL> alter session set container=PDB1;

Session altered.

SQL> alter pluggable database add supplemental log data;

Pluggable database altered.

SQL> alter session set container=PDB2;

Session altered.

SQL> alter pluggable database add supplemental log data;

Pluggable database altered.
```

Table Supplemental Logging

Logical replication also needs supplemental logging on primary keys enabled to ensure that primary key information is `always` written to the redo log. This is necessary for every logical replication tool, because Oracle writes only changed values to redo, by default. That is OK for recovery or physical standbys, as they identify rows by rowid (data block and row position in the block), but logical replication does not preserve rowids and thus needs a primary key, or similar, to uniquely identify the row affected.

In our example, we want to replicate an entire schema, so we use `SCHEMATRANDATA` to set it for all tables in the schema—it would be `TRANDATA` if we wanted to set it for a table only. We do this through the GoldenGate Software Command Interface (GGSCI), the command interface between users and Oracle GoldenGate functional components.

```
[oracle@source ogg]$ ./ggsci

Oracle GoldenGate Command Interpreter for Oracle
Version 12.2.0.1.1 OGGCORE_12.2.0.1.0_PLATFORMS_151211.1401_FBO
Linux, x64, 64bit (optimized), Oracle 12c on Dec 12 2015 02:56:48
Operating system character set identified as US-ASCII.

Copyright (C) 1995, 2015, Oracle and/or its affiliates. All rights reserved.

GGSCI (source) 1> dblogin userid hr@PDB1, password hr
Successfully logged into database PDB1.

GGSCI (source as hr@CDBSRC/PDB1) 2> add schematrandata PDB1.hr

2016-04-04 17:51:16  INFO    OGG-01788  SCHEMATRANDATA has been added on schema hr.

2016-04-04 17:51:16  INFO    OGG-01976  SCHEMATRANDATA for scheduling columns has
been added on schema hr.

GGSCI (source as hr@CDBSRC/PDB1) 3> dblogin userid scott@PDB2, password tiger
Successfully logged into database PDB2.

GGSCI (source as scott@CDBSRC/PDB2) 4> add schematrandata pdb2.scott

2016-04-04 17:52:37  INFO    OGG-01788  SCHEMATRANDATA has been added on schema scott.

2016-04-04 17:52:37  INFO    OGG-01976  SCHEMATRANDATA for scheduling columns has been
added on schema scott.
```

Configure and Start Manager

The next step is simple: let GGSCI create all necessary directories and start the manager processes.

First, the directories:

```
GGSCI (source) 2> create subdirs

Creating subdirectories under current directory /home/oracle/ogg
Parameter files                /home/oracle/ogg/dirprm: created
Report files                   /home/oracle/ogg/dirrpt: created
Checkpoint files               /home/oracle/ogg/dirchk: created
Process status files           /home/oracle/ogg/dirpcs: created
SQL script files               /home/oracle/ogg/dirsql: created
Database definitions files     /home/oracle/ogg/dirdef: created
Extract data files             /home/oracle/ogg/dirdat: created
Temporary files                /home/oracle/ogg/dirtmp: created
Credential store files         /home/oracle/ogg/dircrd: created
Masterkey wallet files         /home/oracle/ogg/dirwlt: created
Dump files                     /home/oracle/ogg/dirdmp: created
```

Next, we configure the manager parameter file:

```
GGSCI (source) 3> edit params mgr
```

Then, in the editor, we simply enter the following:

```
PORT 7809
```

Back in GGSCI, we can now start the manager:

```
GGSCI (source) 4> start manager
```

And we also need to repeat these steps for the target server.

Extract Configuration File

Now we configure the extract process to read from the source database redo:

```
GGSCI (source) 5> edit params extora
```

In our example, we will specify only the basic settings:

```
EXTRACT EXTORA
USERID c##ggadmin@CDBSRC, PASSWORD ggadmin
RMTHOST localhost, MGRPORT 7809
RMTTRAIL ./dirdat/rt

DDL INCLUDE MAPPED
LOGALLSUPCOLS
UPDATERECORDFORMAT COMPACT

TABLE pdb1.hr.*;
TABLE pdb2.scott.*;
```

Because this is not a GoldenGate handbook, we have simply referred to the original product documentation for explanation on the settings as well as the steps to encrypt the password. However, notice the second and the last two lines, where you can see that the extract process connects to the container database (CDB) as a common user. In addition, we specify the tables to be replicated in a single configuration file, specifying the names with three parts —PDB, schema, and table.

Now let’s register the extract process just defined and configure the remote trail:

```
GGSCI (source) 1> dblogin userid c##ggadmin@CDBSRC, password ggadmin
Successfully logged into database CDB$ROOT.

GGSCI (source as c##ggadmin@CDBSRC/CDB$ROOT) 2> register extract extora database
container (PDB1,PDB2);

2016-04-04 18:05:11  INFO    OGG-02003  Extract EXTORA successfully registered with
database at SCN 1894436.

GGSCI (source as c##ggadmin@CDBSRC) 3> add extract extora, integrated tranlog, begin now
EXTRACT (Integrated) added.


GGSCI (source as c##ggadmin@CDBSRC) 4> add rmttrail ./dirdat/rt, extract extora,
megabytes 100

RMTTRAIL added.
```

Set Up the Target Database

Let’s now move on to the target database. Unlike the extract, a replicat process connects to one specific target PDB. We have two PDBs replicated, so we have to create users in both of these databases. First of all, we must enable the “magic parameter” as we did for the source database, as follows:

```
SQL> alter system set enable_goldengate_replication=true;

System altered.

SQL> alter session set container=PDBT1;

Session altered.

SQL> create user repuser identified by rep_pass container=current;

User created.

SQL> grant dba to repuser;

Grant succeeded.

SQL> exec dbms_goldengate_auth.grant_admin_privilege('REPUSER',container=>'PDBT1');

PL/SQL procedure successfully completed.
SQL> alter session set container=PDBT2;

Session altered.

SQL> create user repuser identified by rep_pass container=current;

User created.

SQL> grant dba to repuser;

Grant succeeded.

SQL> exec dbms_goldengate_auth.grant_admin_privilege('REPUSER',container=>'PDBT2');

PL/SQL procedure successfully completed.
```

Configure Parameter Files for Replicat Processes

Now let’s configure two replicat processes—one for each PDB:

```
GGSCI (target) 1> edit params rept1
REPLICAT REPT1
ASSUMETARGETDEFS
DISCARDFILE ./dirrpt/repora1.dsc, PURGE, MEGABYTES 100
DDL INCLUDE MAPPED
DBOPTIONS INTEGRATEDPARAMS(parallelism 6)
USERID repuser@pdbt1, PASSWORD rep_pass
MAP pdb1.hr.*, TARGET pdbt1.hr.*;
```

And here's the second one:

```
GGSCI (target) 2> edit params rept2
REPLICAT REPT2
ASSUMETARGETDEFS
DISCARDFILE ./dirrpt/repora2.dsc, PURGE, MEGABYTES 100
DDL INCLUDE MAPPED
DBOPTIONS INTEGRATEDPARAMS(parallelism 6)
USERID repuser@pdbt2, PASSWORD rep_pass
MAP pdb2.scott.*, TARGET pdbt2.scott.*;
```

Now we can register the replicat processes:

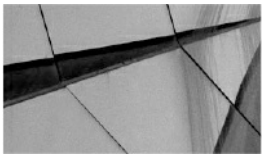
```
GGSCI (target) 3> add replicat rept1, integrated, exttrail ./dirdat/rt
REPLICAT (Integrated) added.

GGSCI (target) 4> add replicat rept2, integrated, exttrail./dirdat/rt
REPLICAT (Integrated) added.
```

Initial Extract

As with every replication, we must provision the source data to the target to create a baseline, so they match as of a specific starting point. Let's set this starting point to the current database system change number (SCN):

```
SQL> SELECT current_scn from v$database;

CURRENT_SCN
-----------
    1939272
```

We use GoldenGate's initial extract for the provisioning, which means we ask GoldenGate to load the initial data as a set of inserts, obtained by a select into the source database.



NOTE

We have manually created empty tables at the target databases prior to this—for example, using metadata only Data Pump import.

```
GGSCI (source) 1> edit params initext1
```

Let's use two extracts, as an example to demonstrate that we are not limited to a single extract in a CDB:

```
EXTRACT INITEXT1
USERID c##ggadmin@cdbsrc, PASSWORD ggadmin
RMTHOST localhost, MGRPORT 7809
RMTTASK REPLICAT, GROUP INITREP1
TABLE pdb1.hr.*, SQLPREDICATE 'AS OF SCN 1939272';
```

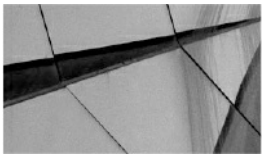
And similarly, for the second extract:

```
GGSCI (source) 2> edit params initext2
```

We use the SCN in the predicate for the tables, effectively turning this

into a flashback query:

```
EXTRACT INITEXT2
USERID c##ggadmin@cdbsrc, PASSWORD ggadmin
RMTHOST localhost, MGRPORT 7809
RMTTASK REPLICAT, GROUP INITREP2
TABLE pdb2.scott.*, SQLPREDICATE 'AS OF SCN 1939272';
```



NOTE

The TABLE clause can also accept a VIEW. This does not matter for regular extract processing, although a VIEW generates no redo, so no changes will ever be captured. However, for an initial extract, in which data is queried using a select, the data would be extracted, and replicat would fail, because it can't insert into that VIEW. If it could, we would otherwise end up with primary key violations or duplicate data in the underlying table.

Let's configure the replicat processes now:

```
GGSCI (target) 1> edit params initrep1
```

There is nothing special about the configuration in this file:

```
REPLICAT INITREP1
ASSUMETARGETDEFS
DISCARDFILE ./dirrpt/init1.dsc, APPEND
USERID repuser@PDBT1, PASSWORD rep_pass
MAP pdb1.hr.*, TARGET pdbt1.hr.*;
```

Again, the second file differs from the first only in the replicat and

filenames, as well as login and tables:

```
GGSCI (target) 2> edit params initrep2
REPLICAT INITREP2
ASSUMETARGETDEFS
DISCARDFILE ./dirrpt/init2.dsc, APPEND
USERID repuser@PDBT2, PASSWORD rep_pass
MAP pdb2.scott.*, TARGET pdbt2.scott.*;
```

Finally, we can register the processes:

```
GGSCI (source) 1> add extract initext1, sourceistable
EXTRACT added.

GGSCI (source) 2> add extract initext2, sourceistable
EXTRACT added.
```

And then also on the target:

```
GGSCI (target) 1> add replicat initrep1, specialrun
REPLICAT added.

GGSCI (target) 2> add replicat initrep2, specialrun
REPLICAT added.
```

Run the Initial Extract

Now we run the provisioning load:

```
GGSCI (source) 1> start extract initext1

Sending START request to MANAGER ...
EXTRACT INITEXT1 starting

GGSCI (source) 2> start extract initext2

Sending START request to MANAGER ...
EXTRACT INITEXT2 starting
```

We verify that both ran successfully, and through to completion:

```
GGSCI (5212e334b10e) 71> info extract initext1

EXTRACT    INITEXT1  Last Started 2016-04-04 19:20   Status STOPPED
Checkpoint Lag       Not Available
Log Read Checkpoint  Table PDB1.HR.REGIONS
                     2016-04-04 19:20:55  Record 4
Task                 SOURCEISTABLE
```

In the detailed report, which is too long to list here, we verify there were no errors:

```
GGSCI (5212e334b10e) 72> view report initext1
…
```

Obviously, we need to perform the same check for the three other processes.

Start Extract and Replicats

Now, with the data loaded, we are at the place where can start the replication. Note, however, that we could actually start the extract much sooner, because only the replicats have to wait for all the data to be loaded:

```
GGSCI (source) 1> start extract extora

Sending START request to MANAGER ...
EXTRACT EXTORA starting

GGSCI (5212e334b10e) 110> info extract extora

EXTRACT    EXTORA    Last Started 2016-04-04 19:34   Status RUNNING
Checkpoint Lag       00:00:10 (updated 00:00:03 ago)
Process ID           31008
Log Read Checkpoint  Oracle Integrated Redo Logs
                     2016-04-04 19:34:26
                     SCN 0.1951977 (1951977)
```

We then start the replicats:

```
GGSCI (target) 1> start replicat rept1, aftercsn 1939272;


Sending START request to MANAGER ...
REPLICAT REPT1 starting

GGSCI (target) 2> start replicat rept2, aftercsn 1939272;


Sending START request to MANAGER ...
REPLICAT REPT2 starting
```

Finally, we check that the processes are up and running correctly:

```
GGSCI (target) 3> info replicat rept1

REPLICAT   REPT1     Last Started 2016-04-04 19:29   Status RUNNING
INTEGRATED
Checkpoint Lag       00:00:00 (updated 00:00:09 ago)
Process ID           30825
Log Read Checkpoint  File ./dirdat/rt000000000
                     2016-04-04 19:29:55.118374  RBA 0

GGSCI (target) 4> info replicat rept2

REPLICAT   REPT2     Last Started 2016-04-04 19:30   Status RUNNING
INTEGRATED
Checkpoint Lag       00:00:00 (updated 00:00:00 ago)
Process ID           30911
Log Read Checkpoint  File ./dirdat/rt000000000
                     2016-04-04 19:30:45.269029  RBA 0
```

Big Data Adapters

Oracle GoldenGate now supports various Big Data or event streaming targets, and these adapters are available for Hadoop HDFS, HBase, Hive, Flume, Kafka, and Spark. Collectively, this is marketed as Oracle GoldenGate for Big Data. A complete example would span many pages, so let's succinctly summarize how to use these Big Data adapters.

On the source side, nothing really changes, and we still have to define an extract process to get the data. For the target side, there is still a replicat process, but it needs to be pointed to a Java class and parameter file.

The easiest way is to start with the example for Kafka (Apache Kafka is fast, scalable, and durable publish-subscribe messaging, rethought as a distributed commit log, and distributed by design) provided in AdapterExamples/big-data/kafka directory in the Big Data adapters installation. We won't go into detail on Kafka here; we recommend that you visit the Apache Kafka project web page for more details, documentation, and use cases. Here, we set up a very basic example.

We can use the example replicat definition right away, just changing the tables to be included in the replication:

```
REPLICAT rkafka
-- Trail file for this example is located in "AdapterExamples/trail" directory
-- Command to add REPLICAT
-- add replicat rkafka, exttrail AdapterExamples/trail/tr
TARGETDB LIBFILE libggjava.so SET property=dirprm/kafka.props
REPORTCOUNT EVERY 1 MINUTES, RATE
GROUPTRANSOPS 10000MAP PDB1.hr.*, TARGET PDB1.hr.*;
```

The kafka.props file referenced here is where all the Kafka-specific settings are stored. Again, we can start off with the supplied example, although we will want to change at least the topic (stream of events describing changes on a table; in this example we put all source tables to a single event stream) and schema topic (event stream where replicat puts Avro schemas for the replicated tables) names, and possibly expand the classpath to include Avro libraries (if we go with the Avro format, as set in the example file):

```
gg.handlerlist=kafkahandler
gg.handler.kafkahandler.type=kafka
gg.handler.kafkahandler.KafkaProducerConfigFile=custom_kafka_producer.properties
gg.handler.kafkahandler.TopicName=oggtopic
gg.handler.kafkahandler.format=avro_op
gg.handler.kafkahandler.SchemaTopicName=mySchemaTopic
gg.handler.kafkahandler.BlockingSend =false
gg.handler.kafkahandler.includeTokens=false

gg.handler.kafkahandler.mode =tx
#gg.handler.kafkahandler.maxGroupSize =100, 1Mb
#gg.handler.kafkahandler.minGroupSize =50, 500Kb


goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE

gg.log=log4j
gg.log.level=INFO

gg.report.time=30sec

gg.classpath=dirprm/:/var/lib/kafka/libs/*:

javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=ggjava/ggjava.jar
```

This configuration in turn references one more configuration file: the description of the Kafka server we want to connect to. The only reference we need to change in this custom_kafka_producer.properties file is the first line —bootstrap servers (Kafka servers that provide information on the Kafka cluster configuration; in the simple case of a standalone server, just specify this server):

```
bootstrap.servers=host:port
acks=1
compression.type=gzip
reconnect.backoff.ms=1000

value.serializer=org.apache.kafka.common.serialization.ByteArraySerializer
key.serializer=org.apache.kafka.common.serialization.ByteArraySerializer
# 100KB per partition
batch.size=102400
linger.ms=10000
```

As discussed earlier, it is a simple process to register the replicat, as follows:

```
GGSCI (target) 1> add replicat rkafka, exttrail dirdat/or, begin now
REPLICAT added.
```

There is much more to be configured and set up for various use cases; see Oracle Golden Gate for Big Data documentation.

Oracle XStream

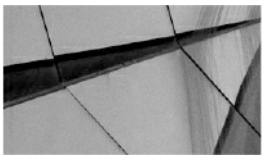
As we mentioned previously, Oracle GoldenGate now uses the `integrated extract`. This code, which sits in the Oracle executable itself, is based on LogMiner and Oracle Streams original code, further developed to satisfy the needs of Oracle GoldenGate and to accommodate all the new features.

However, since the Oracle GoldenGate extract and the Oracle Database executables are separate processes, Oracle had to introduce an API to enable them to talk to each other.

Oracle decided to make this API public and called it XStream. This API consists of PL/SQL packages and v$ views for management. For the actual processing of changes, there are Oracle Call Interface functions for C, along with a Java API. Logically, the API is split in two: one for data coming out of the database (“extract” in GoldenGate, XStream Out) and one for data coming into the database (“replicat” in GoldenGate, XStream In).

Oracle also decided that since all the code for GoldenGate for Oracle

source databases is contained within XStream, we need an Oracle GoldenGate license to use it. This relegates the use of XStream to special use cases that GoldenGate does not directly support, meaning that it is anything but a “limited, but free” solution like the original Streams offering was.



NOTE

The XStream API is indeed an API, meaning it’s intended for application programmers, not for database administrators. You need to write a C or Java program to make use of it.

Logical Standby

In Chapter 11, we covered Oracle Data Guard physical standby, ignoring at that time the fact that there is also another type available: the often forgotten logical standby.

Conceptually, this is similar to Streams or GoldenGate, in that redo is mined for changes, and these changes are then applied to the target using SQL statements. This implies similar limitations to those for Streams and GoldenGate, including limited datatype support and no duplicate rows in tables. Unlike Streams, however, logical standby is not deprecated, and unlike GoldenGate, it does not require any additional licensing. In reality, though, even logical standby is destined to be replaced by GoldenGate—with one notable use case exception, as you will see.

One special aspect of a logical standby to be aware of is that it is instantiated from a physical standby—that is, its setup begins with the same steps used for a physical standby, including the selection of PDBs to include in the standby, and the same steps also apply for including PDBs created later. In having been generated from a physical standby, it is always built at the CDB level and starts out with all PDBs, with one notable exception: application containers are skipped. Generally, we can’t convert a logical standby back to a physical one, because the two are no longer one-to-one copies of each other. However, in the special case of an upgrade (see the next section), it is indeed possible.

This instantiation from a physical standby is both a curse and a blessing. On the positive side of the ledger, it is very easy to do it right and to be 100-percent sure that all the source data is initially copied correctly to the target. On the other hand, though, it imposes the same restrictions that a physical standby does: it requires the same Oracle versions (when the logical standby is created) and platform compatibility.

Use in Upgrade

The unique feature of a logical standby, and the use case that has become its point of difference, is its function in a rolling upgrade. In this scenario, a logical standby is automatically converted from a physical standby. Oracle is then upgraded on this logical standby and it becomes the new primary, and then the other database in the configuration is upgraded. This makes the upgrade much simpler than it would have been with GoldenGate—we don't need to rebuild any database after the upgrade is completed, because logical standby can convert back to physical standby in this case.

Oracle Database 12.1 introduced the DBMS_ROLLING package that, along with Active Data Guard, enables automation of such an upgrade process and can be especially beneficial if there are multiple standby databases in the configuration.

As for a multitenant database, not much is different, although in this special case, application containers are supported. Note, however, that an upgrade of an application container cannot happen at the same time a rolling upgrade is performed, and vice versa.

To ensure that there is no data loss during this process, all PDBs must be plugged in and opened when the transient logical standby becomes the new primary.

Other Third-Party Options

As mentioned, other companies provide alternatives to Oracle GoldenGate. In general, they lag behind Oracle GoldenGate in terms of features and performance, but they provide competitive alternatives by majoring on ease-of-use and price.

Dbvisit Replicate

Dbvisit Replicate, not to be confused with Dbvisit Standby (which is similar to physical standby), is a logical replication tool that uses its own implementation of a redo parsing engine. Only Oracle Database as source is supported.

PDBs are supported. The mine process connects to a single PDB at the source and replicates this sole PDB. Replication of multiple or all PDBs at the same time using a single replication is not supported as of the time of writing.

Although not as feature-rich as Oracle GoldenGate, Dbvisit Replicate's strength lies in its ease-of-use. As a point of comparison, the example we went through earlier to configure GoldenGate requires significantly less manual work. In fact, most of the steps are performed automatically by a single script that the Replicate Setup Wizard creates, after asking a handful of simple questions about your databases and the tables/schemas you want replicated. Unfortunately, there is no GUI available at this point in time.

Replicate also supports Event Streaming mode, producing a stream of changes, similar to the output Oracle CDC produced. Using this functionality, or published APIs, it supports for example Apache Kafka and Apache NiFi targets.

Dell SharePlex

Dell SharePlex, known as Quest SharePlex before Dell acquired Quest Software, is another alternative to Oracle GoldenGate. Again, it is more cost-effective than GoldenGate, but it also supports only the Oracle Database as a source.

Since late 2014, SharePlex has supported PDBs, and each of the PDBs requires its own configuration and capture process.

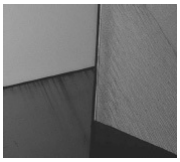
The configuration is text-based, but the SharePlex Manager does offer a basic GUI for configuration and monitoring.

Summary

We have only briefly touched on the merits of logical replication. To be fair,

it is a topic that warrants a book in its own right, and indeed, there are already multiple books on Oracle GoldenGate alone. With the advent of Big Data and event streaming, these tools are even more important to consider in our environments.

Oracle Multitenant is the direction that the Oracle Database is going in, so all logical replication tools need to support it. Unfortunately, this has become a stumbling block for many of the features we were familiar with prior to 12*c*, and the Oracle-recommended replacement is an expensive one.



Index

Please note that index links point to page beginnings from the print edition. Locations are approximate in e-readers, and you may need to page down one or more times after clicking a link to get to the indexed material.

A

AaaS (Application as a Service), Oracle Database 12c, 5

ACFS (ASM Clustered File System), 241

Active Data Guard

- automating upgrades, 363
- changing UNDO mode, 223
- cloning local PDB, 240
- enabling read-only database in recovery, 291
- physical replication 100-percent copy, 347
- simple report offloading, 347

Add Standby Database wizard, Cloud Control, 304–305

`administer key management`

- creating keystore, 160–161
- creating master key, 162–163
- opening keystore in TDE, 162

`ADMINISTER_RESOURCE_MANAGER` system privilege, 263

administration. *See also* DBA (database administrator)

- creating CDB using DBCA GUI, 39
- multitenant. *See* day-to-day management
- root container database link only for, 321

Advanced Configuration, creating CDB with DBCA GUI, 41–42

`AFTER CLONE` trigger, 252–253

`AFTER DB_ROLE_CHANGE` trigger, CDB-only, 252

`AFTER LOGON ON PLUGGABLE DATABASE`, 26

`AFTER SET CONTAINER ON PLUGGABLE DATABASE`, 26

AFTER SET CONTAINER trigger, 26, 252
alert logs, 121–122, 184
ALL option, ALTER LOCKDOWN PROFILE, 156
ALTER DATABASE, modifying PDB, 98
ALTER LOCKDOWN PROFILE, 155–156
ALTER PLUGGABLE DATABASE CLOSE, 93–94, 96
ALTER PLUGGABLE DATABASE OPEN, 93–94, 96
ALTER PLUGGABLE DATABASE SAVE STATE, 95, 96
ALTER SESSION SET CONTAINER trigger, 23, 26, 252
ALTER SESSION SET CURRENT_SCHEMA, 23
ALTER SYSTEM
    changing parameters in SPFILE, 113
    disabling with lockdown profiles, 156–157
    limiting PDB I/O parameter, 283
    PDB memory allocation, 282
    PDB SPFILE equivalent, 114
    single-tenant security lockdown, 75–76
ALTER SYSTEM KILL SESSION, thread processes, 125
ALTER SYSTEM RESET, removing persistent setting for parameter, 116
ALTER SYSTEM SET
    administer key management replacing, 161
    CDB SPFILE, 114
    lockdown profile, 157
ALTER USER
    administering temporary tablespaces, 92
    authorizing container access, 151
Amazon Web Services (AWS), back up to cloud, 195
AMM (Automatic Memory Management), 281
APEX (Oracle Application Express), 30–31, 107
application containers
    considerations, 247
    linking tables across, 330–338
    prefix for, 80–83, 151

application metadata
separating system metadata from, 18–19
vs. system metadata, 13–14
application servers, Oracle Database 11*g*, 5
`APPLICATION_USER_PREFIX`, application containers, 151
`APPOWNER`, proxy users, 154
archive files, plug-in/unplug PDB, 238–239
`archive log list`, hot backups, 172
archive logs
backup schedule for, 183
in full CDB backups, 176
full CDB recovery with, 185
hot backups generating, 171–172
PDB backup restrictions, 183
ARCHIVELOG mode
cloning local PDB, 241–242
cloning remote PDB, 242
enabling `FLASHBACK ON`, 214
flashback logging, 215
hot backups, 171–172
hot cloning remote PDB, 241–242
modifying entire CDB, 90
arguments, creating PDB with catcon.pl, 68–69
`AS APPLICATION CONTAINER` clause
application containers, 80
`CREATE PLUGGABLE DATABASE` with, 57
linking tables across containers, 330
`AS CLONE` clause
`CREATE PLUGGABLE DATABASE` with, 57
deciding if PDB should be on standby, 310
plugin PDB as clone, 237–238
ASM (Automatic Storage Management)
creating CDB using DBCA GUI, 39

creating CDB using OMF, 44–45
Oracle Managed Files used with, 37
recommended for database files, 36
ASM Clustered File System (ACFS), 241
ASMM (Automatic Shared Memory Management), 281
atomicity of transactions, PITR of UNDO tablespace, 200
auditing of log in, as proxy user, 153–154
Automatic Memory Management (AMM), 281
Automatic Shared Memory Management (ASMM), 281
Automatic Workload Repository (AWR), collecting statistics, 31–32
autotask directive, CDB resource plan, 268, 270–273
auxiliary instance
PDBPITR in local UNDO mode, 212
point-in-time recovery, 205–206
restore/recover PDB with RMAN, 202–205
availability
multitenant option, 80
single-tenant consolidation, 76
Availability menu, creating standby in Cloud Control, 304–308
AWR (Automatic Workload Repository), collecting statistics, 31–32
AWS (Amazon Web Services), back up to cloud, 195

B

backup
of archive logs, 183
block corruption check, 193–194
CDB recovery and, 175–180
before CDB upgrade, 103–104
to cloud, 195–196
Cloud Control for, 194–195
creating physical standby for, 291–292
hot vs. cold, 170–172
overview of, 170

PDB, 175–176, 180–183
and recovery. *See* recovery
RMAN backup redundancy, 174
RMAN default configuration, 173–174
RMAN optimization, 189–193
of source to standby with RMAN, 293
SYSBACKUP for, 174–175
`backup database` commands, 173–174, 181
`BACKUP OPTIMIZATION ON`, consolidating single-tenant CDB, 77
`BEFORE SET CONTAINER` trigger, 26, 252
`BEFORE UNPLUG` trigger, 252–253
Big Data adapters, 359–361
`BIGFILE/SMALLFILE` datafile, 92, 98
block change tracking, RMAN, 191–192
block corruption checks, 193–194
BPU (Bundle Patch Update), 112

C

case-insensitivity, common user prefix, 151
case-sensitivity, `DB_UNIQUE_NAME` clause, 118
`catalog` command, PDB recovery on standby, 313
catalog options, creating CDB in SQL*Plus, 53
catcon.pl script
creating CDB in SQL*Plus, 50–51
creating PDB, 67–69
pre-upgrade check for CDB, 102
recompiling invalid objects for CDB, 55
running preupgrade script, 104–105
catctl.pl utility, upgrades, 105–106
catoctk.sql script, creating CDB, 53
catproc.sql script, creating CDB, 53
catupgrd.sql, upgrading CDB, 105–107
CDB backup and recovery

full backups, 176–178
overview of, 175–176
partial backups, 178–179
reporting using RMAN, 179–180
restore and recovery, 184–187
CDB (container database)
avoid putting all databases in same, 79–80
cloning local PDB, 240–241
connecting to containers, 23–26
creating common users for admin, 140–141
dictionary views, 26–27
dropping, 90
files common to all containers, 27–30
holding multiple pluggable databases, 14–16
identifying containers, 20–23
managing, 89–92
managing physical standby, 308–314
managing resources in Resource Manager, 264
modifying entire, 90–91
as multitenant database, 14
multitenant dictionaries and, 16–20
PDB level parameters vs., 113–118
recovery scenarios, 183–187
removing running database, 47
Resource Manager requirements, 263
single-tenant, 74–79
startup and shutdown, 90–91
unplugging PDB/plugging into different, 230–231
upgrades. *See* upgrades, CDB
CDB, creating
Oracle Managed Files and, 36–37
overview of, 36
using `CREATE DATABASE` or `DBCA`, 90
using DBCA CLI, 42–44

using DBCA GUI, 37–42
using one PDB (FS and non-OMF), 46–47
using SQL*Plus, 47–55
using two PDBs (ASM and OMF), 44–45
CDB resource plan, Resource Manager
creating, 269–273
creating with PDB profiles, 275–277
default and autotask directives, 267–268
defined, 262
enabling/removing, 274–275
managing, 273–274
overview of, 265
removing directive for, 275
removing directive for PDB profile, 277
resource allocation and utilization limits, 265–267
viewing, 285
CDB SPFILE, 114
CDB_PDB_HISTORY view, 97
CDB$ROOT
block change tracking, 191
changing UNDO mode, 211
common users, local users and, 140
complete recovery of, 185–186
connecting to CDB via, 89–90
creating CDB, 54, 90
creating common users, 144–146
creating database with local UNDO, 208
data links, 20
Data Recovery Advisor run only from, 193
dictionary views from, 17–18, 26–27
full CDB backup of, 176–177
full PDB backup of, 180–181
granting common privileges, 147–148
identifying containers, 22–23

metadata links, 18–19
modifying entire CDB, 90–91
modifying root container, 92
multitenant dictionaries and, 16–17
overview of, 15–16
partial backup details, 179–180
partial PDB backup from, 182
recompiling invalid objects, 55
recovering from lost datafile, 186
recovering from lost tablespace, 186
recovering PDB nonsystem datafile, 188
recovering PDB system datafile, 188
recovering PDB tablespace, 188–189
recovering PDB until time, 203–204
in Resource Manager, 284–285
restore points at CDB/PDB levels, 216–218
restricting access to container data, 151–153
results of querying from, 325, 328
specifying DATABASE for backups, 175
UNDO for all transactions in, 211
CDB_views, common users, 151
CDC (Change Data Capture), 348
changed blocks, incremental backups, 189–191
CLI (command-line interface)
creating CDB with DBCA, 42–44
creating PDB with DBCA, 65–66
CLONEDB initialization parameter, snapshots, 241
cloning
copying database by, 228
with Delphix, 324
non-CDB, 250–251
to Oracle Cloud in OEM, 251–252
overview of, 239–240
plug in a PDB as, 237–238

remote. *See* remote clone
with TDE, 164–165
close state, PDBs, 93–94
cloud
backup options for, 195–196
consolidating PDBs, 14–15
moving PDBs to, 251–252
Oracle Database 12c, 5
running standby in, 315–318
Cloud Control
backup options, 194–195
creating CDB resource plan, 271–273
creating database link, 321
creating PDB, 66–67
creating physical standby, 29s
creating standby, 304–308
monitoring PDBs managed by Resource Manager, 285
monitoring Resource Manager, 284
opening/closing PDBs, 95–96
setting up TDE, 161
transportable tablespaces, 322–323
Cloud Services, back up to Oracle, 195–196
cluster database, opening PDB in, 96
Codd's rules, 11
cold (consistent) backups, 170–172, 185
columns, TDE encrypting all table, 163–164
command-line interface (CLI)
creating CDB with DBCA, 42–44
creating PDB with DBCA, 65–66
commands, pluggable databases accepting all, 15
`COMMENT` clause, for permanently changed parameters, 115
common roles, creating, 153
common users

avoid mixing local users in same namespace as, 144
conflict resolution between local and, 148–150
connecting through local proxy user to, 153–154
creating with CONTAINER=ALL option, 144–146
granting common privileges to, 147–148
granting local privileges to, 146–147
keeping conflict resolution clear and simple, 150–151
querying V$ views for information, 151
restricting access to container data, 151–153
stored in every single PDB, 140–141
COMMON_USER_PREFIX
for all common user names, 144–145
keeping clear and simple, 151
compatibility checks
cloning remote PDB, 242
converting non-CDB by plugging in, 248–250
running, 236
compatibility, plug-in, 235–237
COMPATIBLE parameter
multisection incremental backups, 192–193
upgrade CDB with catupgrd.sql, 105
compressed backup set output, full CDB backups, 176–177
CON_DBID, identifying pluggable database, 31
configuration file, extract process in GoldenGate, 353–354
conflict resolution, 148–151
CON_ID, identifying containers, 20–22, 31
CONNECT role, creating new PDB from PDB$SEED, 59
connection
to application containers, 81
basics of traditional database, 120–121
to CDB, via root container, 89–90
to containers, 23–26
default services and PDB, 125–129

connection brokers. multithreaded networking, 124
CONNECTION_BROKERS database parameter, 124
consistent (cold) backups, 170–172, 185
consolidated data
    container map, 339–342
    containers default, 339
    cross-PDB DML, 338–339
    linking tables across containers, 330–338
    location independence, 342–343
    sharing data across PDBs, 329–330
consolidated servers, 5
consolidation
    data/metadata at CDB level, 30–33
    files common to all CDBs, 27–30
    history of database use, 5
    managing multiple databases with one instance, 10
    with multitenant option, 14–15, 83, 123–124
    pros and cons of, 11
    road to multitenant architecture, 5–6
    schema. *See* schema consolidation
    server, 9–10
    single-tenant CDBs, 76–77
    single-tenant vs. multitenant options, 83
    summary of strategies for, 10–11
    table, 9
consumer groups
    creating PDB resource plan, 277–280
    mapping, 277
container map, sharing data across PDBs, 339–342
CONTAINER=ALL clause
    changing parameter for all PDBs, 116–117
    CREATE USER/CREATE ROLE option, 142
    creating master key, 163

opening keystore in TDE, 162
CONTAINER=CURRENT clause, 142, 146–147
CONTAINER_DATA, restricting common user access to, 151–153
containers
application, 80–83
CDB$ROOT. *See* CDB$ROOT
choosing, 88–89
connecting to, 23–26
dictionary views from, 26–27
identifying, 20–23
linking tables across, 330–338
multitenant, 14
Oracle database vs. lightweight, 6
PDB$SEED, 16
separating system/application metadata, 13–14
setting trigger, 26
CONTAINERS() clause
cross-PDB DML, 338–339
cross-PDB views, 324–325
default value in grant statement, 148
ENABLE CONTAINERS_DEFAULT vs., 339
gathering data from local/proxy PDBs, 342–343
querying own user tables, 325–329
containers default, sharing data across PDBs, 339
control files
common to all containers in CDB, 27–28
in full CDB backups, 176
modifying entire CDB, 90
CON_UID, identifying containers, 21–23
CONVERTING status, non-CDBs, 21
COPIES clause, RMAN backup redundancy, 174
copy
database. *See* cloning

database to different database with logical replication, 346–347
password file for standby with RMAN, 293–295
copy on write, sharing data across PDBs, 323–324
CPU (Critical Patch Update), 112
CPU usage of PDBs, Resource Manager limiting, 261
Create A Database option, DBCA GUI, 38–39
Create As Container Database option, DBCA GUI, 40–41
CREATE_ CDB_PROFILE_DIRECTIVE, 276
CREATE DATABASE
creating CDB, 51–52
in local UNDO mode, 207–208
managing CDB, 90
CREATE PFILE, 115, 294
CREATE PLUGGABLE DATABASE
connecting to containers, 23
creating new PDB from PDB$SEED, 56–59
creating PDB in SQL*PLUS, 55
locating new PDB created with clone/plug-in, 247
relocating database, 246
CREATE ROLE, 142
CREATE SESSION, privileges, 147–148
CREATE USER
as common user with CONTAINER=ALL, 144–146
as local user with CONTAINER=CURRENT, 142–143
overview of, 142
CREATE_CDB_PLAN_DIRECTIVE, 274
CREATE_CONSUMER_GROUP, 277, 280
CREATE_FILE_DEST, 159
CREATE_PLAN, 277, 280
CREATE_PLAN_DIRECTIVE, 277, 280
credentials, Direct NFS, 241
Critical Patch Update (CPU), 112
cross-database replication, data sharing with, 343

cross-PDB DML, 338–339
cross-PDB views
consolidated data, 329–330
container map, 339–342
containers default, 339
cross-PDB DML, 338–339
linking tables across containers, 330–338
location independence, 342–343
overview of, 324–325
simple user tables, 325–329
cumulative incremental backups, RMAN, 189–191
cursor sharing, schema consolidation, 8–9
cursors, connecting to containers, 24–26

D

DAS (direct attached disks), Oracle Database 8*i* and 9*i*, 4–5
data
analysis, with logical replication, 347
consolidating at CDB level, 30–33
moving. *See* moving data
in multitenant containers, 14
sharing. *See* sharing data across PDBs
storing metadata with, 11–12
Data Definition Language (DDL), 8, 326–327
data dictionary table, user/role definitions, 141
Data Guard
Active Data Guard, 291
Cloud Control, 314–315
creating physical standby, 291–292
creating physical standby in multitenantt, 308–314
creating standby with Cloud Control, 304–308
duplicating standby with RMAN, 292–304
overview of, 290

standby in cloud, 315–318
Data Guard Broker
configure network setup for standby, 293
Data Guard configuration and, 292
start processes/finish configuration, 297–299
data-linked objects, linking tables across containers, 336–338
data links (object links), 18–19, 80
Data Manipulation Language (DML), 89
Data Pump, 75, 254–255
Data Recovery Advisor, 193
data security
`CONTAINER_DATA`, 151–153
`CREATE_FILE_DEST`, 159
creating roles, 153
encryption. *See* TDE (Transparent Data Encryption)
lockdown profiles, 155–158
`PATH_PREFIX`, 158
PDB isolation, 158–159
proxy users, 153–155
data table, application containers, 81
database administrator. *See* DBA (database administrator)
Database as a Service (DBaaS), 5, 79
Database Configuration Assistant. *See* DBCA (Database Configuration Assistant)
Database Express 12.2
creating CDB resource plan, 270–273
monitoring PDBs managed by Resource Manager, 285–286
monitoring Resource Manager, 284
`DATABASE` key word, CDB and PDB backups, 175
database links, 8, 320–321
database offerings, Oracle Public Cloud, 316
database time zone, 92, 98
Database Upgrade Assistant. *See* DBUA (Database Upgrade Assistant)

DATABASE_PROPERTIES, cross-PDB DML, 339
databases, creating
container. *See* CDB (container database)
overview of, 36
pluggable. *See* PDBs, creating
datafiles
for all containers in CDB, 30
CDB administrator restricting, 159
creating standby with RMAN, 294–295
enabling PDB recovery on standby, 313
in full CDB backups, 176
nodata clones and, 243
partial PDB backup of, 181–182
recovering from CDB$ROOT, 186
recovering PDB nonsystem, 188
recovering PDB system, 187–188
remove existing PDB from standby, 312–313
restoring PDB to previous state, 200
restoring/recovering PDB, 202–204
setting transportable tablespaces to read-only, 322
day-to-day management
CDBs, 89–92
choosing container, 88–89
overview of, 88
patching, 112–113
patching and upgrades, 100–101
PDBs. *See* PDBs, managing
plug-in, 111–112
upgrading CDB, 101–111
using CDB-level vs. PDB-level parameters. *See* parameters, CDB-level vs. PDB-level
DBA (database administrator)
lockdown profiles, 155–159
management tasks. *See* day-to-day management

as proxy user, 153–155
separation of roles, 155
DBaaS (Database as a Service), 5, 79
DBA_CDB_RSRC_PLAN_DIRECTIVES, CDB resource plan, 277
DBA_CONTAINER_DATA dictionary view, 152
DBA_ENCRYPTED_COLUMNS, TDE-encrypted columns, 164
DBA_HIST views, AWR, 31
DBA_OBJECTS, Oracle-maintained objects, 13
DBA_PDBS dictionary view, PDBs/their status, 21–22
DBA_PDB_SAVED_ STATES view, opening/closing PDBs, 96
DBA_ROLES, Oracle-maintained objects, 13
DBA_SOURCE view, 16–18
DBA_USERS view, 13, 324–325
+DBBACKUP disk group, full CDB backup, 177
DBCA CLI, creating CDB, 42–44
dbca command, creating CDB, 38
DBCA (Database Configuration Assistant)
creating CDB, 90
creating CDB using DBCA CLI, 42–44
creating CDB using DBCA GUI, 37–42
creating database in local UNDO, 207
creating PDB, 65–66
DBCA GUI, creating CDB, 37–42
DB_CACHE_SIZE parameter, PDB memory allocation, 282
DB_FILE_NAME_CONVERT settings, PDB recovery on standby, 313
DBID, identifying containers, 22–23, 31
DBMS_ SQL.PARSE, SQL on multiple PDBs, 98
dbms_preup package, pre-upgrade check for CDB, 102
DBMS_RESOURCE_MANAGER package
adjusting default directive, 268
creating CDB resource plan, 269, 276
modifying CDB resource plan, 273–274

removing PDB resource plan, 281
Resource Manager configuration/management, 263
updating autotask directive, 268
DBMS_RESOURCE_MANAGER_PRIVS package, 264
DBMS_ROLLING package, automatic upgrades, 363
DBMS_SERVICE package, creating services, 133–134
DB_PERFORMANCE_PROFILE, CDB resource plan, 275–276
DBUA (Database Upgrade Assistant)
automating database backup, 104
automating upgrade process, 103
overview of, 110–111
DB_UNIQUE_NAME clause, primary vs. standby databases, 117–118
Dbvisit Replicate, logical replication tool, 363
DDL (Data Definition Language), 8, 326–327
dedicated listener, creating for PDB, 134–137
default directives, CDB resource plan, 267–268, 270–273
DELETE_PLAN, PDB resource plan, 281
deleting (removing)
CDB resource plan, 275
CDB resource plan directive, 275, 277
PDB resource plan, 281
Dell SharePlex, 363
Delphix, cloning/storage de-duplication with, 324
describe file, PDB SPFILE equivalent, 115
dictionaries
multitenant, 16–20, 72
system. *See* system dictionary
dictionary tables, 16
dictionary views
from containers, 26–27
of dictionary objects, 19
listing all PDBs and their status, 21–22

overview of, 17–18
querying metadata with, 11
stored in CDB$ROOT, 16
differential incremental backups, RMAN, 189–190
direct attached disks (DAS), Oracle Database 8*i* and 9*i*, 4–5
Direct NFS, Oracle favoring/promoting, 241
directories
configuring/starting manager in GoldenGate, 353
creating standby with RMAN, 294
schema consolidation and, 8
`DISABLE` clause, `ALTER LOCKDOWN PROFILE`, 156
disable database options, with lockdown profiles, 155–156
disaster recovery. *See* Data Guard
DML (Data Manipulation Language)
cross-PDB DML, 338–339
limiting to current container, 89
drop operations
`DROP DATABASE`, 90
`DROP PLUGGABLE DATABASE`, 90, 99–100, 311–312
`DUPLICATE` command, create standby with RMAN, 295
duplication, RMAN backup redundancy, 174

E

`ENABLE` clause, `ALTER LOCKDOWN PROFILE`, 156
`ENABLE CONTAINERS_DEFAULT`, 339
`ENABLE PLUGGABLE DATABASE`
creating CDB in SQL*Plus, 48–52
creating database in local UNDO, 207–208
defining multitenant architecture, 72
`enable_goldengate_replication`, source database setup, 350–351
`ENCRYPT` keyword, TDE, 163–164
encryption. *See* TDE (Transparent Data Encryption)
Enterprise Edition, single-tenant in, 77–79

Enterprise Manager
Cloud Control. *See* Cloud Control
Database Express 12.2. *See* Database Express 12.2
opening/closing PDBs, 95–96
error messages
Data Guard broker and, 298–299
documentation book for, 298
query user tables for, 325–327
Event Streaming mode, Dbvisit Replicate, 363
event triggers, on PDB operations, 252–253
expire all unused accounts, creating CDB, 54
export, full transportable, 253–255
extended data metadata-linked objects, 333–336
extract process, GoldenGate
BigData support, 359–361
initial extract, 356–357
multitenant support, 353–354
running initial extract, 358
starting, 358–359

F

fallback scenario, planning before upgrades, 104
Fast Recovery Area (FRA), 39, 104, 212
features
disabling with lockdown profiles, 158
not compatible in multitenant, 73–74
file storage (FS), creating CDB without OMF, 46
`FILE_ID`, datafiles, 30
`FILE_NAME_CONVERT` clause, creating new PDB from PDB$SEED, 58–59
files, CDB container
control files, 27–28
datafiles, 30
overview of, 27–30

redo logs, 29
SPFILE, 27
temporary tablespaces, 28–29
UNDO tablespace, 28
files, naming database, 36–37
FILESPERSET, multisection incremental backups, 192–193
flashback database, 300–302
flashback logs
in auxiliary instance cleanup, 225
modifying root container, 92
overview of, 213–215
at PDB level, 218–219
planning backup for CDB upgrades, 104
FLASHBACK OFF, 216, 218
FLASHBACK ON, 213–215
flashback PDB
auxiliary instance cleanup, 225
cleaning restore point, 219–220
flashback logging, 213–215
impacting standby database, 223–224
with local UNDO, 215–216
overview of, 213
PITR vs., 221–222
resetlogs, 221–222
restore points at CDB/PDB levels, 216–219
in shared UNDO, 216
single-tenant in Enterprise Edition, 77–78
FORCE LOGGING mode, modifying PDB, 99
FORMAT option, full CDB backups, 176–177
FRA (Fast Recovery Area), 39, 104, 212
FS (file storage), creating CDB without OMF, 46
full CDB backups, 176–178
full CDB recovery, 185

full PDB backups, 180–181
full transportable export/import, 253–255
FULL=Y, IMPDP options, 12

G

-generateScripts, database, 48
GGSCI (GoldenGate Software Command Interface), 352–353
global database name, modifying entire CDB, 90
GoldenGate
    configure and start manager, 352–353
    configure parameter files for replicat, 355–357
    extract configuration file, 353–354
    logical replication with, 349
    multitenant support in, 349–351
    replacing Oracle Streams in multitenant, 73
    run initial extract, 358
    set up target database, 354–355
    start extract and replicat, 358–359
    supporting big data adapters, 359–361
    table supplemental logging, 352
GoldenGate Software Command Interface (GGSCI), 352–353
grant statement, privileges, 146–148
GRANT_SWITCH_CONSUMER_GROUP, Resource Manager, 264, 279–280
GRANT_SYSTEM_PRIVILEGE, Resource Manager, 264
groups, temporary tablespace, 92
guaranteed restore point, flashback logs, 214–215
GUI (graphical user interface)
    creating CDB, 37–42
    creating PDB, 65
GUID
    clone operations, 240
    identifying containers, 22

H

hard-coded users, 142
hard-coding schema name, 7–8
HASH schema, for container map, 342
history
    of database use, 4–5
    viewing PDB operation, 97
HOST, proxy DBs, 247
hot clone (refreshable copy), 244, 245–246
hot (online, inconsistent) backups, 171–172
HugePages, impact on memory management, 281

I

I/O (input/output)
    managing PDB via initialization parameters, 281–283
    Resource Manager limiting PDB, 261
    setting parameters at PDB level, 267
IaaS (Infrastructure as a Service), 5
IDs (identifiers)
    CON_ID for containers, 20–22
    CON_UID and DBID for containers, 22–23
    DBA_HIST views, AWR, 31
    FILE_ID, datafiles, 30
image copies
    full CDB backups, 176–177
    multisection incremental backups, 192–193
    restore and recover PDBs, 187
    view full CDB backup details with LIST, 177
importing
    full database into new database, 12
    full transportable database, 253–255
incremental backups, RMAN

block change tracking for, 91, 191–192
combining with multisection backups, 192–193
keeping image copies up to date with, 176
for optimization, 189–191
Information Lifecycle Management, 74
Infrastructure as a Service (IaaS), 5
inheritance, conflict resolution, 149
initial extract, Golden Gate replication, 356–357, 358
initialization parameters
creating CDB resource plan with PDB profile, 275–276
enabling/disabling PDB resource plan, 281
managing PDB memory and I/O, 281–283
init.ora, storing parameters in older versions in, 113
instance caging, 283–284
instance recovery, CDBs, 184
instance(s)
AWR collecting, 31–32
creating service with SRVCTL, 130–132
initialization parameters controlling, 113
multiple databases managed by one, 4, 10–11
RMAN commands to restore/recover PDB, 203
server consolidation and, 9–10
in traditional database connection, 120–121
invalid objects, recompiling, 55
`ISPDB_MODIFIABLE`, 116
`ISSYS_MODIFIABLE`, 116

J

Java, preupgrade.jar file, 103
JDBC (Java Database Connectivity), connecting to containers from, 25

K

`KEEP DATAFILES`, unplugged PDB, 231

keystore, TDE, 160–163

L

levels, Resource Manager, 264–265
LGWRs (Log Writers), redo logs, 29
licenses
    cloud virtual machines requiring, 317
    creating PDB in Enterprise Edition, 78–79
    multitenant requiring, 72–73
    single-tenant not requiring, 72
    virtualization and, 10
links, public synonyms and database, 8
`LIST` command, view full backups, 177–178, 181
listener registration (LREG) background process, 121–123, 127
`listener_address` string, dedicated listener for PDB, 135
`LISTENER_NETWORKS`, dedicated listener for PDB, 135
listener.ora file
    creating dedicated listener for PDB, 135–136
    setting listener to allow threads, 124
    static network definition setup, 293
listeners
    creating PDB dedicated, 134–137
    creating service with SRVCTL, 131–132
    default services/connecting to PDBs, 125–129
    Oracle Net Listener, 120–121
    registration process, 121–123
    setting to allow threads, 124
load options, creating CDB with SQL*Plus, 53
local clone
    create new PDB for standby, 310
    creating inside single PDB, 240–241
    creating new PDB, 59–60
local roles, creating, 153

local UNDO
cloning local PDB, 241
cloning remote PDB, 242
flashback with, 215–216
hot cloning remote PDB, 241–242
local UNDO, in 12.2
changing to shared UNDO from, 223
changing UNDO mode, 209–211
changing UNDO tablespace, 208–209
creating database, 207–208
database properties, 207
overview of, 206–207
PDBPITR in, 212–213
using shared or, 211
`LOCAL UNDO OFF` state, shared UNDO, 211–213, 216
`LOCAL UNDO ON` state, local UNDO, 211, 215–216
local users
conflict resolution between common and, 148–150
creating with `CONTAINER=CURRENT`, 142–143
granting local privileges to, 146–147
keeping conflict resolution clear and simple, 150–151
namespace for, 144
as proxy users in multitenant, 153
stored only in single PDB, 140–141
SYSBACKUP privilege for, 174–175
`LOCAL_LISTENER` parameter, create standby, 294
locally managed tablespaces, 12
`LOCAL_UNDO_ENABLED` property, database, 207
location independence, sharing data across PDBs, 342–343
locations, altering PDB file. *See* PDBs, moving files
lock all unused accounts, creating CDB with SQL*Plus, 54
lockdown profiles
disable ALTER SYSTEM, 156–157

disable database options, 155–156
disable features, 158
overview of, 155
PDB isolation, 158–159
single-tenant security and, 76
log in, auditing as proxy users, 153–155
Log Writers (LGWRs), redo logs, 29
logical replication
defined, 346
deprecated features, 348–349
Oracle LogMiner and, 347–348
with Oracle XStream, 361–363
with other third-party options, 363
overview of, 346–347
logical replication, Oracle GoldenGate
configure and start manager, 352–353
configure parameter files for replicat processes, 355–357
extract configuration file, 353–354
multitenant support, 349–351
overview of, 349
run initial extract, 358
set up target database, 354–355
start extract and replicats, 358–359
supporting big data adapters, 359–361
table supplemental logging, 352
logical standby database, 362
logs
flashback, 213–215
multitenant support in GoldenGate, 352
pre-upgrade check for CDB, 102–103
resetlogs, 221–222
resuming CDB upgrade after failure, 108
loopback database link, non-CDBs, 320–321

LREG (listener registration) background process, 121–123, 127
`lsnrctl start listener_pdb` command, 135–136

M

maintenance, autotask directive, 268
management. *See* day-to-day management
manager, configure GoldenGate, 352–353
mapping, container, 339–342
master key
    create and store in keystore, 162–163
    shipping when plugging in/unplugging, 164–165
    Transparent Data Encryption setup, 159–160
    verifying created keys, 163
`MAX_IOPS` parameter, limiting PDB I/O, 282
`MAX_MBPS` parameter, limiting PDB I/O, 282
`MAX_SHARED_ TEMP_SIZE`, temporary tablespace quota, 28
memory
    creating CDB resource plan, 269–271
    managing PDB, 281–283
    Resource Manager limiting usage of PDBs, 261
    server consolidation and, 9–10
    setting memory limits, 266–267
    setting utilization limits for PDBs, 266–267
    virtualization and, 10
`memory_limit`
    CDB resource plan, 266–267, 270–271, 273
    CDB resource plan using PDB profiles, 276–277
    PDB resource plan, 279
`memory_min` limit
    CDB resource plan, 266, 270–271, 273
    CDB resource plan using PDB profiles, 276–277
    PDB resource plan, 279
metadata

application container table, 81
Codd's rules for RDBMS, 11
consolidating at CDB level, 30–33
links, 18–19
in multitenant containers, 14
in pluggable database, 72
stored in system dictionary, 11–12
system vs. application, 13–14
transportable tablespaces in multitenant and, 7
metadata-linked objects, 331–336
MIGRATE state, PDBs, 93
migration, logical replication use case for, 346
`modify service`, for Oracle RAC PDB, 133
monitoring, Resource Manager, 284–286
MOUNTED state, PDBs, 93, 212
mounted state, performing cold backups with RMAN in, 171
moving data
altering PDB file locations. *See* PDBs, moving files
in application containers, 247
by cloning, 239–240
by cloning local PDB, 240–241
by cloning remote PDB, 242–247
converting non-CDB, 247–250
full transportable export/import for, 253–255
overview of, 228
in PDBs to cloud, 251–256
transportable tablespaces for, 255–256
triggers on PDB operations for, 252–253
multiple channel backup, RMAN, 192
multisection incremental backup, RMAN, 192–193
multitenant administration. *See* day-to-day management
multitenant databases
application containers, 80–83

consolidation, 5, 83
data movement, 75
database links, 321
non-CDB depreciation and, 72–73
not an option, 72–74
overview of, 79–80
purchasing license for, 72
supported in GoldenGate, 349–351
unsupported features in, 73–74
multitenant, introduction to
connecting to containers, 23–24
consolidation at CDB level, 27–33
consolidation pros and cons, 10–11
data and metadata at CDB level, 30–33
dictionary views from containers, 26–27
history of database use, 4–5
identifying containers, 20–23
list of containers, 21–22
multiple databases managed by one instance, 10
multitenant containers, 14–16
multitenant dictionaries, 16–20
overview of, 4
road to multitenant, 5–6
schema consolidation, 6–9
server consolidation, 9–10
system dictionary. *See* system dictionary
system dictionary, containers, 14–16
table consolidation, 9
virtualization, 10
multithreaded mode
LREG running in, 122–123
and multitenant, 123–125
My Oracle Support (MOS), 100, 101

N

name collision, schema consolidation and, 7–8
names
    avoid hard-coding schema, 7–8
    creating CDB using DBCA GUI, 40
    identifying containers with, 20–22
    identifying objects with schema/object, 88–89
    public synonyms and schema, 8
    service, 125–129
    tablespace, 8
    using OMF for database file, 36–37
namespace
    avoid mixing common/local users in same, 144
    keeping clear and simple, 150–151
NAS (network-attached storage), 5
`NEED UPGRADE` status, PDBs, 21
network-attached storage (NAS), 5
networking and services
    creating dedicated listener for PDB, 134–137
    creating services, 129–134
    LREG process, 121–123
    networking, multithreaded and multitenant, 123–125
    Oracle Net, 120
    Oracle Net Listener, 120–121
    overview of, 120
    service names, 125–129
networking packages, disabling, 158
`NEW` status, PDBs, 21–22
`NLS_DATE_FORMAT` environment variable, RMAN, 173
NOARCHIVELOG mode
    backing up database in, 104
    modifying entire CDB, 90
    performing cold backups in, 171, 185

nodata clone, remote PDB, 243
NOFILENAMECHECK, RMAN, 295
NOLOGGING mode, modifying PDB, 99
NOMOUNT state
creating CDB using SQL*Plus, 51–52
restoring CDB using cold backup, 185
unavailable for PDBs, 22, 93
non-CDB (non-container database)
container database vs., 14
converting, 247–251
Data Recovery Advisor used in, 193
database link categories in, 320–321
depreciation of, 72–73
instance caging to CDB resource plan, 283–284
movement away from. *See* day-to-day management
noncompatible features, 73–74
object identification in, 88
running APEX on, 30–31
as system dictionary in past, 11–14
NORMAL status, PDBs, 21–22
null string, avoid setting prefix to, 144

O

object links (data links), 18–19, 80
objects
creating CDBs by recompiling invalid, 55
identifying in non-CDB, 88
identifying in PDB, 88–89
identifying Oracle-maintained, 12–13
obsolete logical replication features, 348–349
OCI (Oracle Call Interface), connecting to containers, 25
OEM (Oracle Enterprise Manager), 144–146
oerr utility, 298

OMF (Oracle Managed Files)
cloning PDB within same CDB, 59
creating CDB using file storage without, 46
creating CDE with two PDBs, 45–46
creating new PDB from PDB$SEED, 56–58
enabling/disabling block change tracking, 191–192
enabling PDB recovery on standby, 314
file naming with, 36–37
identifying PDB in directory structure, 22
restoring/recovering PDB with RMAN, 203–204
open normally, end of CDB upgrade, 108
`OPEN RESETLOGS`, 221–222
open state, PDBs, 93–94
operating systems. *See* OSs (operating systems)
`OPT_PARAM` hint, 113
Oracle Advanced Replication, as deprecated, 349
Oracle Application Express (APEX), 30–31, 107
Oracle Call Interface (OCI), connecting to containers, 25
Oracle Change Data Capture (CDC), as deprecated, 348
Oracle Cloud Services, back up to, 195–196
Oracle Database Resource Manager. *See* Resource Manager
Oracle Database versions, history of, 4–5
Oracle Enterprise Manager (OEM), creating local/common user, 143–146
Oracle GoldenGate. *See* GoldenGate
Oracle LogMiner, 347–348
Oracle Managed Files. *See* OMF (Oracle Managed Files)
Oracle Net, 120
Oracle Net Listener, 120–121
Oracle Net Services, 120
Oracle Public Cloud, running standby in, 315–318
Oracle Streams
as deprecated, 348–349

only supported in non-CDB architecture, 73
Oracle XStream, logical replication, 361–363
ORACLE_HOME directories
patching using, 112–113
pre-upgrade check for CDB, 101–103
upgrade CDB with catupgrd.sql, 105–106
ORACLE_MAINTAINED column, 142
ORACLE_MAINTAINED flag, 13
ORACLE_SID, consolidating single-tenant CDBs, 76
ora_lreg_*SID*, LREG process, 121
OSs (operating systems)
credentials in single-tenant security, 76
database resource allocation, 260
distributing resources with Resource Manager, 261
limiting interaction with file system/processes, 158

P

parallel execution servers, Resource Manager, 261
parallel_server_limit, CDB resource plan, 266
parameters
configure replicat processes, 355
create CDB using SQL*Plus, 49–50
create standby with RMAN, 293–295
modifying entire CDB, 90–92
modifying PDB, 98–99
plugging in non-CDB, 249
resuming CDB upgrade after failure, 108–109
running catupgrd.sql to upgrade CDB, 105–107
parameters, CDB-level vs. PDB-level
ALTER SYSTEM RESET, 116
CDB SPFILE, 114
Container=ALL, 116–117
DB_UNIQUE_NAME, 117–118

ISPDB_MODIFIABLE, 116
overview of, 113–114
PDB SPFILE equivalent, 114–115
SCOPE=MEMORY, 116
parent cursors, consolidation of, 8–9
partial CDB backups, 178–179
partial PDB backups, 181–182
partitions, table consolidation and, 9
passwords
create standby with RMAN, 293–295
creating CDB using DBCA CLI, 44–45
creating CDB using SQL*Plus, 50–51
proxy users and, 153–155
Patch Set Update (PSU), 112
patching
CDBs (container databases), 112–113
overview of, 100–101
PATH_PREFIX, lockdown profile, 158
PDB level, Resource Manager, 265
PDB profiles
CDB resource plan, 266–267, 275–276
Resource Manager, 263
PDB SPFILE equivalent, 114–115
PDBADMIN user, creating CDB, 39, 41
PDB_DBA role, 41
PDB_LOCKDOWN parameter
disable ALTER SYSTEM, 157
disable database options, 156
disable features, 158
PDB_OS_CREDENTIALS, lockdown profile, 158
PDBPITR (PDB point-in-time recovery). *See also* flashback PDB
in local UNDO mode, 212–213

overview of, 200–201
restore/recover PDB with RMAN, 201–204
in shared UNDO mode, 211–212
summary of 12.1, 205–206
where is the UNDO? 204–205
PDB_PLUG_IN_VIOLATIONS
conflict resolution, 151
running compatibility check, 236
upgrading PDBs, 111–112
PDBs, creating
overview of, 55–56, 92–93
from PDB$SEED, 56–59
using catcon.pl script, 67–69
using Cloud Control, 66–67
using DBCA, 65–66
using local clone method, 59–60
using SQL Developer, 60–64
PDBs, managing
dropping, 99–100
modifying, 98–99
new, 92–93
opening and closing, 93–96
overview of, 92
running SQL on multiple PDBs, 97–98
view operation history, 97
view state of, 97
PDBs, moving files
altering file locations, 228–229
application container considerations, 247
cloning, 239–240
cloning local PDB, 240–241
cloning remote PDB, 242–247
to cloud, 251–252
converting non-CDB database, 247–251

full transportable export/import, 253–255
transportable tablespaces, 255–256
triggers, 252–253
PDBs, moving files with plug-in/unplug
application containers, 247
check compatibility for plug-in, 235–237
overview of, 229
PDB archive files, 238–239
plug in PDB as clone, 237–238
plug in/unplug operations, 229
plugging in non-CDB, 248–250
unplug/plug in PDB, 230–231
unplugged database stays in source, 231–232
XML file contents, 232–235
PDBs (pluggable databases)
accepting all commands, 15
backups, 175–176, 180–183
CDB level parameters vs., 113–118
connecting to containers, 23–26
creating CDB, examples, 44–47
creating CDB resource plan, 269–273
creating dedicated listener, 134–137
creating in SQL*PLUS, 55
creating on source database, 309–311
creating services, 129–134
creating with PDB$SEED container, 17
default services and connecting to, 125–129
dropping, 90
enabling recovery on standby, 313–314
encryption key when plugging in/unplugging, 164–165
isolation, 158–159
listing all/status of all, 21–22
managed by Resource Manager, 285–286
managing memory and I/O, 281–283

modified when modifying CDB, 90–91
one instance opening multiple, 4
querying data, 328–329
removing from source database, 311–312
removing from standby, 312–313
Resource Manager requirements, 263
resource plan, 262, 277–281
restoring and recovering, 187–189
sharing data across. *See* sharing data across PDBs
single-tenant in Enterprise Edition, 77–79
single-tenant in Standard Edition, 74–77
temporary tablespaces in, 28–29
understanding, 14–15
upgrading, 111–112
PDB$SEED
consolidating single-tenant CDBs, 77
create new PDB on source database, 310
created as part of `CREATE DATABASE`, 52
creating new PDB, 56–59
creating UNDO tablespace, 210–211
full CDB backup of, 176
opening pluggable database, 53
overview of, 16
PFILE
creating CDB using SQL*Plus, 51–52
creating SPFILE for standby with RMAN, 294
creating SPILE from, 54
recreating SPFILE from, 113
storing parameters in older versions, 113
`PGA_AGGREGATE_TARGET`, PDB memory allocation, 282
`PHASE_TIME` number, CDB upgrade after failure, 108–109
physical copy, of database, 346
physical replication, 346, 347

physical standby database, 362
PITR (point-in-time recovery). *See also* PDBPITR (PDB point-in-time recovery)
auxiliary instance cleanup, 225
flashback PDB. *See* flashback PDB
flashback vs., 222–224
impact on standby database, 223–224
local UNDO. *See* local UNDO, in 12.2
overview of, 200
resetlogs, 221–222
plug, and clone with TDE, 164–165
plug-in/unplug PDB
moving files. *See* PDBs, moving files with plug-in/unplug
for standby database, 311–312
`PLUGGABLE DATABASE` keywords
full CDB backup, 177–178
partial CDB backup, 178–179
PDB backup, 175
PDB backup without, 181
pluggable databases. *See* PDBs (pluggable databases)
`pluggable` keyword, cross-PDB DML, 339
PMON (Process Monitor) process, 121
point-in-time recovery. *See* PDBPITR (PDB point-in-time recovery)
PORT, proxy DBs, 247
pre-upgrade, CDB, 101–104
prefix
application container, 151
C##, 144–145, 151
preupgrade.jar file, CDB, 103
preupgrd.sql script
post-upgrade scripts generated by, 109
pre-upgrade check for CDB, 101–103
prechecking fast recovery area, 104

running, 104–105
privileges
cloning remote PDB, 245
conflict resolution between common/local users, 148–150
defining roles for same group of, 141
granting common, 147–148
granting local, 146–147
keeping clear and simple, 150–151
Resource Manager configuring, 263–264
SYSBACKUP system, 174–175
user, 141–142
Process Monitor (PMON) process, 121
profiles, lockdown. *See* lockdown profiles
properties
changing UNDO mode from database, 207–209
creating PDB, 61, 63
Provision Pluggable Databases, creating PDB, 66–67
provisioning
running initial extract, 358
source data to target, to create baseline, 356–357
proxy PDBs, 246–247, 342–343
proxy users, 153–155
`ps` command
identifying LREG thread, 123
viewing thread and process details, 125
PSU (Patch Set Update), 112
Public Cloud, standby in Oracle, 315–318
public synonyms, schema consolidation and, 8

Q

queries
BOOTSTRAP$ table, 14
CDB_ views, 151

current state of all PDBs, 97
flashback, 356
user table, 325–329
_query_on_physical=no, prevent use of Active Data Guard, 291
Quest SharePlex, 363
quota, temporary tablespace, 28

R

RAC (Real Application Clusters)
adding service for PDB, 132–133
in Oracle Database 8i and 9i, 4
Oracle Net Listener and, 120–121
RANGE schema, container map, 342
RDBMS, Codd's rules for, 11
read-only data
Active Data Guard enabling, 291
cloning local PDB, 241
multiple databases sharing common, 322–324
for transportable database during import process, 253
READ ONLY state, PDBs, 93
read/write mode
cloning local PDB in 12.2, 241
cloning remote PDB, 242
cloning remote PDB as refreshable copy, 244–245
opening PDB$SEED, 211
READ WRITE state, PDBs, 93
RECOVER command
PDBPITR in local UNDO mode, 212
syntax, 204–205
RECOVER PLUGGABLE DATABASE, PDBPITR, 212–213
recovery. *See also* Data Guard; flashback PDB; PITR (point-in-time recovery)
backup and. *See* backup

CDB restore and, 184–187
with Data Recovery Advisor, 193
DBA must never fail in, 290
disabling on standby, 224
enabling for PDB with standby, 313–314
instance, 184
overview of, 170
PDB restore and, 187–189
SYSBACKUP privilege for, 174–175
Recovery Manager. *See* RMAN (Recovery Manager)
redo apply, PDB recovery on standby, 313–314
redo logs
common to all containers in CDB, 29
create standby with Cloud Control, 306
create standby with RMAN, 300–302
Data Guard broker/duplicate database and, 298–299
LogMiner extracting/displaying changes from, 347–348
modifying entire CDB, 90
testing standby created with RMAN, 302–303
REDO, PDB point-in-time recovery, 200
REDO stream, resetlogs, 221–222
redundancy, RMAN backup, 174
refreshable copy (hot clone), 244, 245–246
registration
creating new PDB, 125–126
listener, 121–123
in traditional database connection, 120–121
`RELOCATE` statement, 96, 245–246
`RELOCATED` status, 246
`RELOCATING` status, 245
remote clone
of non-CDB, 250–251
plug/unplug operations in, 75

from previous versions, 112
remote clone PDB
create new PDB for standby, 311
nodata, 243
overview of, 242–243
proxy PDB, 246–247
refreshable copy (hot clone), 244–245
`RELOCATE` statement, 245–246
root container database link used by, 321
splitting PDB, 243
for standby database, 311
remote database link, non-CDBs, 320–321
remote file server (RFS), remote cloning from previous version, 112
`remove service`, Oracle RAC PDB, 133
rename of PDB, 312
replicat processes, BigData support in GoldenGate, 359–361
replicat processes, GoldenGate
configure parameter files, 355
defined, 349
initial extract, 357
setup target database, 354–355
topology example, 350
replication
data sharing with cross-database, 343
logical. *See* logical replication
physical vs. logical, 346–347
reports
Automatic Workload Repository, 31–32
CDB backup, 179–180
PDB backup, 181, 183
`RERESHING` status, 244
resource consumer groups, Resource Manager allocating resources to, 261
Resource Manager

basics, 260–261
creating CDB resource plan. *See* CDB resource plan, Resource Manager
creating PDB resource plan, 277–281
instance caging, 283–284
levels, 264–265
managing PDB memory/I/O, 281–283
monitoring, 284–286
overview of, 260
requirements, 263–264
terminologies, 261–263
resource plan directives
adding to CDB resource plan, 274
adjusting default and autotask, 267–268
creating CDB resource plan, 269–273
default, 267–268
removing CDB, 275, 277
Resource Manager, 261
specifying resource utilization limits, 267
updating CDB resource plan, 273–274
viewing CDB, 285
resource plan, Resource Manager
creating CDB. *See* CDB resource plan, Resource Manager
creating PDB, 277–280
defined, 261–262
enabling/disabling/removing PDB, 281
viewing CDB, 285
`RESOURCE_MANAGER_PLAN` parameter, enable/disable PDB resource plan, 281
resources, prioritizing with Resource Manager. *See* Resource Manager
response file, creating CDB, 43–47
`restore` command
enabling PDB recovery on standby, 314
PDBPITR in local UNDO mode, 212–213

restore points
at CDB and PDB levels, 216–219
for CDB upgrades, 103–104
clean, 219–220
dropping when testing upgraded application, 109
flashback logs using guaranteed, 214–215
retention period, flashback log, 214
`REVOKE_SWITCH_CONSUMER_GROUP`, Resource Manager, 264
`REVOKE_SYSTEM_PRIVILEGE`, Resource Manager, 264
RFS (remote file server), remote cloning from previous version, 112
RMAN (Recovery Manager)
backup optimization, 189–193
backup redundancy, 174
CDB and PDB backups, 175
CDB reporting, 179–180
cold backups, 171
commands to restore/recover PDB, 201–204
creating standby database. *See* standby database, creating with RMAN
default backup, 173–174
incremental backups, 90
PDB backup restrictions, 183
running standby in the cloud, 317
setting transportable tablespaces to read-only, 322
`TAG` option, 170
unplugged databases part of, 231
roles
creating with `CREATE ROLE`, 142
lockdown profile, 155–158
proxy user, 153–155
schema consolidation and, 8
understanding, 141–142
ROLES clause, creating PDBs with SQL*Plus, 41
roll-forward recovery phase, point-in time, 200–201

rollback recovery phase, point-in time, 200–201
rolling upgrades, logical standby used in, 362–363
root container database link, administration only, 321

S

SAN (storage area network), 5
`SAVE STATE` command, 95–96
scheduler window, CDB resource plan, 274–275
schema
    names, 88–89
    synonymous to user, 141
schema consolidation
    cursor sharing, 8–9
    multitenant overcoming limitation of, 23–26
    overview of, 6
    pros and cons, 11
    public synonyms and database links, 8
    roles, tablespace names, and directories, 8
    schema name collision, 7–8
    transportable tablespaces, 7
SCN (system change number), Golden Gate replication, 356–357
`SCOPE=MEMORY` clause, 116
`SCOPE=SPFILE` clause, 116–117
scp command, copying files to target database location, 230
`scriptDest`, database creation scripts, 48
security
    creating CDB using SQL*Plus, 54
    overview of, 140
    proxy users and, 153–155
    single-tenant and, 75–76
security, data
    `CONTAINER_DATA`, 151–153
    lockdown profiles, 155–158

PDB isolation, 158–159
roles, 153
Transparent Data Encryption, 159–165
Security Patch Update (SPU), 112
security, user
common grants, 147–148
common vs. local, 140–141
conflict resolution, 148–150
`CONTAINER-CURRENT` clause, 142–143
`CONTAINER=COMMON` clause, 144–146
keeping clear and simple, 150–151
local grants, 146–147
understanding, 141–142
servers
consolidation of, 9–10
Oracle Database 11g application, 5
Oracle Database 8i and 9i client, 4–5
service names, 125–129
services
creating generally, 129–130
creating with DBMS_SERVICE, 133–134
creating with SRVCTL, 130–133
updating with LREG process, 122
sessions
initialization parameters controlling behavior of, 113
user, 141
`SESSIONS` parameter value, PDBs vs. CDB, 91
`SET CONTAINER` privilege, users, 154
`setasmgidwrap` utility, creating CDB in SQL*Plus, 49
`SET_CONSUMER_ GROUP_MAPPING`, PDB resource plan, 277
`SGA_MIN_SIZE` parameter, PDB memory allocation, 282
`SGA_TARGET` parameter, PDB memory allocation, 282
sharding, and multitenant, 74

share values, CDB resource plan, 265–266
shared UNDO mode
changing, 209–210
changing to local UNDO mode from, 223
clean restore point in, 219–220
flashback in, 216
`LOCAL_UNDO_ENABLED` in, 207
no reason to use in 12.2, 211
PDBPITR in, 211–213
problem with, 206–207
`SHARED_POOL_SIZE` parameter, PDB memory allocation, 282
shares, Resource Manager, 263
sharing data across PDBs
common read-only data, 322–324
cross-database replication, 343
database links, 320–321
overview of, 320
`show all`, RMAN configuration, 173–174
`SHOW CONFIGURATION`, 302
`SHOW SPPARAMETER`, 114–116
shutdown, CDB, 90–91
shutdown pluggable database, PDBs, 94–95
`-silent` parameter, CLI
creating CDB using DBCA, 42, 44
creating CDB using DBCA CLI, 44–45
single-instance CDB, Data Recovery Advisor, 193
single-tenant configuration
defined, 72
in Enterprise Edition, 77–79
estimating length of time for CDB upgrade, 110
in Standard Edition, 75–77
upgrading CDB with needed components, 107
skip scan index optimization, table consolidation, 9

snapshot copy, cloning local PDB, 241
software keystore. *See* keystore, TDE
source database
create new PDB on, 309–311
create standby with RMAN, 293, 295–296
logical replication in sync with, 346–347
plugging in non-CDB, 248–250
setup multitenant for GoldenGate replication, 350–351
unplugged database stays in, 231–232
SOURCE$ dictionary table, multitenant dictionaries and, 16–17
SPFILE parameters
CDB, 114
common to all containers in CDB, 27
creating from PFILE, 54
creating standby with RMAN, 294
full CDB backup of, 176
PDB equivalent to, 114–115
for site vs. service, 118
storing in current versions, 113
upgrading CDB with catupgrd.sql, 105
SPU (Security Patch Update), 112
SQL
creating PDB using SQL Developer, 61–63
running on multiple PDBs, 97–98
SQL Developer
configuring Resource Manager, 271
creating PDB, 60–64
monitoring Resource Manager, 285
sqlnet.ora, keystore location setup for TDE, 160
SQL*Plus
creating CDB resource plan, 269
monitoring Resource Manager, 285
`startup pluggable database` and, 94–95

SQL*Plus, create CDB manually
add default USERS tablespace, 52
adding database to Oracle Restart, 55
creating basic parameter file, 49–50
creating catalog and load options, 53–54
creating password file, 51
creating pluggable database, 55
creating SPFILE from PFILE, 54
lock/expire all unused accounts, 54
opening PDB$SEED PDB, 53
overview of, 47–48
prerequisite steps, 49
recompiling all invalid objects, 55
setting up for catcon.pl script, 50–51
starting database instance in nomount, 51–52
updating /etc/oratab file, 50–51
SRVCTL utility
adding service for Oracle RAC PDB, 132–133
creating service, 130–133
stopping service, 129
Standard Edition, single-tenant in, 74–77
standby database
Active Data Guard option for, 291
in cloud, 315–318
Cloud Control, 314–315
create physical, 291–292
creating with Cloud Control, 304–308
duplicating with RMAN. *See* standby database, creating with RMAN
evolving into Data Guard, 290
as exact copy of primary database, 290
impact of flashback on, 223–224
logical, 362–363
managing physical, in multitenant, 308–314
standby database, creating with RMAN

back up source database, 293
choose subset of source database, 295–296
further configuration, 304
overview of, 292
password file/ temporary parameter file, 293–295
run duplicate process, 295
set up network, 293
set up static network services, 292–293
start Data Guard broker/configuration, 297–299
test configuration, 302–303
verify configuration/fill in missing pieces, 299–302
`STANDBY_FILE_MANAGEMENT=AUTO` parameter, create new PDB on source database, 309–311
`standbys` clause, 309–311, 313
`START` request, multitenant support in GoldenGate, 358
startup, CDB, 90–91
`startup pluggable database` statement, opening/closing PDBs, 94
state
modifying PDB, 61, 64
opening/closing PDBs, 95–96
viewing PDB, 93, 97
static connection, testing RMAN standby database, 302–303
static network definition, creating RMAN standby database, 292–293
Statspack, statistics collection, 33
status
adding service for Oracle RAC PDB, 132–133
listing all containers and their, 21–22
reviewing PDB, 64
storage
consolidation, 5
limit, for PDBs, 98
snapshots, 104, 323–324
storage area network (SAN), 5

STORAGE clause, creating new PDB from PDB$SEED, 59
strace utility, viewing LREG, 122
Streams
    as deprecated, 348–349
    non-CDBs supporting, 73
SUBMIT_PENDING_AREA procedure, 268
subplan, Resource Manager, 262
subsetting, create standby with RMAN, 295–296
supplemental logging, multitenant support in GoldenGate, 352
switchover, testing standby database, 302–303
synonyms
    database links and public, 8
    schema names using private, 8
SYSAUX tablespace
    cloning with proxy PDBs, 246
    restore/recover PDB with RMAN, 202–204
    storing metadata in, 11–12
    tablespace point-in-time recovery, 200
SYSBACKUP privilege
    for backup and recovery, 174–175
    in full PDB backup, 180
SYS.COL$, column metadata stored in, 13
SYSDBA privilege
    dropping PDB, 99–100
    flashback PDB, 222
SYS.TAB$, table/dictionary table metadata stored in, 13–14
system change number (SCN), Golden Gate replication, 356–357
system data, in CDB$ROOT, 72
system dictionary
    connecting to containers, 23–24
    databases storing metadata in, 11–12
    dictionary views from containers, 26–27

identifying containers, 20–23
list of containers, 21–22
multitenant containers, 14–16
multitenant dictionaries, 16–20
previous versions of, 11–14
system metadata
in CDB$ROOT, 72
separating application metadata from, 18–19
vs. application metadata, 13–14
system packages, stored in CDB$ROOT, 16
system privilege, Resource Manager, 263–264
system roles, avoid mixing with user roles, 153
SYSTEM tablespace
CDB$ROOT container and, 16–17
metadata stored in, 11–12
proxy DBs cloning, 246
restore/recover PDB with RMAN, 202–204
in tablespace point-in-time recovery, 200

T

tables
consolidation of, 9, 11
creating CDB resource plan, 269
definitions stored in dictionary, 13
linking across containers, 330–338
sharing data across PDBs, 325–329
system vs. application metadata, 13–14
TDE encrypting columns of, 163
tablespace point-in-time recovery (TSPITR), 200–201
tablespace(s)
application containers, 81
backing up database, 104
CDB administrator restricting, 159

metadata stored in, 11–12
nodata clones and, 243
partial CDB backup of, 178–179
partial PDB backup of, 181–182
recover CDB$ROOT, 186
recover lost PDB, 188–189
restore complete PDB, 201
schema consolidation and, 8
splitting PDB and, 243
storage snapshots/copy on write, 323–324
SYSAUX. *See* SYSAUX tablespace
SYSTEM. *See* SYSTEM tablespace
TDE encrypting all data in, 164
temporary. *See* temporary tablespace
transportable. *See* transportable tablespace
UNDO. *See* UNDO tablespace
USERS. *See* USERS tablespace
`TAG` option, RMAN, 170, 176–177
target database, multitenant support in GoldenGate, 354–355
TDE (Transparent Data Encryption)
Cloud Control setup, 161
creating keystore, 160–161
creating master key, 162–163
encrypting data, 163–164
keystore location setup, 160
opening keystore, 162
overview of, 159
plug and clone with, 164–165
setting up, 159–160
summary of, 165
verifying created keys, 163
TEMP file trap, plugging in non-CDB, 249–250
`TEMPFILE REUSE`, plugging in non-CDB, 150
tempfiles, recovery from lost CDB, 186–187

temporary tablespace
common to all containers in CDB, 28–29
modifying entire CDB, 92
modifying PDB, 98
proxy DBs cloning, 246–247
terminologies, Resource Manager, 261–263
testing, standby database created with RMAN, 302–303
third-parties
logical replication, 363
LogMiner/creating own log miner, 348
thread processes, killing, 125
`THREADED_EXECUTION=TRUE`, configuring multithreaded option, 124
TIMESTAMP WITH LOCAL TIME ZONE, modifying root container, 92
TNS (Transparent Network Substrate) connection string, 342
tnsnames.ora file
dedicated listener for PDBs, 137
network setup for standby with RMAN, 293
service names and, 127–129
topology, multitenant support in GoldenGate, 350
transactions
opening in another container, 24
point-in-time recovery using auxiliary instance, 205–206
roll-forward recovery phase and, 200–201
rollback recovery phase and, 200–201, 204
UNDO containing information about all, 206
Transparent Data Encryption. *See* TDE (Transparent Data Encryption)
Transparent Network Substrate (TNS) connection string, 342
`TRANSPORT TABLESPACE` command, 322
transportable database, 253–255
transportable tablespace
data movement with single-tenant and, 75
limitations of, 255
schema consolidation and, 7

sharing common read-only data, 322–323
transportable database vs., 253–254
use case for, 255–256
triggers
Oracle LogMiner vs. use of, 348
PDB operations, 252–253
setting container, 26
TSPITR (tablespace point-in-time recovery), 200–201

U

UID, in clone operations, 240
UNDO mode
changing, 208–209
defining to local in 12.2, 206–207
impact on standby of changing, 223
UNDO tablespace vs., 28
UNDO tablespace
common to all containers in CDB, 28
local UNDO in 12.2, 206–211
modifying root container, 92
PDBPITR in local UNDO, 212–213
point-in-time recovery and, 200
recover PDB to point-in-time, 204–205
restore/recover PDB with RMAN, 202–204
unplug operation. *See* PDBs, moving files with plug-in/unplug
UNPLUGGED status, PDBs, 21–22
UNUSABLE status, PDBs, 21–22
`UPDATE_CDB_ DEFAULT_DIRECTIVE`, CDB resource plan directive, 268
`UPDATE_CDB_PLAN`, CDB resource plan, 273–274
`UPDATE_CDB_PLAN_DIRECTIVE`, CDB resource plan, 273–274
updates
autotask directive in CDB resource plan, 268
CDB resource plan directives, 273–274

default directive in CDB resource plan, 268
upgrades
compatibility check for, 236
logical standby used in rolling, 362–363
overview of, 100–101
upgrades, CDB
backup or restore point, 103–104
with catupgrd.sql, 105–107
estimating length of time for, 110
open normally, 108
overview of, 101
patching, 112–113
plugging in, 111–112
post-upgrade scripts, 109
pre-upgrade, 101–103
pre-upgrade script, 104–105
test and open service, 109–110
upgrade resume after failure, 108–109
using Database Upgrade Assistant, 110–111
USER$ data dictionary table, user and role definitions, 141
user security
common vs. local users, 140–141
conflict resolution, 148–150
creating common users, 144–146
creating local users, 142–143
creating users with CREATE USER, 142
granting common privileges, 147–148
granting local privileges, 146–147
identifying system users, 142
understanding, 141–142
user tables
adding query hint, 328
querying data from PDB, 328–329

querying only some PDBs, 328
simple status query on, 325–327
users
creating roles, 153
proxy, 153
USERS tablespace
creating CDB using SQL*Plus, 52
partial CDB backup of, 178–179
partial PDB backup of, 182
restoring/recovering PDB with RMAN, 202
`USER_TABLESPACES` clause, cloning remote PDB, 243
utilization limits
autotask resource plan directives, 267–268
creating CDB resource plan, 270–273
setting in CDB resource plan, 266–267
using instance caging with Resource Manager, 283–284
`utilization_limit` parameter, CDB resource plan, 266
utluppkg.sql script, CDB, 101
utrlp.sql script, 55

V

V$ views, 151
`VALIDATE` command, block corruption check, 193–194
`VALIDATE DATABASE` command, standby configuration, 299–300
`VALIDATE_PENDING_ AREA` procedure, default resource plan directive, 268
validation, default resource plan directive, 268
/var/tmp, restore/recover PDBs in RMAN, 203
V$DATABASE view, 27
`v$encrypted_tablespaces`, viewing, 164
versions, logical replication pairing databases with mismatched, 346
views. *See also* V$ views
dictionary, 17–18

PDB operation history, 97
state of PDBs, 97
virtual machines, 5
virtualization
Oracle Database 12c, 5
overview of, 10
pros and cons of consolidation, 11
v$logmnr_contents view, LogMiner, 348
V$PARAMETER view, 116
V$PDB_INCARNATION view, 221–222
V$PDBS view, 96
V$PROCESS view, 125
V$RMAN_*CONFIGURATION* view, 173–174
V$SERVICES view, 131–132
V$SESSION view, 177

W

wallet directory, specifying backup, 163

X

XML file
contents of, 232–235
moving PDB archive files, 238–239
plug-in compatibility for, 235
plugging in non-CDBs, 248–249
unplugging/plugging in PDBs, 230–231
XStream, logical replication with, 361–363