Oracle Database 12*c* Release 2 Multitenant

Administer a High-Performance Multitenant Database Architecture

Anton Els

Vít Špinka

Franck Pachot

Oracle Press™

Oracle Press™

Oracle Database 12c Release 2 Multitenant

Anton Els
Vít Špinka
Franck Pachot

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

ISBN: 978-1-25-983610-7
MHID: 1-25-983610-X.

The material in this eBook also appears in the print version of this title:
ISBN: 978-1-25-983609-1, MHID: 1-25-983609-6.

eBook conversion by codeMantra
Version 1.0

I would like to dedicate this to my wife, Mildie.
This would not have been possible without
your encouragement and support.
Thank you, you are the best!

—Anton Els

To my wife and kids, who have to put up with me
spending too much time investigating Oracle internals.

—Vít Špinka

I dedicate this book to the great people I've encountered during my professional life. I was lucky to work, from the beginning, with the kind of people who trust you and help you to improve your skills. I wish the same to every beginner starting in IT.

—Franck Pachot

About the Authors

Anton Els, **Oracle ACE**, is Vice President Engineering at Dbvisit Software Limited. Anton has more than 15 years of experience with Oracle technology, with a particular focus on the Oracle Database, backup and recovery, standby databases, Oracle Linux, virtualization, and docker. Anton is an active member of the Independent Oracle Users Group (IOUG) and vice president of the New Zealand OUG (NZOUG). He is an Oracle Certified Master (OCM) Oracle Database 11*g;* Oracle Certified Professional (OCP) Oracle Database 8*i* to 12*c*; Oracle Certified Expert (OCE) Oracle Real Application Clusters 11*g* and Grid Infrastructure Administrator; Red Hat 5 RHSA; and Oracle Solaris 10 SCSA. He regularly presents at industry and user group conferences, such as Collaborate, Oracle OpenWorld dbtech showcase Japan, NZOUG, and the Asia-Pacific and Latin America Oracle Technology Network Tour. He can be reached on Twitter @aelsnz or through his blog at www.oraclekiwi.co.nz.

Vít Špinka, **Oracle ACE Associate**, is Chief Architect at Dbvisit Software Limited. He has more than 15 years of experience with Oracle technology, with a particular focus on the Oracle Database. Vít is an active member of IOUG and a regular presenter at Oracle OpenWorld, Collaborate, UKOUG, DOAG, and NZOUG. He is an Oracle Certified Master (OCM) Oracle Database 10*g*, 11*g*, and 12*c*; Oracle Certified Professional (OCP) Oracle Database 9*i* to 12*c*; Oracle Database 10*g* Real Application Cluster Administration Expert; and LPIC-2 Linux Network Professional. He can be reached on Twitter @vitspinka or through his blog at http://vitspinka.blogspot.com/.

Franck Pachot, **Oracle ACE Director**, is Principal Consultant, trainer, and Oracle Technology Leader at dbi services (Switzerland) with more than 20 years of experience with Oracle technology. Franck is a regular presenter at Oracle OpenWorld, IOUG Collaborate, DOAG, SOUG, and UKOUG; an active member of the SOUG and DOAG user groups; and a proud member of the OraWorld Team. He is an Oracle Certified Master (OCM) Oracle

Database 11*g* and 12*c*, Oracle Certified Professional (OCP) Oracle Database 8*i* to 12*c*, and Oracle Certified Expert (OCE) Oracle Database 12*c*: Performance Management and Tuning, and also holds an Oracle Exadata Database Machine 2014 Implementation Essentials certification. Franck can be reached on Twitter @franckpachot or through his blog at http://blog.pachot.net.

About the Technical Editors

Deiby Gómez was both the youngest Oracle ACE (23 years old) and ACE Director (25 years old) in the world and the first in his home country, Guatemala. He is also the youngest Oracle Certified Master 11*g* (OCM 11*g*, February 2015) in Latin America (24 years old) and the first in Guatemala. In addition, Deiby also became the youngest Oracle Certified Master 12*c* in the world (26 years old, April 2016) and the first OCM 12*c* in Central America. He is the recent winner of *SELECT Journal* Editor's Choice Award 2016 (Las Vegas, NV) and a frequent speaker at Oracle events around the world, including Oracle Technology Network Latin American Tour 2013, 2014, 2015, and 2016; Collaborate (United States); and Oracle Open World (Brazil and United States). The first Guatemalan accepted as a beta tester by Oracle Database (version 12*c*R2), Deiby has had several articles published in English, Spanish, and Portuguese on Oracle's website, DELL's Toad World, and hundreds more on his blog. He appeared in *Oracle Magazine* in Nov/Dec 2014 as an outstanding expert and currently serves as the President of Oracle Users Group of Guatemala (GOUG), Director of Support Quality in Latin American Oracle Users Group Community (LAOUC), and co-founder of OraWorld Team. He also currently provides Oracle services in Latin America with his own company, NUVOLA, S.A.

Arup Nanda has been an Oracle DBA for more than 20 years, with experience spanning all aspects, from modeling to performance tuning and Exadata. He has written about 500 published articles, co-authored five books, delivered 300 sessions, blogs at arup.blogspot.com, and mentors new and seasoned DBAs. He won Oracle's DBA of the Year in 2003 and Enterprise Architect of the Year in 2012, and he is an ACE Director and a member of Oak Table Network.

About the Technical and Language Editor

Mike Donovan joined the Dbvisit team in 2007, where he has held a number of different roles, including head of the Global Support team and Digital Business Development pioneer. He has recently been appointed to the role of Chief Technology Officer (CTO). He is enthusiastic about new technologies and working with customers and partners to conceive of and build bridges between the existing RDBMS world and the new frontiers of Big Data, for businesses' benefit. He is motivated by championing smart, cost-effective approaches and alternatives. Mike has a diverse background in technology and the arts and considerable experience in technical customer support and software development. He is passionate about Oracle Database technology, having worked with it for more than a decade; spoken at numerous industry conferences including OOW, RMOUG, dbtech showcase Japan, and Collaborate; spent time as a production DBA; and gained certifications on this RDBMS platform in versions 9*i* to 12*c*.

Contents at a Glance

PART I

What Multitenant Means

1 Introduction to Multitenant

2 Creating the Database

3 Single-Tenant, Multitenant, and Application Containers

PART II

Multitenant Administration

4 Day-to-Day Management

5 Networking and Services

6 Security

PART III

Backup, Recovery, and Database Movement

7 Backup and Recovery

8 Flashback and Point-in-time-Recovery

9 Moving Data

PART IV

Advanced Multitenant

10 Oracle Database Resource Manager

11 Data Guard

12 Sharing Data Across PDBs

13 Logical Replication

Index

Contents

Introduction

PART I
What Multitenant Means

1 Introduction to Multitenant
History Lesson: A New Era in IT
The Road to Multitenant
Schema Consolidation
Table Consolidation
Server Consolidation
Virtualization
Multiple Databases Managed by One Instance
Summary of Consolidation Strategies
The System Dictionary and Multitenant Architecture
The Past: Non-CDB
Multitenant Containers
Multitenant Dictionaries
Working with Containers
What Is Consolidated at CDB Level
Data and Metadata at CDB Level
Summary

2 Creating the Database
Creating a Container Database (CDB)
What About OMF?
CDB Creation Options
Creating a Pluggable Database
Create a New PDB from PDB$SEED
Create a New PDB Using the Local Clone Method

Create a PDB Using SQL Developer
Create a PDB Using the DBCA
Create a PDB Using Cloud Control
Using the catcon.pl Script
Summary

3 Single-Tenant, Multitenant, and Application Containers
Multitenant Architecture Is Not an Option
Non-CDB Deprecation
Noncompatible Features
Single-Tenant in Standard Edition
Data Movement
Security
Consolidation with Standard Edition 2
Single-Tenant in Enterprise Edition
Flashback PDB
Maximum Number of PDBs
Using the Multitenant Option
Application Containers
Consolidation with Multitenant Option
Summary

PART II
Multitenant Administration

4 Day-to-Day Management
Choosing a Container to Work With
Managing the CDB
Create the Database
Database Startup and Shutdown
Drop the Database
Modify the Entire CDB
Modify the Root
Managing PDBs
Create a New PDB

Open and Close a PDB
View the State of PDBs
View PDB Operation History
Run SQL on Multiple PDBs
Modify the PDB
Drop a PDB
Patching and Upgrades
Upgrade CDB
Plugging In
Patching
Using CDB-Level vs. PDB-Level Parameters
CDB SPFILE
PDB SPFILE Equivalent
SCOPE=MEMORY
Alter System Reset
ISPDB_MODIFIABLE
Container=ALL
DB_UNIQUE_NAME
Summary
5 Networking and Services
Oracle Net
The Oracle Net Listener
The LREG Process
Networking: Multithreaded and Multitenant
Service Names
Default Services and Connecting to PDBs
Creating Services
Create a Dedicated Listener for a PDB
Summary
6 Security
Users, Roles, and Permissions
Common or Local?
What Is a User?
CONTAINER=CURRENT

CONTAINER=COMMON
Local Grant
Common Grant
Conflicts Resolution
Keep It Clear and Simple
CONTAINER_DATA
Roles
Proxy Users
Lockdown Profiles
Disable Database Options
Disable Alter System
Disable Features
PDB Isolation
PDB_OS_CREDENTIALS
PATH_PREFIX
CREATE_FILE_DEST
Transparent Data Encryption
Setting Up TDE
Plug and Clone with TDE
TDE Summary
Summary

PART III

Backup, Recovery, and Database Movement

7 Backup and Recovery

Back to Basics
Hot vs. Cold Backups
RMAN: The Default Configuration
RMAN Backup Redundancy
The SYSBACKUP Privilege
CDB and PDB Backups
CDB Backups
PDB Backups
Do Not Forget Archive Logs!

Recovery Scenarios
Instance Recovery
Restore and Recover a CDB
Restore and Recover a PDB
RMAN Optimization Considerations
The Data Recovery Advisor
Block Corruption
Using Cloud Control for Backups
Back Up to the Cloud
Summary

8 Flashback and Point-in-time Recovery
Pluggable Database Point-in-Time
Recover PDB Until Time
Where Is the UNDO?
Summary of 12.1 PDBPITR
Local UNDO in 12.2
Database Properties
Create Database
Changing UNDO Tablespace
Changing UNDO Mode
Shared or Local UNDO?
PDB Point-in-Time Recovery in 12.2
PDBPITR in Shared UNDO Mode
PDBPITR in Local UNDO Mode
Flashback PDB
Flashback Logging
Flashback with Local UNDO
Flashback in Shared UNDO
Restore Points at the CDB and PDB Levels
Clean Restore Point
Resetlogs
Flashback and PITR
When Do You Need PITR or Flashback?
Impact on the Standby Database

Auxiliary Instance Cleanup
Summary

9 Moving Data
Grappling with PDB File Locations
Plugging In and Unplugging
Unplug and Plug In a PDB
An Unplugged Database Stays in the Source
What Exactly Is in the XML File?
Check Compatibility for Plug-In
Plug In a PDB as Clone
PDB Archive File
Cloning
Cloning a Local PDB
Cloning a Remote PDB
Application Container Considerations
Converting Non-CDB Database
Plug In a Non-CDB
Cloning a Non-CDB
Moving PDBs to the Cloud
Triggers on PDB Operations
Full Transportable Export/Import
Transportable Tablespaces
Summary

PART IV
Advanced Multitenant

10 Oracle Database Resource Manager
Resource Manager Basics
Key Resource Manager Terminologies
Resource Manager Requirements
Resource Manager Levels
The CDB Resource Plan
Resource Allocation and Utilization Limits

Default and Autotask Directives
Creating a CDB Resource Plan
The PDB Resource Plan
Creating a PDB Resource Plan
Enable or Disable a PDB Resource Plan
Removing a PDB Resource Plan
Manage PDB Memory and I/O via Initialization Parameters
PDB Memory Allocations
Limit PDB I/O
Instance Caging
Instance Caging to Resource Manager
Monitoring Resource Manager
Viewing the Resource Plan and Plan Directives
Monitoring PDBs Managed by Resource Manager
Summary
11 Data Guard
Active Data Guard Option
Creating a Physical Standby
Duplicate with RMAN
Create a Standby with Cloud Control
Managing a Physical Standby in a Multitenant Environment
Creating a New PDB on the Source
Removing PDB from Source
Changing the Subset
Cloud Control
Standby in the Cloud
Summary
12 Sharing Data Across PDBs
Database Links
Sharing Common Read-Only Data
Transportable Tablespaces
Storage Snapshots and Copy on Write
Cross-PDB Views
Simple User Tables

Consolidated Data
Cross-Database Replication
Summary

13 Logical Replication
Oracle LogMiner
Obsolete Features
Oracle CDC
Oracle Streams
Oracle Advanced Replication
Oracle GoldenGate
Multitenant Support in Oracle GoldenGate
Big Data Adapters
Oracle XStream
Logical Standby
Use in Upgrade
Other Third-Party Options
Dbvisit Replicate
Dell SharePlex
Summary

Index

Introduction

Oracle Database 12*c* release 1 (12.1) introduced the new multitenant option to the world, and since this time the phrase “pluggable database” has been bandied about, often without a clear understanding of this functionality or its implications. But, simply put, multitenant is one of the most significant architectural changes to have been implemented in the Oracle Database software since its first release. It brings new features, but it also changes many of the ways in which we Oracle DBAs perform administrative tasks on a day-to-day basis. And with the second release of 12*c* (12.2), the features available with the multitenant option have been extended even further. What is clear is that the old architecture is deprecated and that multitenant is here to stay—and cannot be ignored.

The arrival of Oracle Database 12*c* Multitenant requires that DBAs adjust the way they think about and perform daily tasks. And whether you are running a single tenant or a large number of pluggable databases, there will be a substantial learning curve. So, more than covering what’s new about multitenant alone, the aim of this book is to encompass how this relates to DBA tasks, from the core day-to-day operations to advanced tasks. You can read it as an Oracle Database administration guide for the 12*c* era, and beyond, equipping you with essential “on-the-job” knowledge about new features, syntax changes, and best practice options. This book has been written by three experienced and certified DBAs and enhanced by highly skilled reviewers, all with a passion to see Oracle Database administration done with insight and excellence.

Part I introduces multitenant functionality. Key questions include why Oracle Corporation introduced it, in what ways this mimicked other RDBMSs, and whether it is actually required by the way we design and deploy our applications nowadays. Chapter 1 addresses these questions and explains the multitenant architecture. In Chapter 2, we cover the container database (CDB) creation process and how to do it properly. Because creating a CDB is now the default option in 12*c*, believe it or not we actually

encounter people who have created a CDB without knowing it. Before launching into detail on all the new multitenant features, Chapter 3 helps guide you toward making informed decisions between CDB or non-CDB, and it outlines the different editions and options available to you.

Part II gives you a sense of what will change in your day-to-day tasks when you use multitenant. The section starts with Chapter 4, which focuses on pluggable database (PDB) creation and administration. Upgrading to 12*c* is also covered, and then networking and services are detailed in Chapter 5. This is followed, in Chapter 6, by an important topic—security—in which we address isolation of PDBs, user commonality, and encryption.

Part III covers the greatly enhanced backup and duplicate operations possible at the PDB level. Chapter 7 kicks this off with a detailed look at one of the most important areas every DBA should be familiar with—backup and recovery—and how we might revert to the previous state of a database, if needed. Chapter 8 discusses how this can be achieved with flashback technology, and then follows with details on how you can perform a point-in-time recovery at the PDB level. The final chapter in this section, Chapter 9, tackles unplugging/plugging a PDB as well as cloning, transporting, and online relocation of PDBs

In Part IV you learn how to take multitenant to the next level. When consolidating multiple PDBs resource consumption, you must be aware of Resource Manager, which is the focus of Chapter 10; this often underused facility is expected to become more crucial in multitenant environments. Chapter 11 takes a look at protecting your multitenant database environment with Data Guard, because PDBs may have to interact with it, and Chapter 12 builds on this in its discussion on data sharing within a CDB. Cloud-based solutions tend to require that data be delivered in a more flexible way than physical cloning or synchronization, and this is the focus of the discussion about logical replication in Chapter 13.

PART I

What Multitenant Means

CHAPTER 1

Introduction to Multitenant

With Oracle Database 12*c*, Oracle introduced a major change to its database architecture. Before Oracle Database 12*c*, an instance could open only one database. If you had multiple databases, you would need to start multiple instances, because they were totally isolated structures, even when hosted on the same server. This differs from most other RDBMSs, where a single system can manage multiple databases.

With the release of Oracle Database 12*c*, one instance can open multiple *pluggable databases* or PDBs. Oracle has signaled that the new multitenant architecture is here to stay, with the deprecation of the old style. With or without the multitenant option, all future Oracle databases will run on the multitenant architecture, a fact that Oracle database administrators cannot ignore.

History Lesson: A New Era in IT

Let's start by taking a brief look at the history of database use, before introducing the architecture of the future. As you can see in Figure 1-1, we will not refer to dates, but version numbers, going back to the time when the Oracle Database became predominant.

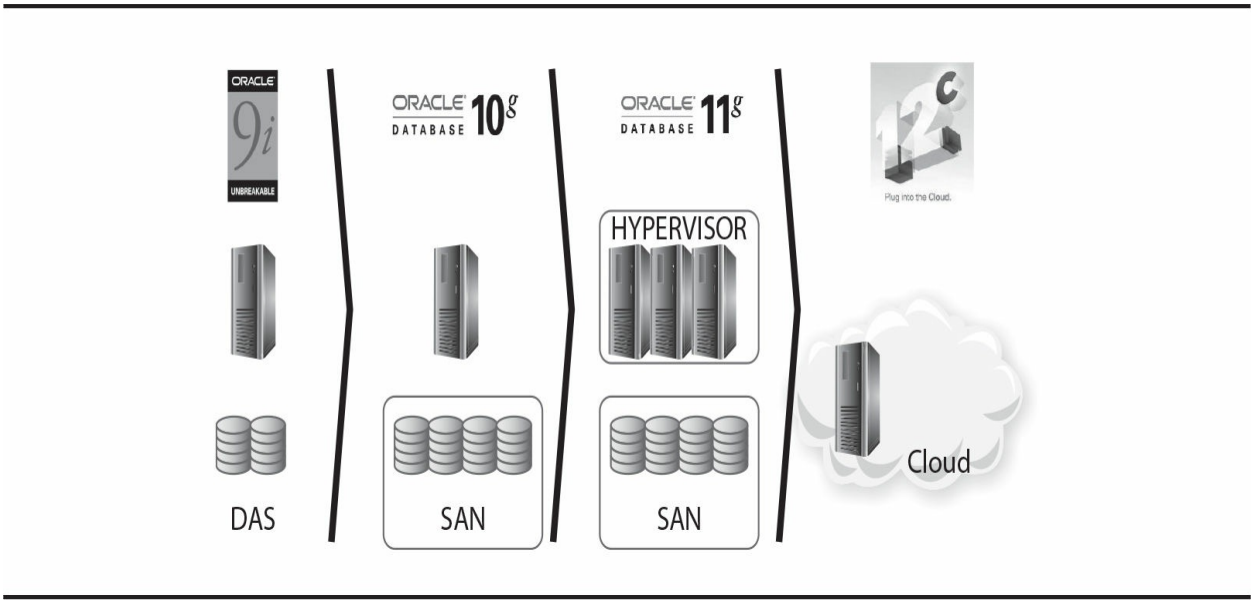


FIGURE 1-1. *From IT consolidation to the cloud*

When Oracle Database 8*i* and 9*i* were on the market, midrange computers became prevalent in data centers. We were moving from the mainframe era to a client/server era, and Oracle architecture was very well suited for that. Written in the C programming language, it was available on multiple platforms and with all user management contained within its data dictionary. It was ready for the client/server architecture, using the OS only to listen on a TCP/IP port and to store files. Furthermore, the architecture was scalable on minicomputers, thanks to the parallel server feature, which would later become RAC (Real Application Clusters).

These databases were growing along with the overall number of servers. It was common at that time to have a lot of physical servers with direct attached disks (DAS), each running one or two Oracle Database 8*i* or 9*i* instances.

With the number of databases increasing, it became a nightmare to maintain all those servers and disks. Having internal disks made capacity planning tremendously difficult when facing the exponential increase of data. By the time Oracle Database 10*g* was introduced, we needed to consolidate storage, so we put the database datafiles into a storage array, shared by all servers through a storage area network (SAN); that was storage consolidation.

By the time Oracle Database 11*g* rolled around, the prevailing thought was to do with servers what had been done with disks earlier. Instead of sizing and maintaining multiple servers, virtualization software brought us the possibility of putting our physical servers in a farm and provisioning virtual machines on top of them. That was the way in this previous era: application servers, SAN or network-attached storage (NAS), and virtual machines.

And now Oracle Database 12*c* accompanies a new day. Even organizations with consolidated storage and servers now realize that operating this infrastructure is not their core business, and instead, they now demand IT as a service, which is both scalable and flexible. Small companies want to provision their IT from a public cloud, and larger companies build their own private clouds. In both cases, virtualization can provide Infrastructure as a Service (IaaS). But we also *want* Application as a Service (AaaS) and *need* Database as a Service (DBaaS). This is a significant change in the technology ecosystem, similar in scale and importance to the earlier move from client/server to application servers. This new direction will not be

immediate—it will take time. But it is safe to predict that over the next ten years, the hybrid mixed model (on-premise/cloud) will start strong, but be slowly supplanted by the cloud.

As is expected, a new era has different requirements, and the future of databases seems bound up with consolidation, agile development, and rapid provisioning. For Oracle, some such features came progressively from Oracle Database 9*i* to 11*g*, such as easy transport of data, cloning, and thin provisioning. But two core architectural foundations came from the previous era and were not ready to accommodate consolidation: the need to run one instance per database, and having one data dictionary per database. Oracle Database 12*c* provides the answer: multitenancy. Retaining its portability philosophy, Oracle has designed this architecture to enable you to run your application on the same database, with the same software running from small server to large cloud.

The Road to Multitenant

This new era is about consolidation. Some people can imagine it as a centralized system and centralized administration, recalling the time of mainframes. But there is another challenge that comes with it: we need more and more agility. Provisioning a database is not an easy operation today, and we cannot make it worse.

Consider an example. You are an Oracle DBA. A developer comes to your desk and asks for a new database; she is under the impression that this is a simple demand, merely requiring a few clicks of an administration interface. You look at her, wide eyed, and tell her she has to fill out a request with specifics related to storage, memory, CPU, and availability requirements. Furthermore, you explain, the request will have to be approved by management, and then it will take a few days or a week to set up. And here begins a pattern of misunderstanding between dev and ops.

The developer probably hasn’t worked with Oracle databases before, so she has some notion of a database as a lightweight container for her application tables—and in many other non-Oracle RDBMSs, this is actually what is referred to as a “database.”

In Oracle, however, we have lightweight containers—schemas at logical level and tablespaces at physical level—but a database is an entity that

comprises much more than that. An Oracle database is a set of schemas and tablespaces, plus all the metadata required to manage them (the data dictionary), along with a significant amount of PL/SQL code to implement the features (the DBMS packages). Each database must have its own instance, including a number of background processes and shared memory. And each database also has a structure to protect the transactions, comprising undo tablespaces and redo logs.

For these reasons, provisioning a new database is not a trivial operation. To do so, you must interact with the system administrators and the storage teams, because you need server and disk resources for it. You don't want to put too many instances on the same server, but you can't have a dedicated physical server for each database. Because of this, today we often virtualize and have a dedicated virtual machine (VM) for each instance, but this is not possible for every application, for every environment, in any agile sort of way—there are just too many of them. Furthermore, you end up wasting a lot of resources when you have to allocate server, storage, and instance for each database.

Prior to Oracle Database 12*c*, the answer to the developer, in this scenario, probably was to create a new schema for her in an existing database. But this solution is not always possible or feasible. Let's explain why.

Schema Consolidation

Schema was exactly the objective prior to 12*c*. Each application had one schema owner, or a set of schemas if you wanted to separate tables and procedures. They were logically independent of each other, and security was controlled by grants.

Physically, you dedicated tablespaces to each application. This meant that, in case of a datafile loss, only one application was offline during the restore, which would also be the case if you wanted to relocate the tablespace to another filesystem. However, everything else was shared to optimize resource usage: instance processes and memory, SYSTEM and SYSAUX tablespaces, with dictionary.

The backup strategy is common, and the high availability (HA) policy is common. One DBA administers one database, and several applications run

on it. This is exactly what the Oracle Database was designed for from its first versions.

Transportable Tablespaces

A large number of operations in the Oracle Database can be performed at the tablespace level. This is especially true since the inception of the transportable tablespaces feature, which enables you to physically copy your application datafiles to another database, and even to a newer version. Transportable tablespaces are significant because they were a forerunner to, and an ancestor of, multitenant. The Oracle Corporation patent for Transportable Tablespaces published in 1997 was entitled “Pluggable tablespaces for database systems.” And the multitenant architecture is the foundation for pluggable databases.

In this context, *pluggable* means that you can directly plug a physical structure (datafile) into a database and have it become part of that database. The transportable tablespaces feature enabled user tablespace datafiles to be plugged into the database. Then only the metadata (dictionary entries) had to be imported so that the logical object definitions matched what was stored physically in the datafiles.

In 12*c* you can transport databases, which is nothing less than transporting all user tablespaces: a “FULL=Y” transportable tablespace. But metadata still has to be transferred logically, and that operation can be lengthy if you have thousands of tables, even if those tables are empty. For example, if you want to migrate a PeopleSoft database, which has 20,000+ tables, the metadata import alone can take hours to create all those empty tables.

As you will see, with the superior implementation in multitenant, the transport of pluggable databases is actually the transport of all datafiles, including SYSTEM and SYSAUX, which stores the data dictionary, and perhaps even the UNDO. This means that all metadata is also imported physically and, as such, is a relatively quick operation.

Schema Name Collision

Schema consolidation is in fact difficult to achieve in real life. You want to consolidate multiple applications into the same database, along with multiple

test environments of the same application, but you are faced with a number of application constraints.

What do you do if the schema owner is hard-coded into the application and you cannot change it? We were once involved in installing a telco billing application that had to be deployed in a schema called PB, and we wanted to consolidate multiple environments into the test database, but that was forbidden. The reason was that the schema name was hard-coded into the application, and in packages, and so on. We better understood that strange schema name when we hosted a consultant from the application vendor. You may be able to guess what his initials were.

If the application design is under your control you can avoid this problem, and needless to say, you should never hard-code the schema name. You can connect with any user and then simply set `ALTER SESSION SET CURRENT_SCHEMA` to have all referenced objects prefixed by the application schema owner. And if you have multiple schemas? It's not a bad idea to have multiple schemas for your application. For example, you can separate data (tables) from code (PL/SQL packages). That makes for good isolation and encapsulation of data. But even in that case, you don't need to hard-code the table schema name into the package. Just create synonyms for them into the package schema, which will reference the objects from the table schemas. You reference them from your PL/SQL code without the schema name (synonyms are in the same schema), and they are resolved to the other schema. If a name changes, you have to re-create only those synonyms. That can be done very easily and automatically.

Public Synonyms and Database Links

With the above-mentioned synonyms, we were talking about private synonyms, of course. Don't use public synonyms. They cannibalize the whole namespace. When an application creates public synonyms, you cannot consolidate anything else on it. That's a limitation for schema consolidation: objects that do not belong to a specific schema can collide with other applications and other versions or environments of the same application.

Roles, Tablespace Names, and Directories

An application can define and reference other objects, which are in the database public namespace—such is the case for roles, directories, and

tablespace names. An application for which several environments can be consolidated into the same database must have parameters in the Data Definition Language (DDL) scripts so that those database objects names can be personalized for each environment. If this is not the case, schema consolidation will be difficult.

Those public objects that do not pertain to a schema also make data movement more complex. For example, when you use Data Pump to import a schema, those objects may need to have been created earlier.

Cursor Sharing

Even with an application that is designed for schema consolidation, you may encounter performance issues when consolidating everything into the same database. We once had a database with 3000 identical schemas. They were data marts: same structure, different data.

And, obviously, the application code was the same. The user connected to one data mart and ran the queries that were coded into the applications. This meant that the same queries—exactly the same SQL text—were run on different schemas. If you know how cursor sharing works in Oracle, you can immediately see the problem: one cursor has thousands of child cursors. A parent cursor is shared by all identical SQL text, and child cursors are created when the objects are different, which is the case when you are not on the same schema. Parsing has to follow a long chained list of children cursors, holding latches during that time, and that means huge library cache contention.

In multitenant, the parent cursors are shared for consolidation purposes, but enhancements may be implemented in the child cursor search to alleviate this problem.

Table Consolidation

When you want to consolidate the data for multiple environments of the same application and same version of an application, which means that the tables have exactly the same structure, you can put everything into the same table. This is usually done by adding an environment identifier (company, country, market, and so on) into each primary key. The advantage of this is that you can manage everything at once. For example, when you want to add an index,

you can add it for all environments.

For performance and maintenance reasons, you can separate the data physically by partitioning those tables on the environment ID and put the partitions into different tablespaces. However, the level of isolation is very low, and that affects performance, security, and availability.

Actually most of the applications that were designed like this usually store only one environment. In most cases, the ID that is added in front of each primary key has only one value, and this is why Oracle introduced the skip scan index optimization. You can build virtual private database policy to manage access on those environments. You can manage the partitions independently, even at physical level, with exchange partitions. If you want to see an example of that, look at the RMAN repository: information for all registered databases is stored in the same tables. However, the isolation is not sufficient to store different environments (test, development, production), or to store different versions (where the data model is different).

Server Consolidation

If you want several independent databases but don't want to manage one server for each, you can consolidate several instances on the same server. If you go to Oracle's Ask Tom site (asktom.oracle.com/) for questions about the recommended number of instances per server, Tom Kyte's answer is this: "We recommend no more than ONE instance per host—a host can be a virtual machine, a real machine, we don't care—but you want ONE HOST = ONE INSTANCE." In real life, however, most database servers we have seen have several instances running on them. You can install multiple versions of Oracle (the ORACLE_HOME), and you can have a lot of instances running on one server—and you often have to do it. We have seen servers running as many as 70 instances.

There are few ways to isolate the resources between instances. As for memory, you can divide the physical memory among instances by setting the shared memory with `SGA_MAX_SIZE`, and in 12*c* you can even limit the process memory with `PGA_AGGREGATE_LIMIT`. You can also limit the CPU used by each instance with instance caging, setting for each instance the maximum number of processes that can run in the CPU. And with the latest license, Standard Edition 2, you don't even need Enterprise Edition to do instance caging. We will come back to this in Chapter 3.

Running a lot of instances on one server is still a problem, however. For example, when you reboot the server, you will have lot of processes to start and memory to allocate. A server outage, planned or not, will penalize a lot of applications. And you waste a lot of resources by multiplying the System Global Area (SGA) and database dictionaries.

Virtualization

Today, virtualization is a good way to run only one instance per server without having to manage a lot of physical servers. You have good isolation of environments, you can allocate CPU, memory, and I/O bandwidth, within limits. And you can even isolate them on different networks. However, even if those servers are virtual machines, you don't solve the resource wastage of multiple OSs, Oracle software, memory, and the dictionary. And you still have multiple databases to manage—to back up, to put in high-availability, in Data Guard, and so on. And you have multiple OSs to patch and monitor.

In addition to that, virtualization can be a licensing nightmare. When Oracle software is licensed by the processors where the software is installed, Oracle considers that, on some virtualization technologies, the software is installed everywhere the VM can run. The rules depend on the hypervisor vendor and on the version of this hypervisor.

Multiple Databases Managed by One Instance

The idea, then, is to find the consolidation level that fits both the isolation of the environment and the consolidation of resources. This is at a higher level than schema consolidation, but at a lower level than the instance and the database as we know it today. It means you can have several databases managed by the same instance on the same server.

This did not exist in versions of Oracle Database prior to 12*c*, but it is now possible with multitenant architecture. Now, one consolidation database can manage multiple pluggable databases. In addition to a new level that can be seen as an independent database, the pluggable database architecture brings agility in provisioning, moving, and upgrading.

Summary of Consolidation Strategies

Table 1-1 briefly summarizes the different consolidation alternatives prior to multitenant.

Consolidation	Pros	Cons
Table	Manage all as one	Very limited isolation Not for different environments
Schema	Share instance, dictionary, and HA	Public objects collision Limited isolation
Database	Only one server to administer	Several SGA, background processes Multiplies backup and HA configuration
Virtualization	Best isolation, separation of duties HA and vMotion features	Licensing nightmare New technology to learn Lot of hosts to run and manage

TABLE 1-1. *Consolidation Strategies Pros and Cons*

The System Dictionary and Multitenant Architecture

The major change in the multitenant architecture regards the system dictionary. Let's see how it was implemented in all previous versions and what changed in 12*c*.

The Past: Non-CDB

A database stores both data and metadata. For example, suppose you have the EMP table in the SCOTT schema. The description of the table—its name, columns, datatypes, and so on—are also stored within the database. This description—the metadata—is stored in a system table that is part of the dictionary.

The Dictionary

Codd's rules (created by E. F. Codd, who invented the relational model) defines that a RDBMS must represent metadata in the same way as the data: you can query both using SQL queries. As a database administrator, you do that every day. You query the dictionary views, such as DBA_TABLES, to get information about your database objects. This rule is for logical representation only, and the dictionary views provide that. But Oracle went further by deciding to store physically the metadata information in relational tables—the same kind of tables as application tables, but they are owned by SYS schema and stored in the system's tablespaces (SYSTEM and SYSAUX).

Without using the actual name and details of the Oracle dictionary, Figure 1-2 gives you the idea. Table SCOTT.DEPT stores user data. The definition of the table is stored in a dictionary table, SYS.COLUMNS, because it stores column information here. And because this table is itself a table, we have to store its definition in the same way.

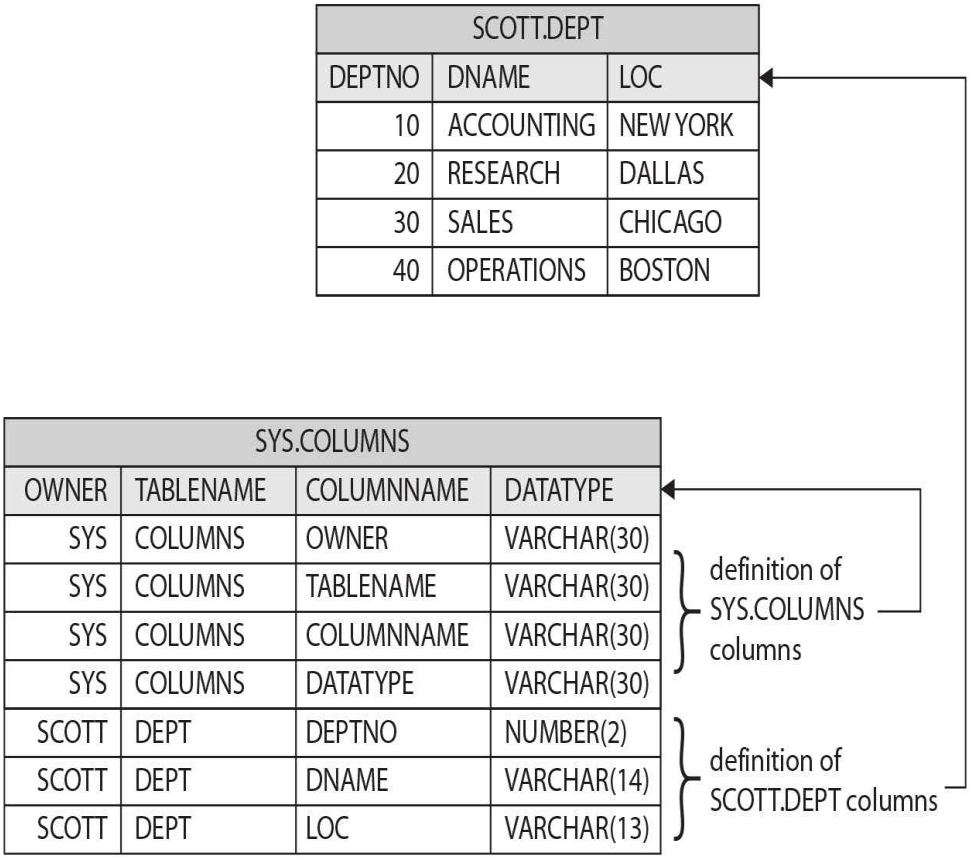


FIGURE 1-2. *Storing metadata with data*

And there are not only table definitions in the dictionary. Until version 8*i*, even the physical description of the data storage (table extents) were stored in the dictionary. That changed with Locally Managed Tablespaces, however, when tablespace became more self-contained in preparation of pluggable features. On the other hand, at each new version, a lot of new information was added to the dictionary. In the current version, a large part of the Oracle Database software is implemented as PL/SQL packages, which are stored in the dictionary.

Oracle-Maintained Objects

The implementation choice we've described is specific to the Oracle RDBMS: The dictionary is stored in the database. Each database has its own

dictionary. And if you used logical export/import (EXP/IMP or Data Pump) to move a database, you probably have seen how it is difficult to distinguish the dictionary objects belonging to the system from the user objects belonging to the applications. When you import a full database (`FULL=Y` in IMPDP options, as discussed in Chapter 8) into a newly created database, you don't want to import the dictionary because it already exists in the target.

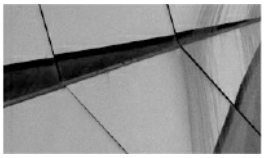
Of course, objects in the SYS schema are dictionary objects, and they are ignored by Data Pump. But if someone has created user objects, then they are lost. Grants on SYS objects are lost. And you can find system objects elsewhere, such as OUTLN, MDSYS, XDB, and so on. A lot of roles come from the system, and you can create your own role. It's difficult to distinguish them easily.

Fortunately, 12*c* includes a flag in DBA_OBJECTS, DBA_USERS, and DBA_ROLES to identify the Oracle-maintained objects that are created with the database and that do not belong to your application. Let's query the Oracle-maintained schemas list from a 12*c* database:

```
SQL> select listagg(username,',' on overflow truncate with count) within group
(order by created) from dba_users where oracle_maintained='Y';

LISTAGG(USERNAME,','ONOVERFLOWTRUNCATEWITHCOUNT)WITHINGROUP(ORDERBYCREATED)
--------------------------------------------------------------------------------
AUDSYS,SYS,SYSBACKUP,SYSDG,SYSKM,SYSRAC,SYSTEM,OUTLN,GSMADMIN_INTERNAL,GSMUSER,
DIP,XS$NULL,REMOTE_SCHEDULER_AGENT,DBSFWUSER,ORACLE_OCM,SYS$UMF,DBSNMP,APPQOSSYS,
GSMCATUSER,GGSYS,XDB,ANONYMOUS,WMSYS,OJVMSYS,CTXSYS,MDSYS,ORDDATA,ORDPLUGINS,OR
DSYS,SI_INFORMTN_SCHEMA,OLAPSYS,MDDATA,SPATIAL_WFS_ADMIN_USR,SPATIAL_CSW_ADMIN_
USR,LBACSYS,APEX_050000,APEX_PUBLIC_USER,FLOWS_FILES,DVF,DVSYS
```

This is a big improvement in 12*c*. You can easily determine what belongs to your application and what belongs to the system itself. The ORACLE_MAINTAINED flag is present in DBA_OBJECTS, DBA_USERS, and DBA_ROLES views and it's now easy to distinguish the objects created at database creation from those created by your application.



NOTE

Before 12c, you could try to list the objects from different views used by Oracle internally. There are the views used by the data movement, listing what they must ignore: EXU8USR for EXP/IMP, KU_NOEXP_TAB for Data Pump, and LOGSTDBY$SKIP_SUPPORT for Data Guard. There is also the DEFAULT_PWD$ table to identify some pre-created schemas. And you can also query the V$SYSAUX_OCCUPANTS or DBA_REGISTRY views.

System Metadata vs. Application Metadata

We described the metadata structures: schemas, objects, and roles. Let's go inside them, into the data. You know that table definitions are stored in dictionary tables, and in Figure 1-2 we simplified this in a SYS.COLUMN table. But the dictionary data model is more complex than that. Actually, object names are in SYS.OBJ$, table information is in SYS.TAB$, column information is in SYS.COL$, and so on. Those are tables, and each has its own definition—the metadata—which is stored in that dictionary: SYS.TAB$, for example, has rows for your tables, but it also has rows for all dictionary tables.

SYS.TAB$ has a row to store the SYS.TAB$ definition itself. You may ask how that row is inserted at the table creation (which is database creation), because the table does not yet exist. Oracle has a special bootstrap code that is visible in the ORACLE_HOME. (It's beyond the scope of this book, but you can look at the dcore.bsq file in ORACLE_HOME/rdbms/admin directory. You can also query the BOOTSTRAP$ table to see the code that creates those tables at startup in dictionary cache, so that basic metadata is available immediately to allow access to the remaining metadata.)

All metadata is stored in those tables, but this is a problem in non-multitenant databases: system information (which belongs to the RDBMS itself) is mixed with user information (which belongs to the application). Both of their metadata is stored in the same tables, and everything is stored into the same container: the database.

This is what has changed with multitenant architecture: we have now multiple containers to separate system information from application information.

Multitenant Containers

The multitenant database's most important structure is the container. A container contains data and metadata. What is different in multitenant is that a container can itself contain several containers inside it in order to separate objects logically and physically. A container database contains several pluggable databases, and an additional one, the root, contains the common objects.

A multitenant database is a container database (CDB). The old architecture, in which a database is a single container that contains no subdivisions, is called a non-CDB. In 12*c* you can choose which one you want to create. You create a CDB, the multitenant one, by setting `ENABLE_PLUGGABLE=true` in the instance parameters and by adding the `ENABLE PLUGGABLE` to the `CREATE DATABASE` statement. (More details are in Chapter 2.)

This creates the CDB, which will contain other containers identified by a number, the container ID, and a name. It will contain at least a root container and a seed container, and you will be able to add your own containers, up to 252 in version 12.1, and thousands in 12.2.

Pluggable Database

The goal of multitenant is consolidation. Instead of having multiple databases on a server, we can now create only one consolidated database, the CDB, which contains multiple pluggable databases (PDBs). And each PDB will appear as a whole database to its users, with multiple schemas, public objects, system tablespaces, dictionary views, and so on.

The multitenant architecture will be used to consolidate on private or public clouds, with hundreds or thousands of pluggable databases. The goal is to provision those pluggable databases quickly and expose them as if they were a single database. By design, anyone connected to a pluggable database cannot distinguish it from a standalone database.

In addition, all commands used in previous Oracle Database versions are compatible. For example, you can run `shutdown` when you are connected to a PDB and it will close your PDB. It will not actually shut down the instance, however, because other PDBs are managed by that instance, but the user will see exactly what he would see if he shut down a standalone database.

Consider another example. We are connected to a pluggable database and we can't have undo tablespaces because they are at CDB level only (we can change this in 12.2, but you'll learn about that in Chapter 8). Let's try to create one:

```
SQL> show con_name
CON_NAME
------------------------------
PDB
SQL> create undo tablespace WHATEVER datafile '/nowhere' size 100T;
Tablespace created.
SQL> create undo tablespace WHATEVER datafile '/nowhere' size 100T;
Tablespace created.
SQL> create undo tablespace WHATEVER datafile '/nowhere' size 100T;
Tablespace created.
```

There are no errors, but the undo tablespaces are obviously not created. It's impossible to create a 100-terabyte datafile. My statements have just been ignored. The idea is that a script made to run in a database can create an undo tablespace, so the syntax must be accepted in a pluggable database. It's allowed because everything you can do in a non-CDB must be accepted in a PDB, but it is ignored because here the undo tablespace is at the CDB level only.

With multitenant, you have new commands, and all commands you know are accepted by a PDB. You can give the DBA role to a PDB user, and she will be able to do everything a DBA can do with regard to her database. And the PDB user will be isolated from the other pluggable databases and will not see what is at the CDB level.

CDB$ROOT

How big is your SYSTEM tablespace? Just after database creation, it's already a few gigabytes. By database creation, we aren't referring to the `CREATE DATABASE` statement, but running catalog.sql and catproc.sql. (Well, you don't call those directly in multitenant; it's catcdb.sql but it runs the same scripts.) The dictionary of an empty database holds gigabytes of

dictionary structures and system packages that are part of the Oracle software —as the ORACLE_HOME binaries—but they are deployed as stored procedures and packages inside the database. And if you put 50 databases on a server, you have 50 SYSTEM tablespaces that hold the same thing (assuming that they are the same version and same patche level). If you want to consolidate hundreds or thousands of databases, as you can do with PDBs, you don't want to store the same data in each one. Instead, you can put all the common data into only one container and share it with the others. This is exactly what CDB$ROOT is: it's the only container in a CDB that is not a PDB but stores everything that is common to the PDBs.

Basically, CDB$ROOT will store all the dictionary tables, the dictionary views, the system packages (those that start with *dbms_*), and the system users (SYS, SYSTEM, and so on)—and nothing else. The user data should not go in CDB$ROOT. You can create your own users only if you need them on all PDBs. You will see more about common users in Chapter 6.

You can think of CDB$ROOT as an extension of the ORACLE_HOME. It's the part of the software that is stored in the database. It's specific to the version of the ORACLE_HOME, and it is the same in all CDBs that are in the same version. Our 12.2.0.1 CDB$ROOT is mostly the same as yours.

PDB$SEED

The goal of a multitenant database, a CDB, is to create a lot of PDBs. More than that, it should be easy and quick to create PDBs on demand. It's the architecture focused on Database as a Service (DBaaS). How do you create a database quickly with the Database Configuration Assistant (DBCA)? You create it from a template that has all datafiles. No need to re-create everything (as catalog.sql and catproc.sql do) if you can clone an empty database that already exists. This is exactly what the PDB$SEED is: it's an empty PDB that you can clone to create another PDB. You don't change it, it is read-only, and you can use it only as the source of a new PDB.

A CDB has at minimum one CDB$ROOT container and one PDB$SEED container. You can't change them; you can only use them. Their structure will change only if you upgrade or patch the CDB.

Multitenant Dictionaries

One goal of the multitenant architecture is to separate system metadata from application metadata. System metadata, common to all PDBs, is stored in the CDB$ROOT, as are all system objects. Consider, for example, the package definitions. They are stored in the dictionary table SOURCE$, which we can query through the DBA_SOURCE dictionary views. In a non-CDB, this table contains both the system packages and the packages that you create—the packages owned by SYS, as well as the packages owned by your application schema; let's call it ERP. In multitenant, the CDB$ROOT contains only system metadata, so in our previous example, that means all the SYS packages.

In our PDB dedicated to the application, let's call it PDBERP, the SOURCE$ contains only the packages of our application, the ERP ones. Let's see an example. We are in the CDB$ROOT and we count the lines in SOURCE$. We join with the DBA_OBJECT that shows which are the Oracle-maintained objects (the system objects):

```
SQL> select o.oracle_maintained,count(*)
from sys.source$ s join dba_objects o on obj#=object_id
group by o.oracle_maintained;
ORACLE_MAINTAINED   COUNT(*)
----------------- ----------
Y                     362030
```

All the lines in SOURCE$ are for Oracle-maintained objects which are system packages.

Now let's have a look in a PDB:

```
ORACLE_MAINTAINED   COUNT(*)
----------------- ----------
N                       8318
```

The lines here are not for system packages, but for our own application packages. You may have a different result, but basically this is how the dictionaries are separated in multitenant: the metadata that was stored in the same dictionary in non-CDBs is now stored in identical tables but in different

containers, to keep the Oracle metadata separated from the application metadata. Note that this is not the same as partitioning; it's more like these are actually different databases for the dictionaries.

Dictionary Views

Do you know why we've queried the SOURCE$ table and not the DBA_SOURCE, which is supposed to give the same rows? Check this out:

```
SQL> select o.oracle_maintained,count(*)
from dba_source s join dba_objects o on s.owner=o.owner and s.name=o.
object_name and s.type=o.object_type
group by o.oracle_maintained;
ORACLE_MAINTAINED   COUNT(*)
----------------- ----------
Y                     362030
```

Same number of lines here in the CDB$ROOT. But when we connect to the PDB,

```
SQL> select o.oracle_maintained,count(*)
from dba_source s join dba_objects o on s.owner=o.owner and s.name=o.
object_name and s.type=o.object_type
group by o.oracle_maintained;
ORACLE_MAINTAINED   COUNT(*)
----------------- ----------
Y                     362030
N                       8318
```

We see more rows here. Actually, we see the rows from CDB$ROOT. There are two reasons for this. First, we said that what is in the CDB$ROOT is common, so it makes sense to see this from the PDB. Second, we said that a user connected to a PDB must see everything as if she were on a standalone database. And on a standalone database, a query on DBA_SOURCE shows all sources from both the system and the application. It's not the case when you query SOURCE$, but you're not expected to do that. Only the views are

documented, and you're expected to query those ones.

The dictionary views in a PDB show information from the PDB and from the CDB$ROOT. It's not partitioning, and it's not a database link. We will see how Oracle does this in the next section.

When you are connected to CDB$ROOT, the DBA_SOURCE view shows only what is in your container. But new views starting with *CDB_* can show what is in all containers, as you will see later in the chapter in the section "Dictionary Views from Containers."

So, physically, the dictionaries are separated. Each container stores metadata for its user objects, and the root stores the common ones—mainly the system metadata. Logically, from the views, we see everything, because this is what we have always seen in non-CDBs, and PDBs are compatible with that.

Metadata Links

Oracle has introduced a new way to link objects from one container to another: *metadata link*. Each container has all the dictionary objects (stored in OBJ$ and visible through DBA_OBJECTS), such as the system package names in the example used earlier. But more definitions (such as the packages source text) are not stored in all of the containers, but only in the CDB$ROOT. Each container has a flag in OBJ$, visible as the SHARING column of DBA_OBJECTS, that tells Oracle to switch to the CDB$ROOT container when it needs to get the metadata for it.

Here is some information about one of those packages, which includes the same definition in all containers, from DBA_OBJECTS in the CDB$ROOT:

```
SQL> select owner,object_name,object_id,object_type,sharing,
oracle_maintained from dba_objects where object_name = 'DBMS_SYSTEM';

OWNER  OBJECT_NAME OBJECT_ID OBJECT_TYPE  SHARING       ORACLE_MAINTAINED
------ ----------- --------- ------------ ------------- -----------------
SYS    DBMS_SYSTEM     14087 PACKAGE      METADATA LINK Y
SYS    DBMS_SYSTEM     14088 PACKAGE BODY METADATA LINK Y
```

And this is from the PDB:

```
 SQL> select owner,object_name,object_id,object_type,sharing,
oracle_maintained from dba_objects where object_name = 'DBMS_SYSTEM';

OWNER  OBJECT_NAME OBJECT_ID OBJECT_TYPE  SHARING       ORACLE_MAINTAINED
------ ----------- --------- ------------ ------------- -----------------
SYS    DBMS_SYSTEM     14081 PACKAGE      METADATA LINK Y
SYS    DBMS_SYSTEM     14082 PACKAGE BODY METADATA LINK Y
```

You can see the same object names and types, defined as Oracle maintained and with METADATA LINK sharing. They have different object IDs. Only the name and an internal signature is used to link them. From this, we know that those objects are system objects (Oracle maintained), and when we query one of them from a PDB, the Oracle code knows that it has to switch to the root container to get some of its information. This is the behavior of metadata links. The dictionary objects that are big are stored in only one place, the CDB$ROOT, but they can be seen from everywhere via dictionary views.

This concerns metadata. Metadata for your application is on your PDB. Metadata for Oracle-maintained objects is stored on the CDB$ROOT. The latter is static information: it’s updated only by upgrades and patches. You can see that the benefit is not only a reduction in duplication, but also the acceleration of the upgrades of the PDBs, as there are only links.

Figure 1-3 shows the expanded simplified dictionary from Figure 1-2 to reveal the dictionary separation.

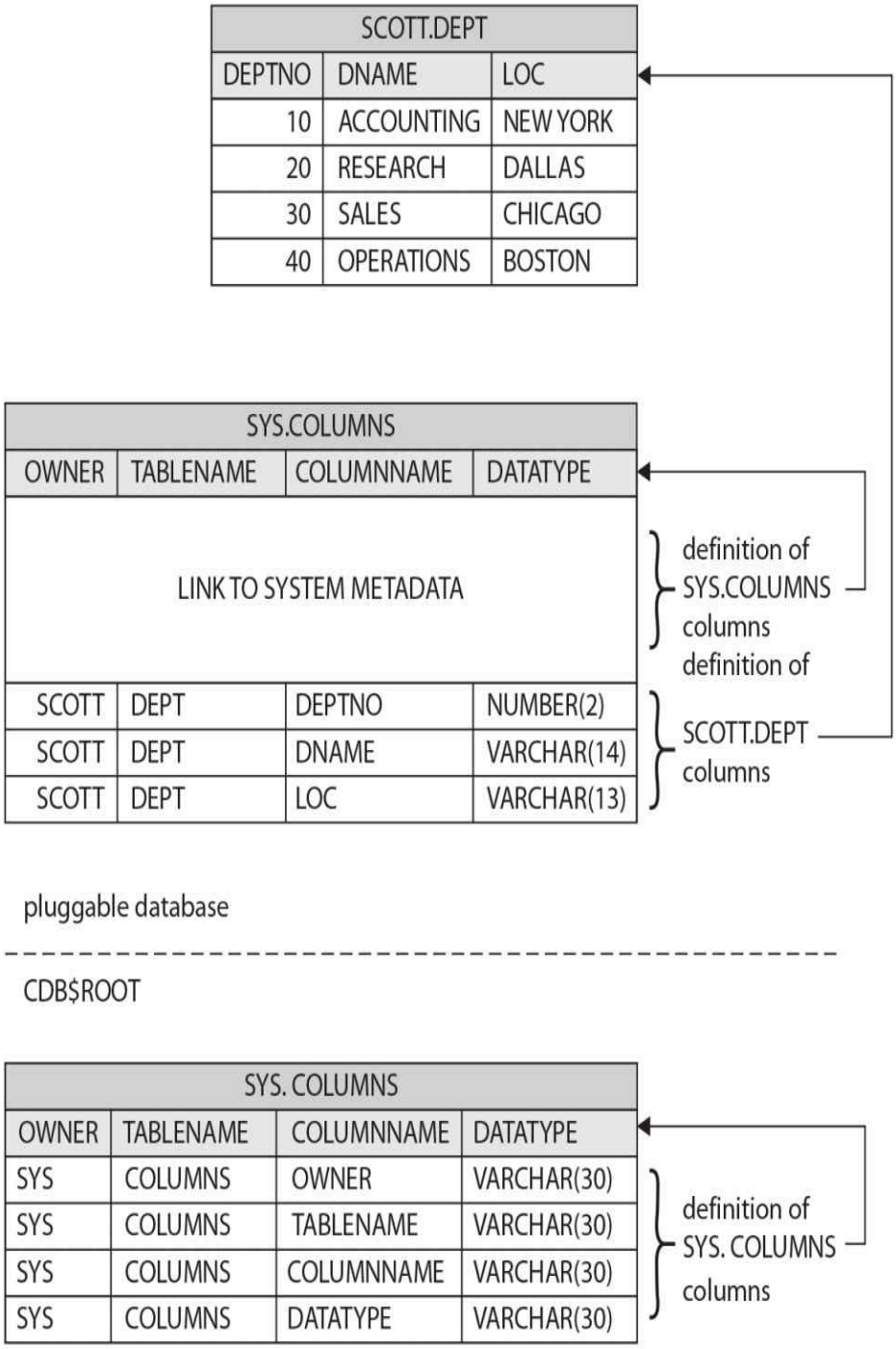


FIGURE 1-3. *Separating system and user metadata*

Data Links (called Object Links in 12.1)

There is not only metadata in the dictionary. The multitenant database also contains some data to store at the CDB level. Here's a simple example. Suppose the CDB must keep a list of its containers, which is stored in the system table CONTAINER$, which is exposed through the dictionary view DBA_PDBS. This data is updated to store the status of the containers. However, this data, which makes sense only at the CDB level, can be queried from all PDBs. Let's see how this is shared.

This is from the CDB$ROOT:

```
SQL> select owner,object_name,object_id,object_type,sharing,
oracle_maintained from dba_objects where object_name in ('CONTAINER$','DBA_PDBS');

OWNER  OBJECT_NAME OBJECT_ID OBJECT_TYPE SHARING       ORACLE_MAINTAINED
------ ----------- --------- ----------- ------------- -----------------
SYS    CONTAINER$        161 TABLE       METADATA LINK Y
SYS    DBA_PDBS         4755 VIEW        DATA LINK     Y
PUBLIC DBA_PDBS         4756 SYNONYM     METADATA LINK Y
```

And this is from the PDB:

```
OWNER  OBJECT_NAME OBJECT_ID OBJECT_TYPE SHARING       ORACLE_MAINTAINED
------ ----------- --------- ----------- ------------- -----------------
SYS    CONTAINER$        161 TABLE       METADATA LINK Y
SYS    DBA_PDBS         4753 VIEW        DATA LINK     Y
PUBLIC DBA_PDBS         4754 SYNONYM     METADATA LINK Y
```

You can see that the view that accesses CONTAINER$ is a data link, which means that the session that queries the view will read from the CDB$ROOT. Actually, the CONTAINER$ table is present in all containers, but it is always empty except in root.

Working with Containers

How do you work with so many PDBs? You begin by identifying them.

Identifying Containers by Name and ID

A consolidated CDB contains multiple containers that are identified by a name and a number, the CON_ID. All the V$ views that show what you have in an instance have an additional column in 12*c* to display the CON_ID to which the object is related. The CDB itself is a container, identified by container CON_ID=0. Objects that are at CDB level and not related to any container are identified by CON_ID=0.

For example, here's what we get if we query V$DATABASE from the root:

```
SQL> select dbid,name,cdb,con_id,con_dbid from v$database;
      DBID NAME      CDB     CON_ID   CON_DBID
---------- --------- --- ---------- ----------
2013933390 CDB       YES          0 2013933390
```

And this is from the PDB:

```
SQL> select dbid,name,cdb,con_id,con_dbid from v$database;
      DBID NAME      CDB     CON_ID   CON_DBID
---------- --------- --- ---------- ----------
2013933390 CDB       YES          0 2621919399
```

The information may be different when viewed from different containers, but in all cases, the database information is located at the CDB level only. Information in that view comes from the control file, and you will see that it is in the common ground. So the CON_ID is 0. If you are in non-CDB, CON_ID=0 for all objects. But if you are in multitenant architecture, most of the objects pertain to a container.

The first container that you have in any CDB is the root, named CDB$ROOT and with CON_ID=1. All the other containers are PDBs.

The first PDB that is present in every CDB is the seed, named PDB$SEED, which is the second container, with CON_ID=2.

Then CON_ID>2 are your PDBs. In 12.1, you can create 252 additional PDBs. In 12.2, you can create 4,098 of them.

List of Containers

The dictionary view DBA_PDBS lists all PDBs (all containers except root) with their status:

```
SQL> select pdb_id, pdb_name, status,con_id from dba_pdbs;

    PDB_ID PDB_NAME     STATUS          CON_ID
---------- ------------ ---------- ----------
         2 PDB$SEED     NORMAL              2
         3 PDB1         NORMAL              3
         4 PDB2         NEW                 4
         5 PDB3         UNUSABLE            5
         6 PDB4         UNPLUGGED           6
```

The status is NEW when you create it and is changed to NORMAL at first open read/write, because some operations must be completed at first open. UNUSABLE is displayed when the creation failed and the only operation we can do is DROP it. UNPLUGGED is a way to transport it to another CDB, and the only operation that can be done on the source CDB is to DROP it.

Instead of NEW you can see the following in 12.1 only: NEED UPGRADE indicates it came from a different version, and CONVERTING indicates it came from a non-CDB. You'll learn about three others, the RELOCATING, REFRESHING and RELOCATED statuses, in Chapter 9.

That was the information from the database dictionary. We can list the containers known by the instance, which show the open status:

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

    CON_ID NAME         OPEN_MODE  OPEN_TIME
---------- ------------ ---------- ---------------------------------
         2 PDB$SEED     READ ONLY  05-JAN-16 03.57.35.171 PM +01:00
         3 PDB1         READ WRITE 05-JAN-16 04.49.27.558 PM +01:00
         4 PDB2         MOUNTED
         5 PDB3         MOUNTED
         6 PDB4         MOUNTED
```

In non-CDB, the MOUNTED state occurs when the control file is read but the datafiles are not yet opened by the instance processes. It's the same idea here: a closed PDB does not yet open the datafiles. There is no NOMOUNT state for PDBs because the control file is common.

Note that SQL*Plus and SQL Developer have a shortcut you can use to show your PDBs or all PDBs when you are in the root container:

```
SQL> show pdbs

    CON_ID CON_NAME                       OPEN MODE  RESTRICTED
---------- ------------------------------ ---------- ----------
         2 PDB$SEED                       READ ONLY  NO
         3 PDB1                           READ WRITE NO
         4 PDB2                           MOUNTED    NO
         5 PDB3                           MOUNTED    NO
         6 PDB4                           MOUNTED    NO
```

Identify Containers by CON_UID and DBID

You have seen that in addition to its name, a container is identified by an ID, the CON_ID within the CDB. The CON_ID can change when you move the PDB. For that reason, you also have a unique identifier, the CON_UID, which is a number that identifies the PDB even after it has been moved. The CDB$ROOT that is a container but not a PDB, and does not move, has a CON_UID=1.

Because of the compatibility with databases, each container has also a DBID. CDB$ROOT is the DBID of the CDB. The DBID of PDBs is the

CON_UID.

In addition, each container has a GUID, a 16-byte RAW value that is assigned at PDB creation time and never changes after that. It is used as a unique identifier of the PDB in the directory structure when using Oracle Managed Files (OMF).

All those identifiers are in V$CONTAINER, but you can also use the functions `CON_NAME_TO_ID`, `CON_DBID_TO_ID`, `CON_UID_TO_ID`, and `CON_GUID_TO_ID` to get the ID of a container. A null is returned if a container is not there. Here are some examples:

```
SQL> select CON_NAME_TO_ID('PDB$SEED'),CON_DBID_TO_ID(794366768),CON_NAME_TO_ID('XXX')
from dual;

CON_NAME_TO_ID('PDB$SEED') CON_DBID_TO_ID(794366768) CON_NAME_TO_ID('XXX')
-------------------------- ------------------------ --------------------
                         2                        4
```

Connecting to Containers

We talked about multitenant as a way to overcome the schema-based consolidation limitation. So how do you switch between schemas, other than connecting directly as the schema user? You `ALTER SESSION SET CURRENT_SCHEMA`.

Of course, you can connect directly to a PDB, but we will explain that in Chapter 5 about services, because that is the right way to connect from users or application. But when, as a CDB administrator, you are already connected to the CDB, you can simply switch your session to a new container with `ALTER SESSION SET CONTAINER`.

Here, we are connected to CDB$ROOT:

```
SQL> show con_id
CON_ID
------------------------------
1
SQL> show con_name
CON_NAME
------------------------------
CDB$ROOT
```

We change our current container:

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

And we are now in the PDB:

```
SQL> show con_id
CON_ID
------------------------------
3
SQL> show con_name
CON_NAME
------------------------------
PDB
```

Transactions If you have started a transaction in a container, you cannot open another transaction in another container.

```
SQL> alter session set container=PDB1;
Session altered.
SQL> insert into SCOTT.DEPT(deptno) values(50);
1 row created.
```

You can leave the transaction and change the container:

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

But you can't run DML that requires a transaction:

```
SQL> delete from SCOTT.EMP;
                    *
ERROR at line 1:
ORA-65023: active transaction exists in container PDB1
```

First you must return to the previous container and finish your transaction:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> commit;
Commit complete.
```

And then you can open a new transaction in another container:

```
SQL> alter session set container=PDB2;
Session altered.
SQL> insert into DEPT(deptno) values(50);
1 row created.
```

Cursors If you open a cursor in one container, you cannot fetch from it in another container. You need to go back to the cursor's container to fetch from it:

```
SQL> variable C1 refcursor;
SQL> exec open :C1 for select * from DEMO;
PL/SQL procedure successfully completed.
SQL> alter session set container=PDB2;
Session altered.
SQL> print C1
ERROR:
ORA-65108: invalid use of a cursor belonging to another container
```

Basically, it's easy to switch from one container to another, but what we do to them is isolated. Nothing is shared with the previous container state.

For example, connected to PDB1, we set `serveroutput` to `on` and use `dbms_output`:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> set serveroutput on
SQL> exec dbms_output.put_line('==> '||sys_context('userenv','con_name'));
==> PDB1
PL/SQL procedure successfully completed.
```

The `dbms_output` line was displayed. You can see that the USERENV context shows the current container name. Now we switch to PDB2:

```
SQL> alter session set container=PDB2;
Session altered.
SQL> exec dbms_output.put_line('==> '||sys_context('userenv','con_name'));
PL/SQL procedure successfully completed.
```

Nothing is displayed here. `serveroutput` was set for PDB1, and we have to set it for PDB2:

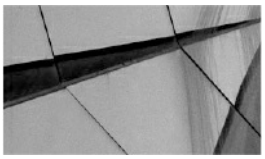
```
SQL> set serveroutput on
SQL> exec dbms_output.put_line('==> '||sys_context('userenv','con_name'));
==> PDB2
PL/SQL procedure successfully completed.
```

Now we go back to PDB1:

```
SQL> alter session set container=PDB1;
Session altered.
SQL> exec dbms_output.put_line('==> '||sys_context('userenv','con_name'));
==> PDB1
PL/SQL procedure successfully completed.
```

No need to set `serveroutput` again. When switching back, we regained the state.

From JDBC or OCI My examples were run on SQL*Plus, but any client can do this. As long as you are connected with a user that is defined in the root (a common user) and that has been granted the SET CONTAINER system privilege on a PDB, you can switch the session to the PDB. You can do it from Java Database Connectivity (JDBC) or from the Oracle Call Interface (OCI). For example, you can have a connection pool from an application server that will switch to the required container when connections are grabbed. This is a way to have a common application server for multiple database tenants.



NOTE

If you want to use some container features that are new in 12.2, such as switching to a different character set PDB, you need to have a 12.2 client or you'll get a ORA-24964: ALTER SESSION SET CONTAINER error.

Set Container Trigger If, for any reason, you want to run something when a session changes to another container, such as setting specific optimizer

parameters, you can create a BEFORE SET CONTAINER and AFTER SET CONTAINER.

Here is how it works:

- The `BEFORE SET CONTAINER` created in PDB1 will be raised when you are in PDB1 and you execute an `ALTER SESSION SET CONTAINER`. If the trigger reads the container name, it will be PDB1.
- The `AFTER SET CONTAINER` created in PDB2 will be raised when you have executed an `ALTER SESSION SET CONTAINER=PDB2`.

This means that if both PDB1 and PDB2 have before and after triggers, changing from PDB1 to PDB2 will raise a `BEFORE SET CONTAINER` in PDB1 and an `AFTER SET CONTAINER` in PDB2.

There are two ways to work in a PDB. You can connect to it through its services, which is described in Chapter 5, and then `AFTER LOGON ON PLUGGABLE DATABASE` can be used to run some code at the beginning of the session. Or you can `SET CONTAINER`, and then use the `AFTER SET CONTAINER ON PLUGGABLE DATABASE`. If you want to be sure to set some sessions settings for a user working on a PDB, you will probably define both. Note that the word `PLUGGABLE` is not mandatory because of the syntax compatibility with database behavior.

Dictionary Views from Containers

A PDB includes everything that you expect from a database, which means that the queries on the dictionary views have the same behavior. You have all the DBA_/ALL_/USER_ views to show metadata for the PDB objects, or those you have access to, or those that you own. The fact that system objects are stored elsewhere is transparent: you see system objects in DBA_OBJECTS, system tables in DBA_TABLES, and instance information in the V$ views, but only the rows that are of concern in your PDB.

When you are in CDB$ROOT, you have additional CDB_ views that are like a `UNION ALL` of all opened containers' DBA_ views. It's a way for a CDB database administrator to view all objects. For this CDB$ROOT user, the V$ views show information about all containers.

Finally, you may want to know if you are in non-CDB or in multitenant. The CDB column in V$DATABASE provides the answer:

```
SQL> select name,cdb from v$database;

NAME      CDB
--------- ---
CDB       YES
```

What Is Consolidated at CDB Level

Sharing resources that can be common is the main goal of consolidation. Beyond the instance and the dictionary, many database structures are managed at the CDB level. We are not talking about datafiles here, because they are specific to each container, and the only commonality among them is that they must have the same character set (except when a container is transported from another CDB, but that's a topic for Chapter 9). Several other files are common to all containers in a container database.

SPFILE

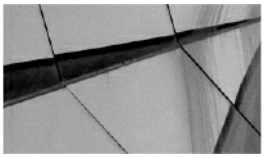
The database instance is common to all containers, and the SPFILE holds the instance parameters, storing these settings for the entire CDB. The SPFILE contains the configuration elements that cannot be stored within the database or the control file, because they must be available before the database is mounted.

Some parameters can be set at the PDB level (those with `ISPDB_MODIFIABLE=TRUE` in `V$PARAMETER`). These changes can also be persisted, but even if the syntax for such an operation is `SCOPE=SPFILE`, this is for syntax compatibility only, as PDB level parameters are actually stored in the CDB dictionary (PDB_SPFILE$). They are not stored in the PDB itself because the parameter must be accessed before opening the PDB. We will see later that when moving pluggable databases (unplug/plug), these parameters are extracted into an XML file that you ship with the PDB datafiles.

Control Files

The control file references all other structures in the database. For example, this is the only location in which the datafile names are actually stored, and in

the dictionary it is the FILE_ID alone that is used to reference them. In multitenant, the control file resides at the CDB level and holds records for all pluggable database datafiles. You will see in Chapter 9 that information relating to pluggable databases that is stored in the control file will also be exported to an XML file when a pluggable database is moved by unplug/plug.



NOTE

When we refer to the "control file," we are actually referring to the control files (plural), because you can, and must, multiplex them to protect your database. This fundamental best practice does not change, and, if anything, the importance of database availability is even higher in multitenant, because an outage will have impact on multiple tenants.

While on the subject of database files, there is a parameter that controls the maximum number of files opened by an instance, DB_FILES, which defaults to 200. Be aware that if you create hundreds of pluggable databases, you will very quickly reach this limit, and you will then be unable to create new tablespaces or new pluggable databases until you restart the instance. In multitenant, restarting the instance may mean an outage for a large number of applications, so you want to avoid this. So don't forget to size DB_FILES properly when you expect your container to house several pluggable databases.

UNDO

In 12.1, the first release with multitenant architecture, the UNDO tablespace, is common and implemented at the CDB level. However, in 12.2, we now also have the choice to run the CDB in local UNDO mode. If `LOCAL UNDO` is `ON`, each pluggable database has its own UNDO tablespace and all sessions writing data into PDB data blocks will place UNDO record information in this local tablespace. Only the changes performed on CDB$ROOT will put the UNDO record in the root UNDO tablespace.

In short, it is better to run in local UNDO mode if possible. UNDO contains application data, and we don't achieve pluggable database isolation if we then store this in common UNDO datafiles. One reason for this preference is that local UNDO is required for efficient flashback pluggable database and point-in-time-recovery. We will explain this in Chapter 8.

Temporary Tablespaces

Temporary tablespaces can be created at both the CDB and PDB levels. If a user is not assigned a temporary tablespace within the PDB in which the session is running, and the PDB does not have a default temporary tablespace, the session will use the CDB temporary tablespace—but this is not recommended. A quota (`MAX_SHARED_TEMP_SIZE`) can be set on pluggable database usage of the root temporary tablespace.

The CDB$ROOT temporary tablespace is normally used by sessions connected to the root container, or from a pluggable database, when a work area needs to be allocated for a recursive query on object linked views.

Once a temporary tablespace has been defined as the default, you can change it to another that you have created in that PDB, but you cannot reinstate the CDB temporary tablespace as a default after this point.

Redo Logs

The redo logs protect the instance, and because of this, they are also common. Their main objective is to record all changes made in the buffer cache and to ensure that those changes are persisted for committed transactions.

The redo stream in multitenant is similar to the stream in previous versions, except for additional information in each redo record to identify the container. Redo format is crucial for recovery operations, and, as such, Oracle changes this code very infrequently.

Having the redo thread cover all pluggable databases brings more multitenant benefits to the DBA managing the CDB. In non-CDB contexts, when you provision a new database, it takes a lot of effort and time to size the recovery area, establish the backups, and build and configure the Data Guard physical standby, when used. With multitenant, you do it only once, for your CDB, because this is where functionality related to availability runs from:

your backups, your Data Guard, and your RAC configuration. Simply create a new pluggable database and it will benefit from the same already-configured availability: it is automatically backed-up with the CDB, automatically created on the physical standby (note that you need Active Data Guard for this), and automatically accessible from all RAC instances. Again, this is because the main structure used for availability, the redo stream, operates at the CDB level.

However, having only one redo stream may be a concern in terms of performance. If you have ever encountered log writer performance issues, such as long waits on “log file sync” events, then you can imagine what happens when the Log Writer (LGWR) has to write the redo from *all* pluggable databases. And the upshot is that if the LGWR cannot keep up with the redo rate, users will be forced to wait when committing.

So, for the purpose of LGWR scalability, Oracle introduced multiple LGWRs in 12*c*. LGWR is the coordinator process, and multiple slaves (LG00, LG01, and so on) are associated with this, so that the instance redo stream is written in parallel. Of course, RAC is still another means of implementing parallel threads of redo. What you must keep in mind is that tuning the LGWR and the amount of redo written is of crucial importance in multitenant. When you consolidate, pay special attention to the performance of the disks on which you put the redo logs.

Datafiles

The datafiles, which store the tablespace data blocks, belong to each container, but they are also managed by the CDB. They have a unique identifier for the CDB, which is the `FILE_ID`:

```
SQL> select file_id,con_id,tablespace_name,relative_fno,file_name
 from cdb_data_files order by 1;

FILE_ID CON_ID TABLESPACE_NAME RELATIVE_FNO FILE_NAME
------- ------ --------------- ------------ --------------------------------
      1      1 SYSTEM                     1 /u02/oradata/CDB/system01.dbf
      3      1 SYSAUX                     3 /u02/oradata/CDB/sysaux01.dbf
      5      1 UNDOTBS1                   5 /u02/oradata/CDB/undotbs01.dbf
      7      1 USERS                      7 /u02/oradata/CDB/users01.dbf
      8      3 SYSTEM                     1 /u02/oradata/CDB/PDB1/system01.dbf
      9      3 SYSAUX                     4 /u02/oradata/CDB/PDB1/sysaux01.dbf
     10      3 UNDOTBS1                   6 /u02/oradata/CDB/PDB1/undotbs01.dbf
     11      3 USERS                     11 /u02/oradata/CDB/PDB1/users01.dbf
     12      4 SYSTEM                     1 /u02/oradata/CDB/PDB2/system01.dbf
     13      4 SYSAUX                     4 /u02/oradata/CDB/PDB2/sysaux01.dbf
     14      4 UNDOTBS1                   6 /u02/oradata/CDB/PDB2/undotbs01.dbf
     15      4 USERS                     11 /u02/oradata/CDB/PDB2/users01.dbf
```

The concept of a *relative file number* was introduced along with transportable tablespaces, so this aspect of the architecture of Oracle was ready long before 12*c*. In multitenant, within a pluggable database, the datafiles are identified by the tablespace number and the file number relative to the tablespace (RELATIVE_FNO). Furthermore, these do not have to be changed when you move, clone, or plug in the pluggable database, and it is only the absolute file number (FILE_ID) that will be renumbered to ensure that it is unique within the CDB—but that is a very quick change in the control file and datafile headers.

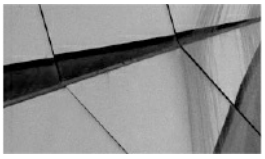
Data and Metadata at CDB Level

We have explained that the dictionary related to system objects, stored in the CDB$ROOT SYSTEM and SYSAUX tablespaces, is common and can be accessed by the PDBs. Beyond the basic database objects (created by catalog.sql and catproc.sql) more common information can be stored in the root.

APEX

By default, if you install APEX (in 12.2, you can choose the components), it is placed at the CDB level. The idea is that APEX, like the system dictionary,

houses metadata that does not need to be installed in PDB$SEED or in all pluggable databases created. However, there is a big drawback in this approach: you have only one APEX version in your CDB and you will have problems if you want to plug a non-CDB running APEX 5.0, for example, into Oracle Cloud Services, where the CDB is installed with APEX 4.2.



NOTE

This issue has been detailed by Mike Dietrich on his blog, blogs.oracle.com/UPGRADE/entry/apex_in_pdb_does_not. The APEX 5.0 documentation states, "Oracle recommends removing Oracle Application Express from the root container database for the majority of use cases, except for hosting companies or installations where all pluggable databases (PDBs) utilize Oracle Application Express and they all need to run the exact same release and patch set of Oracle Application Express."

Automatic Workload Repository

Automatic Workload Repository (AWR) collects a large amount of instance information (statistics, wait events, and so on) from the instance views, and in multitenant, this is done at the CDB level. Only one job collects all statistics for all containers, and they are stored in the CDB$ROOT. This is actually the main use case for object-link views: AWR views (those that start with DBA_HIST) can be queried from each pluggable database but actually read data that is stored in root.

There are two important consequences relating to this. The first is that if you move a pluggable database, the AWR history does not follow along with the container; instead, it remains in the original CDB. You can read the views from the original database, or export it elsewhere. But the CON_ID that is stored in AWR should be the container ID at the time the snapshot was taken, so you need to check the `CON_DBID` to identify a specific pluggable database. There are actually three different identifiers in each of the `DBA_HIST` views:

- DBID is the DBID of the CDB, same as in a non-CDB environment.

This uniquely identifies the snapshot, along with the SNAP_ID and INSTANCE_NUMBER.

- CON_ID is the container ID that comes from the `V$` views queried to take the snapshot. Some rows are not related to any container, and so have CON_ID=0, while others record statistics for a container object and so take the related CON_ID at the time of the snapshot.
- CON_DBID uniquely identifies a pluggable database, in the same way that `DBID` uniquely identifies a database.

The second consequence of having AWR collected at the CDB level is that, when you run an AWR report at PDB level, it will filter only the statistics relevant to your container, and this is exactly the same as querying `V$` views from a PDB. But you must keep in mind that you still see some statistics that are at CDB level (those that have CON_ID=0) in the same report. This means that, for example, you will see the amount of logical reads done by the instance in the instance statistics section, but the details (in SQL sections or Segment section) reveal only what is relevant to your pluggable database. Let's take a look at an example.

Before reading an AWR report details, we always check that most of the SQL statements are captured, because there is no point in continuing if we cannot go down to the statement-level detail. Here is the `SQL ordered by Gets` header from an AWR report:

```
SQL ordered by Gets                              DB/Inst: CDB/CDB  Snaps: 139-143
…
-> Total Buffer Gets:      24,958,807
-> Captured SQL account for   88.9% of Total
```

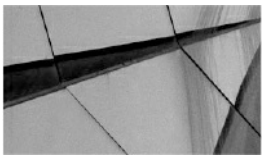
This shows that 89 percent of the SQL statements were captured, and we know that we will have the detail we need when we investigate the high logical reads issue. When the percentage is low, that usually means that we are analyzing a report that covers a time window that is too large, and most of the SQL statements have been aged out of the shared pool before the end snapshot. But you can see another reason for low percentages when you run the AWR report from a pluggable database:

```
SQL ordered by Gets                              DB/Inst: CDB/CDB  Snaps: 139-143
…
-> Total Buffer Gets:      24,958,807
-> Captured SQL account for   21.6% of Total
```

You don’t see anything different here, except that only 21 percent of the statements have been captured. You have to check the AWR report header to see that it covers only one PDB. Actually we had two PDBs active at that time, and here is the other one:

```
SQL ordered by Gets                              DB/Inst: CDB/CDB  Snaps: 139-143
…
-> Total Buffer Gets:      24,958,807
-> Captured SQL account for   60.3% of Total
```

From one PDB, you have no way to see if all statements for that PDB have been captured. There are no statistics such as total logical reads for that pluggable database.



NOTE

This is the behavior in 12.1, where per-PDB instance statistics are available in `V$CON_SYSSTAT` but not collected by AWR. In 12.2, they are collected in `DBA_HIST_CON_SYSSTAT`, and you have similar views from time model (DBA_HIST_CON_SYS_TIME_MODEL) and system events (DBA_HIST_CON_SYSTEM_EVENT). In 12.2 the AWR report uses them and the inconsistency above does not appear.

Statspack If you don’t have the Diagnostic Pack, you can’t use AWR and you will probably install Statspack. According to the documentation (spdoc.txt), Statspack can be installed only at the PDB level. We think that it makes sense to install it at the CDB level as well, because you may want to analyze CDB$ROOT activity. Each PDB from which you want to collect

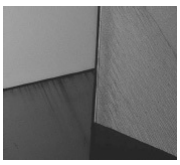
snapshots will store its own statistics. Because statistics are now collected at PDB level, the behavior is different from the behavior with AWR. We've taken Statspack snapshots at the same time as the AWR snapshots in the preceding example. The session logical reads from spreport.sql show 24,956,570 logical reads when the report is run on CDB$ROOT, and 5,709,168 (22 percent) for one PDB and 17,138,586 (68 percent) for the other. Statspack collects statistics related with the CDB when run from root and with the container when run from a PDB.

Summary

In this long introduction, we have explained why multitenant was introduced by Oracle 12*c* in 2013. We have seen the different consolidation alternatives, and perhaps you think that you don't need to run in multitenant. However, this new architecture will eventually become the only one supported, the non-CDB being already deprecated. So even if you don't want multiple pluggable databases per instance now, you will have to run what we will call "single-tenant" in Chapter 3, and you will have to administer container databases.

In addition to consolidation, the new architecture separates the application data and metadata from the system dictionary, bringing more agility for data movement and location transparency. This will be covered in Chapter 9.

The next chapter will start from the beginning by creating a consolidated database.



CHAPTER 2

Creating the Database

In Chapter 1 we introduced the new Oracle 12*c* Database Multitenant option. It is interesting that we still reference this as *new*—the fact is, even though it was introduced in 2013, many DBAs and organizations have not started using this technology yet. In learning new technologies, many DBA's most effective approaches include starting simple. The acronym KISS—Keep It Simple Stupid—is still in many cases the best approach. In this chapter, we will follow such an approach to help you lay a good foundation for the world of Oracle Database 12*c* Multitenant, which, as you will see, scales in complexity rather quickly.

As with earlier versions, there are many ways to create an Oracle Database, and some might expect that experienced, hardcore DBA would naturally favor the command line. But you might be surprised to learn that many of them actually make use of graphical interfaces, whether it is the well-known Database Configuration Assistant (DBCA) or Oracle Database Express, and Cloud Control for the larger configurations. In this chapter, we will cover two key topics:

- Creation of the container database (CDB)
- Creation of pluggable databases (PDBs)

Creating a Container Database (CDB)

Before we dive into the CDB creation steps, we need to discuss a few crucial introductory topics that will assist you in gaining a better understanding of what happens when you create a CDB database with one or more PDBs. As detailed in Chapter 1, when you create a CDB, you end up with at least two containers: the root container (CDB$ROOT) and the seed container (PDB$SEED).

In most Oracle configurations nowadays, two main storage options are used for database files: Automatic Storage Management (ASM), which is highly recommended, or the traditional file system–based storage. In earlier versions of the Oracle Database, many system administrators were kind of scared to use ASM; because database files were not easily viewable, they felt they had no control over them. However, times have changed, and with ASM's improvements and its excellent feature set, it has now been widely

adopted. In this chapter, therefore, both options will be discussed in conjunction with creating a CDB and PDBs. For more detail on the use and configuration of ASM, review the Oracle documentation.

What About OMF?

Oracle Managed Files (OMF) is an option that charges Oracle with the responsibility for database file naming, and in a multitenant environment it is highly recommended. For those unfamiliar with OMF, this may sound dangerous, but there is no need for alarm, because it is an easy-to-use and proven option that can make a DBA's life a lot easier.

Imagine creating a tablespace. Traditionally, you would have to specify the full path and name of the datafile you are creating. But with OMF, you can simply state that you want a file of a specific size for the tablespace, and the naming of the file will be taken care of for you. An example of the file structure and naming in an OMF-enabled environment is shown in Figure 2-1.

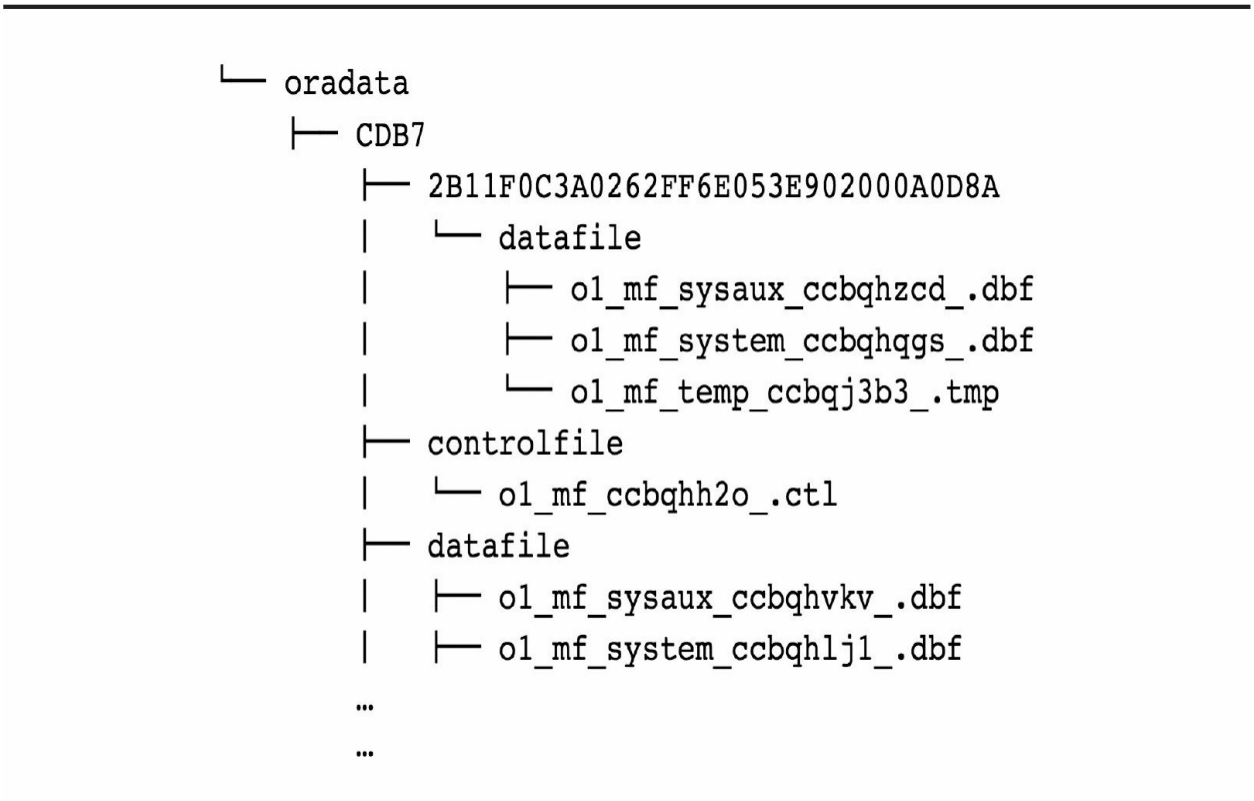


FIGURE 2-1. *Example OMF directory structure*

To the uninitiated, this might take some getting used to. Note especially the subdirectory naming for the PDB. When you're working with OMF, a long hash is used as a subdirectory name for the PDB datafiles, which in this example is 2B11F0C3A0262FF6E053E902000A0D8A.

Where does this come from, and what does it mean? If you review V$CONTAINER from the CDB$ROOT, you will notice that the value used for this directory name is in fact the GUID value for the pluggable database. In short, the OMF directory structure takes the following format:

```
<DB_CREATE_FILE_DEST DiskGroup>/<db_unique_name>/<GUID>/DATAFILE/
```

Again, when using ASM, it is recommended that you use OMF. This brings us to the next step—the creation of the CDB.

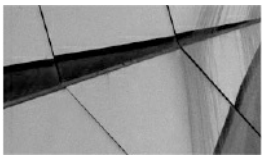
CDB Creation Options

A number of different methods are available for creating the CDB, but the most popular—and the recommended—method is to use the DBCA. For those interested, it is possible to create the CDB manually by running a number of SQL statements, and this option will be outlined in the next section.

The DBCA is an extremely powerful tool for creating databases, both via its GUI and with its lesser known command line (CLI) option. First, let's take a look at creating a CDB using the GUI.

Using the DBCA GUI

Over the years, the DBCA has become a much more reliable and stable tool, and it is a widely accepted and trusted method for creating new Oracle databases. A number of options are available that would require a book of their own to document, so we will focus instead on those key aspects related to the Multitenant option.



NOTE

It is recommended that you have a default listener already configured on the database server where you will be running the DBCA.

The first step is to start the DBCA by issuing the `dbca` command to invoke the executable located in the $ORACLE_HOME/bin directory. Once DBCA starts, you will see the GUI, as shown in Figure 2-2.

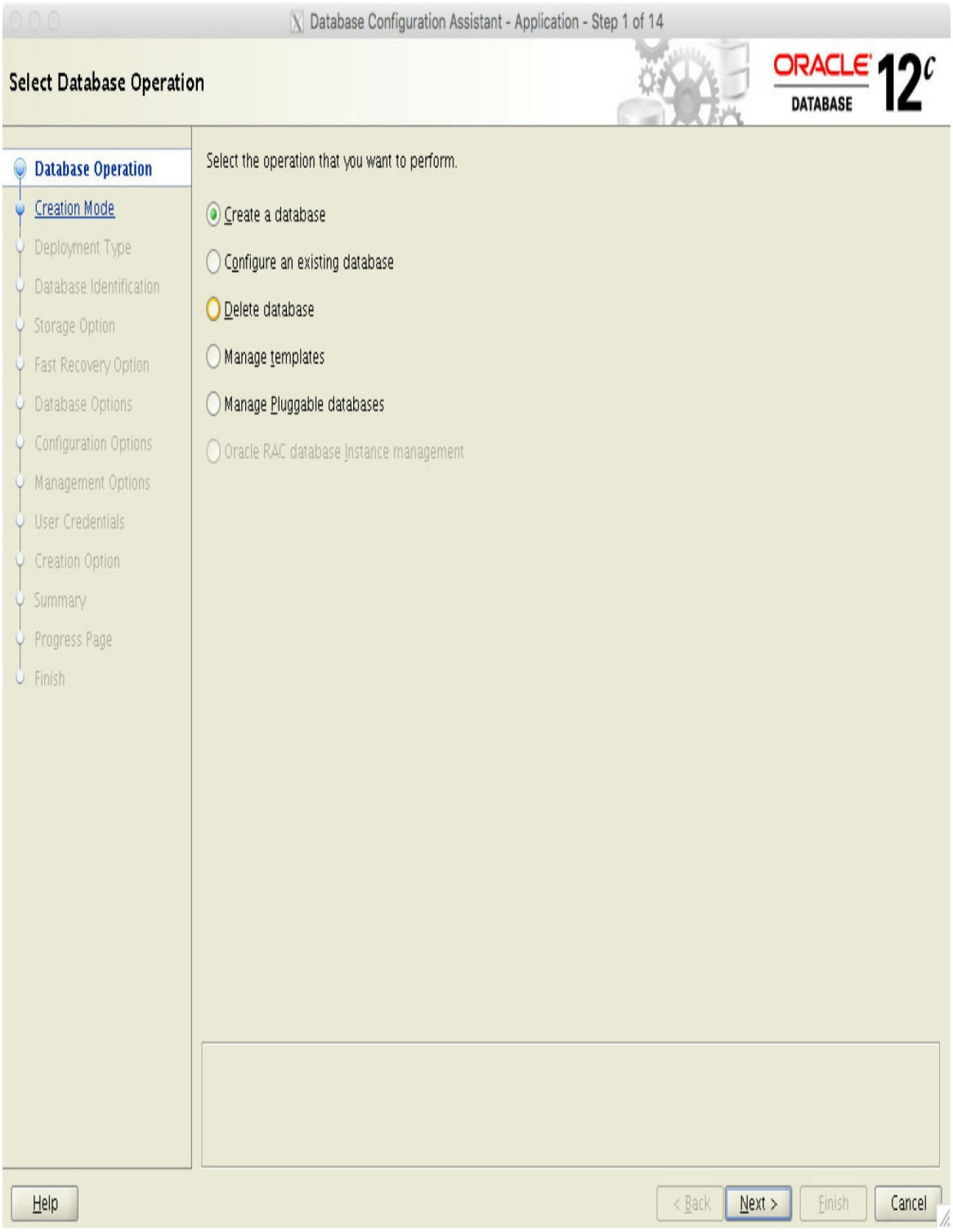
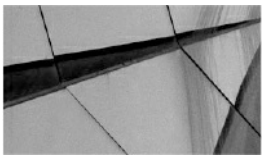


FIGURE 2-2. *DBCA step 1: Create a database*



NOTE

A number of the DBCA options (such as Delete Database) will be available only if DBCA detects at startup that the system is already running at least one other Oracle Database.

Typical Configuration To create a new database, select the Create A Database option, and click Next.

The next screen presents more detail (Figure 2-3) and two options: Typical Configuration or Advanced Configuration. In most cases choosing Typical Configuration is sufficient, but if you are more familiar with the Oracle Database and need to configure advanced options, such as memory allocations, choose Advanced Configuration.

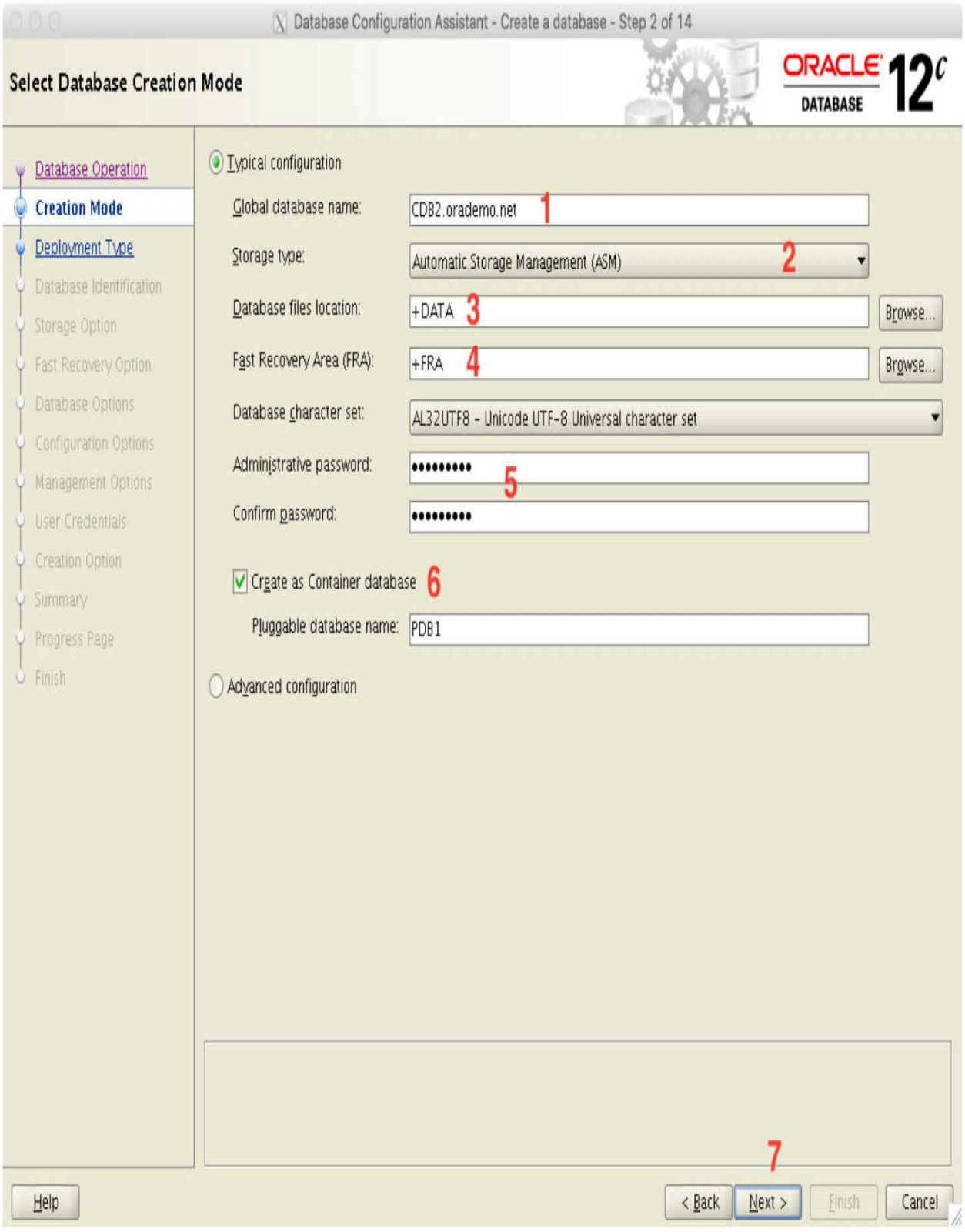
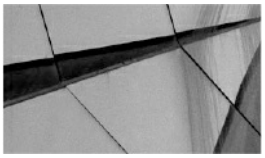

Database Configuration Assistant - Create a database - Step 2 of 14
Select Database Creation Mode
ORACLE DATABASE 12c
Database Operation
Creation Mode
Deployment Type
Database Identification
Storage Option
Fast Recovery Option
Database Options
Configuration Options
Management Options
User Credentials
Creation Option
Summary
Progress Page
Finish
Typical configuration
Global database name: CDB2.orademo.net 1
Storage type: Automatic Storage Management (ASM) 2
Database files location: +DATA 3 Browse...
Fast Recovery Area (FRA): +FRA 4 Browse...
Database character set: AL32UTF8 - Unicode UTF-8 Universal character set
Administrative password: ••••••••• 5
Confirm password: •••••••••
Create as Container database 6
Pluggable database name: PDB1
Advanced configuration
7
Help
< Back
Next >
Finish
Cancel

FIGURE 2-3. *DBCA—Typical Configuration steps*

The first example discussed will show the Typical Configuration steps. Seven steps are highlighted in Figure 2-2:

1. Provide the default Global Database Name—in this example, CDB2.orademo.net.
2. For Storage Type, specify whether you want to use filesystem-based storage or ASM. Here, we chose ASM.
3. The default Database Files Location is ASM disk group +DATA (note that these disk groups must exist prior to running the DBCA).
4. The Fast Recovery Area (FRA) is updated to point to the +FRA disk group.
5. The Administrative Passwords in this configuration will be used for all administrative users, such as SYS, SYSTEM, and PDBADMIN (the local administrator user for the PDB).
6. For multitenancy this is an important step: select the Create As Container Database option, and provide a name for the single default PDB that will be created for you.
7. Click Next to continue.

From here, the summary screen is displayed, providing an opportunity to review the options. Click Finish to start the database creation process. You will be presented with a progress page that updates on the stages of the database creation process steps until complete.



NOTE

You probably noticed references to the PDBADMIN user. This is the local admin user created during PDB creation. This user will be assigned the PDB_DBA role by default. The PDBADMIN user and

PDB_DBA role by default have no assigned privileges.

*Later in the chapter, when we create PDBs using SQL*Plus, the ROLES clause is provided as part of the PDB creation statement. This clause is used to specify the roles you want to assign to the PDB_DBA role locally in the PDB. The PDBADMIN user does not have to be called PDBADMIN, but in most cases this is the default name. For more detail on LOCAL and COMMON users, see Chapter 6.*

Advanced Configuration If you selected Advanced Configuration in Step 2, you'll see additional options, and the number of steps increases from 5 to 14. These are easy to follow, but we want to highlight Step 4, where you will be presented with a screen similar to that shown in Figure 2-4.

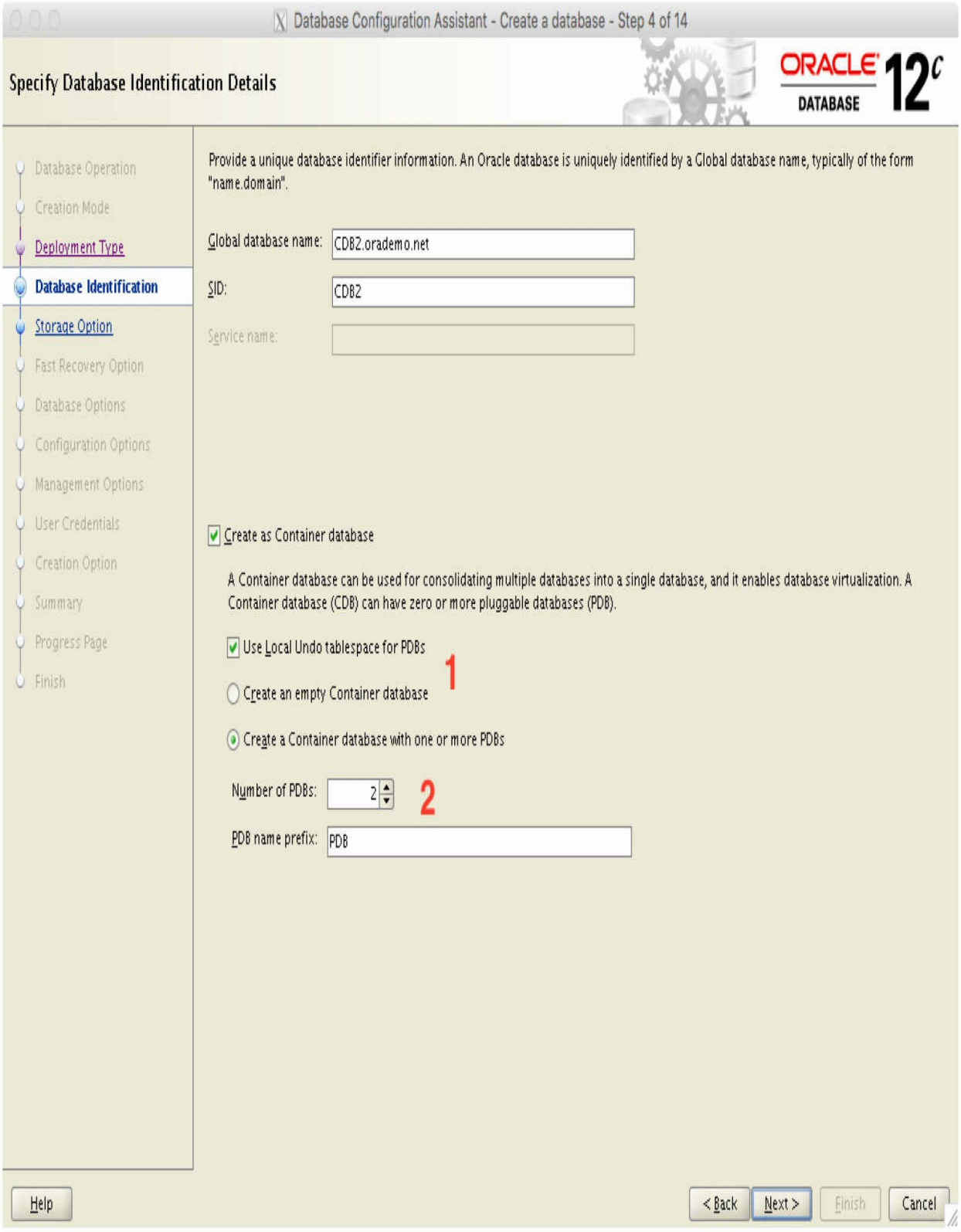
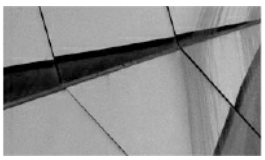

Database Configuration Assistant - Create a database - Step 4 of 14
Specify Database Identification Details
ORACLE 12c
DATABASE
Database Operation
Creation Mode
Deployment Type
Database Identification
Storage Option
Fast Recovery Option
Database Options
Configuration Options
Management Options
User Credentials
Creation Option
Summary
Progress Page
Finish
Provide a unique database identifier information. An Oracle database is uniquely identified by a Global database name, typically of the form "name.domain".
Global database name: CDB2.orademo.net
SID: CDB2
Service name:
Create as Container database
A Container database can be used for consolidating multiple databases into a single database, and it enables database virtualization. A Container database (CDB) can have zero or more pluggable databases (PDB).
Use Local Undo tablespace for PDBs
1
Create an empty Container database
Create a Container database with one or more PDBs
Number of PDBs: 2
2
PDB name prefix: PDB
Help
< Back
Next >
Finish
Cancel

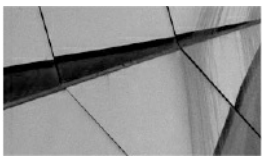
FIGURE 2-4. *DBCA—Advanced Configuration*

Here you can specify the Global Database Name as well as the option to create a CDB and any PDBs, as highlighted by 1 and 2. Note that you can specify the number of PDBs to be created, and a prefix will be used if more than one PDB will be created. You will also have the option from 12.2 to specify if you want to use Local UNDO tablespaces for the PDBs. This is a new feature introduced in 12.2 allowing PDBs to store their undo records in a local UNDO tablespace. For more detail please see Chapter 8.



NOTE
In 12.1.0.x, the maximum PDBs per CDB is 253 (including the PDB$SEED). In 12.2.0.x, the limit was increased to a maximum of 4K (4096) per CDB.

The end result is that we have a CDB called CDB1, which includes the CDB$ROOT, PDB$SEED, and two additional PDBs, PDB1 and PDB2, which will be based on the seed pluggable database.

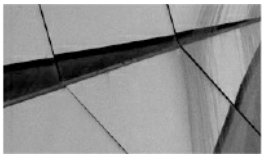


NOTE
The default templates used during the creation of a database are located in: $ORACLE_HOME/assistants/dbca/templates.

Using the DBCA CLI

Sometimes working with a GUI isn't possible, such as when you need to perform an installation and database creation via a remote connection and the network bandwidth is inadequate, or the connection is extremely slow. Don't be alarmed, because there are options available to you. One is to perform a silent, or unattended, installation using a response file, and a similar approach

can be followed when creating a database. Another is to use the DBCA CLI and the `-silent` parameter, which invokes the same executable as the GUI via the `-silent` parameter, but presents the command line alternative. This is not a black box operation, because output is pushed to screen, and if you are interested in the details, you can interrogate the log files generated by this process.



NOTE

The default character set (AL32UTF8) should be sufficient for most database implementations, but this can be adjusted, as required.

Response File Format Change There are some small differences between Oracle 12.1.0.x and 12.2.0.x with respect to the DBCA utility, including the use of response files. The latter actually features a new response file format, which will be a change welcomed by many.

On review of the sample response files, you will notice that the 12.1.0.x version of the response file makes use of grouping or sections, noted in square brackets—for example, `[CREATEDATABASE]`. These are used to identify the command being executed. This was improved in 12.2.0.x so that the response file contains only key-value pairs, and the command to be executed is passed to the dbca utility as an additional argument, such as *`dbca -createDatabase -responseFile`*.

Following is an example of a response file that can be used in 12.1.0.x. It will create a new CDB database called CDB1 with one PDB called PDB1, using the General Purpose template. The database uses ASM as the default storage type:

```
[GENERAL]
RESPONSEFILE_VERSION = "12.1.0"
OPERATION_TYPE = "createDatabase"
[CREATEDATABASE]
GDBNAME = "CDB1.orademo.net"
SID = "CDB1"
CREATEASCONTAINERDATABASE = true
NUMBEROFPDBS = 1
PDBNAME = PDB
PDBADMINPASSWORD = " Password1234"
TEMPLATENAME = "General_Purpose.dbc"
SYSPASSWORD = "Password1234"
SYSTEMPASSWORD = "Password1234"
DATAFILEDESTINATION ="+DATA"
RECOVERYAREADESTINATION="+DATA"
STORAGETYPE="ASM"
DISKGROUPNAME=DATA
AUTOMATICMEMORYMANAGEMENT = "FALSE"
TOTALMEMORY = "850"
```

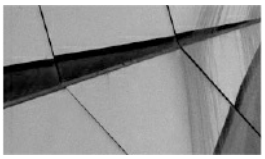
If you do not specify values for at least one of the key commands (`createDatabase`, `createTemplateFromDB`, or `createCloneTemplate`) in the 12.1.0.x release response file (such as `[CREATEDATABASE]` as shown on line 4 of the preceding example), you will receive an error when running the `dbca` command—as per the following output, which indicates that at least one of these key commands must be configured:

```
dbca -silent -responseFile /home/oracle/responsefiles/cdb1-121.rsp
No command specified to perform. Please specify one of following commands:
createDatabase, createTemplateFromDB or createCloneTemplate
```

DBCA, Response Files, and 12.2.0.x Using the `silent` option of the DBCA —that is, running it from the command line without starting the GUI—can be a big time-saver and is finding increased favor. And, as mentioned earlier, the CLI for the DBCA makes use of the same `dbca` executable. A substantial amount of detail is provided when executing the `dbca` command with the `-help` argument, and in 12.2.0.x the output is better structured and easier to

read, compared to the initial 12.1.0.x release. You can add the `-help` flag to provide a detailed listing for the specific command you are interested in. For example, running `dbca -createDatabase -help` triggers a full and detailed listing of the available options for this command.

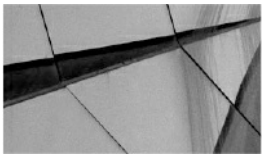
Using Response Files There are two ways to use the CLI. The first is with a response file, which is similar to using a response file for the database software installation.



NOTE

A sample response file that can be customized and used with the DBCA utility to create databases is provided as part of the database software installation at $ORACLE_HOME/assistants/dbca/ and is called dbca.rsp. Focusing on the uncommented lines is a good starting point.

Alternatively, you can create, or save, your own response file based on the steps you perform in the DBCA interface. You may have noticed at the end of the configuration process an option to generate a response file based on all your selections and input. This is a quick-and-easy way to create one of these files to meet your requirements.



NOTE

When using a response file, you do not have to assign values to all parameters. Most parameters have default values, which will suit most configurations and can be used without modifications.

Let’s look at two examples of using response files to create two CDB databases.

Example 1: Create CDB with two PDBs (ASM and OMF)

Here, we'll create a container database called CDB1 with two PDBs: PDB1 and PDB2. The Oracle-supplied template General_Purpose.dbc is used, and the database will be located in ASM with OMF enabled, using disk groups +DATA and +FRA. The key parameters specified in the response file are

- createAsContainerDatabase = true
- numberOfPDBs = 2
- pdbAdminPassword = Password12345
- pdbName = PDB

It is recommended that the passwords for the SYS, SYSTEM, and PDBADMIN accounts conform to Oracle standards. If you do not specify these in the response file, the user will be prompted for them on execution of the `dbca` command, because they are mandatory. The values specified in the response file are summarized as follows (in alphabetical order):

```
cdb1.rsp
automaticMemoryManagement=false
characterSet=AL32UTF8
createAsContainerDatabase=true
datafileDestination=+DATA
enableArchive=true
gdbName=CDB1.orademo.net
nationalCharacterSet=AL16UTF16
numberOfPDBs=2
pdbAdminPassword=Password12345
pdbName=PDB
recoveryAreaDestination=+FRA
recoveryAreaDestination=+FRA:
recoveryAreaSize=5120
redoLogFileSize=100
storageType=ASM
sysPassword=Password12345
systemPassword=Password12345
templateName=/u01/app/oracle/product/12.2.0/dbhome_1/assistants/dbca/templates/
General_Purpose.dbc
totalMemory=850
useOMF=true
```

The command to execute the DBCA using the response file in silent mode is

```
dbca -createDatabase -responseFile <full qualified path of responsefile> -silent
```

Here's an example:

```
dbca -createDatabase -responseFile /home/oracle/responsefiles/cdb1.rsp -silent
```

Example 2: Create CDB with one PDB (FS and non-OMF)

Now let's create a container database, CDB1, with one PDB called PDB. The Oracle supplied template General_Purpose.dbc is used. The database uses

normal file system–based storage, without OMF. The database files will be located in /u01/app/oracle/oradata with a FRA (recovery area destination) located at /u01/app/oracle/fast_recovery_area. The values specified in the response file are summarized as follows (in alphabetical order):

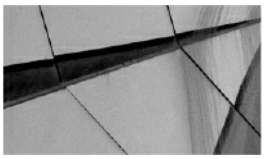
```
cdb2.rsp
automaticMemoryManagement=false
characterSet=AL32UTF8
createAsContainerDatabase=true
datafileDestination=/u01/app/oracle/oradata
enableArchive=true
gdbName=CDB2.orademo.net
nationalCharacterSet=AL16UTF16
numberOfPDBs=1
pdbAdminPassword=Password12345
pdbName=PDB
recoveryAreaDestination=/u01/app/oracle/fast_recovery_area
redoLogFileSize=100
storageType=FS
sysPassword=Password12345
systemPassword=Password12345
templateName=/u01/app/oracle/product/12.2.0/dbhome_1/assistants/dbca/templates/
General_Purpose.dbc
totalMemory=850
useOMF=false
```

If the `numberOfPDBs=1`, then the `pdbName` parameter will be taken as the actual name of the PDB. So in example 2, the PDB name would be PDB. However, if the `numberOfPDBs` is greater than `1`, the `pdbName` parameter is used as a prefix for the PDBs that will be created. For example, if `numberOfPDBs=3`, the end result will be PDB1, PDB2, and PDB3, which will be created from the default PDB$SEED. The command used to execute the DBCA, using the response file cdb2.rsp in silent mode is shown here:

```
dbca -createDatabase -responseFile /home/oracle/responsefiles/cdb2.rsp -silent
```

The end result of the command is a CDB database with one PDB, located

on file system–based storage with archive logging enabled. For more details, you can always review the log files generated under the cfgtoollogs/dbca subdirectory located in the $ORACLE_BASE location.



NOTE

To remove a running database, such as CDB1, dbca can be used: `dbca -deleteDatabase -sourceDB CDB1 -silent`. *But be careful when invoking this command on a running instance, because it will shut it down and remove all datafiles. But the most important part: it will remove all the datafiles* and *it will remove any known backups. This might not be your desired outcome when executing this command.*

Using the DBCA CLI Without Response Files Instead of using a response file, which contains all the options, you can specify the key-value pairs as arguments to the *dbca* executable. Let’s take the first example from earlier to see how this works. Using the values in the response file, we can rewrite this as follows. (To make it easier to read, the format with the \ line delimiter is used.)

```
dbca -createDatabase \
-silent \
-automaticMemoryManagement false \
-characterSet AL32UTF8 \
-createAsContainerDatabase true \
-datafileDestination +DATA \
-enableArchive true \
-gdbName CDB1.orademo.net \
-nationalCharacterSet AL16UTF16 \
-numberOfPDBs 2 \
-pdbAdminPassword Password12345 \
-pdbName PDB \
-recoveryAreaDestination +FRA \
-redoLogFileSize 100 \
-storageType ASM \
-sysPassword Password12345 \
-systemPassword Password12345 \
-templateName General_Purpose.dbc \
-totalMemory 850 \
-useOMF true
```

This method of creating a CDB database works in both 12.1.0.x and 12.2.0.x; however, a small number of options such as `-enableArchive` and `-useOMF` are not available in 12.1.0.x. This method is a quick-and-easy way to establish container databases along with a number of required pluggable databases.

Using SQL*Plus

If you are looking at creating customized configurations, particularly when certain database options are not required in the database, using SQL*Plus might be a good option. However, before doing this, you should first take note of the options provided by the DBCA—and this is where the DBCA CLI can be extremely useful. You can actually use the `-generateScripts` option to let the DBCA create scripts for you, and then review and update them before executing them manually. If we take the earlier CDB1 database creation example, you would add `-generateScripts` and `scriptDest` to the command, which will result in the database creation scripts being created in the /u01/app/oracle/admin/CDB1/scripts directory:

```
dbca -generateScripts \
-scriptDest /u01/app/oracle/admin/CDB1/scripts
-silent \
…
```

You can then review or update these settings and remove options, for example, if needed. Then simply execute the CDB1.sh master script created in the designated scripts folder to invoke the process. This is an easy way to prepare the database creations scripts, but let's review some of the manual processes at a high level.

The steps to create a CDB database using SQL*Plus may look similar to what you would have performed for non-CDB databases, but a closer look at the details reveals a number of new options—or steps—to be performed when creating a CDB. At a high level these are as follows:

- Create a password file - `orapw<SID>`.
- Create a parameter file - init<SID>.ora.
- Set the parameter enable_pluggable_database=TRUE.
- Start the new instance using the parameter file.
- Execute the CREATE DATABASE statement.
- Include the ENABLE PLUGGABLE DATABASE clause.
- Create the database catalog and options required.

The following example will take you through the high-level steps to create a CDB manually using SQL*Plus.

Example: Creating a CDB Using SQL*Plus In this example, a CDB database called CDB2 will be created using SQL*Plus. The database will be using Oracle ASM as default storage with disk group +DATA as the primary location for the database files and disk group +FRA for the recovery area.

Step 1: Prerequisite steps

If you are using role separation when installing your Oracle Database

software, which means the Grid Infrastructure (GI) is installed as a different user, such as grid, and the database software is installed as the oracle user, for example, and if you have not created any databases yet, ensure that you set the correct permission on the Oracle executable inside the Oracle Database software home to allow access to ASM storage. In most cases, the ASM disks would be owned by the grid user with the default group set as asmadmin.

Oracle has made this easier for you by introducing the `setasmgidwrap` in 11*g*, and it is still available and used in Oracle Database 12*c*. This utility is located in the GI home and should be executed while logged in as the grid user. An example of the command to be executed is shown next:

```
/u01/app/12.2.0/bin/setasmgidwrap o=/u01/app/oracle/product/12.2.0/
dbhome_1/bin/oracle
```

As part of the prerequisite steps, make sure you create the required directories, as follows:

```
cd /u01/app/oracle/admin/CDB2
mkdir adump dpdump pfile scripts
```

If you are using ASM storage create the required base directory for the database in the ASM disk groups. This can be done in a number of ways, one of which is to run these two SQL statements while connected to the ASM instance:

```
SQL> alter diskgroup DATA add directory '+DATA/CDB2';
SQL> alter diskgroup FRA add directory '+FRA/CDB2';
```

Step 2: Create a basic parameter file

Remember that some values can easily be adjusted following the creation of the CDB, and therefore in most cases it is recommended that you start with a

basic parameter file. Then, once you have the database up and running, you can adjust the values as required.

Because we are creating a CDB database, it is important to ensure that the `enable_pluggable_database=TRUE` is specified in the parameter file. As mentioned earlier, this database will be making use of ASM. Note that the control file parameter values will be added once the database is created. The parameter file initCDB2.ora is created with the options that follow and saved in the $ORACLE_BASE/admin/CDB2/pfile/ directory.

```
log_archive_format=%t_%s_%r.arc
db_block_size=8192
open_cursors=300
db_domain=orademo.net
```

```
db_name="CDB2"
db_create_file_dest="+DATA"
db_create_online_log_dest_1=+DATA
db_create_online_log_dest_2=+FRA
db_recovery_file_dest="+FRA"
db_recovery_file_dest_size=10g
instance_name=CDB2
compatible=12.2.0.0.0
diagnostic_dest=/u01/app/oracle
nls_language="AMERICAN"
nls_territory="AMERICA"
processes=320
sga_target=1512M
audit_file_dest="/u01/app/oracle/admin/CDB2/adump"
audit_trail=db
remote_login_passwordfile=EXCLUSIVE
dispatchers="(PROTOCOL=TCP) (SERVICE=CDB2XDB)"
pga_aggregate_target=512m
undo_tablespace=UNDOTBS1
enable_pluggable_database=true
```

Step 3: Update /etc/oratab file

This step is optional, but if you are using UNIX-based systems, updating the oratab file is highly recommended. This will make it easy for you to switch between database environments, especially if you have multiple Oracle Database software installations on the same system.

On Oracle Linux, the oratab file is located at /etc/oratab. The following entry is added for the CDB2 database:

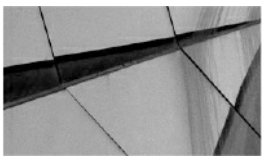
```
CDB2:/u01/app/oracle/product/12.2.0/dbhome_1:N
```

Once you have added the entry, you can make use of the oraenv utility to set the correct environment. Run the command `. oraenv`, and when asked to provide the ORACLE_SID, specify CDB2. You will notice that the required environment variables such as ORACLE_HOME will now be set. Here's an example:

```
oracle@linux3[/home/oracle]: . oraenv
ORACLE_SID = [CDB1] ? CDB2
The Oracle base remains unchanged with value /u01/app/oracle
```

Step 4: Set the correct environment for the catcon.pl script execution

This brings us to an interesting point: the catcon.pl script. This script will be discussed in more detail at the end of this chapter.



NOTE

As of 12.2.0.x, the default Perl version that is shipped with the Oracle Database software is 5.22.

The following commands are executed to set the correct PATH and PERL5LIB environment variable values:

```
export PERL5LIB=$ORACLE_HOME/rdbms/admin:$PERL5LIB
export PATH=$ORACLE_HOME/bin:$ORACLE_HOME/perl/bin:$PATH
```

Step 5: Create a password file

There are a few new options introduced in Oracle Database 12*c*, including a new password file format:

```
orapwd file=$ORACLE_HOME/dbs/orapwCDB2 force=y format=12 password=Password1234
```

Step 6: Start the Database instance in nomount state with pfile

Start the database using the parameter file created in Step 2 in a nomount state:

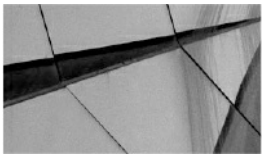
```
SQL> connect / as sysdba
SQL> startup nomount pfile='/u01/app/oracle/admin/CDB2/pfile/initCDB2.ora';
```

We now get to the CREATE DATABASE statement, and because we are using OMF, there is no need to specify datafile names and locations. The key words to notice are ENABLE PLUGGABLE DATABASE.

```
CREATE DATABASE "CDB2"
MAXINSTANCES 8
MAXLOGHISTORY 1
MAXLOGFILES 16
MAXLOGMEMBERS 3
MAXDATAFILES 1024
DATAFILE SIZE 700M AUTOEXTEND ON NEXT 20M MAXSIZE UNLIMITED EXTENT MANAGEMENT LOCAL
SYSAUX DATAFILE SIZE 700M AUTOEXTEND ON NEXT 20M MAXSIZE UNLIMITED
SMALLFILE DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE SIZE 1G AUTOEXTEND ON NEXT 100M
MAXSIZE 5G
SMALLFILE UNDO TABLESPACE "UNDOTBS1" DATAFILE SIZE 501M AUTOEXTEND ON NEXT  100M
MAXSIZE 5G
CHARACTER SET AL32UTF8
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1  SIZE 100M,
        GROUP 2  SIZE 100M,
        GROUP 3  SIZE 100M
USER SYS IDENTIFIED BY Password1234 USER SYSTEM IDENTIFIED BY Password1234
ENABLE PLUGGABLE DATABASE;
```

If you're using non-OMF, the `SEED FILE_NAME_CONVERT` parameter can be used to specify a location for the PDB$SEED datafiles—here's an example:

```
…
ENABLE PLUGGABLE DATABASE
SEED file_name_convert = ('/u01/app/oracle/oradata/CDB1,'/u01/app/
oracle/oradata/CDB1/SEED');
```



NOTE

When you're creating a CDB database, always make sure you specify high enough value for `MAXDATAFILES`.

When executing the `CREATE DATABASE` statement, you can ignore the following error: "ORA-06553: PLS-213: package STANDARD not accessible." This occurs because catalog objects have not been created yet.

Notice the control files created after the `CREATE DATABASE` statement.

You will need to update the parameter file to include their location and names. Use `show parameter control_files` to obtain the created control file names. In this example, the control files were

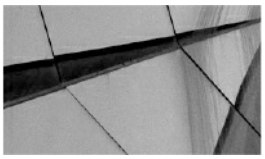
```
'+DATA/CDB2/CONTROLFILE/current.266.903389817'
```

and

```
'+FRA/CDB2/CONTROLFILE/current.297.903389817'
```

The following can be used to add the control file names to the parameter file:

```
echo "control_files='+DATA/CDB2/CONTROLFILE/current.266.903389817', '+FRA/CDB2/
CONTROLFILE/current.297.903389817'" >> /u01/app/oracle/admin/CDB2/pfile/initCDB2.ora
```



NOTE

The SEED pluggable database (PDB$SEED) is created as part of the `CREATE DATABASE` statement. The sql.bsq is executed, which will run dcore.bsq twice—once for the root container and once for the SEED container. This was noted when we reviewed the alert log during database creation and reviewed the dcore.bsq script.

Step 7: Add default USERS tablespace

Here’s how to add a default USERS tablespace to the CDB$ROOT:

```
CREATE SMALLFILE TABLESPACE "USERS" LOGGING
DATAFILE  SIZE 5M AUTOEXTEND ON NEXT  10M MAXSIZE UNLIMITED  EXTENT
MANAGEMENT LOCAL  SEGMENT SPACE MANAGEMENT  AUTO;
ALTER DATABASE DEFAULT TABLESPACE "USERS";
```

Step 8: Open the PDB$SEED pluggable database

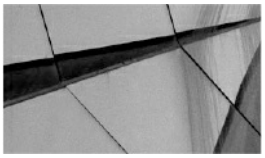
We are now getting to one of the interesting parts—the SEED database. To enable you to open or close the SEED database, you have to alter the session and set `_oracle_script=true;`. Once this is done, you will be able to close and then open the SEED database:

```
SQL> alter session set "_oracle_script"=true;
SQL> alter pluggable database pdb$seed close;
SQL> alter pluggable database pdb$seed open;
```

Once this is done, both the ROOT and SEED containers will be open read/write, allowing you to continue creating the database catalog and loading the required options.

Step 9: Create catalog and load options—catcdb.sql

The next step is to run the catcdb.sql script located in $ORACLE_HOME/rdbms/admin. This script will make use of the catcon.pl script and will create the catalog and load the default options.



NOTE

The catcdb.sql script was missing in the initial 12.1.0.1 release and was later added as part of the pat set updates—12.1.0.1.4 DB PSU and higher.

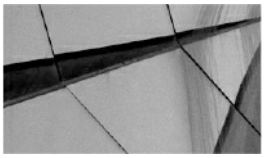
The catcdb.sql script can be run as follows (using the SYS user):

```
SQL> @$ORACLE_HOME/rdbms/admin/catcdb.sql
```

This script can take a long time to run. It will ask for the SYS and SYSTEM users' passwords and run the required scripts to create the catalog and load the default options. If you do require or want to customize the CDB options loaded, you can use the catcon.pl script to create the catalog and load the required options. The bare minimum recommended (options) for a CDB environment is to run the catalog.sql, catproc.sql, and catoctk.sql scripts. An example execution of one of these scripts is shown here:

```
perl $ORACLE_HOME/rdbms/admin/catcon.pl -n 1 -l $ORACLE_BASE/admin/CDB2 -v  -b catalog
-U SYS/Password1234 $ORACLE_HOME/rdbms/admin/catalog.sql
```

These are the minimum recommended options, but for most configurations, it is highly recommended that you use the catcdb.sql script and load all the default options such as Oracle JVM and Oracle Text.



NOTE

If you want more information on loading only certain options, review MOS note 2001512.1

Step 10: Lock/expire all unused accounts (optional)

From a security point of view, it is recommended that at this stage you lock all accounts that will not be used. This should be done in the CDB$ROOT as well as the PDB$SEED. To perform these tasks on the PDB$SEED, follow these high level steps:

```
SQL> alter session set "_oracle_script"=true;
SQL> alter pluggable database pdb$seed close;
SQL> alter pluggable database pdb$seed open;
```

Next, run the required commands to lock unused users in the CDB$ROOT, followed by locking the required users in the PDB$SEED as well. This can be done by first setting the container to PDB$SEED:

```
SQL> alter session set container=pdb$seed;
```

Once the required commands to lock unused users in the PDB$SEED were executed sucessfully, you can set the container back to CDB$ROOT:

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

You can also close the PDB$SEED at this stage if required:

```
SQL> alter pluggable database pdb$seed close;
SQL> alter session set "_oracle_script"=false;
```

This gives you some insight into how you can customize the PDB$SEED pluggable database.

Step 11: Create a spfile from the pfile created in Step 2

One of the final steps is to create a server parameter file (spfile) from the parameter file you created in step 2. This is the command:

```
SQL> create spfile from pfile='/u01/app/oracle/admin/CDB2/pfile/
initCDB2.ora';
```

The spfile can also be created inside ASM:

```
SQL> create spfile='+DATA' from pfile='/u01/app/oracle/admin/CDB2/
pfile/initCDB2.ora';
```

In this example, the end result is a spfile being created in +DATA/CDB2/PARAMETERFILE/ spfile.268.903390805.

Step 12: Recompile all invalid objects

This is a highly recommended option and should be a well-known step to most DBAs. This script we are referring to is utrlp.sql. It is recommended that you make use of the catcon.pl script to ensure that you run this against the CDB$ROOT as well as the PDB$SEED, which at this stage is the only pluggable database in the CDB2 database.

Step 13: Optional—Add the database to Oracle Restart

This last step is optional. If GI is installed, you can make use of Oracle Restart. This enables the option to start or stop the database as part of a system restart. If offers a number of other advantages, but these are probably the most well-known reasons for using Oracle Restart.

The `srvctl` command is used to perform these tasks:

```
srvctl add database -db CDB2 -oraclehome /u01/app/oracle/
product/12.2.0/dbhome_1 -spfile +DATA/CDB2/PARAMETERFILE/
spfile.268.903390805 -diskgroup "DATA,FRA" -startoption OPEN
-stopoption IMMEDIATE
```

Step 14: Create a pluggable database

At this stage, we haven’t created a CDB database, which includes the root and the SEED pluggable database. Before getting into the details in the next section, we can list the basic `CREATE PLUGGABLE DATABASE` statement here for completeness of the example:

```
CREATE PLUGGABLE DATABASE PDB1
  ADMIN USER PDBADMIN IDENTIFIED BY "Password1234" ROLES=(CONNECT);
```

This will create the PDB called PDB1 based on the PDB$SEED PDB. Once created, the PDB will be in a mounted state. To open it, execute the following:

```
ALTER PLUGGABLE DATABASE PDB1 OPEN;
```

The next section continues with the options for creating the PDB. Note that if you were following the DBCA options discussed earlier, you can create PDBs as part of the CDB creation. This is one reason why using the DBCA is highly recommended and used by most DBAs: it takes away most of the complexities and helps keep it simple.

Creating a Pluggable Database

Instead of jumping straight into the PDB creation process, let’s take a look at the surroundings and context before diving into the details.

In a minimal Oracle PDB creation, you end up with only the CDB$ROOT and PDB$SEED PDBs. This is probably the best way to start. Why? Imagine, for example, that you are planning to create 50 PDBs and want them all to look exactly the same. You could create a simple golden image first, and then, when you need more databases, simply clone the golden image to create any new ones.

You can create new pluggable databases using a variety of tools:

- SQL*Plus
- Database Configuration Assistant (DBCA)
- SQL Developer
- Oracle Enterprise Manager Database Express
- Oracle Enterprise Manager Cloud Control

In Chapter 9, we will cover plug-in and plug-out, conversions from non-CDB to PDB, cloning, and a number of other options used to create pluggable databases. We will also discuss the new proxy PDB option introduced in release 12.2. In this chapter, the focus will be on helping you get started in creating PDBs using two basic methods:

- **Create a new PDB from the CDB SEED (PDB$SEED)** This is used mainly for new configurations. A PDB is created based on the template SEED database called PDB$SEED, which resides inside the same CDB. This method is fast, easy, and seems almost instantaneous.
- **Clone a PDB within the same CDB (also known as the local clone method)** This can be extremely useful in many scenarios, such as cloning an application PDB to create a secondary PDB on which you can test upgrade scripts prior to executing them the production PDB. There are a number of requirements when using this method, including that the cloned PDB name must be unique within the same CDB.

Create a New PDB from PDB$SEED

This method, as illustrated in Figure 2-5, is a quick and easy way to create a PDB based on PDB$SEED. During this process, the new PDB is generated by creating a copy of the PDB$SEEED, which should be in read-only mode.

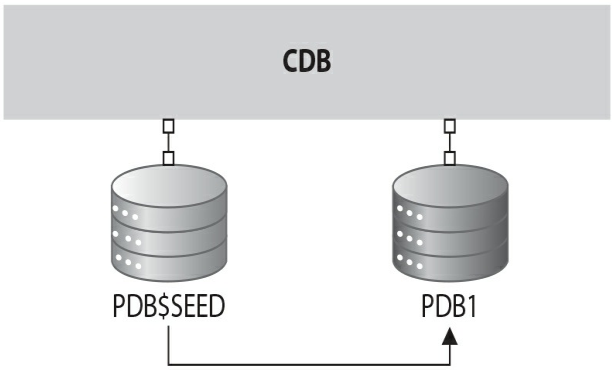


FIGURE 2-5. *Create A PDB from PDB$SEED*

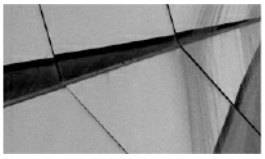
When using OMF, the process is simplified, and a basic `CREATE PLUGGABLE DATABASE` command like the following can be used to create a PDB from the CDB SEED:

```
create pluggable database PDB1
admin user PDB1ADMIN identified by Password12345
ROLES = (CONNECT);
```

The end result of this statement is the creation of PDB1, a copy of the CDB SEED—PDB$SEED. The default PDB administrator is created as PDB1ADMIN, and the default role, assigned locally to the PDB_DBA role, is CONNECT. A number of additional clauses can be used with the `CREATE PLUGGABLE DATABASE` statement and provide the ability to specify a number of customization options for the newly created PDB. Some of the key clauses include these:

- `AS APPLICATION CONTAINER`
- `AS CLONE`
- `AS SEED`
- `CREATE_FILE_DEST`
- `DEFAULT TABLESPACE`
- `FILE_NAME_CONVERT`
- `HOST`
- `PORT`
- `NOCOPY, COPY, MOVE`
- `NO DATA`
- `PARALLEL`
- `ROLES`
- `SNAPSHOT COPY`
- `SOURCE_FILE_DIRECTORY`
- `SOURCE_FILE_NAME_CONVERT`

- STANDBYS
- STORAGE
- TEMPFILE REUSE
- USER_TABLESPACES



NOTE

Throughout this book we will reference a number of these clauses, but for full details on each, refer to the "Oracle Database SQL Language Reference" for Oracle Database 12c Releases 1 and 2.

The CREATE_FILE_DEST clause can be specified as part of the create statement if you want to overwrite the default OMF location, which is specified by the CDB's DB_CREATE_FILE_DEST instance parameter. For example, if the requirement is to place the PDB on a different ASM disk group called +PDBDATA, the CREATE PLUGGABLE DATABASE statement can be adjusted as follows:

```
create pluggable database PDB1
admin user PDB1ADMIN identified by Password12345 ROLES = (CONNECT)
CREATE_FILE_DEST='+PDBDATA';
```

The end result will be that an OMF file structure will be created in the disk group +PDBDATA. So, for example, after the command is executed, PDB1 was created with CON_ID=4:

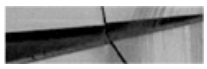
```
SQL> select name from v$datafile where con_id=4;
Name
--------------------------------------------------------------------------------
+PDBDATA/CDB1/2B674893CA800203E053E902000A5C20/DATAFILE/system.296.903476189
+PDBDATA /CDB1/2B674893CA800203E053E902000A5C20/DATAFILE/sysaux.293.903476189
```

If OMF is not used, the `FILE_NAME_CONVERT` parameter must be included when creating a new PDB from the CDB SEED. Creating a new PDB1 in a CDB, which does not make use of OMF, is illustrated next:

```
create pluggable database PDB1
admin user pdb1admin identified by Password12345 roles=(connect)
file_name_convert=('/u01/app/oracle/oradata/CDB2/pdbseed','/u01/app/oracle/oradata/
CDB2/pdb1');
```

The end result here is that the new PDB is created as a copy of the PDB$SEED, and its files are located in the directory /u01/app/oracle/oradata/CDB2/pdb1.

Before moving on to the next section, let's have a look at a slightly more complex case using a number of the clauses available in the `create` statement:



```
create pluggable database PDB2
admin user PDB2ADMIN identified by Password12345 roles=(connect)
file_name_convert=('/u01/app/oracle/oradata/CDB2/pdbseed'
                  ,'/u01/app/oracle/oradata/CDB2/pdb2')
default tablespace USERS
  datafile '/u01/app/oracle/oradata/CDB2/pdb2/users01.dbf' size 10M
  autoextend on next 100M maxsize 20G
storage (maxsize 100G max_shared_temp_size 5G)
```

Breaking down the example, observe the following:

- We are creating a new PDB called PDB2.
- The PDB admin user is called PDB2ADMIN and a password is supplied.
- The CONNECT role is assigned to the local PDB_DBA role.
- The `FILE_NAME_CONVERT` clause is specified to ensure that the new PDB subfolder pdb2 is used.

- A new default permanent tablespace is created for the new PDB, the data file name is specified, and the size of this file is 10M with the option to grow to 20G.
- The STORAGE clause is used to limit the size of the PDB to a maximum size of 100G, and only a maximum of 5G shared temporary space can be used by this PDB.

Create a New PDB Using the Local Clone Method

The second method is creating a new PDB using the clone option (see Figure 2-6) from a local PDB located in the same CDB. This method is also referred to as creating a local clone.

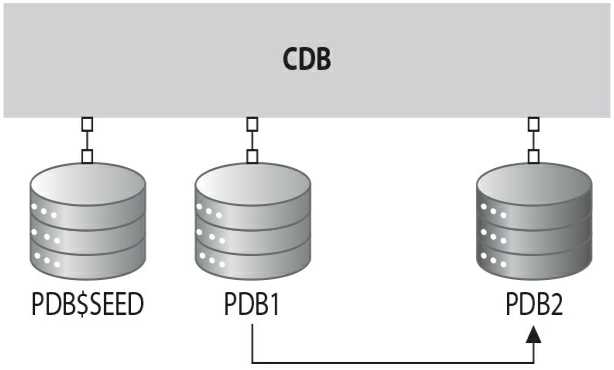


FIGURE 2-6. *Create PDB2 from PDB1 (clone PDB1)*

When using this approach, take note of the following:

- If using version 12.1.0.x, the source PDB must be in a read-only state. (As of 12.2.0.x, the source PDB can be open, as long as the CDB is in ARCHIVELOG mode with local UNDO enabled.)
- Each PDB in a CDB must be uniquely identifiable.
- Once the clone is complete, the new PDB must be opened read-write at least once to allow further operations.

When using the local cloning process, the datafiles of a source PDB (which is in a read-only state in 12.1.0.x) are read and then copied to a new uniquely identifiable PDB.

Performing a local clone in an OMF environment is easy and can be done without any additional clauses being specified. For example, to create a new PDB5 database as a clone from PDB1 in CDB1, we can execute the following two statements, cloning and then opening the new PDB in read-write mode:

```
SQL> create pluggable database PDB5 from PDB1;
SQL> alter pluggable database PDB5 open;
```

When using a non-OMF environment, the `create` statement needs to include the additional `FILE_NAME_CONVERT` clause, as follows:

```
SQL> create pluggable database PDB5 from PDB2
     file_name_convert=('/u01/app/oracle/oradata/CDB2/pdb2',
                        '/u02/oracle/oradata/CDB2/pdb5');
SQL> alter pluggable database pdb5 open;
```

Create a PDB Using SQL Developer

As mentioned in a previous section, a number of tools can be used to create PDBs. One of the utilities that is growing rapidly in popularity, and we highly recommend it if you have not tried it, is Oracle SQL Developer (Figure 2-7).

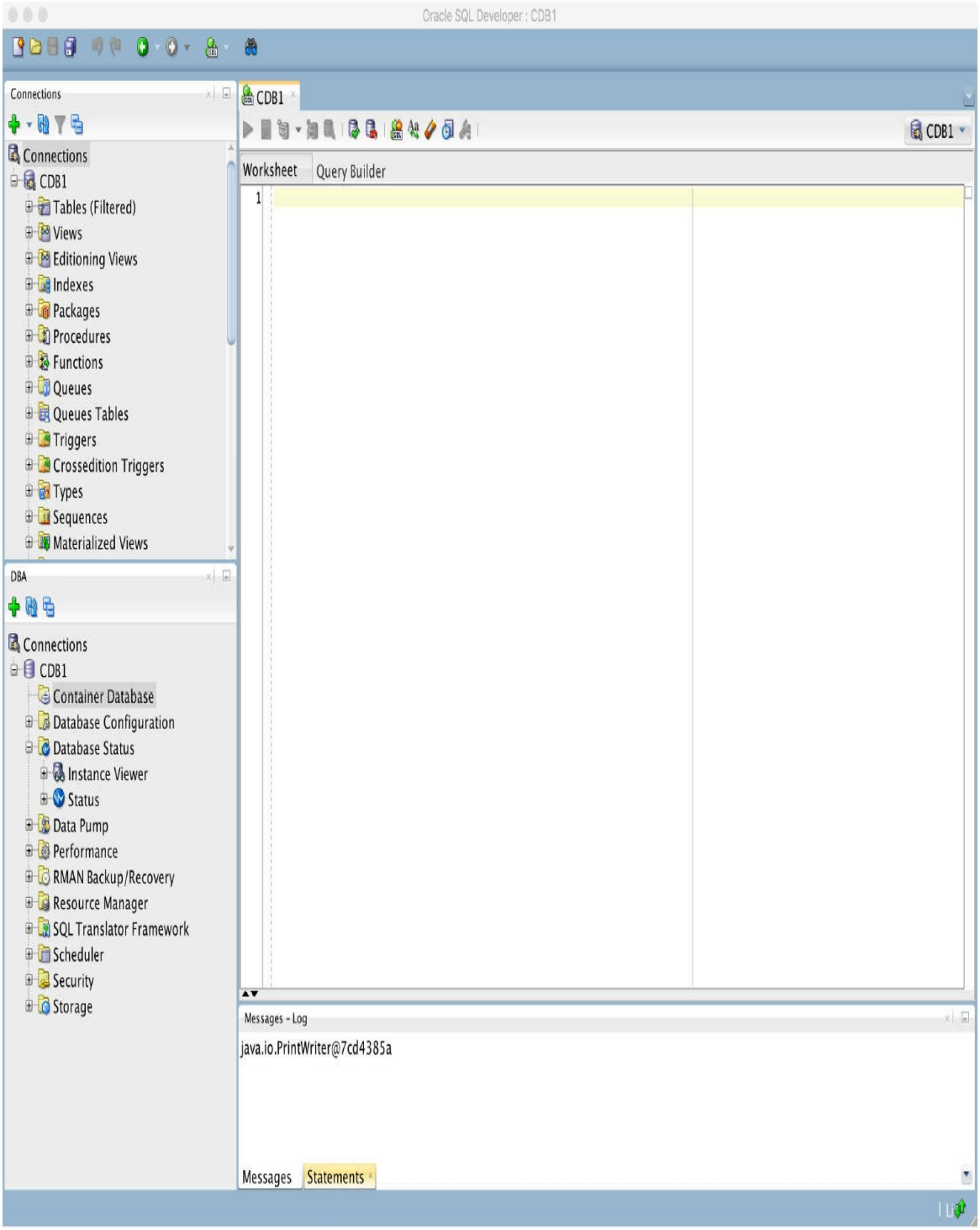


FIGURE 2-7. *Oracle SQL Developer*

This example will use SQL Developer to create a PDB called PDB1 from the CDB1 SEED.

1. Log into the CDB (CDB1) as the SYS user by establishing a connection to the database under the DBA option on the bottom right of the main SQL Developer screen. Once connected, various administration options and areas will be listed, with the first one being Container Database. In this example, as shown in Figure 2-7, no PDBs have been created, so the CDB1 database contains only CDB$ROOT and PDB$SEED.
2. Right-click Container Database, and you will be presented with a number of options, as shown in Figure 2-8. Select Create Pluggable Database.

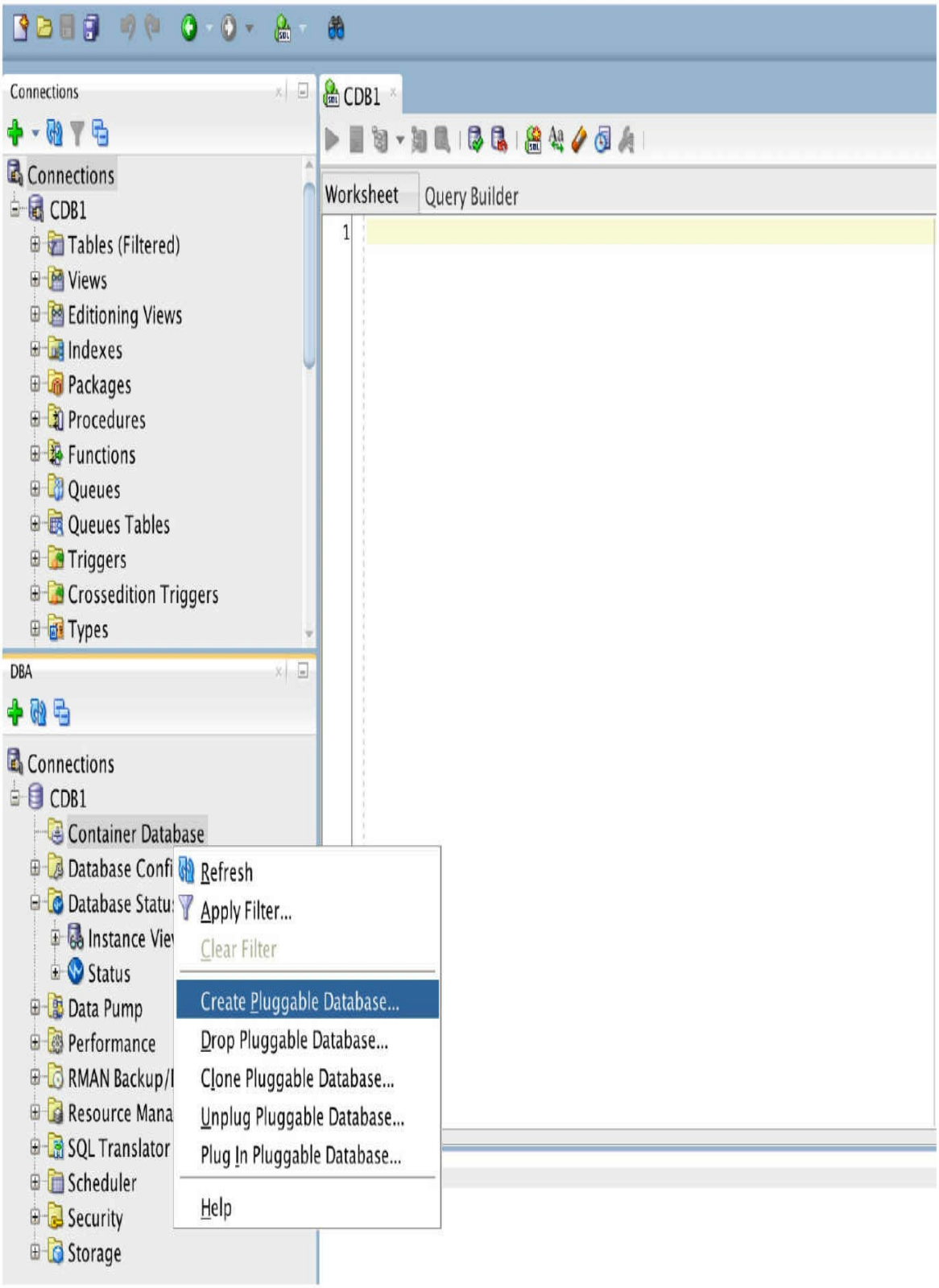

Connections
CDB1
Worksheet
Query Builder
1
Connections
CDB1
Tables (Filtered)
Views
Editioning Views
Indexes
Packages
Procedures
Functions
Queues
Queues Tables
Triggers
Crossedition Triggers
Types
DBA
Connections
CDB1
Container Database
Database Confi
Database Statu
Instance Vie
Status
Data Pump
Performance
RMAN Backup/
Resource Mana
SQL Translator
Scheduler
Security
Storage
Refresh
Apply Filter...
Clear Filter
Create Pluggable Database...
Drop Pluggable Database...
Clone Pluggable Database...
Unplug Pluggable Database...
Plug In Pluggable Database...
Help

FIGURE 2-8. *Create the PDB.*

3. In the next screen, add a new PDB name, Admin Name, and Password, and specify any storage requirements. As shown in Figure 2-9, a number of options are available with regard to the storage configuration. In the example, we will leave them set to the defaults, supplying only the new PDB name PDB1, and the admin username and password. As OMF is used in this configuration, the File Name Conversions are set at the default, None.

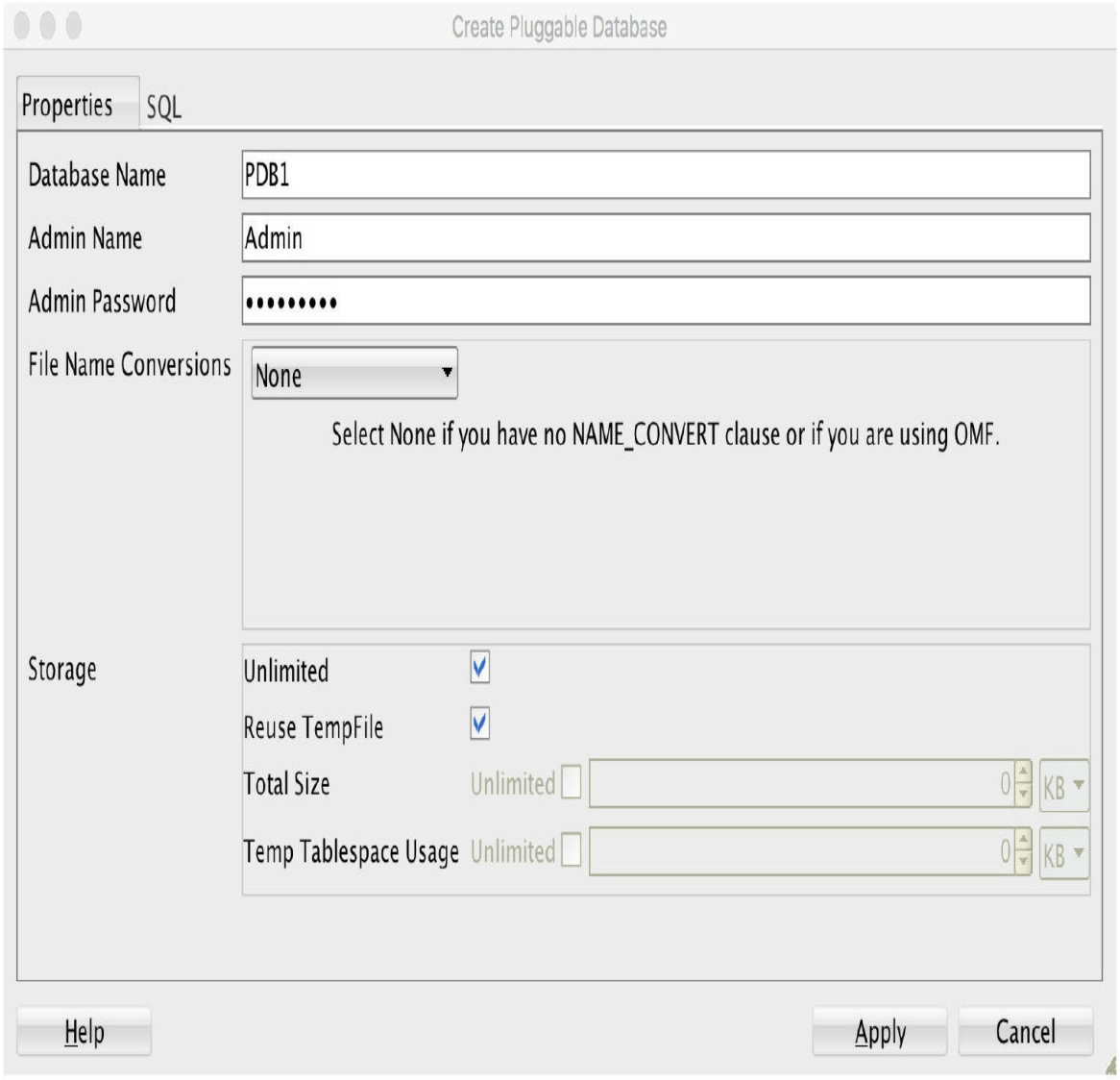


FIGURE 2-9. *Create PDB properties.*

4. Optionally, review the SQL tab shown in Figure 2-10.

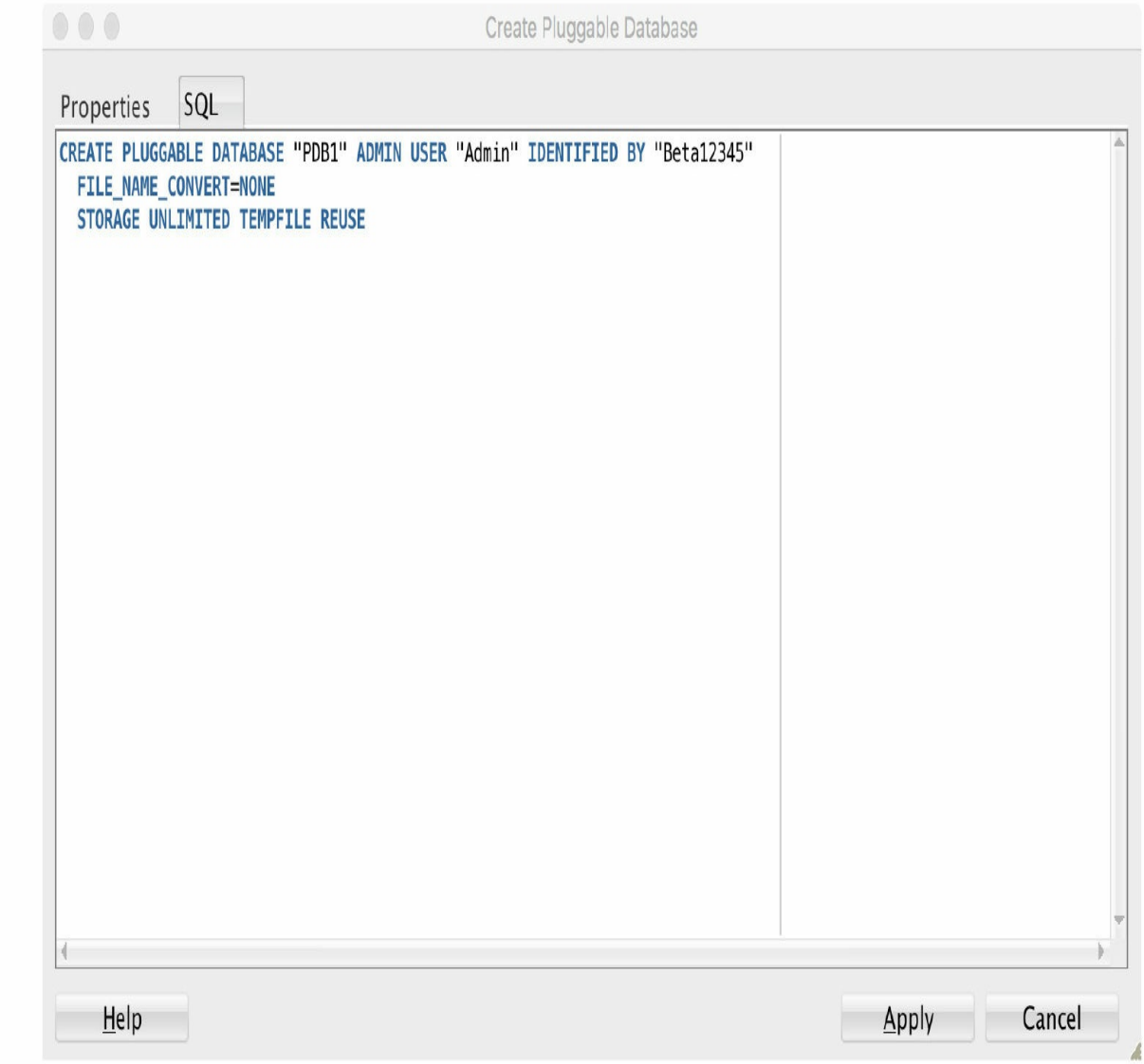


FIGURE 2-10. *Review the SQL statement.*

5. Click Apply and the new PDB will be created. If you now refresh the screen you will see that the new PDB1 is displayed under the Container Database folder. As shown in Figure 2-11, when you select PDB1, more information about the PDB is displayed on the right side of the screen. We can see that the PDB is currently MOUNTED and not yet open read-write.

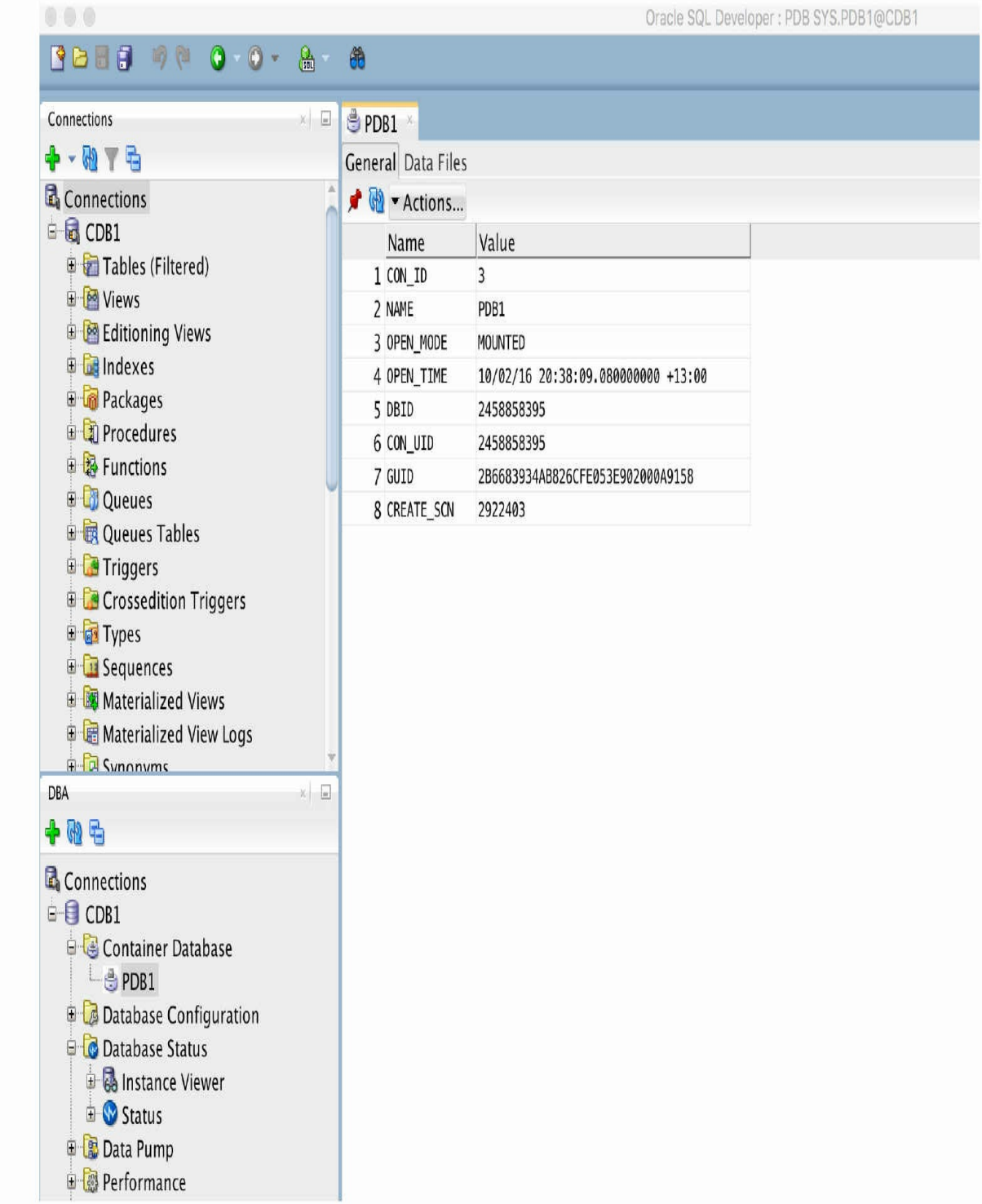


FIGURE 2-11. *Review the PDB status.*

6. Open the newly created PDB read-write by selecting and right-clicking the PDB. Choose Modify State, and a new screen will enable you to set the PDB to a specific open mode (state)—see Figure 2-12.

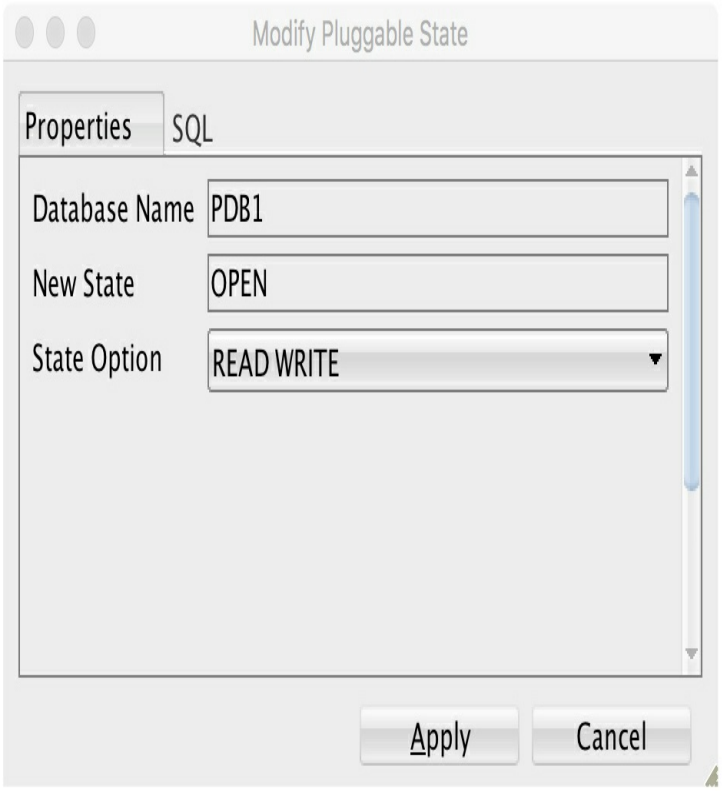


FIGURE 2-12. *Modify the PDB state*.

7. Choose the required state from the State Option drop-down list (READ WRITE, READ ONLY, RESTRICTED), and click Apply.

Create a PDB Using the DBCA

You can also use the DBCA to create new PDBs. As with the creation of CDBs, there are two options for doing this: the GUI or the CLI.

When using the DBCA GUI, you are presented with the option of creating a PDB, along with other options. Figure 2-13 shows these options on

the DBCA opening screen.

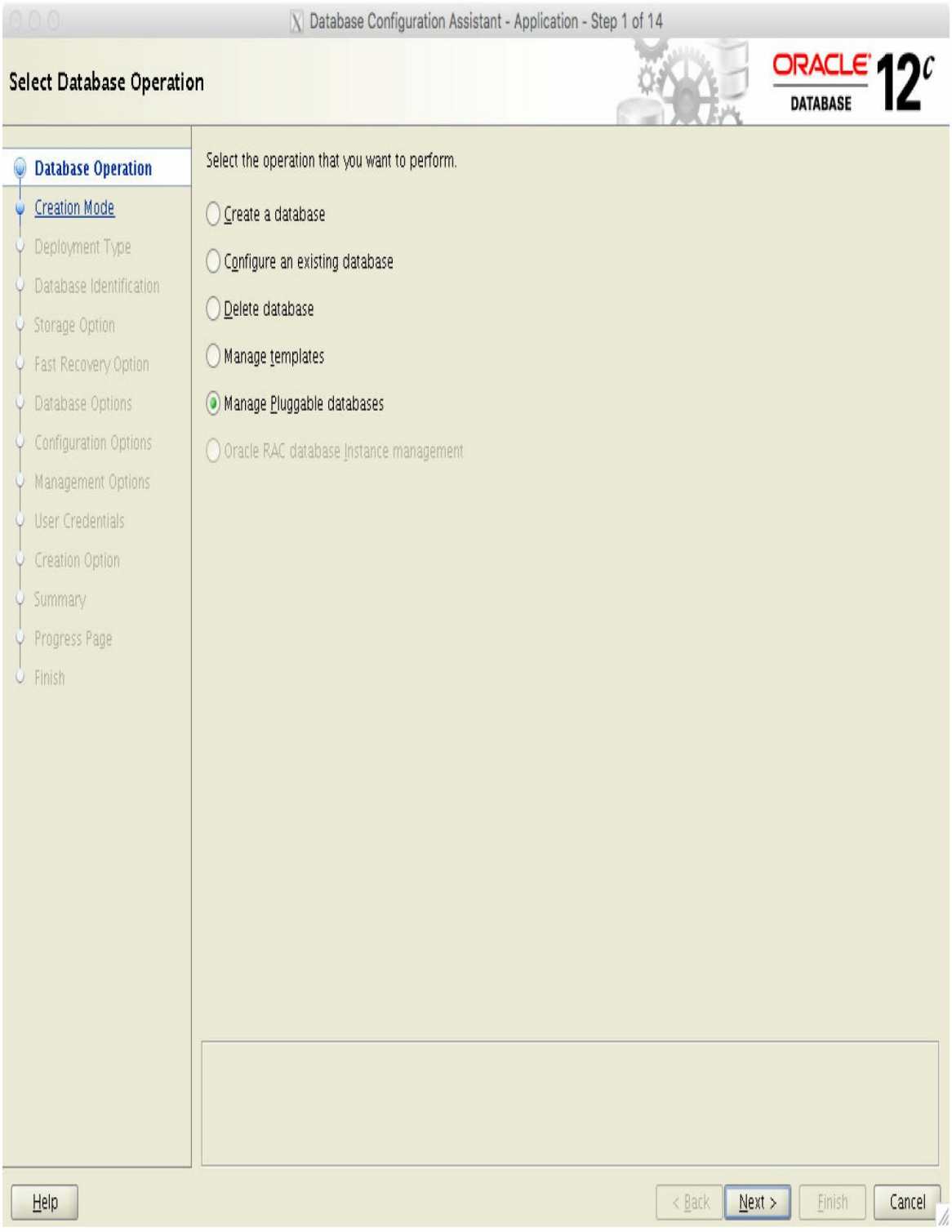


FIGURE 2-13. *DBCA—Manage Pluggable Database*

Choose Manage Pluggable Databases and you will be guided through a eight-step process to create a new PDB.

Using the DBCA is a straightforward method in which you click through a number of screens and provide basic input required to create a new PDB. But if you are looking at creating a new PDB from the CDB SEED, using SQL*Plus, SQL Developer, or the DBCA CLI (shown next) might be a much faster way to achieve this.

When using the DBCA CLI, you can use the `-createPluggableDatabase` command option. As mentioned earlier, using the `-help` keyword with this option will display all the available arguments. Following is a basic example that demonstrates the creation of a PDB database called PDB9 in a CDB called CDB1, which uses ASM and OMF:

```
dbca -createPluggableDatabase \
-silent \
-sourceDB CDB1 \
-pdbName PDB9 \
-createPDBFrom DEFAULT \
-pdbAdminUsername PDB9ADMIN \
-pdbAdminPassword Password12345
```

Create a PDB Using Cloud Control

Another option for creating ODBs is Enterprise Manager Cloud Control. Using this method is straightforward, because the tool will guide you through the process. From the Oracle Database drop-down menu path, navigate to Provisioning, and then select the Provision Pluggable Databases option. This will start the process of creating a PDB. Figures 2-14 and 2-15 show the start of the wizard-driven process that assists in creating a PDB.

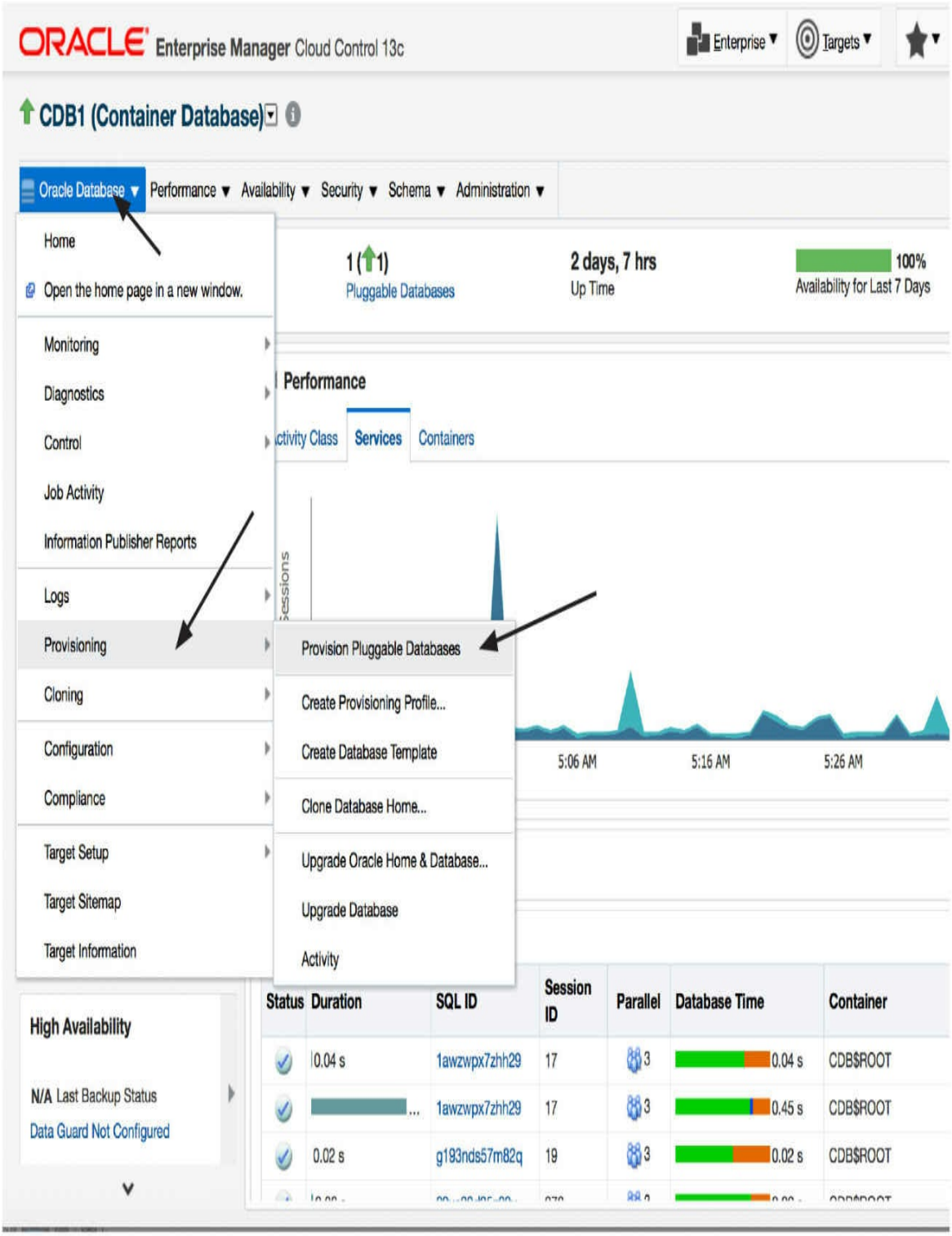


FIGURE 2-14. *Choose Provision Pluggable Databases*

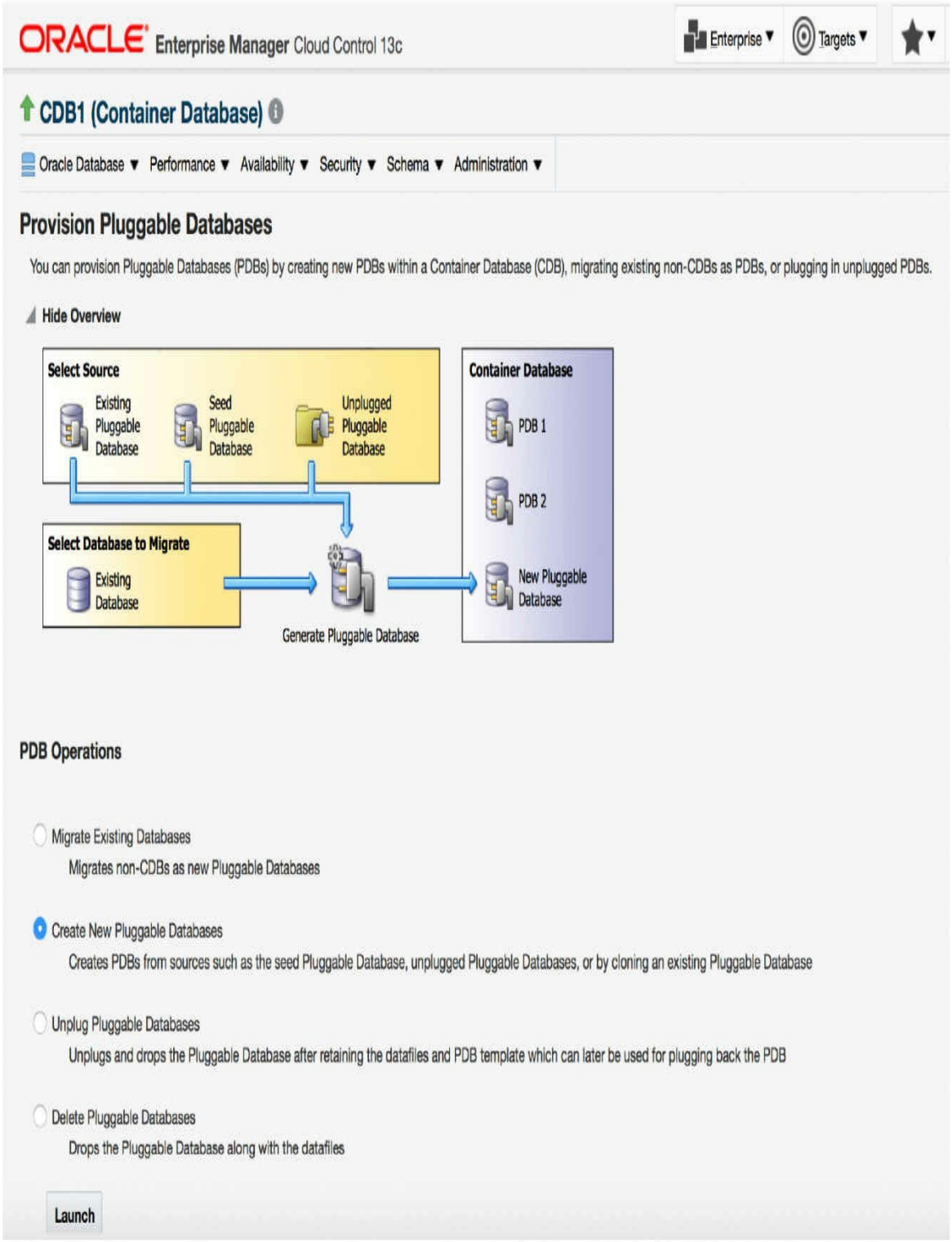


FIGURE 2-15. *Creating a new PDB*

For more information on using Cloud Control, refer to the online documentation for EM Cloud Control 13*c*.

Using the catcon.pl Script

Imagine that you have a CDB with 100 PDBs, and each database is used for the same application, but by 100 different customers, so that each has its own copy of the data. These might be production customers, or perhaps the end customer is 100 developers, again each with his or her own copy of the application database (PDB). The application vendor issues an update script to be executed against every PDB to upgrade to the latest application version.

This entails running a single script, which could be either basic or complex, on each of these PDBs. Needless to say, this can be a time-consuming job, perhaps alleviated only by writing some clever additional scripts to assist with the process. But before you launch into such efforts, the good news is that this is no longer necessary, because Oracle Database 12*c* provides a Perl-based script that can assist with precisely these types of operations!

It is to be expected that some DBAs may be reluctant to use this script, but you can be confident that this is a well-tested method and piece of code. Furthermore, if you look closely at the DBCA and the scripts it invokes against a CDB, you will notice that Oracle has actually implemented the use of the catcon.pl script in its own processes. This is now a critical component under the hood of the DBCA, and it is key in the process of creating and upgrading Oracle databases.

Before we look at some examples, let’s first highlight some of the key requirements, along with a summary of the commands and arguments, used by the catcon.pl script. Perhaps most importantly, before executing this script, ensure that you update the `PERL5LIB` and `PATH` environment variables, and both of these should include the $ORACLE_BASE/rdbms/admin path. Here’s an example:

```
export PERL5LIB=$ORACLE_HOME/rdbms/admin:$PERL5LIB
export PATH=$ORACLE_HOME/bin:$ORACLE_HOME/perl/bin:$PATH
```

Once these are set, you are ready to begin using the catcon.pl script. The next consideration is the key input flags used by this script. You need to be aware, first of all, that there are two mandatory argument requirements:

- **`-b log-file-name-base`** The first option `-b` takes a parameter that specifies the base name that will be used for the log files that will be generated when the catcon.pl script is executed.
- **`--<sqlplus-script>`** The second option is the name of a SQL*Plus script that should be executed.

Or

- **`--x<sql-statement>`** The second option can be a standalone SQL statement.

Other key arguments for the catcon.pl script include these:

- **`-d`** Directory where script to be executed is located
- **`-l`** Directory to be used for spool files
- **`-c`** Container(s) in which scripts/SQL are to be executed
- **`-C`** Container(s) in which scripts/SQL are *not* to be executed
- **`-u`** Username/password (optional) to run user-supplied scripts (defaults to / as sysdba)
- **`-w`** Environment variable that will hold the user password for user specified with `-u`
- **`-U`** Username/password (optional) to run internal tasks (defaults to / as sysdba)
- **`-W`** Environment variable that will hold the user password for user specified with `-U`
- **`-e`** Sets echo on while running SQL*Plus scripts

For more detail on the options available, execute the catcon.pl script without any options specified to generate a full usage listing and description of each.

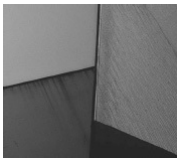
So, for example, to run a script called xyz.sql in all PDBs except CDB$ROOT and PDB$SEED, you would enter the following:

```
export PERL5LIB=$ORACLE_HOME/rdbms/admin:$PERL5LIB
export PATH=$ORACLE_HOME/bin:$ORACLE_HOME/perl/bin:$PATH
cd $ORACLE_HOME/rdbms/admin
perl catcon.pl -e -C 'CDB$ROOT PDB$SEED' -l /home/oracle/logs -b xyz-
out -d /home/oracle/sql xyz.sql
```

Summary

This chapter covered the basics to get you started with Oracle Database 12*c* Multitenant. It detailed the creation of the CDB as well as one or several PDBs. However, there is a lot more to multitenant. What is clear is that some of the benefits and advantages of using multitenant are becoming apparent—especially the ability to create a PDB in a few seconds and the flexibility it can bring to provisioning.

But at this stage you probably have more questions. Should you stay with the old architecture, or move to multitenant? You may not be sure that you can even move to multitenant, but, if possible, how is it done? And what if you are using Standard Edition (SE, SE1 or SE2)—is multitenant even an option? Or perhaps you are using Enterprise Edition and cannot wait to get your hands on the new technology—did you know it is an additional licensed option? The next chapter will assist you in answering these questions, and much more.



CHAPTER 3

Single-Tenant, Multitenant, and Application Containers

In the previous chapter we detailed how to create a container database, an indispensable foundation of the multitenant architecture. However, at this point you may be wondering whether multitenant is for you. Perhaps you have heard the term "multitenant option," which suggests that additional licensing is required, or you exclude yourself because you are running Oracle Database Standard Edition. On the other hand, you are also aware that the old architecture, known as non-CDB (non–container database), is now deprecated. So what should you do?

These are important questions and concerns that we will consider before going on to describe the multitenant features in detail. Of course, to describe what is available with the different editions and options, we will make mention of specific features—but don't worry, because we will return to address the features thoroughly in later chapters.

Multitenant Architecture Is Not an Option

Let's make it clear from the get-go: the multitenant architecture is available in 12*c* for all editions, with no additional licensing option required. The main characteristic of multitenant is the separation of dictionaries: system metadata and system data are in CDB$ROOT, whereas user data and metadata are in a pluggable database (PDB). This new architecture is defined at the database creation stage with `ENABLE PLUGGABLE DATABASE`. It is available at no additional cost for the Enterprise Edition or Standard Edition. At the time of writing, there is no XE edition for 12*c*, but we expect that when it is released, it will be multitenant as well.

You need to be aware of one limitation regarding multitenant: You cannot create more than one PDB per CDB unless you have purchased a multitenant option license. This means that without purchasing the option, your multitenant database has at maximum one tenant—it is a *single-tenant configuration*. Although the multitenant architecture was obviously introduced with the multitenant option in view (hence its name), it is also available and usable without this.

If you don't intend to purchase the multitenant option, or if you are using Standard Edition, you may think that you don't need to delve into this new

architecture and acquire new skills for its administration. But you need to rethink this approach. In this chapter we will explain the advantages of using a single-tenant database, but before we get to this, there is one fundamental reason to move ahead with it.

Non-CDB Deprecation

Chapter 1 introduced multitenant as a major change in the Oracle Database. There are likely scores of places in the Oracle Database code where the developers had to introduce additional operations for the multitenant context, but the existing code for non-CDB is still there. Obviously, as the software evolves in the future, new features will be implemented on the new architecture as a priority. The old architecture is still supported, so you can create non-CDB databases in 12*c* and obtain support and fixes, but if you want to benefit from all the latest features (and innovation of development focus), then it is recommended that you move to the multitenant architecture. This is exactly what deprecation means, as it relates to the non-CDB architecture, and the "Oracle Database Upgrade Guide" is clear that "deprecated features are features that are no longer being enhanced but are still supported for the full life of the release."

Is it bad to use deprecated features? Absolutely not! But it's a bad idea to completely ignore new features that are introduced to replace the deprecated ones. For example, when you start work on a brand-new project, the usual recommendation is to avoid using deprecated features, because you want to benefit from all the latest features available, and because deprecated features may become desupported in the foreseeable future. However, if you are upgrading a current application that will likely exist for only a few more years, you probably just want to keep it as it is, without major changes.

For multitenant, the same principle holds, and you probably won't move all your existing production databases to multitenant architecture when upgrading to 12*c*; instead, you'll retain some of them as non-CDB. Still, it is a good idea that you begin to learn about and create CDB databases, perhaps for a test environment to begin with, so that you build your familiarity with multitenant. With this approach, you have time to learn and adapt your scripts to this new architecture, not because you need it immediately, but because it's sure to be in your future.

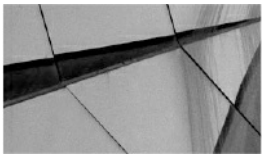
Our message about the non-CDB deprecation is this: Don't worry; the

non-CDB that you know will still be around for years, and you can stay with it as long as you are not comfortable with multitenant. It is still supported in 12*c* and will probably continue to be for several future releases. But that should not prevent you from learning and trying out multitenant on noncritical databases.

Noncompatible Features

Another reason to keep the non-CDB architecture for your database is to make use of one of the few features that is not yet supported in multitenant. There are two possible reasons for such a scenario:

- The feature/functionality itself was already deprecated when multitenant arrived, so it has not been enhanced to work in a CDB. This is the case, for example, with Oracle Streams. Oracle has deprecated this and recommends using an additional alternative product, Golden Gate. Oracle Streams is still supported in 12*c*, but only with the non-CDB architecture. Note that alternatives will be covered in Chapter 13.
- Some new features have been developed independently of multitenant and may not be immediately available in CDB. That was the case with Information Lifecycle Management features (Heat Map and Automatic Data optimization), which arrived in 12.1, but only for non-CDB. This limitation has now been removed with the release of 12.2. Another example is *sharding*, a new feature in 12.2, which is not compatible with multitenant at the time of writing. Even features that are available in multitenant may be available only at the CDB level. This is the case with Real Application Testing in 12.1, for example, which cannot be used at the PDB level, although this limitation has been overcome with the release of 12.2.



NOTE

At the time of writing, features such as Change Notification,

Continuous Query Notification, and Client Side Result Cache are not yet available for a CDB in release 12.2.

Single-Tenant in Standard Edition

In the Standard Edition, you don't have access to the multitenant option that allows for more than one user PDB; this is an inherent limitation of this edition. Furthermore, in the Standard Edition, features that are not available are disabled at installation time. For example, the following shows a container database with one pluggable database, PDB, in addition to the seed:

```
SQL> show pdbs
    CON_ID CON_NAME                        OPEN MODE  RESTRICTED
---------- ------------------------------ ---------- ----------
         2 PDB$SEED                       READ ONLY  NO
         3 PDB                            MOUNTED
```

Creating more pluggable databases is explicitly disallowed:

```
SQL> create pluggable database NEWPDB admin user admin identified by oracle
file_name_convert=('pdbseed','newpdb');
create pluggable database NEWPDB admin user admin identified by oracle
ile_name_convert=('pdbseed','newpdb')
                           *
ERROR at line 1:
ORA-65010: maximum number of pluggable databases created
```

The maximum number of PDBs in a Standard Edition CDB is two: one PDB$SEED (that you are not allowed to change except for modifying the undo tablespace) and one user PDB.

Data Movement

Given the limited features available in the Standard Edition, you may ask if it's better to have a single-tenant CDB rather than a non-CDB. When you

create a CDB, you immediately see the overhead: three containers for only one database, which means more datafiles to store the CDB$ROOT and the PDB$SEED. Multitenant is designed for consolidation, but here we see the opposite, because each database uses more space. But you must remember that multitenant is also beneficial for agility in data movement, and this is a great feature even for single-tenant.

There are three ways in which to transport data. The first is a very flexible, but very slow, method: logical transport through Data Pump. Data is extracted by Data Manipulation Language (DML), target tables are created by Data Definition Language (DDL), rows are inserted by DML, and then indexes are rebuilt. When you want something faster than this, you can use transportable tablespaces, where data is shipped physically with the datafiles and only metadata is created logically. This is generally quick, except when you have thousands of tables, as many enterprise resource planners (ERPs) do, and it still takes considerable time to create those tables, even though most are empty. Note, however, that the transportable tablespaces' import functionality is not available in Standard Edition.

In multitenant architecture, the user metadata is stored separately from the system metadata. Each PDB has its own SYSTEM tablespace that contains this user metadata only, meaning that it can be transported physically. This is a key feature that PDBs provide: the ability to transport by unplug/plug. This mechanism is even superior to transportable tablespaces, which is why Oracle product manager Bryn Llewellyn calls it the "third generation of Data Pump" in his Oracle multitenant white paper. The really good news is that plug/unplug operations are allowed in the Standard Edition and in remote cloning, so you can clone a PDB into a new CDB with a simple command.

Thanks to the agility of PDBs, the small overhead of the CDB structure pales against the ease and efficiency of database movement and cloning operations across servers and versions. This more than makes up for the missing transportable tablespaces functionality in the Standard Edition.

Security

Multitenant has been developed to cater to large numbers of PDBs, such as in cloud environments, and for this reason, many new security and isolation features have been introduced. We will cover those features in Chapter 6, but

some of them also are very interesting from the single-tenant perspective, and perhaps the best example is the ALTER SYSTEM lockdown. In a developer database, you may be tempted to give ALTER SYSTEM privileges to the developers in case they want to test new settings, kill their own sessions, and the like. But this privilege is definitely too broad and powerful, as with ALTER SYSTEM you can basically do whatever you like on the system. With the new PDB lockdown profiles you can, alternatively, grant ALTER SYSTEM and enable or disable exactly those operations and access to parameters you deem appropriate.

Lockdown profiles and OS credentials are also very interesting for those applications in which the application owner must be granted powerful privileges. In multitenant, you can give the CREATE DIRECTORY privilege and constrain the absolute path where the PDB users (even the DBA) can create a directory. Multitenant architecture brings a separation of roles between root and the PDB, and you will appreciate this even in the Standard Edition.

Consolidation with Standard Edition 2

Because you can have only one PDB per CDB, you will probably have a number of CDBs on a server. Even among different CDBs, each PDB must have a unique name because, as you will see in Chapter 5, its name will be a service registered to the listener, and you may not want one listener per CDB in this context. Take care when naming CDBs, because you may be tempted to differentiate them with a trailing number, but it is recommended that you ensure that the ORACLE_SID remains unique for each server even when ignoring trailing numerics. Figure 3-1 shows the warning displayed in the Database Configuration Assistant (DBCA).

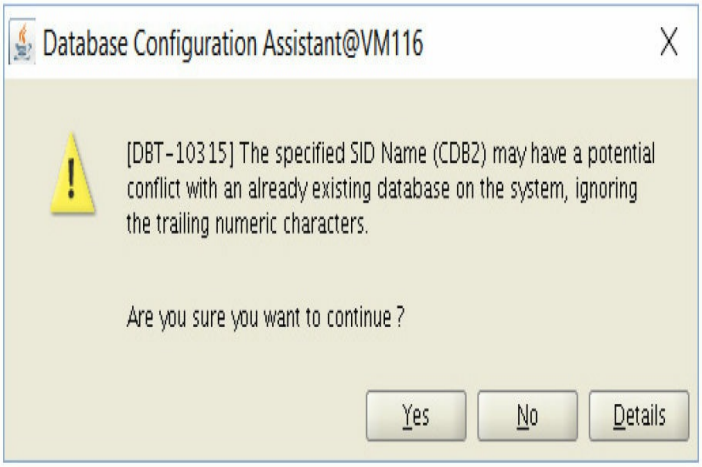


FIGURE 3-1. *Warning when only suffix numbers differ in ORACLE_SID*

This advice probably results from Real Application Clusters (RAC) considerations, where the database name includes an instance number suffix. And remember that in Standard Edition 2, you can have high availability, because RAC is still available on a two-socket total cluster.

Because several CDBs are on the same server, you don't want one instance to use all the CPU available. Standard Edition 2 limits each database to run, at maximum, 16 user processes on CPU, but you can lower this by decreasing the `CPU_COUNT` parameter. This *instance caging* was not available in the Standard Edition before 12.1.0.2, but it is now. It's a side effect of the new SE2 limitation that we can turn to our advantage to ensure that critical databases have enough resources.

Another consideration with several single-tenant CDBs is the overhead of having one PDB$SEED for each CDB. Because PDB$SEED is needed only to create new PDBs, you might surmise that it is not needed after you have created your PDB. Although you may be tempted to drop it, you must remember that dropping this is not a supported operation. However, there is something you can do to avoid to backing up PDB$SEED every day, and that is to configure `BACKUP OPTIMIZATION ON`, so that it will be skipped as any read-only tablespace. Just be careful that you don't have an external expiration policy set that is shorter than the RMAN one if you do this.

We hope that we have convinced you that multitenant is also important in

the future of Standard Edition. And even if it seems like a paradox at the outset, you can also consider consolidation with Standard Edition 2 alongside single-tenant agility.

Single-Tenant in Enterprise Edition

If you are using the Enterprise Edition without the multitenant option, you can, and should, run single-tenant container databases. If you haven't read the preceding section about the Standard Edition, you should do so now, because all that has been said there is relevant for Enterprise Edition, and we will only detail here what is possible specifically in Enterprise Edition.

As a sample, here is an operation that can be done in multitenant that you cannot do in a non-CDB:

```
SQL> shutdown;
SQL> alter database open;
```

In non-CDB, or in a CDB at root level, when you close the database, you have to shut down the instance. Here, the statements have been run within a PDB and the shutdown closes only the PDB, but the instance, which is common, is still up. Then it is possible to open the PDB again. You can open and close a PDB as many times as you want without restarting the instance. This may sound inconsequential, but when you think about operations that require mount mode and a closed database, you soon realize its value.

Flashback PDB

The flashback database feature in the Enterprise Edition can revert the state of the database to a previous point in time; this was available only at the CDB level in 12.1. But with the release of 12.2, you can flashback a PDB—this will be covered in Chapter 8.

For now, let's consider a test database in which the developers perform continuous integration tests. They often need to revert to the initial data before each run, but a restore or Data Pump import can take a long time to complete. The quickest way to resume the same set of data is to create a

guaranteed restore point for the initial state and execute flashback database between each run. This is a fast operation except when you have to restart the instance. But in multitenant, as outlined in the previous section, this is not required, and you can now close, flashback, and open the database in mere seconds, which makes the operation possible to run hundreds or thousands of times.

Maximum Number of PDBs

With the Enterprise Edition, you must be careful when you create a PDB, because creating more than one PDB will activate the usage of the multitenant option. If you haven't licensed this option, you must prevent this from happening. There are no easy ways to achieve this in 12.1, but 12.2 introduced the "max_pdbs" parameter which, when set on CDB$ROOT, is the maximum number of user PDBs allowed. It defaults to 4098 in Enterprise Edition but should be set to 1 when the option has not been purchased.

In the Enterprise Edition you must monitor the usage of features to be sure that only licensed options are used. The feature name of CDB is "Oracle Multitenant" (or "Oracle Pluggable Databases" in 12.1.0.2 because of Bug 20718081).

Figure 3-2 shows the Enterprise Manager Database Express interface, which indicates that multitenant is used. This means that the database is a CDB. It does not provide further indication of the usage of the option beyond this, because a single-tenant is still counted as a multitenant database.

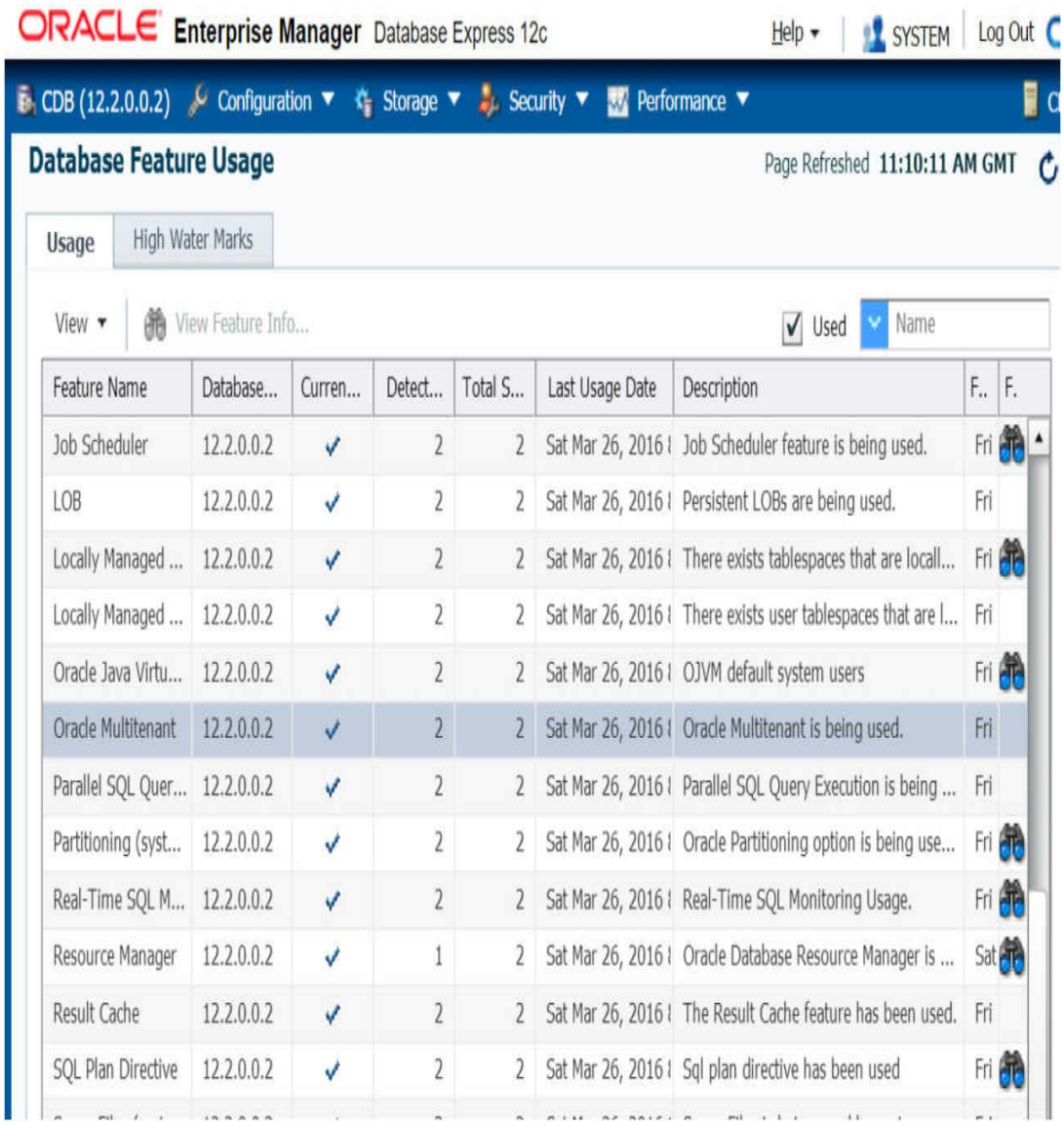
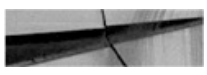


FIGURE 3-2. *Multitenant feature usage from the Enterprise Manager Express*

More detail is available from the DBA_FEATURE_USAGE_STATISTICS view, where the AUX_COUNT column shows the total number of user PDBs (not including PDB$SEED).



```
SQL> select name, version, detected_usages, currently_used, aux_count, feature_info
from dba_feature_usage_statistics where name = 'Oracle Multitenant';
NAME                VERSION     DETECTED_USAGES CURRE  AUX_COUNT FEATURE_IN
------------------- ----------- --------------- ----- ---------- ----------
Oracle Multitenant 12.2.0.0.2                2 TRUE           2
```

This example comes from a CDB in which two PDBs have been created. Basically, AUX_COUNT is calculated as the number of PDBs that have `CON_ID` > 2, so that PDB_SEED is excluded. If AUX_COUNT is 1, you don't need the option. In the preceding example, you must either license the option or drop (move it to a new CDB) one PDB.

What should you do if you have created more than one PDB by mistake? Don't worry, because AUX_COUNT is only the latest value with no history for past values. You can just move the additional PDB to another CDB that you create for it, and then drop the additional PDB to restore the configuration to single-tenant.

The next run of the feature usage sampling will bring back the `AUX_COUNT` = 1, and you can even run the sampling manually if you want:

```
SQL> exec sys.dbms_feature_usage_internal.exec_db_usage_sampling(sysdate); commit;
PL/SQL procedure successfully completed.
```

At this point, we have listed some interesting features that multitenant architecture brings to single-tenant. Nonetheless, the full advantage of multitenant becomes clear only when you consolidate several PDBs in a container database.

Using the Multitenant Option

When you use the multitenant option, you can create hundreds, or even thousands, of PDBs. You can do this in a test CDB to provide a database to each developer, or in a cloud service to provide on-demand databases with Database as a Service (DBaaS). The additional cost of the option can be balanced by the benefit of consolidation, because it enables you to share expensive compute resources at a high level (disk, memory, and background processes). The multitenant option also brings the agility of lightweight PDBs within a consolidated CDB, and you can administer (backup, upgrade, and so

on) as one.

Of course, you will likely not put all your databases into the same CDB, for a number of reasons. First, a CDB is a specific version, so when a new patch set is released, you will probably create a new CDB running in that new version, and then move your PDBs to it. This is the first reason to have several CDBs, and you can think of multiple PDBs in the same way as having different Oracle Homes. A second reason is that you don't want to mix environments (for example, production CDBs will not hold test PDBs). Third, you may also have different availability requirements: one CDB protected in Data Guard and/or in RAC. Changing the availability features for one PDB is as easy as moving it to the appropriate CDB. What is clear is that you cannot consolidate everything and will probably have multiple CDBs to manage.

On the other hand, this does not mean that your CDBs will have only a few PDBs. First, you will probably create different PDBs for the schemas that were consolidated into the same database, aware that multitenant offers better isolation than schema consolidation. Furthermore, the amazing cloning capabilities covered in Chapter 9 will cause you to think differently about managing development databases. You can offer many more environments for development, without increasing the complexity of the CDBs you manage.

Of course, multitenant is new, and our recommendation is to learn it slowly. Take your time, and do not attempt to manage hundreds of PDBs immediately.

Application Containers

You may have a case where your CDB will contain multiple PDBs that run the same application—for example, you may provide your application as a service to multiple customers. In this scenario, you can provision one PDB for each customer, and each will have its own data, but with the same data model. You might immediately think about what Oracle did with the metadata and object links, where the common metadata and objects are stored once in the root only, and ask whether the same can be done for your application. Well, the good news is that in 12.2 with the multitenant option, you can do that with *application containers*. You create one PDB that will behave as the root for your application, and then create another PDB for each

“application tenant” that links to that application root.

We will not go into the details of application containers here, because probably very few readers of this book are SaaS providers (Software as a Service). However, a quick example will demonstrate this feature and serve as an occasion to strengthen what we said in Chapter 1 about metadata and object links. Note that with the introduction of application containers in 12.2, “object links” are now called “data links” and “common data” is called “extended data.”

First, we create the application root, which is itself a PDB, with the `AS APPLICATION CONTAINER` clause:

```
SQL> connect sys/oracle@//localhost/CDB as sysdba
Connected.
SQL> create pluggable database MYAPP_ROOT as application container admin user MYAPP_
ADMIN identified by oracle roles=(DBA);
Pluggable database created.
SQL> alter pluggable database MYAPP_ROOT open;
Pluggable database altered.
```

Then we connect to it and declare that we are starting installation of the application:

```
SQL> connect sys/oracle@//localhost/MYAPP_ROOT as sysdba
Connected.
SQL> alter pluggable database application MYAPP begin install '1.0';
Pluggable database altered.
```

Next we can create the required tablespaces and users:

```
SQL> create tablespace MYAPP_DATA datafile size 100M autoextend on maxsize 500M;
Tablespace created.
SQL> create user MYAPP_OWNER identified by oracle default tablespace MYAPP_DATA
container=ALL;
User created.
SQL> grant create session, create table to MYAPP_OWNER;
Grant succeeded.
SQL> alter user MYAPP_OWNER quota unlimited on MYAPP_DATA;
User altered.
```

Now that we have the application owner, we can create the application schemas. In this example, we will create one common metadata table (same structure but different data for each application tenant) and a common data table (same data for all tenants), to show the new syntax.

```
SQL> alter session set current_schema=MYAPP_OWNER;
Session altered.
SQL> create table MYAPP_TABLE sharing=metadata (n number primary key);
Table created.
SQL> create table MYAPP_STATIC sharing=data (n number primary key, s varchar2(20));
Table created.
SQL> insert into MYAPP_STATIC values(1,'One');
1 row created.
SQL> insert into MYAPP_STATIC values(2,'Two');
1 row created.
```

When this is done, we declare the end of application installation:

```
SQL> alter pluggable database application MYAPP end install '1.0';
Pluggable database altered.
```

You can see that, in this example, we have provided an application name and version. This is for the initial install, but similar syntax is available to manage patches and upgrades during the application lifecycle.

Now we create a PDB for each application tenant; we must be connected to the application root for this:

```
SQL> connect sys/oracle@//localhost/MYAPP_ROOT as sysdba
Connected.
SQL> create pluggable database MYAPP_ONE admin user MYAPP_ONE_ADMIN identified by
oracle;
Pluggable database created.
SQL> alter pluggable database MYAPP_ONE open;
Pluggable database altered.
```

For the moment, the PDB belongs to the application root, but it is completely empty. The linkage to the application root common metadata and data is achieved by syncing this, as follows:

```
SQL> connect sys/oracle@//localhost/MYAPP_ONE as sysdba
Connected.
SQL> alter pluggable database application MYAPP sync;
Pluggable database altered.
```

At this point everything is now ready; our users and tables are there, as you can see:

```
SQL> connect MYAPP_OWNER/oracle@//localhost/MYAPP_ONE
Connected.

SQL> desc MYAPP_TABLE;
 Name              Null?    Type
 ----------------- -------- ------------------------------------------------
 N                 NOT NULL NUMBER

SQL> select * from MYAPP_STATIC;
         N S
---------- --------------------
         1 One
         2 Two
```

Finally, let’s connect back to CDB$ROOT and check the information about our containers:

```
SQL> select con_id, name, application_root, application_pdb, application_root_con_id
from v$containers;

    CON_ID NAME       APPLICATION_ROOT APPLICATION_PDB APPLICATION_ROOT_CON_ID
---------- ---------- ---------------- ---------------- ------------------------
         1 CDB$ROOT   NO               NO
         2 PDB$SEED   NO               NO
         3 PDB        NO               NO
         4 MYAPP_ROOT YES              NO
         5 MYAPP_ONE  NO               YES                                     4
```

The application containers are identified by and have the CON_ID of their application root. Now our application will have its lifecycle. As we did `begin install` we can `begin upgrade` and `begin patch`.

Application containers can go even further: you can automatically partition your application into several PDBs thanks to the container map that we will cover in Chapter 12.

Consolidation with Multitenant Option

The multitenant option brings two new levels to database consolidation on a server. PDBs can belong to an application root, which belongs to the CDB. Figure 3-3 depicts some example use cases. Without the multitenant option, only single-tenant is possible; thus PDB1 and PDB2 in the example stand alone in their CDB01 and CDB02 container databases. With the multitenant option, you can have several PDBs per CDB, as illustrated by PDB3 to PDB5. Finally, you have the possibility of managing common metadata and data in one place for PDBs that belong to the same application (APP1 to APP4).

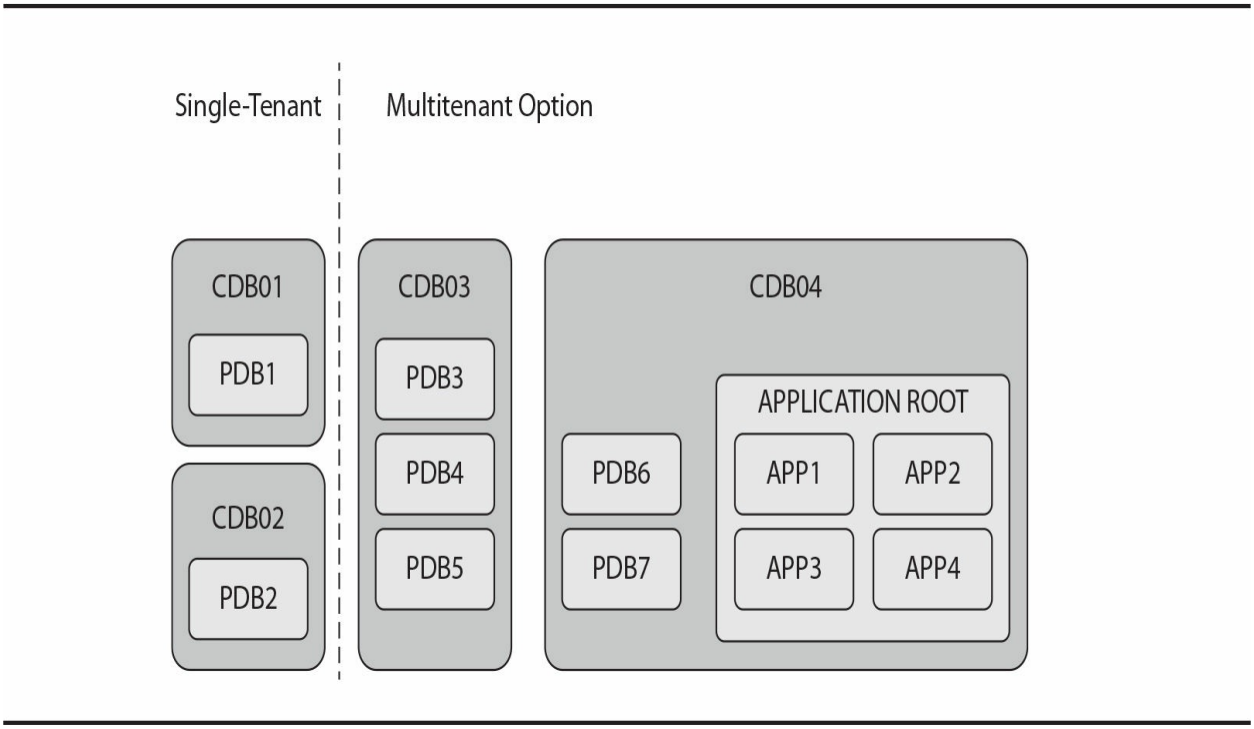


FIGURE 3-3. *Different levels of consolidation on a server*

In addition, of course, the schema level provides logical separation in each PDB, and tablespaces provide physical separation.

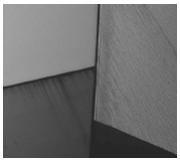
Summary

This chapter ends Part I, which introduced multitenant and its constituent concepts and components. We explained why Oracle Corporation developed this new architecture. As a database professional, you appreciate the Oracle Database because of its reliability, and a major architecture change can raise concerns that this would be compromised. However, as we outlined in Chapter 1, multitenant is not so much a revolution as a logical extension to the transportable tablespaces functionality that has been in place, and working very well, for a number of years. We endeavored to explain that the metadata and object links (data links) are new features and simply provide internal flags to indicate that the code has to switch to the root container to obtain information.

In Chapter 2, we detailed the different ways of creating a CDB, which is new in part, but shares much in common with the approach used for

databases and instances you have known for years. Finally, after reading Chapter 3, we hope that you now realize and appreciate that multitenant is not just for big shops or cloud providers. It is clear that multitenant is the wave of the future for the Oracle Database, and although it brings many changes, it includes a raft of benefits, even for Standard Edition users.

We have named a few features, such as fast provisioning, availability, and cloning, and we hope that this has awakened your curiosity. Let’s turn our attention next to daily practicalities and a consideration of database administration in the context of multitenant.



PART II

Multitenant Administration

CHAPTER 4

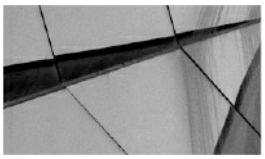
Day-to-Day Management

In the first part of the book, you became familiar with the concepts and promises this new multitenant world brings to the table. Now it's time to get your head out of the clouds, plant your feet firmly on the ground, and begin using it in earnest.

In this chapter, we will focus on a number of common tasks that DBAs perform, concentrating specifically on those that are transformed in the move away from old-fashioned noncontainer databases (non-CDBs).

Choosing a Container to Work With

In a non-CDB, an object is identified by schema name and object name—for example, SCOTT.EMP. These two component labels, plus the object type (which is a table in this example), are sufficient to identify the object uniquely in the database.



NOTE

Specifying the schema name is not always required, because it is implied in the user session itself. Within a session, the current schema is most often the user we connected as, but this can be changed by running the `ALTER SESSION SET CURRENT_SCHEMA` *command within a connected session.*

In a pluggable database (PDB) environment, three pieces of information are required to identify an object: PDB, schema, and object name. However, unlike with schema selection, there is no way to specify the PDB explicitly, because it is always derived from the session. This means that a session always has a PDB container set and, similar to schema, this is implicitly set when the session is initiated. It can also be changed later in an existing session with proper privileges.

We will discuss this in Chapter 5, but let's look at a simple description of the process here. Each PDB offers a service and we connect to the database specifying `SERVICE_NAME`. This implies the container we connect to: the CDB root or a PDB.

If the user account for a connection is a common one and has the appropriate privileges (see Chapter 6), we can change the current container simply with

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

 Session altered.
```

To query the container currently selected, we can either use these SQL*Plus commands,

```
SQL> show con_id

CON_ID
------------------------------
 4

SQL> show con_name

CON_NAME
------------------------------
PDB2
```

Or, should we need to get this information in a query or program, we can use the following:

```
SQL> select sys_context('USERENV', 'CON_ID') con_id,
          sys_context ('USERENV', 'CON_NAME') con_name from dual;

CON_ID     CON_NAME
---------- ----------
4          PDB2
```

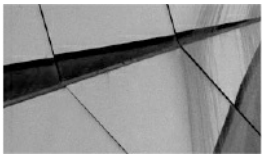
Data Manipulation Language (DML) operations, that is those that

actually access data, are limited to the current container only, unless we apply some of the special options (as outlined in Chapter 12) and the objects accessed are resolved using the implied container, as just described. And although we can change the current container on the fly, a transaction can modify a single container only. You will see, however, that some DDL commands can work on more than one container at a time, while others allow us to specify the PDB explicitly.

Managing the CDB

From the DBA's point of view, a CDB as a whole is very similar to a non-CDB. If we want to use it, we must start an instance; if there is no running instance, it is unavailable to users. Apart from the datafiles, all the files associated with the database are owned by the CDB—this means the SPFILE, control files, password file, alert log, trace files, wallet, and other files.

One major difference, however, is that you cannot connect to a CDB per se, because a connection is always to a container. So, for example, if we want to connect to the CDB, we connect to the root container; this makes the root container a very special type of container, which represents both the database as a whole and the root container itself. This can create confusion, as some of the commands issued in this context affect the database as a whole, while others affect the root CDB$ROOT container only—and, with enough permissions, they can also affect other PDBs.



NOTE

This implies that many tasks are performed in similar ways in the root and in PDBs; it is the context that is different.

Create the Database

Creating the database, either with the `CREATE DATABASE` command or with the Database Configuration Assistant (DBCA), generates a CDB with the root container and the SEED PDB. The database does not have any PDBs when

created, although DBCA offers the option to create some immediately. The CDB$ROOT root container, however, is mandatory because it contains information for the entire database, along with PDB$SEED, which serves as the template on which new PDBs will be based.

Creating a CDB versus a non-CDB involves a single click, which we described in detail in Chapter 2.

Database Startup and Shutdown

Starting up a CDB is no different from starting any other database:

```
SQL> startup
ORACLE instance started.
Total System Global Area  419430400 bytes
Fixed Size                  2925120 bytes
Variable Size             264244672 bytes
Database Buffers          146800640 bytes
Redo Buffers                5459968 bytes
Database mounted.
Database opened.
```

And a shutdown stops the database:

```
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
```

These operations also look the same in the Enterprise Manager Cloud Control, as shown in Figure 4-1.

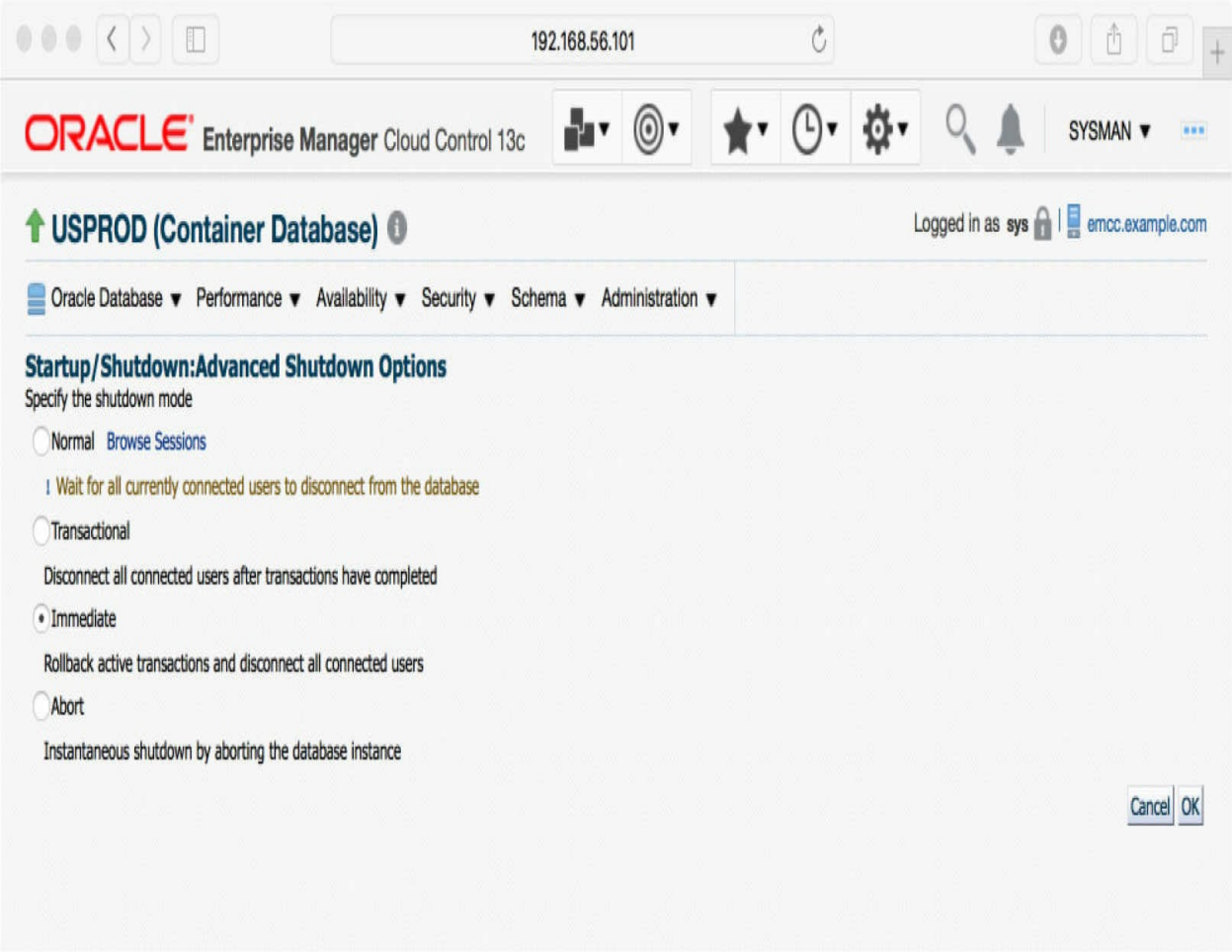


FIGURE 4-1. *Startup/shutdown in Enterprise Manager Cloud Control*

Drop the Database

Version 10*g* introduced the `DROP DATABASE` command, which completely deletes the database. This command has no notion of PDBs, so it deletes all datafiles and all PDBs. To delete a PDB, use the `DROP PLUGGABLE DATABASE` command.

Modify the Entire CDB

Objects and functionality shared by all the containers, including

CDB$ROOT, are managed at the CDB level. And although we are connected to the root for this, we are in reality changing the configuration for the whole database, including every PDB.

First of all, the redo logs are global and they contain changes made by all PDBs. Essentially there is no difference between managing redo logs of a CDB and those of a non-CDB, except that with a CDB, we have to size the logs according to the combined load of all the PDBs plugged into the database.

Consequently, the archivelog or noarchivelog mode is set for the entire CDB, as well as parameters such as archived log destinations, Recovery Manager (RMAN) retention policies, standby databases, and so on. We discuss backups in more detail in Chapter 7 and Data Guard in Chapter 11.

Another database-global file type is the control file. The control files describe the complete structure of the database—that is, all database files and all PDBs. Their content is thus updated by many DDL commands, issued when connected to the root and when connected to a PDB. Commands that work with the control file directly need to be issued at the root container; these include creating standby control files as well backing these up to a file or to trace.

Several important parameters have similar database-wide scope, and a couple of the key ones include global database name (and thus default domain for PDBs) and block change tracking for RMAN incremental backups.

In fact, all database parameters are still set by default at the CDB level, and only a subset can actually be set at the PDB level, in which their specified value overrides that implied by the CDB. With some of these there are additional rules—for example, the `SESSIONS` parameter value in a PDB cannot be higher than the CDB value. In this case, the CDB value determines the hard limit and memory allocations, and the PDB sets only a logical limit.

Modify the Root

Some parameters are set in the root container and determine the default values for the PDBs as well. However, the PDBs are free to set their own value, if required.

One of the simple settings of this sort is the database time zone, which

Oracle uses for storing TIMESTAMP WITH LOCAL TIME ZONE. A similar trivial setting is whether new tablespaces are created as SMALLFILE or BIGFILE by default.

Two considerably more complex settings with broader ramifications are undo management (creation of undo tablespaces in the root and/or PDBs) and the flashback logs configuration. Both of these are covered in more detail in Chapter 8.

Temporary Tablespaces

Every Oracle database can have multiple temporary tablespaces, and 10*g* introduced temporary tablespace groups to assist in their administration. In short, we can use a default temporary tablespace (group) for the database, and we can assign every user a different tablespace.

This is logically expanded in a multitenant database. There is a default set for the whole CDB (set at the root container), and every PDB can override it to use a temporary tablespace created in that PDB; an `ALTER USER` command can override both to set it at the user level.

Managing PDBs

At the broad operational level, the whole CDB resembles a non-CDB for a DBA, and the PDB looks almost indistinguishable from a non-CDB for the ordinary user or the local PDB administrator. But this leaves an administrative gap: management of the PDB by the DBA. It is in this area that we find a number of new tasks and associated commands to invoke them.

Create a New PDB

A PDB first has to be created (see Chapter 1) or copied from another (see Chapter 9). Both approaches are faster and simpler than creating an entire CDB and, after all, this agility is one of the key selling points of the multitenant architecture. With Oracle 12*c*, it is now easy to provision one or more databases, whether it is for testing, development, or production, or because a user has requested a new database in his or her Oracle Public Cloud dashboard.

Open and Close a PDB

Starting a CDB does not imply that all of its member PDBs are opened automatically, as Oracle actually leaves the decision to us. A PDB can be in one of four states, as listed next; notice that there is no NOMOUNT state. Only the CDB as a whole can be started in NOMOUNT state, and in that case there is no control file opened, so the instance does not know which PDBs are in the database.

- **MOUNTED** Data is not accessible, and only an administrator can modify the structure, including files, tablespaces, and so on
- **MIGRATE** Used during various Oracle maintenance operations (such as running scripts for patching)
- **READ ONLY** Accessible to users, in read only
- **READ WRITE** Fully accessible to users for both read and write operations

All three open states can be further constrained to enable only those users with the RESTRICTED SESSION privilege access.

Alter Pluggable Database Statement

The most obvious way to change the open state is by using `ALTER PLUGGABLE DATABASE` statement. The syntax is very similar to `ALTER DATABASE` for the CDB as a whole:

```
alter pluggable database open;
alter pluggable database open read only;
alter pluggable database open upgrade restricted;
alter pluggable database close;
alter pluggable database close immediate;
```

In this basic form, it affects only the currently selected container. It would be a lot of typing to switch to a container, open it, and to repeat this for each and every one. Instead, we can directly specify which container to

modify.

So, for example, when in the root we can execute this:

```
alter pluggable database PDB1 open;
alter pluggable database ALL open;
alter pluggable database ALL EXCEPT PDB2 open;
```

This syntax is permitted even if the current container is a PDB; however, the specified PDB must be the current container, or we get an error:

```
SQL> alter pluggable database PDB2 open;
 alter pluggable database PDB2 open
*
 ERROR at line 1:
 ORA-65118: operation affecting a pluggable database cannot be
performed from another pluggable database
```

Startup Pluggable Database Statement

For those DBAs who like the SQL*Plus `STARTUP` command, this has been enhanced in 12*c* and now also supports PDBs. This is not an SQL command, per se, which constrains it to the SQL*Plus console, and the list of supported options is also limited.

When the current container is the root, you will also notice that the command is often longer than its `ALTER` counterpart:

```
startup pluggable database PDB2;
startup pluggable database PDB2 open;
startup pluggable database PDB2 open read only restrict;
```

The `FORCE` keyword is also available in this context; it closes the database first, before opening it again:

```
startup pluggable database PDB2 force;
```

When the current container is a PDB, and working within that PDB, the `startup` and `shutdown` commands imitate the syntax of a non-CDB. This is part of Oracle's pledge to have "all things work like before" with the move to 12*c*'s multitenant architecture. So the PDB admin can simply connect to the database and issue the old, trusted, and proven `startup`/`shutdown` commands and receive the expected results.

```
SQL> startup

Pluggable Database opened.

SQL> shutdown;

Pluggable Database closed.
```

Note that even the shutdown abort and transactional keywords are accepted in these commands, but, functionally speaking, they are ignored.

Use Enterprise Manager

With Enterprise Manager Cloud Control it is also possible to open and close PDBs, as shown in Figure 4-2.

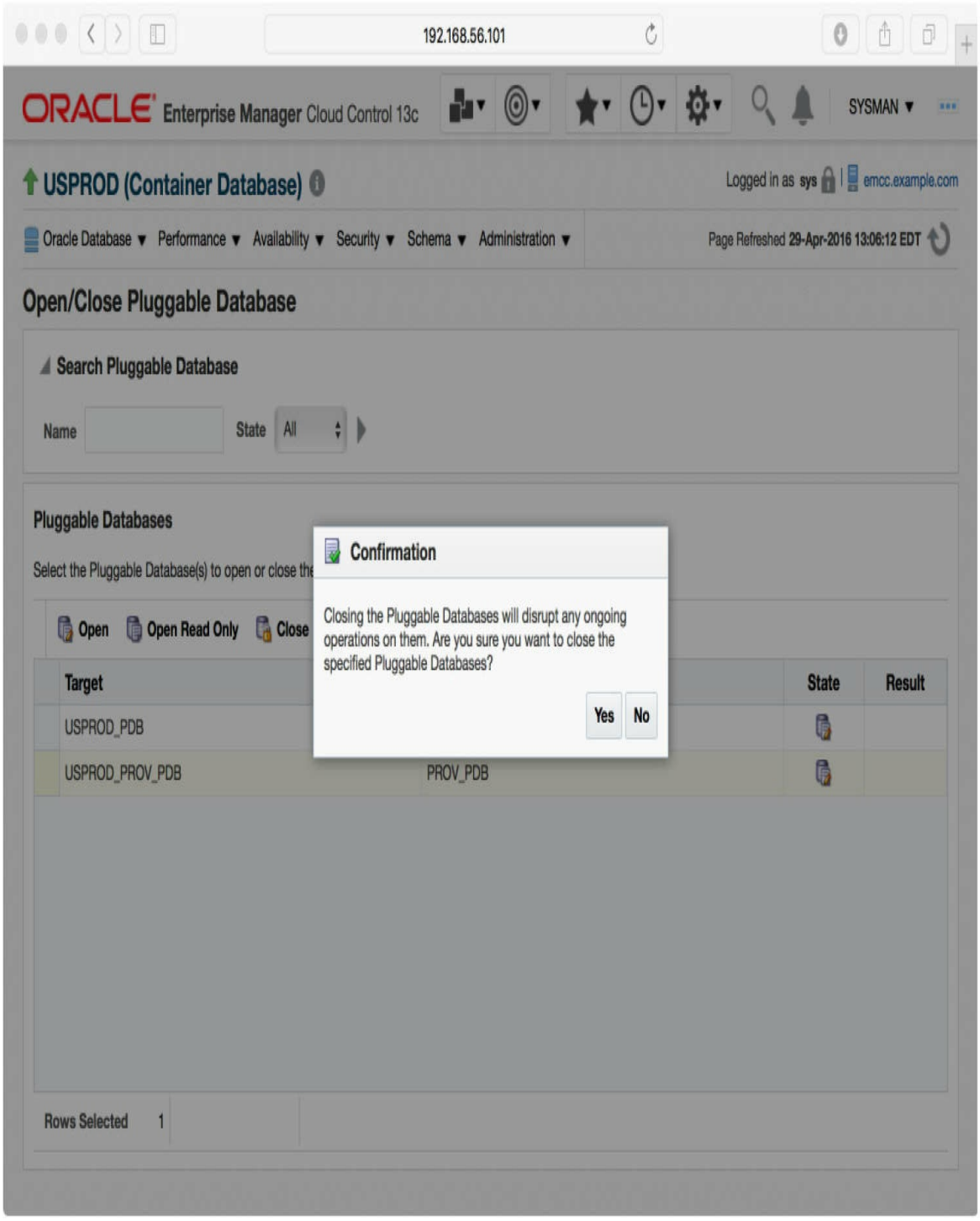


FIGURE 4-2. *Close a pluggable database*

Save the PDB Open State

When 12.1.0.1 was first released, starting up a CDB left all PDBs mounted and inaccessible, and the DBA had to open them manually or write system triggers to do it automatically. However, since 12.1.0.2, there is now a provision to have Oracle do this for us automatically, every time the CDB is started and opened again. We can set the desired state using the SAVE STATE command.

We need to put the PDB into the desired state, and then set this as the requested after-restart mode using this command:

```
SQL> alter pluggable database PDB2 save state;

Pluggable database altered.
```

The command stores the current state of the PDB in the DBA_PDB_SAVED_STATES view:

```
SQL> select con_name, state from DBA_PDB_SAVED_STATES;

CON_NAME   STATE
---------- --------------
PDB2       OPEN READ ONLY
```

In this example, whatever state the PDB is in, upon restart it will be opened read-only again.

The DBA_PDB_SAVED_STATES view contains records in all the databases for which we have issued save states, provided the current state is not MOUNTED. To clear this setting and remove the row for a PDB, either execute save state when it is MOUNTED or use DISCARD STATE. With no record, the PDB will not be opened after CDB restart, and instead remains in the MOUNTED state.

Open the PDB in a Cluster Database

A PDB does not have to be open on all instances, and we can actually pick and choose where we open it. Where it has been opened, though, the mode must be the same among all instances, though we can mix only one open mode and the mounted mode. This is, after all, the same as for an entire CDB or non-CDB.

The `ALTER PLUGGABLE DATABASE OPEN` and `ALTER PLUGGABLE DATABASE CLOSE` commands can specify which instances to affect:

```
alter pluggable database open instances = ('DB1', 'DB2');
alter pluggable database open instances = all;
alter pluggable database open instances = all except ('DB3', 'DB4');
```

One interesting option in this context is the `RELOCATE` command. This is shorthand syntax for "close here, open somewhere else." The `CLOSE` statement closes the PDB on the current instance (specifying relocation is mutually exclusive with listing instances to affect) and the `RELOCATE` keyword instructs Oracle to open the PDB on an instance either specified by us or chosen by Oracle:

```
alter pluggable database close relocate to DB4;
alter pluggable database close relocate;
```

Saving the state is also a per-instance operation, so we must issue the `ALTER PLUGGABLE DATABASE SAVE STATE` on each of the instances. However, this may not be needed if we use grid infrastructure to manage database services—for example, in a RAC environment. Starting a service on an instance automatically opens that PDB (see Chapter 5).

View the State of PDBs

We can query the current state of all PDBs easily with the V$PDBS view:

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

NAME         OPEN_MODE  RES
------------ ---------- ---
PDB$SEED     READ ONLY  NO
PDB1         READ WRITE NO
PDB2         READ ONLY  NO
```

View PDB Operation History

The simplest journey of events over the lifetime of a PDB would see it starting and ending with its creation. However, as you will see in Chapter 9, things can get decidedly more complicated than this. The CDB_PDB_HISTORY view provides a way of reviewing this, and even for a simple PDB, it's a handy way to see its inception date:

```
SQL> select pdb_name, op_timestamp, operation from cdb_pdb_history
order by 2;

PDB_NAME   OP_TIMEST OPERATION
---------- --------- ---------------
PDB$SEED   07-JUL-14 UNPLUG
PDB$SEED   07-JUL-14 UNPLUG
PDB$SEED   23-FEB-16 PLUG
PDB$SEED   23-FEB-16 PLUG
PDB1       23-FEB-16 CREATE
PDB2       28-FEB-16 CREATE
```

You might wonder where the first four records come from. These refer to when the DBCA template was created and then when DBCA created the database from that template.

Run SQL on Multiple PDBs

There are some restrictions to keep in view when an SQL statement, which is usually intended for a single PDB, needs to be run on more than one PDB.

First, user management and privilege grants behave differently for a PDB than for a non-CDB, as you will see later in this chapter. Second, Chapter 12 shows some examples of PDBs working together in tandem, sharing the data structures and data.

Aside from these options, it is possible to implement a simple workaround. If we log in to the root, we can change the current container on the fly, privileges permitting. So it follows that we can execute the desired SQL in various containers, one by one, by selecting the container in the session, running the SQL, and repeating. This can be done by using an SQL script or with dynamic SQL; execute `immediate` or `DBMS_SQL`.

We don't even have to code the container switch ourselves, because `DBMS_SQL.PARSE` has a new parameter, `CONTAINER`, that allows us to specify where the statement should be run.

Note that the rule that a transaction cannot span multiple containers is still in effect, so we have to commit the changes before running SQL in a different container.

```
DECLARE
 cur INTEGER;
 rc INTEGER;
 cmd VARCHAR2(32767) :=
  'BEGIN update SCOTT.EMP set SAL=SAL*1.1 where EMPNO=1; COMMIT; END;';
BEGIN
 cur := DBMS_SQL.OPEN_CURSOR;
 DBMS_SQL.PARSE (c => cur, statement => cmd,
       language_flag => DBMS_SQL.NATIVE, container => 'PDB1');
 rc := DBMS_SQL.EXECUTE(c => cur);
 DBMS_SQL.PARSE (c => cur, statement => cmd,
       language_flag => DBMS_SQL.NATIVE, container => 'PDB2');
 rc := DBMS_SQL.EXECUTE(c => cur);
END;
/
```

Modify the PDB

The datafiles are owned by the PDBs and the usual SQL statements still apply, so there is no change in creating tablespaces, adding datafiles, or taking them online or offline.

As discussed, in the PDB we can override some of the default parameters set at the CDB level: database time zone, default temporary tablespace, and the choice of BIGFILE/SMALLFILE datafile default, for example.

What is set only at the PDB level is the default tablespace. Because permanent tablespaces are private to PDBs, there is no CDB-level setting. Similarly, the default PL/SQL edition is also set at the PDB level only.

All of these features are present in a non-CDB so, for SQL compatibility, they can also be issued using the `ALTER DATABASE` command when connected to a PDB. Again, this means that the PDB administrator can use the existing trusted and proven commands, which are useful especially if the admin has built up a library of favorite SQL scripts over time.

A unique feature for multitenant PDBs is storage limit. This parameter enables us to determine the maximum allowed size of all tablespaces as well as the maximum space occupied in a CDB-level temporary tablespace:

```
SQL> alter pluggable database storage (maxsize 2G max_shared_temp_size 100m);
Pluggable database altered.
```

It is also possible to specify `FORCE LOGGING` mode in a PDB. Normally this is done at the CDB level to ensure that any `NOLOGGING` operations are actually logged, and thus the integrity and completeness of a standby database, for one, is ensured. But in some cases, we may want to change this setting at the PDB level, possibly disabling the `FORCE LOGGING` mode for that particular PDB (it's a temporary testing database or is not included in any standby, for example). Alternatively, we can set a specific PDB to force logging, while keeping CDB level force logging disabled. (Note that the usual options for setting `FORCE LOGGING` at tablespace level and `NOLOGGING` at table level still apply. In other words, it is possible to create a messy, multilayered setup, although we would advise against this and recommend keeping things simple instead.)

The database must be in restricted mode to run these commands, and then we can invoke the desired setting:

```
SQL> startup restrict

Pluggable Database opened.

SQL> alter pluggable database enable force logging;

Pluggable database altered.

SQL> alter pluggable database disable force logging;

Pluggable database altered.

SQL> alter pluggable database enable force nologging;

Pluggable database altered.
```

Note that there is also a `LOGGING` and `NOLOGGING` clause for a PDB, and this establishes the default mode for any new tablespace created in it. As these clauses set only the default setting for tablespaces created in the future, they do not have an immediate effect on the database, unlike the `FORCE LOGGING` clauses.

```
SQL> alter pluggable database nologging;

Pluggable database altered.

SQL> alter pluggable database logging;

Pluggable database altered.
```

Drop a PDB

The life of a PDB ends with a drop operation. It requires a SYSDBA connection, but it is one of the simplest operations to enact. The syntax

suggests that the only additional input we have in this decision is whether to keep the datafiles on disk or not. In reality, however, only an unplugged database can keep its datafiles, while others have to drop them. (See Chapter 9 for a detailed description on unplugging a PDB.)

To effect this operation, the PDB has to be closed (or unplugged), and this must be performed from the root container.

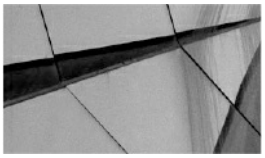
The default option is to keep the datafiles, and Oracle apparently assumes that moving a database by virtue of unplugging and plugging is a more frequent operation than outright dropping.

```
SQL> drop pluggable database PDB1;

Pluggable database dropped.
```

Patching and Upgrades

Each Oracle software major version (such as 12*c*) has a number of additional releases (for example, the 12*c* first release is 12.1, and the second release, 12.2, is also called 12*c*R2). Between these releases, patch sets are released. For example, the 11*g*R2 final patchset is 11.2.0.4. The patch sets, despite the name, are actually distributed as full, standalone software installations. All these levels of software distribution (edition, release, patch set) provide new features and bug fixes. Additionally, on top of patch sets, you can (and should) apply the Patch Set Updates (PSUs, or bundle patches in Windows, which include the patch sets).



NOTE

Upgrading and patching are inherently dynamic processes, but to understand the features available for their administration, you should refer to the Oracle documentation. Information about upgrades is constantly evolving as bugs and issues are encountered, so, for this reason, the standard documentation is not always

sufficient, and you must also review the My Oracle Support (MOS) notes about them. For a friendly and helpful read in light of this, we recommend you bookmark and regularly monitor the excellent upgrade blog from Mike Dietrich, the Master Product Manager for Upgrades and Migrations at Oracle: https://blogs.oracle.com/UPGRADE/.

You can upgrade to 12.2 directly from 11.2.0.3, 11.2.0.4, 12.1.0.1, and 12.1.0.2. If you are using a previous version of the Oracle Database, you can upgrade in several steps, or choose a logical migration approach with a utility such as Data Pump. In this chapter, we discuss physical upgrades, in which data in the actual datafiles remains unchanged.

Upgrade CDB

In the same way that you can upgrade a database, you can upgrade a CDB. This is the simplest way to upgrade all PDBs at the same time, provided you have a maintenance window in which you can stop all their services at the same time.

To upgrade all PDBs, they must first be opened:

```
SQL> alter pluggable database all open;
Pluggable database altered.
```

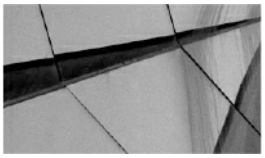
If you want one or more PDBs to remain closed to postpone their upgrade, they will need to be explicitly excluded within the catcon.pl and catctl.pl utility commands, using the `-c` argument.

Pre-Upgrade

In Oracle Database 11*g*R2 you used the utlu112i.sql script to check the database you wanted to upgrade. Oracle Database 12*c* comes with a new pre-upgrade script that installs a package and runs it, detailing suggestions for manual or automatic actions to perform before and after the upgrade. The script is preupgrd.sql, and it calls utluppkg.sql, which installs the package.

There are two important points to note about this script. First, it is

shipped in the rdbms/admin directory of the ORACLE_HOME of the new version of the Oracle software, but remember that it will be run in the database that you want to upgrade, with its older ORACLE_HOME. If both ORACLE_HOME directories are located on the same server, you can call the scripts from the new directory. But you can also copy the preupgrd.sql and utluppkg.sql files.

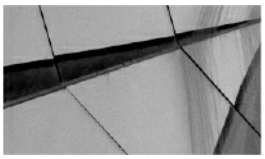


NOTE

The upgrade files shipped in an ORACLE_HOME are actually useless for the databases running from this ORACLE_HOME. Those files are to be used for a database from a previous version ORACLE_HOME.

The second important point is that the shipped upgrade files come from the release of the patch set, but those scripts may evolve over time. You can, and should, always download the latest version from My Oracle Support MOS ID 884522.1 at support.oracle.com/epmos/faces/DocContentDisplay?id=884522.1.

When you run preupgrd.sql, it installs the dbms_preup package and generates a log file and fixup scripts in the following directory: $ORACLE_BASE/cfgtoollogs/<db_unique_name>/preupgrade/.



NOTE

If ORACLE_BASE is not set, it will be replaced by ORACLE_HOME.

Remember that you are in a multitenant environment, so running the script from SQL*Plus will execute it only on CDB$ROOT. You must use the catcon.pl utility introduced in Chapter 2 to enact this at the PDB level. Here is an example in which we run the preupgrd.sql from the future ORACLE_HOME:

```
cd $ORACLE_BASE/product/12202EE
```

First we create the log directory that we will specify with the `-l` argument:

```
mkdir $ORACLE_BASE/admin/$ORACLE_SID/preupgrd
```

Then we run the preupgrd.sql script located in rdbms/admin of the future ORACLE_HOME, specified with the `-d` argument with an absolute path to the current directory:

```
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catcon.pl \
  -d ./rdbms/admin \
  -l $ORACLE_BASE/admin/$ORACLE_SID/preupgrd -b $(date +%Y%m%d%H%M%S) \
  preupgrd.sql
```

The output of catcon.pl goes into the directory and files defined with `-l` and `-b`, but the pre-upgrade package still writes its output in $ORACLE_BASE/cfgtoollogs. Here is an example of the files generated from a two PDB, CDB upgrade:

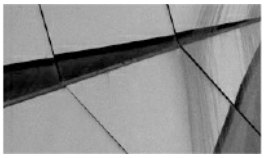
```
cd $ORACLE_BASE/cfgtoollogs/CDB/preupgrade ; tree
.
├── pdbfiles
│   ├── postupgrade_fixups_pdb1.sql
│   ├── postupgrade_fixups_pdb2.sql
│   ├── postupgrade_fixups_pdb_seed.sql
│   ├── preupgrade_fixups_pdb1.sql
│   ├── preupgrade_fixups_pdb2.sql
│   ├── preupgrade_fixups_pdb_seed.sql
│   ├── preupgrade_pdb1.log
│   ├── preupgrade_pdb2.log
│   └── preupgrade_pdb_seed.log
├── postupgrade_fixups.sql
├── preupgrade_fixups.sql
└── preupgrade.log
```

You have a set of logs, pre-upgrade fixes, and post-upgrade fixes for the CDB, and in the pdbfiles subdirectory, one for each PDB.

Note that preupgrade_fixups.sql and postupgrade_fixups.sql contain the code for all containers (CDB$ROOT and all those in the pdbfiles subdirectory), so you can run those scripts with catcon.pl on all containers.

preupgrade.jar In 12.2, the pre-upgrade process received a further enhancement, being bundled into a Java utility, and the previous versions were deprecated. Making use of the same environment as before, where ORACLE_HOME is set to the current environment and the new one is installed in the 12202EE at the same level, we can run this:

```
cd $ORACLE_HOME/../12202EE
./jdk/bin/java -jar ./rdbms/admin/preupgrade.jar FILE TEXT
```



NOTE

Use the `-help` flag to show the options for changing the output

directories.

At the time of writing, the Java solution does not automatically generate a master script to run with catcon.pl, but you can achieve this by concatenating the per-container scripts:

```
cd $ORACLE_BASE/cfgtoollogs/CDB/preupgrade
cat preupgrade_fixups_*.sql > preupgrade_fixups.sql
cat postupgrade_fixups_*.sql > postupgrade_fixups.sql
```

Then, as with the previous method, you end up with one file with “if con_name” conditions that you can run for each container.

Backup or Restore Point

Invoking an upgrade process is a simple operation and is fully automated when you use the Database Upgrade Assistant (DBUA). But problems may occur at any time during the upgrade process. Imagine, for example, that you plan one hour of downtime to upgrade a 10TB database, and there’s no problem with that because the time to upgrade does not depend on the size of the database per se. But what do you do if the upgrade fails in the middle—perhaps due to a bug, server crash, remote connection unexpectedly closed, and so on? Did you count the fallback scenario within the one-hour outage? How long will it take to restore 10TB and recover all the redo generated since the time of this backup?

What is critical is that, before starting an upgrade, you must plan out the fallback scenario. The easiest method for doing this, if you are using Enterprise Edition, is to create a guaranteed restore point:

```
SQL> create restore point BEFORE_UPGRADE guarantee flashback database;
```

Then, in case of failure, you can flashback the database quickly to revert to this pre-upgrade point:

```
SQL> shutdown immediate
SQL> startup mount
SQL> flashback database to restore point BEFORE_UPGRADE;
SQL> alter database open resetlogs;
```

Of course, this assumes that your database is in archivelog mode and that you have enough space allocated for the flashback logs. Helpfully, the required size for the fast recovery area (FRA) is one of the precheck activities performed by the preupgrd.sql script.

You may choose other means by which to back up the database, such as taking a storage snapshot when the database is closed. If your database is in noarchivelog mode, and you want to do a cold backup before the upgrade, keep in mind that the upgrade will not update anything in your user data. Then you can put your user tablespaces in read-only mode and don't need to back them up. Only the system tablespaces remain open (SYSTEM, SYSAUX, UNDO, and all those that have system components objects) and so require backup. This means that you don't need to back up the user tablespaces, which reduces the time to restore if you do need to perform a point-in-time recovery before the upgrade. This can also be automatically achieved with the upgrade utilities: the `-T` option of `catctl.pl` or `-changeUserTablespacesReadOnly` of the DBUA.

Pre-Upgrade Script

The pre-upgrade scripts generated by preupgrd.sql in the cfgtoollogs/<db_unique_name>/preupgrade directory can be run on each container with catcon.pl, as follows:

```
mkdir $ORACLE_BASE/admin/$ORACLE_SID/preupgrade_fixups
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catcon.pl \
  -d $ORACLE_BASE/cfgtoollogs/CDB/preupgrade \
  -l $ORACLE_BASE/admin/$ORACLE_SID/preupgrade_fixups -b $(date
+%Y%m%d%H%M%S) \
  preupgrade_fixups.sql
```

You also have to fix any manual recommendations, but don't hesitate to invoke the pre-upgrade script again to check and confirm that the recommended actions have been completed.

Here is a quick check of the manual tasks that were recommended in one test example we ran:

```
$ grep -E "Executing in container|MANUAL" *
201602261856320.log:Executing in container:  CDB$ROOT
201602261856320.log:Previously failed CHECK pga_aggregate_target is still presently
failing.  It must be resolved MANUALLY by the DBA.
201602261856320.log:Executing in container:  PDB$SEED
201602261856320.log:Previously failed CHECK pga_aggregate_target is still presently
failing.  It must be resolved MANUALLY by the DBA.
201602261856321.log:Executing in container:  PDB1
201602261856321.log:Previously failed CHECK pga_aggregate_target is still presently
failing.  It must be resolved MANUALLY by the DBA.
201602261856322.log:Executing in container:  PDB2
201602261856322.log:Previously failed CHECK pga_aggregate_target is still presently
failing.  It must be resolved MANUALLY by the DBA.
```

These fixes required manual intervention. In this example, we had to increase `PGA_AGGREGATE_TARGET`. Note that we did it manually, and only at CDB level, because the CDB value is the default for the PDB. More detail about parameters will follow later in this chapter in the section "CDB SPFILE."

Upgrade with catupgrd.sql

Now, to upgrade, you must shut down the CDB. The downtime begins. Prior to this, check to ensure that the `COMPATIBLE` parameter is set. You can set this parameter to the current value in case you need to downgrade later. If you have created a restore point, this is actually mandatory; otherwise you will get an ORA-38880 error when mounting from the new ORACLE_HOME.

From here, you can copy the SPFILE to the new ORACLE_HOME, and then change the /etc/oratab file. Then you can start the database from the new ORACLE_HOME in upgrade mode, ensuring that all PDBs are in the required upgrade mode:

```
SQL> shutdown immediate
```

Change /etc/oratab to

```
CDB:/u01/app/oracle/product/12202EE
```

And set the environment like so:

```
. oraenv <<< CDB
```

Then you are ready to startup upgrade:

```
SQL> startup upgrade
SQL> alter pluggable database all close;
SQL> alter pluggable database all open upgrade;
```

Note that the OPEN_MODE displayed by `show pdbs` is MIGRATE here:.

```
SQL> show pdbs
    CON_ID CON_NAME                       OPEN MODE  RESTRICTED
---------- ------------------------------ ---------- ----------
         2 PDB$SEED                       MIGRATE    YES
         3 PDB1                           MIGRATE    YES
```

Time to run the upgrade. The utility to run the catupgrd.sql script is not SQL*Plus but catctl.pl, the parallel upgrade utility optimized to minimize the time the process takes by parallelizing and reducing the number of restarts. In 12.2, running catupgrd.sql directly is not supported.

```
mkdir $ORACLE_BASE/admin/$ORACLE_SID/upgrade
cd $ORACLE_HOME/rdbms/admin
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catctl.pl \
  -n 4 -N 2 -M \
  -d $ORACLE_HOME/rdbms/admin \
  -l $ORACLE_BASE/admin/$ORACLE_SID/upgrade \
   catupgrd.sql
```

In this example we have added the `-n 4` to parallelize with four processes in total, and `-N 2` to use two processes per container, which means that the upgrade of PDBs will take place on two PDBs at a time. One additional point worth noting: Parallel operations are usually unavailable in the Standard Edition (SE), but this is not the case with upgrades. They work the same in SE as in Enterprise Edition (EE), meaning that you can perform parallel upgrades in SE.

You will also see that we have added another parameter, `-M`. By default, once the CDB$ROOT has been upgraded, the instance that was in upgrade mode is restarted to normal mode before the PDBs are upgraded. This is good if you need to open PDBs as soon as they are upgraded, without waiting for the others to complete. Here, with the `-M` flag, the CDB$ROOT stays in upgrade mode until the end. It's faster, but we will have to wait until all of the PDB upgrades finish for them to be open and accessible.

You may choose to have the PDBs open as soon as they are upgraded, and in 12.2 you can even prioritize the way they are processed by using the `-L` argument.

The main log file for the upgrade process is upg_summary.log, which details the elapsed time of the upgrade per container and per component. You can also review the results again like this:

```
sqlplus / as sysdba @ ?/rdbms/admin/catresults.sql
```

At the time of writing, the PDBs upgrade process takes the same time as the CDB$ROOT upgrade process, or even longer, but this will be improved in the future. Actually, thanks to metadata links, a large part of the upgrade DDL does not need to be run on PDBs, but this optimization is implemented only partially now.

Let’s review an example of time taken by an upgrade. The following is the CDSB$ROOT part of the upg_summary.log from 12.1 to 12.2. You can see that the Oracle Server section is only 21 minutes within an overall elapsed period of 92 minutes:

```
Oracle Database 12.2 Post-Upgrade Status Tool           03-07-2016 16:07:02
                            [CDB$ROOT]

Component                               Current         Version  Elapsed Time
Name                                    Status          Number   HH:MM:SS

Oracle Server                             VALID      12.2.0.0.2  00:21:31
JServer JAVA Virtual Machine              VALID      12.2.0.0.2  00:07:37
Oracle Real Application Clusters     OPTION OFF      12.2.0.0.2  00:00:01
Oracle Workspace Manager                  VALID      12.2.0.0.2  00:02:11
OLAP Analytic Workspace                   VALID      12.2.0.0.2  00:00:42
Oracle OLAP API                           VALID      12.2.0.0.2  00:00:24
Oracle Label Security                     VALID      12.2.0.0.2  00:00:17
Oracle XDK                                VALID      12.2.0.0.2  00:02:11
Oracle Text                               VALID      12.2.0.0.2  00:01:00
Oracle XML Database                       VALID      12.2.0.0.2  00:02:42
Oracle Database Java Packages             VALID      12.2.0.0.2  00:00:26
Oracle Multimedia                         VALID      12.2.0.0.2  00:02:41
Spatial                                   VALID      12.2.0.0.2  00:10:51
Oracle Application Express                VALID     5.0.3.00.02  00:14:44
Oracle Database Vault                     VALID      12.2.0.0.2  00:00:32
Final Actions                                                    00:03:49
Post Upgrade                                                     00:01:41
Post Compile                                                     00:13:46

Total Upgrade Time: 01:32:39 [CDB$ROOT]

Database time zone version is 25. It meets current release needs.

Summary Report File = /u01/app/oracle/cfgtoollogs/dbua/upgrade2016-03-06_06-20-12-PM/
CDB/upg_summary.log
```

Our recommendation, in view of this, is not to install any components that you don’t need. With the multitenant option, you will likely create lots of PDBs, so perhaps you will choose to install all components in case you need them in the future. But that will make upgrades take longer. Note that without

the multitenant option, in single-tenant, it’s always better to choose only what is actually needed.

Taking our previous example as a test case, you may choose to upgrade Oracle Application Express (APEX) in advance. Or even better, remove it from CDB$ROOT completely with $ORACLE_HOME/apex/apxremov_con.sql, because it’s not a good idea to have APEX in the root container or the APEX version will be tied to the database version.

Open Normally

At the end of the upgrade process, when everything has completed successfully, we can open the CDB and the PDBs:

```
SQL> shutdown immediate
SQL> startup;
SQL> alter pluggable database all open;
```

Upgrade Resume

In 12.2, if the upgrade fails, look at the `PHASE_TIME` number. Then, if you are able to fix the problem, you can continue the upgrade with the `-p` option. You can also use the `-c` option to choose (and prioritize) the containers.

For example, let’s imagine our upgrade has been interrupted. The catresult.sql shows that the post-upgrade step has not been performed. The log for PDB phases PDBSOracle_Server.log shows nothing after Phase 105 for PDB:

```
Serial   Phase #:102  [PDB$SEED] Files:1    Time: 2s
******************   Migration   ******************
Serial   Phase #:102  [PDB] Files:1    Time: 223s
Serial   Phase #:103  [PDB$SEED] Files:1    Time: 19s
Serial   Phase #:104  [PDB$SEED] Files:1    Time: 245s
Serial   Phase #:103  [PDB] Files:1    Time: 9s
Serial   Phase #:104  [PDB] Files:1    Time: 167s
*****************   Post Upgrade   *****************
Serial   Phase #:105  [PDB$SEED] Files:1    Time: 239s
*****************   Post Upgrade   *****************
Serial   Phase #:105  [PDB] Files:1    Time: 181s
***************   Summary report   ***************
Serial   Phase #:106  [PDB$SEED] Files:1    Time: 3s
Serial   Phase #:107  [PDB$SEED] Files:1    Time: 7s
Serial   Phase #:108  [PDB$SEED] Files:1     Time: 0s
```

And the log file for PDB shows that it started the `PHASE_TIME` number 106 but didn't finish:

```
grep -H PHASE_TIME catupgrdpdb0.log | tail -2
catupgrdpdb0.log:PHASE_TIME___END 105 16-03-06 10:35:06
catupgrdpdb0.log:PHASE_TIME___START 106 16-03-06 10:35:08
```

To resume, we can run it again from this specific phase—that is, only for the PDB container and starting at `PHASE_TIME 105`:

```
cd $ORACLE_HOME/rdbms/admin
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catctl.pl \
  -n 4 -N 2 -M \
  -d $ORACLE_HOME/rdbms/admin \
  -l $ORACLE_BASE/admin/$ORACLE_SID/upgrade \
  -c PDB -p 105 \
catupgrd.sql
```

Without the `-P` flag, which specifies an end phase, it will run all the

remaining phases until the end—and in our example the output shows that this will be Phase 108:

```
-----------------------------------------------------
Phases [105-108]         Start Time:[2016_03_07 16:18:21]
Container Lists Inclusion:[PDB] Exclusion:[NONE]
-----------------------------------------------------
   Time: 8s
*****************   Post Upgrade   *****************
Serial   Phase #:105  [PDB] Files:1    Time: 17s
****************   Summary report   ****************
Serial   Phase #:106  [PDB] Files:1    Time: 1s
Serial   Phase #:107  [PDB] Files:1    Time: 1s
Serial   Phase #:108  [PDB] Files:1     Time: 0s
-----------------------------------------------------
Phases [105-108]         End Time:[2016_03_07 16:18:46]
Container Lists Inclusion:[PDB] Exclusion:[NONE]
-----------------------------------------------------
```

If you encounter an issue with this process, you can obtain a detailed debug trace with the `-z` option, but that is beyond the scope of this book.

Post-upgrade Script

The post-upgrade scripts that have been generated by the preupgrd.sql in the cfgtoollogs/<db_unique_name>/preupgrade directory can be run with catcon.pl:

```
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catcon.pl \
  -d $ORACLE_BASE/cfgtoollogs/CDBPRD01/preupgrade \
  -l $ORACLE_BASE/admin/$ORACLE_SID -b postupgrade_fixups$(date +%Y%m%d%H%M%S) \
  -- postupgrade_fixups.sql
```

And then you can recompile all invalid objects with utlrp:

```
$ORACLE_HOME/perl/bin/perl $ORACLE_HOME/rdbms/admin/catcon.pl \
  -d $ORACLE_home/RDBMS/ADMIN \
  -l $ORACLE_BASE/admin/$ORACLE_SID -b utlrp$(date +%Y%m%d%H%M%S) \
  utlrp.sql
```

Test and Open the Service

From here you can test the application(s) and open the service. Don't forget to drop any guaranteed restore points, if any were used.

How Long Does an Upgrade Take?

Because upgrading is an operation that you perform when an application is offline, you need to estimate the time it will take. Upgrading a CDB takes longer than upgrading a non-CDB because you have multiple containers. However, the upgrade of PDBs is optimized. As the CDB$ROOT is upgraded first, upgrading the metadata and object links should be faster than upgrading a standalone database. Thanks to this, upgrading a multitenant with *N* PDBs should theoretically be faster than upgrading *N* standalone databases. This is, at least, one goal of multitenant architecture from the get-go, but this optimization has not yet been fully implemented. If you are in the single-tenant configuration, upgrading the CDB takes longer than upgrading a non-CDB because three containers are involved: CDB$ROOT, PDB$SEED, and your PDB.

Note that the size of the database, per se, does not determine the time to upgrade, because it is only the data dictionary that is affected. Obviously, a huge number of objects can increase the time to upgrade, but the major cause for extended upgrade duration is the number of components installed.

Database Upgrade Assistant

We have described the manual procedure, which hopefully helps you understand the phases of the upgrade process. However, as in previous versions, the procedure outlined is fully automated with the DBUA, available

from the Oracle Home of the new version. There, you can choose the database to upgrade, along with the different upgrade options.

If you don't have the graphical environment set up, or you want to script the upgrade, you can also run DBUA in silent mode. There is no response file, but all options are available from the command line. You can also run the pre-upgrade step only from the new Oracle Home, as per the following:

```
./bin/dbua -silent -executePreReqs -sid CDB -sysDBAUsername sys -sysDBAPassword …
```

The output directory created under $ORACLE_BASE/cfgtoollogs/dbua is displayed and you will find all log, pre-upgrade, and fixup scripts directed there.

For the upgrade, you can see all options from the inline help:

```
./bin/dbua -silent -help
```

An example of a useful additional option is to create a restore point automatically with the `-createGRP` argument.

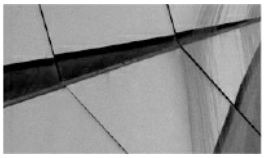
We will not offer any preference here to the manual or DBUA upgrade, because this depends on a number of factors, such as your experience, the requirements, and the complexity of the database. In short, you can have more control with the manual procedure, but everything is possible with DBUA, graphically or in a scripted way.

Plugging In

Upgrading the CDB is fastest when you have a large maintenance window to upgrade all the PDBs at the same time. But in real life, you probably don't want to upgrade all at the same time because PDBs are for multiple applications or "tenants." And here the multitenant architecture can help. Several ways to move PDBs across containers will be covered in Chapter 9, and those movements do not require that you work in the same version. In addition, a very interesting way to upgrade a PDB is to move it to a CDB that

is in a higher version. This is an option with plug-in and remote cloning, and it is even possible without the multitenant option, where you have only one PDB per CDB.

The data movement process is described in Chapter 9, but it's important to note that once you've plugged in or cloned from a previous version, you need to bring your PDBs to the new version. Remember that most of the PDB dictionary objects are links to the CDB$ROOT, so they are usable only when those metadata and object links are upgraded to the CDB version. The same catupgrd.sql script performs this action, and it is run in a specific container with the catctl.pl script, using the `-c` option.



NOTE

If you choose to unplug/plug in, remember that you always need to have the PDB protected by a backup. Drop it from the source only when it is upgraded and backed up in the target.

Basically, upgrading a plugged PDB is the same as upgrading a PDB in a CDB. You specify the container with the `-c` option of the catctl.pl script. Verification of the need to upgrade can be checked from `PDB_PLUG_IN_VIOLATIONS`:

```
SQL> select name,cause,type,status,action from pdb_plug_in_violations;

NAME  CAUSE         TYPE  STATUS
----- ------------- ----- -------
ACTION
------------------------------------------------------------
MESSAGE
--------------------------------------------------------------------------------
MYPDB VSN not match ERROR PENDING
Either upgrade the PDB or reload the components in the PDB.
PDB's version does not match CDB's version: PDB's version 12.1.0.2.0. CDB's version
12.2.0.0.1.
```

Remote Clone from Previous Version

Remote cloning uses the remote file server (RFS) process, which may return an error if the client uses a higher version:

```
ORA-17628: Oracle error 17630 returned by remote Oracle server
ORA-17630: Mismatch in the remote file protocol version client server
```

For example, when cloning into 12.2 from a remote 12.1.0.2 database, you receive the following error, with the remote alert.log showing:

```
ORA-17630: Mismatch in the remote file protocol version client 3 server 2
```

The solution is to apply patch18633374: COPYING ACROSS REMOTE SERVERS on the source (https://updates.oracle.com/download/18633374.html) to allow file transfers to a higher version.

The cloning procedure will be described in Chapter 9, and the dictionary upgrade is performed by running catupgrd.sql on the container.

Patching

Upgrades to new releases (such as from 11.2.0.4 [11*g*R2 Patch set 3] to 12.1.0.1 [12*c*R1]) or to a new patch set (such as 12.1.0.1 [12*c*R1] to 12.1.0.2 [12*c*R1 Patch set 1]) are provided by installing a new Oracle Home and then, from that new Oracle Home, executing `startup upgrade` and catuprgd, as shown earlier. New versions, releases, and even patch sets provide large numbers of bug fixes and new features, and thus change significant amounts of dictionary structure and data.

Between these Oracle releases, partial patches are released, which contain only important fixes and changes in a few libraries in the Oracle Home. These have minimal impact on the dictionary:

- **Security Patch Update (SPU)** Previously referred to as CPU (Critical

Patch Update), these updates provide the quarterly cumulative security fixes.

- **Patch Set Update (PSU)** Provides SPU and additional important fixes. Their goal is to stabilize—that is, to fix critical issues without the risk of regressions.
- **Bundle Patch Update (BPU)** Contains a bunch of patches related to a particular component. For example, CBO patches are not in PSU but can be provided in the Proactive Bundle Patch.

Those include only the libraries or files that are changed and are applied by the OPatch utility. In 12*c*, the changes to the dictionary are done through the datapatch utility provided in the OPatch. Datapatch checks all containers to know which ones have to be patched. However, only open PDBs are verified, so those that are closed will be in restricted mode when they are opened, and datapatch will need to be run again for them.

Here is an example in which we have plugged the PDB2 PDB to a CDB that has a patch set additional to the CDB of origin:

```
$ORACLE_HOME/OPatch/datapatch -verbose
......
Currently installed SQL Patches:
  PDB CDB$ROOT: 17027533
  PDB PDB$SEED: 17027533
  PDB PDB2:
......
For the following PDBs: PDB2
  Nothing to roll back
  The following patches will be applied: 17027533
......
Patch 17027533 apply (pdb PDB2): SUCCESS
```

Only the PDB2 has to be patched.

In this example, the multitenant architecture brings significant benefit even when you have only one PDB (the single-tenant that you can use without the multitenant option). If you want to upgrade or patch the entire

CDB, it will take longer, because there are three container dictionaries to update. But if you create a new CDB with a new version, you can simply unplug/plug in. There is no file movement needed, because it is on the same server; then run catupgrd or datapatch. This is faster because there are only relatively few metadata links to check and update.

Using CDB-Level vs. PDB-Level Parameters

Initialization parameters control the instance behavior. Their first utilization is at instance startup, which is why they are called “initialization parameters.” These parameters are also used later, however, and can control the session behavior. Note that each session actually inherits settings from the instance, and some of these can be changed by the session. It’s also possible to use different parameters at the statement level, with the `OPT_PARAM` hint.

The parameters are not stored within the database itself, because a large number of them are needed before opening the database. In older versions of Oracle Database, they were stored in a text file, which Oracle calls the PFILE (the parameter file), and that many DBAs referred to as the init.ora. This file was read by the session running the startup command, but in current versions of Oracle, the parameters are stored in a server parameter file (SPFILE), which is managed by the instance. We can change parameters in the SPFILE with the `ALTER SYSTEM` command, or we can re-create the SPFILE from a PFILE if we are unable to start the instance.

This does not change in multitenant, except that we have a new level between the instance and the session: the PDB.

CDB SPFILE

The CDB SPFILE is the same as the SPFILE you are familiar with in the non-CDB context. It stores all parameters that are set by `ALTER SYSTEM SET … SCOPE=SPFILE` when this command is run at CDB$ROOT level. Those parameters are read when the instance is started, are used by CDB$ROOT, are the defaults for sessions in CDB$ROOT, and also serve as the default for the PDBs when they do not have their own setting.

PDB SPFILE Equivalent

PDBs offer the same Oracle Database functionality, so you are able to set your initialization parameters with the `ALTER SYSTEM` statement. As an example, when connected to PDB1 we set the following:

```
SQL> alter system set optimizer_dynamic_sampling=8 scope=spfile;
System altered.
```

And we can query it with `SHOW SPPARAMETER`:

```
SQL> show spparameter optimizer_dynamic_sampling

SID      NAME                          TYPE        VALUE
-------- ----------------------------- ----------- -----
*        optimizer_dynamic_sampling    integer     8
```

This is similar to what we know in non-CDBs, but here the settings are not stored in the instance SPFILE. In multitenant, we have the CDB$ROOT to store information about the containers, and this resides in the PDB_SPFILE$ system table:

```
SQL> show con_name
CON_NAME
------------------------------
CDB$ROOT
SQL> select db_uniq_name,pdb_uid,sid,name,value$ from pdb_spfile$;
DB_UNIQ    PDB_UID SID NAME                        VALUE$
------- ---------- --- --------------------------- -----------
CDB_GVA 1342292939 *   db_securefile               'PREFERRED'
CDB_GVA 4064530355 *   db_securefile               'PREFERRED'
CDB_GVA 1532325936 *   db_securefile               'PREFERRED'
CDB_GVA 4064530355 *   optimizer_dynamic_sampling  8
```

You don’t see the CON_ID here, but only the PDB_UID, which we can use to match to our PDB name with V$PDBS:

```
SQL> select con_id,con_uid,name from v$pdbs;
    CON_ID    CON_UID NAME
---------- ---------- ----------
         2 1342292939 PDB$SEED
         4 4064530355 PDB1
         5 1532325936 PDB2
```

In this example, our SPFILE parameter for optimizer_dynamic_sampling in PDB1 is actually stored in the root.

When you are connected to a PDB, you can query the SPFILE parameters with `show spparameter` or with a query on V$SPPARAMETER. You can also dump all parameters defined into a text file, because the `CREATE PFILE` syntax is supported:

```
SQL> create pfile='/tmp/init.ora' from spfile;
File created.

SQL> host cat /tmp/init.ora
*.db_securefile='PREFERRED'
*.optimizer_dynamic_sampling=8
```

Keep in mind, however, that this text file cannot be used to start the instance, and the name PFILE is there only for syntax compatibility. This is because we are connected to a PDB, and an instance is not at that level.

Another possibility is to generate the describe file, the same XML file that is used by the unplug/plug in operations we will cover in Chapter 9, because it houses information (parameters) about the container that are not stored within the container itself.

```
SQL> exec dbms_pdb.describe('/tmp/pdb1.xml','PDB1');
PL/SQL procedure successfully completed.
SQL> host grep "<spfile>" /tmp/pdb1.xml
      <spfile>*.db_securefile='PREFERRED'</spfile>
      <spfile>*.optimizer_dynamic_sampling=8</spfile>
```

These are persisted across operations such as PDB close, instance startup, unplug/plug in, and cloning. The PDB_SPFILE$ in PDBs is usually empty, except when the PDB is unplugged, in which case it serves as a backup for the XML file.

Both the CDB SPFILE file and the PDB SPFILE table can store a comment. It is highly recommended to add the `COMMENT` clause to any parameter changed permanently; you have 255 characters to document the date and the reason for the change.

SCOPE=MEMORY

Setting a parameter with `SCOPE=SPFILE` changes the persistent setting, the value that is displayed by `show spparameter`, for future startup only. The `SCOPE=MEMORY` clause, on the other hand, changes the current value of the parameter until the next startup. Without the `SCOPE` clause the parameter is set both in MEMORY and in the SPFILE. This behavior is the same in root and in PDBs, except that shutdown/startup means close/open in the PDB context.

Alter System Reset

You can use `ALTER SYSTEM RESET` to remove the persistent setting for a parameter, which means that when the database is next opened it will take the CDB memory value as the default. Unfortunately there is no reset with the `SCOPE=MEMORY` option, so either you set the value or you reset it in the SPFILE and close/open the PDB.

ISPDB_MODIFIABLE

The `V$PARAMETER` shows all parameters with their names, descriptions, and the current values for your session. There are flags that identify the parameters that can be changed only by restarting the CDB instance

(ISSYS_MODIFIABLE=FALSE), that can be changed for the instance without restart for future sessions (ISSYS_MODIFIABLE=IMMEDIATE), and that can be changed for current sessions as well (ISSYS_MODIFIABLE=DEFERRED). In 12.2, there is also a flag to identify the parameters that can be changed in a PDB without close/open (ISPDB_MODIFIABLE=YES), and these are a subset of the ISSYS_MODIFIABLE group.

In short, parameters relating to the instance itself, such as all memory sizing options, are not PDB-modifiable. However, a large number of parameters can be set by the session, and they are PDB-modifiable, with this value adopted as the default for new sessions. In addition, some parameters require an instance restart, which necessitates closing all PDBs.

Container=ALL

By default, a parameter is set for your current container (CONTAINER=CURRENT). However, when you are in the CDB$ROOT, you can change a parameter for all PDBs by adding the CONTAINER=ALL clause. The goal is not to enact a parameter change, one by one, in every PDB. Instead, when you want to use the same parameter for all containers, you can set it in CDB$ROOT and let it unset within the PDBs. This command can be used to change the value for all containers immediately, without restarting them, as well as to modify the CDB$ROOT SPFILE for future startup. Note that when the command is combined with SCOPE=SPFILE, this makes the parameter's value the default for the whole instance. For example, from CDB$ROOT, we set the temporary undo to true:

```
SQL> alter session set container=cdb$root;
Session altered.
SQL> alter system set temp_undo_enabled=true container=all scope=both;
System altered.
```

Because of the scope=both, it is changed in memory and also in the SPFILE:

```
SQL> show spparameter temp_undo_enabled
SID      NAME                          TYPE        VALUE
-------- ----------------------------- ----------- -------------------------
*        temp_undo_enabled             boolean     TRUE

SQL> show parameter temp_undo_enabled
NAME                                   TYPE        VALUE
-------------------------------------- ----------- ----------------------------
temp_undo_enabled                      boolean     TRUE
```

Then in all PDBs, it is changed in memory, but not actually stored in PDB_SPFILE$:

```
SQL> alter session set container=pdb1;
Session altered.
SQL> show spparameter temp_undo_enabled
SID      NAME                          TYPE        VALUE
------- ----------------------------- ----------- -------------------------
*        temp_undo_enabled             Boolean

SQL> show parameter temp_undo_enabled
NAME                                   TYPE        VALUE
-------------------------------------- ----------- ----------------------------
temp_undo_enabled                      boolean     TRUE
```

On reflection, this actually makes sense, because there is no need to store a parameter for all containers when the default inherited from the CDB is correct.

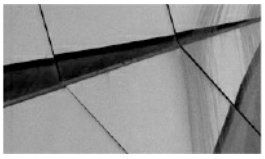
DB_UNIQUE_NAME

In our query on PDB_SPFILE$, we saw a DB_UNIQUE_NAME column with value CDB_GVA, which is the database unique name for our CDB. The parameters that are set in a PDB with `SCOPE=SPFILE` are valid only for one CDB database unique name. We will cover Data Guard in Chapter 11, but you should already know that the DB_UNIQUE_NAME is the correct way to distinguish a primary from its standby database(s). Having the parameter identified by the CDB unique name means that a parameter changed for a

PDB in the primary will not be used on the standby database. This makes sense, especially in the context of Active Data Guard, where the standby can be used for reporting, and you may also want to define different optimizer parameters for the online transaction processing (OLTP) on the primary, and the reporting on the standby. In our preceding example, we set the dynamic sampling to 8, but now we want the default value on the primary CDB_GVA (so we reset it) and the level 8 on the standby CDB_BSL. This can be achieved with the DB_UNIQUE_NAME clause:

```
SQL> alter session set container=pdb1;
Session altered.
SQL> alter system reset optimizer_dynamic_sampling;
System altered.
SQL> alter system set optimizer_dynamic_sampling=8 db_unique_name='CDB_BSL' scope=spfile;
System altered.
```

At the time of writing, there are some limitations with this feature. For example, we cannot reset a parameter with the DB_UNIQUE_NAME clause. A workaround for now is probably to remove the row from pdb_spfile$. Another point is that the DB_UNIQUE_NAME provided is case-sensitive, though we have opened enhancement requests for this.



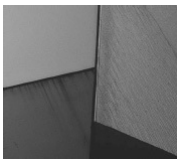
CAUTION

The SPFILE parameters are related to a site and not to a service. In case of switchover or failover, some services will run on another site. For example, the OLTP will failover to the site where the optimizer parameter is defined for reporting. In view of this, particularly when it comes to maintaining different optimizer parameters, it's probably a good idea to set these from a logon trigger that depends on the service name, rather than to a per-site setting.

Summary

In this chapter we covered the basic administration of PDBs after you have a CDB. You can create PDBs, patch them, upgrade them, parameterize them, and drop them. Most of these operations are the same ones that every DBA performs daily, but they are adapted to incorporate two levels in multitenant: CDB and PDB. You will see that a number of new operations are made possible with the multitenant architecture, mostly in Part III, where we delve into recovery and data movement. You need to know and understand the basic operations, and all administration procedures and scripts that are adapted, to function in the new multitenant paradigm.

This chapter focused primarily on the DBA who operates at the CDB level, who is often connected to root, and who invokes `SET CONTAINER` when needed. In the next chapter, you will see how to make use of those PDBs, which provide a service to which you can connect directly.



CHAPTER 5

Networking and Services

Before we dive into Oracle networking, related specifically to the Oracle Database *12c* multitenant option, we need to take a look again at some of the core concepts. We will not go into detail here on Oracle networking features and functionality, because that is a book on its own; for more in-depth information, we recommend you review the "Oracle Net Services Administrator Guide." In this chapter we will cover the new listener registration (LREG) background process, including brief mention of the new Oracle Database 12*c* multithreaded option, before diving into the details of services and Oracle Database 12*c* Multitenant.

Oracle Net

Oracle Net is the software component that provides the network layer for communication between the client and the Oracle database instances. It forms a communication channel between the client and database instance once a connection has been established.

Oracle Net Services consists of a number of components, such as Oracle Net, that facilitate connectivity between distributed environments. It also includes the Oracle Net Listener, the Connection Manager (CMAN), and two key configuration utilities: Oracle Net Configuration Assistant (NETCA) and Oracle Net Manager (netmgr).

The Oracle Net Listener

The listener is one of the most important components to consider when

establishing a new Oracle Database environment, yet many DBAs treat it lightly, paying it only minimal attention. Perhaps this is because, in many cases, once the listener is configured you can forget about it. But in Oracle Database 12*c*, with the introduction of multitenant, there is more to the listener than meets the eye.

Let's begin with a recap of the basics of a database connection in the traditional model. When a database starts it will, by default, register with the listener, providing it with one or more service names. But here is where it can get confusing:

- A service might identify more than one instance (in the case of Oracle Real Application Clusters [RAC]).
- A single instance can be registered by more than one listener.
- The database may register more than one service with the listener.

When looking at the process of a client connecting to a database, we can see there are three high-level steps performed, as shown in Figure 5-1. First, the client initiates communication by requesting a connection from the listener to a particular service. Then the listener identifies the appropriate service and passes the details to the client, after which it will make a direct connection to it. Once the connection is established, it is important to note that the listener is not involved in any way in the communication going forward.

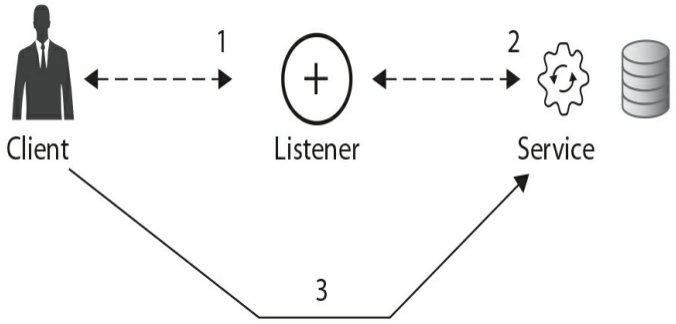


FIGURE 5-1. *Basic connection overview*

This brings us to an interesting change in Oracle Database12*c*: the new listener registration (LREG) background process.

The LREG Process

Instance registration with the listener in Oracle 12*c* behaves a bit differently from that of previous versions. In 12*c*, the LREG is a new background process that will register the database instance with the listener, which was a task performed in earlier versions by the Process Monitor PMON) process. On UNIX, this process is easily identified at the OS level, because “lreg” is included in the visible process name. This can be seen in the following code block:

```
# ps -ef|grep lreg|grep -v grep
oracle    6865     1  0 Feb14 ?        00:00:14 asm_lreg_+ASM
oracle    6996     1  0 Feb14 ?        00:00:20 ora_lreg_CDB1
```

The format of the process is ora_lreg_*SID*.

Note that this background process is classified as critical for the database instance, and, if it is terminated, the Oracle Database instance will actually abort. The code blocks that follow show the output of the alert log when this background process is killed:

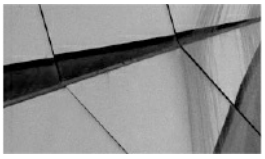
```
# ps -ef|grep lreg|grep -v grep
oracle    6865     1  0 Feb14 ?        00:00:14 asm_lreg_+ASM
oracle    6996     1  0 Feb14 ?        00:00:20 ora_lreg_CDB1
oracle@linux3[/home/oracle]: kill -9 6996
```

From the alert log, notice the following messages when the LREG is terminated:

```
…
Instance Critical Process (pid: 25, ospid: 6996, LREG) died unexpectedly
PMON (ospid: 6938): terminating the instance due to error 500
System state dump requested by (instance=1, osid=6938 (PMON)), summary=[abnormal
instance termination].
System State dumped to trace file /u01/app/oracle/diag/rdbms/cdb1/CDB1/trace/CDB1_
diag_6962_20160220204645.trc
…
Instance terminated by PMON, pid = 6938
USER (ospid: 7185): terminating the instance
Instance terminated by USER, pid = 7185
…
```

The LREG background process also writes out information regarding service updates in the listener log. These include service_update, service_register, and service_died messages; here's an example:

```
20-FEB-2016 20:57:07 * service_update * CDB1 * 0
2016-02-20 20:57:35.907000 +13:00
```



NOTE

When we used the strace utility to view additional detail on the LREG process (though it is not recommended that you use this utility on background processes in production systems), we observed a review of the load average (/proc/loadavg) approximately every 33 seconds.

When running in the new multithreaded mode, LREG does not run as a process on its own, but is run as a thread. Following is an excerpt from a Linux system that shows the LREG process running as a thread (thread ID 4696) under the process ID 4658:

```
SQL> select pid, spid, stid, pname, username, program from v$process where
pname='LREG';

   PID SPID   STID   PNAME USERNAME     PROGRAM
------ ------ ------ ----- ------------ --------------------------------
    29 4658   4696   LREG  oracle       oracle@linux2.orademo.net (LREG)
```

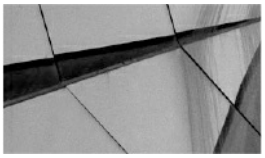
From the preceding example, we now have the operating system process ID 4658. If we review the process listing for the database CDB2, we can easily identify this process:

```
oracle@linux2[/home/oracle]: ps -ef|grep CDB2
oracle    4656     1  0 22:06 ?        00:00:00 ora_pmon_CDB2
oracle    4658     1  0 22:06 ?        00:00:01 ora_u002_CDB2
oracle    4662     1  0 22:06 ?        00:00:00 ora_psp0_CDB2
oracle    4664     1  2 22:06 ?        00:00:21 ora_vktm_CDB2
oracle    4671     1 12 22:06 ?        00:02:03 ora_u005_CDB2
oracle    4675     1  0 22:06 ?        00:00:00 ora_ofsd_CDB2
oracle    4684     1  0 22:06 ?        00:00:00 ora_dbw0_CDB2
oracle    4686     1  0 22:06 ?        00:00:02 ora_lgwr_CDB2
oracle   13247 12521  0 22:23 pts/1    00:00:00 grep CDB2
```

Taking process 4658, we can again use the `ps` command to list the threads associated with this:

```
oracle@linux2[/home/oracle]: ps -eLo "pid tid comm args" |grep 4658
 4658  4658 ora_scmn_cdb2   ora_u002_CDB2
 4658  4659 oracle          ora_u002_CDB2
 4658  4660 ora_clmn_cdb2   ora_u002_CDB2
 4658  4667 ora_gen0_cdb2   ora_u002_CDB2
 4658  4668 ora_mman_cdb2   ora_u002_CDB2
 4658  4678 ora_dbrm_cdb2   ora_u002_CDB2
 4658  4681 ora_pman_cdb2   ora_u002_CDB2
 4658  4689 ora_ckpt_cdb2   ora_u002_CDB2
 4658  4691 ora_smon_cdb2   ora_u002_CDB2
 4658  4696 ora_lreg_cdb2   ora_u002_CDB2
12539 12539 grep            grep 4658
```

By reviewing the output of the `ps` command, we can identify the LREG thread.



NOTE

The new multithreaded option for UNIX-based systems was introduced in 12.1.0.1. To enable the multithreaded mode, the database parameter `THREADED_EXECUTION` must be set to TRUE and the database restarted. The multithreaded model enables Oracle Database processes to execute as operating system threads in separate address spaces. Some background processes run as processes containing only a single thread, but the other Oracle processes run as threads within processes. Multithreaded brings some interesting new options and changes.

Networking: Multithreaded and Multitenant

One of the key advantages of multitenant is consolidation: multiple database (non-CDB) environments can be configured to run as pluggable databases (PDBs) in a single consolidated container database. This enables you to get

the most out of your valuable resources by not having to allocate unnecessary resources. For example, instead of running ten databases on one server, each with its own System Global Area (SGA) and background processes, you could consolidate into a single container database with PDBs, which would mean one SGA and one set of background processes. And the multithreaded mode takes this one step further by reducing the amount of processes on the system, with the possibility of improved performance and scalability.

When you're configuring the multithreaded option (by setting `THREADED_EXECUTION=TRUE` and restarting the CDB), consider that you will also need to make a change in the listener to allow threads, rather than processes, to be spawned. To enable this, set the parameter `DEDICATED_THROUGH_BROKER_<listener_name>=ON` in the listener.ora configuration file, and then restart the listener so that the changes take effect. As a result, when the listener receives a client connection request, it will pass this onto a connection broker (Nnnn), which will then verify the authentication, and a new thread will be spawned in an existing process.

The total number and type of connection brokers can be set using the `CONNECTION_BROKERS` database parameter. By default, two brokers are configured—one of type DEDICATED and one of type EMON. To view the status of the connection brokers (which are also spawned threads), you can review the status of the services using the listener control (`lsnrctl`) command, as follows:

```
# lsnrctl service listener_pdb
LSNRCTL for Linux: Version 12.2.0.0.2 - Beta on 27-FEB-2016 15:37:19
Copyright (c) 1991, 2015, Oracle.  All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=linux2.orademo.net)
(PORT=1531)))
Services Summary...
…
Service "dpdb1.orademo.net" has 1 instance(s).
  Instance "CDB2", status READY, has 12 handler(s) for this service...
    Handler(s):
      "D000" established:0 refused:0 current:0 max:1022 state:ready
         DISPATCHER <machine: linux2.orademo.net, pid: 4634_4664>
         (ADDRESS=(PROTOCOL=tcp)(HOST=linux2.orademo.net)(PORT=23595))
      "N000" established:280 refused:0 state:ready
         CMON <machine: linux2.orademo.net, pid: 4634_4666>
         (ADDRESS=(PROTOCOL=tcp)(HOST=127.0.0.1)(PORT=46921))
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
      "N001" established:234 refused:0 state:ready
         CMON <machine: linux2.orademo.net, pid: 4634_15873>
         (ADDRESS=(PROTOCOL=tcp)(HOST=127.0.0.1)(PORT=57571))
…
The command completed successfully…
```

A quick way to see the thread and process details is to use the ps command:

```
# ps -eLo "pid tid comm args" |grep 4634 |egrep '4666|15873'
 4634  4666 ora_n000_cdb2   ora_u005_CDB2
 4634 15873 ora_n001_cdb2   ora_u005_CDB2
```

And looking further at V$PROCESS, we see the following: