An Oracle White Paper
June 2013

Oracle Multitenant

Executive Overview

Oracle Multitenant is a new option for Oracle Database 12*c* Enterprise Edition that helps customers reduce IT costs by simplifying consolidation, provisioning, upgrades, and more. It is supported by a new architecture that allows a container database to hold many pluggable databases. And it fully complements other options, including Oracle Real Application Clusters and Oracle Active Data Guard. An existing database can be simply adopted, with no change, as a pluggable database; and no changes are needed in the other tiers of the application. The benefits of Oracle Multitenant are brought by implementing a pure deployment choice. The following list calls out the most compelling examples.

- *High consolidation density*. The many pluggable databases in a single container database share its memory and background processes, letting you operate many more pluggable databases on a particular platform than you can single databases that use the old architecture. This is the same benefit that schema-based consolidation brings. But there are significant barriers to adopting schema-based consolidation, and it causes ongoing operating problems. The new architecture removes these adoption barriers and operating problems.
- *Rapid provisioning and cloning using SQL.* A pluggable database can be unplugged from one container database and plugged into another. Alternatively, you can clone one, within the same container database, or from one container database to another. These operations, together with creating a pluggable database, are done with new SQL commands and take just seconds. When the underlying filesystem supports thin provisioning, many terabytes can be cloned almost instantaneously simply by using the keyword *snapshot* in the SQL command.
- *New paradigms for rapid patching and upgrades*. The investment of time and effort to patch one container database results in patching all of its many pluggable databases. To patch a single pluggable database, you simply unplug/plug to a container database at a different Oracle Database software version.
- *Manage many databases as one*. By consolidating existing databases as pluggable databases, administrators can manage many databases as one. For example, tasks like backup and disaster recovery are performed at the container database level.
- *Dynamic between-pluggable database resource management*. Oracle Database 12*c* Resource Manager is extended with specific functionality to instantly control the competition between the pluggable databases within a container database.

This whitepaper explains the new Oracle Database architecture, the new functionality that this brings, and the benefits that ensue.

Disclaimer

How the whitepaper is structured

For readers who don't need a detailed technical understanding

You should read just these sections:

- *"Customer challenges addressed by Oracle Multitenant"* on *page 5*
- *"High-level description of Oracle Multitenant"* on *page 9*
- *"Summary"* on *page 48*

It would help to skim these sections:

- *"How to adopt a non-CDB as a PDB"* on *page 29*
- *"Per CDB choices versus per PDB choices"* on *page 39*
- *"Within-CDB, between-PDBs, resource management"* on *page 43*
- *"Lone-PDB in CDB versus non-CDB"* on *page 47*

For readers who want a full understanding

You should, of course, read the whole paper carefully from start to finish.

Section overview

"Customer challenges addressed by Oracle Multitenant" on *page 5* describes these three issues: striving to consolidate very many databases onto a single platform to maximize return on investment; provisioning databases; and patching the Oracle version of very many databases.

Then, *"High-level description of Oracle Multitenant"* on *page 9* introduces the terminology that characterizes the new architecture, in particular *multitenant container database* (abbreviated to CDB) and *pluggable database* (abbreviated to PDB), and describes, without explanation, the broad outlines the high-level functionality and how this addresses the customer challenges.

Then, in *"The static aspects of the multitenant architecture: the horizontally partitioned data dictionary and pluggability"* on *page 12*, we explain how the customer challenges flow from the pre-12.1[1] architecture for Oracle Database and how the new architecture brought by 12.1 removes the root cause. In short, it brings within-database virtualization so that the "super database", the container database, contains "sub databases", the pluggable databases. We will see why we use the term pluggable database for a customer system.

Next, in *"The operations on PDBs as entities: unplug/plug, clone, create, drop"* on *page 17*, we see how these operations, implemented by SQL statements in the context of the multitenant architecture, bring new paradigms for provisioning and patching the Oracle version.

1. We shall use 12.1 as a shorthand for Oracle Database 12*c*, 11.2 as a shorthand for Oracle Database 11*g*, and so on.

We are now ready to describe very simply, in *"How to adopt a non-CDB as a PDB"* on *page 29*, how a pre-12.1 database can be adopted as a PDB.

Then, in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*, we see why we now distinguish between common users and local users, and how to create a session — especially one that can never escape the PDB to which it connects. We learn, too, how the SGA, the redo log, and the undo tablespaces are *logically* rather than *physically* virtualized by tagging every structure with the identifier of the system to which it belongs.

With the understanding of the new architecture in place, we can now explain, in *"Per CDB choices versus per PDB choices"* on *page 39*, which degrees of freedom enjoyed by a pre-12.1 database are retained by a PDB and which few are inevitably traded out in pursuit of the consolidation goal.

Next, in *"Within-CDB, between-PDBs, resource management"* on *page 43*, follows the account of resource management between the PDBs within a CDB.

In *"Lone-PDB in CDB versus non-CDB"* on *page 47* we compare using a single PDB within a CDB dedicated for that purpose with using a database that has the pre-12.1 architecture when the aim is to host a single application backend. We shall see that the lone-PDB is in no way worse and brings several advantages.

We conclude the whitepaper with *"Summary"* on *page 48*.

There is one appendix:

- *"Appendix A: The treatment of the multitenant architecture in the Oracle Database Documentation Library"* on *page 49*.

Customer challenges addressed by Oracle Multitenant

These days, it is fairly common to find hundreds, or thousands, of databases scattered over almost as many machines, within customer sites of reasonable to large size. The associated expense has driven initiatives to bring lots of databases together, in other words to *consolidate*, to reduce these costs. Over the previous decades, database administrators have been spending more time than they want to, provisioning databases, patching the Oracle version of each of very many databases, planning, setting up, and managing backup and disaster recovery regimes, and otherwise managing each database as its own individual ongoing exercise.

Striving to achieve maximum consolidation density

Standardization is intrinsically beneficial — there is, quite simply, less to understand. It further allows many databases that conform to a particular standard to be consolidated onto a single platform that implements that standard.

Standardization reduces operating expense

The high cost of ownership to an organization that has very many databases is exacerbated by variation between these databases. The most obvious variables are the type of hardware and operating system, the choice of options, and the choice of the Oracle Database software version — characterized by the finest granularity of one-off patch. Other variables, such as how the scheduled backup regime is conducted, and how disaster recovery is catered for, play their part.

Many customers have discovered that the first step to managing complexity is to host as many databases as possible on each of as few as possible hardware platforms. Operating system virtualization seems at first to be an attractive expedient to allow the autonomy of management of each individual database that was enjoyed when each was on its own platform. However, it is exactly by minimizing the variation between databases that is allowed by operating system virtualization, that management cost savings are to be found. Therefore, customers who consolidate many databases onto few machines have realized that operating system virtualization is a solution to a problem that they don't have — and prefer, therefore, to use the native operating system without virtualization and, further, to deploy as few as possible *Oracle Home* installations; in other words, they *standardize* on hardware type, on operating system version and patch-level, and on the particular configurations of Oracle Database that they support.

Standardization brings further opportunities to reduce operating expense and capital expense

Customers who have come this far in pursuit of consolidation have realized that consolidation density is limited by the fact that each additional database brings a noticeable incremental requirement for memory and CPU, because each has its own SGA and set of background processes — multiplied, of course, when Oracle Real Application Clusters is deployed. At this point, they realize that the important metric is not the density of *databases* that a platform can support, but rather the density of *application backends* that it can support.

We will define the term *application backend* to mean the set of artifacts that implement, within a database, the persistence mechanism for a particular application. Inevitably, there exists a mechanical scheme, for every application, to install its application backend. This might consist entirely of SQL*Plus scripts, of a combination of these and Data Pump import files and commands, or of purpose-built client programs that issue the appropriate

SQL statements. But a mechanical scheme will surely exist. We use the term *artifact* to denote the effect of any change to the database made by an application backend's installation scheme. The most obvious examples are objects listed in *DBA_Objetcs* like tables, indexes, views, PL/SQL packages, and so on. But some artifacts, like users, roles, and tablespaces are not objects in this sense. Nor are the effects of granting privileges or roles to users or roles. And nor is data in application tables used as application metadata.

Historically, each application backend has been housed in its own dedicated database. However, many distinct application backends can be installed into the same database by running the installation scheme for each in turn. This approach is usually referred to as *schema-based consolidation*, and we shall use this term in this paper[2].Customers have shown that schema-based consolidation supports a noticeably bigger consolidation density, measured in application backends per platform, than is possible when each application backend is in its own database — hereinafter the *dedicated database model*. The consolidation density depends on properties of the application backends, like the size of database memory that it needs and the throughput that it must support. Moreover, by drastically reducing the number of databases, significant savings in management costs follow.

However, schema-based consolidation has some notorious disadvantages.

- *Name collision might prevent schema-based consolidation.* Some application backends are implemented entirely within a single schema. In such cases, the name of the schema can usually be chosen freely, needing only to be specified in a central client-side configuration file. Such application backends can be usually be trivially installed in the same database (but even so, there are some potential problems, as will be seen below). It is this use case, of course, that gave rise to the term schema-based consolidation.

  Other application backends are implemented in several schemas, which implies cross-schema references and therefore a dependency on the specific schema names that the design has adopted. Of course, it is possible that the same schema names will be used in different application backends. But the name of a schema must be unique within the database as a whole, and so application backends with colliding schema names cannot be installed in the same database unless at least one of them is changed. Such a change would usually be considered to be prohibitively expensive.

  Moreover, the names of other phenomena must also be unique within the database as a whole. Examples are roles, directories, editions, public synonyms, public database links, and tablespaces. Even considering application backends, each of which is implemented in a single schema, the names of these other phenomena might collide and prevent them from being installed in the same database.

- *Schema-based consolidation brings weak security.* The person who installs a particular application backend needs to authenticate as a database user with powerful privileges like *Create User*, *Alter User*, and *Drop User*. This person could then make changes to any other application backend in the same database, and could therefore read, or change, its confidential data. Moreover, a poorly designed application backend whose implementation uses more than one schema might rely on system privileges like *Select Any Table*, *Delete Any Table*, and so on. In such a case, for example, a SQL injection vulnerability might allow a run-time user of one application backend to read, or to change, another application backend's data.

2. We shall see later that schema-based consolidation is one kind of *within-database consolidation*, and we shall contrast this with the kind that the multitenant architecture brings: *PDB-based consolidation*.

An application backend might require that certain privileges are granted to *public*, for example *Execute* on *Sys.Utl_File*. On the other hand, another might specifically prohibit this privilege being granted to *public*. These two application backends could not coexist in the same database.

- *Per application backend point-in-time recovery is prohibitively difficult.* Point-in-time recovery is usually called for when an error in application code, introduced by an application patch, is discovered to have caused irreparable data corruption. Therefore, it is the single application backend, and not the whole database, that needs to be recovered to the time just before the bad application patch was applied. In a fortunate case, tablespace point-in-time recovery might be sufficient. But this can be difficult when the corrupt data spans several tablespaces. It is difficult, too, if an administrator dropped one or more mutually referencing tables by mistake.

- *Resource management between application backends is difficult.* It relies on the humanly imposed convention that one or several specific services will be used to start the sessions that access a particular application backend, and that no service will be used for more than one application backend. In other words, the distinction between each different application backend is known only by the human administrator; the database has no knowledge of these distinctions.

- *Patching the Oracle version for a single application backend is not possible.* If one application backend needs a patch to the Oracle Database software version, this cannot be done without affecting all the other application backends.The only alternative is to use Data Pump to move the application backend that needs the patch to a different database where it has already been applied. However, database artifacts of some kinds, for example XML schemas, cannot be moved using Data Pump.

- *Cloning a single application backend is difficult.* Data Pump is the only option.

Notwithstanding the significant drawbacks of schema-based consolidation, many customers have adopted the practice and have demonstrated significant increase in return on investment. The reason, of course, is that the savings in capital expenditure brought by the high consolidation density, and the savings in operating expenditure brought by having fewer databases to manage, dominate the costs brought by the disadvantages described above.

Provisioning of databases

Database administrators have, over the years, needed to devote significant time, on most typical working days, to creating new databases, to moving existing databases from machine to machine, and to creating maximally current clones of existing databases for various purposes of development, testing, and problem diagnosis. We shall denote this class of chore by the term *provisioning*.

Patching and upgrading the Oracle Database software version

While not as frequent a chore as provisioning, applying one-off patches, bundled patches, patch set updates (critical or ordinary) and patch sets to existing databases, and upgrading them from a point-one release to a point-two release, or a point-two release to the next point-one release, is stressful and time-consuming. We shall refer to this chore as *patching the Oracle version*. We shall make frequent use of this term; it's too long-winded always to distinguish between patching and upgrading, or to call out that we mean the version of the

Oracle Database software, including the executable binaries and the Oracle system within the database, rather than the version of the application backend.

High-level description of Oracle Multitenant

Oracle Database 12*c* supports a new architecture that lets you have many "sub databases" inside a single "super database". From now on, we shall use the official terminology. The "super database" is the *multitenant container database* — abbreviated as *CDB*; and the "sub database" is the *pluggable database* — abbreviated as *PDB*. In other words, the new architecture lets you have many PDBs inside a single CDB. (In 12.1, the maximum number is 252.) We shall refer to the new architecture as the *multitenant architecture*.

We clearly now need a term for the old kind of database, the only kind of database that was supported through Oracle Database 11*g*. We shall call this a *non-CDB*; and we shall refer to the old architecture as the *non-CDB architecture*.

Oracle Database 12*c* Release 1 supports both the new multitenant architecture and the old non-CDB architecture. In other words, you can certainly upgrade a pre-12.1 database, which necessarily is a non-CDB, to a 12.1 non-CDB and continue to operate it as such. But, if you choose, you can adopt the 12.1 non-CDB into a CDB as a PDB[3].

From the point of view of the client connecting via Oracle Net, the PDB *is* the database. A PDB is fully compatible with a non-CDB. We shall refer to this from now on as the *PDB/non-CDB compatibility guarantee*[4]. In other words, the installation scheme for an application backend that ran without error against a non-CDB will run, *with no change*, and without error, in a PDB and will produce the same result. And the run-time behavior of client code that connects to the PDB holding the application backend will be identical to that of client code that connected to the non-CDB holding this application backend. It is intended that a PDB be used to hold a *single* application backend. In this way, the PDB provides a direct, declarative means to contain an application backend so that the Oracle system knows explicitly which artifacts belong to which application backend. In contrast, when schema-based consolidation is used within a non-CDB, the Oracle system has no information about where the various artifacts belong. We shall see that each foreground process sees, at a moment, just a single PDB; indeed, in the simplest usage model, a database user, defined within a PDB, is able to create only sessions that see *only* that PDB — and only, therefore, the artifacts that belong to a single application backend[5].

Recognition of the PDB/non-CDB compatibility guarantee allows very many questions to be answered by assertion: if the answer were not "yes", then the principle would not have been honored. Tests, conducted by Oracle Corporation engineers throughout the whole of the 12.1 development cycle have proved that client code connecting with Oracle Net cannot distinguish between a PDB and a non-CDB. Here are some examples:

- *Can two PDBs in same CDB each have a user called Scott?* Yes, because two non-CDBs can each have a user called *Scott*. By extension, two PDBs in same CDB can each have: a role with the same name; a directory with the same name; an edition with the same name; a public synonym with the same name; a public database link with the same name; and a tablespace

3. The details are explained in *"How to adopt a non-CDB as a PDB"* on *page 29*.

4. A caveat is made about the PDB/non-CDB compatibility guarantee in the section *"The ORA-65040 error"* on *page 42*.

5. This is explained in detail in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*.

with the same name. Moreover, in one PDB you can grant *Execute* on *Sys.Utl_File* to *public* while in another you make no such grant — a grant to *public* in a PDB is seen only in that PDB. Similarly, an *Any* privilege can be exercised only in the PDB where it was granted.

In other words, a PDB defines a global namespace, just as a non-CDB does. And it contains the effects of privileges, just as a non-CDB does.

- *Can you create a database link between two PDBs?* Yes, because you can do that between two non-CDBs. A database link is created by executing a SQL statement. Therefore, such a statement must produce the same result in a PDB as it does in a non-CDB. By extension, you can create a database link from a PDB to a non-CDB, or from a non-CDB to a PDB — and these can cross a difference in Oracle Database software version in the same way as is supported between two non-CDBs.

- *Can you set up GoldenGate replication between two PDBs?* Yes — and by extension, you can set it up between a PDB and a non-CDB, crossing a difference in Oracle Database software version. (The functionality is brought by the first Oracle GoldenGate version that supports 12.1[6].)

From the point of view of the operating system, it is the CDB that is the database. Each RAC instance opens the CDB as a whole, and each SGA can contain data blocks and library cache structures like child cursors from each of the PDBs contained in the CDB[7]. (This is an indirect way of stating that, of course, the multitenant architecture is fully interoperable with Oracle Real Application Clusters.)

We see that the multitenant architecture supports, therefore, a new model for within-database consolidation: PDB-based consolidation.

The population of the SGA in the PDB-based consolidation model is directly comparable to its population in the schema-based consolidation model. There, too, each RAC instance opens the non-CDB, and each SGA can contain data blocks and library cache structures from each of the application backends consolidated into the non-CDB. It follows therefore that it is the CDB that is at a particular Oracle Database patchset level. The PDBs it contains inherit this, just as do the schemas in a non-CDB. It is no more meaningful to ask if two PDBs in the same CDB can be at different Oracle Database patchset levels than it is to ask if the *Scott* and *Blake* schemas can be at different Oracle Database patchset levels in the same non-CDB.

A suitably privileged database user allows a person who knows its password to start a session that can see information from the data dictionary views and the performance views across all the PDBs that the CDB contains[8]. In other words, a single system image is available via SQL — and therefore via tools like SQL Developer and Enterprise Manager. In fact, these tools are extended to expose all the new functionality brought by the multitenant architecture.

6. Engineering work was required because, in the multitenant architecture, each redo log entries is annotated with the identifier of the container where it originated. See *"Data Guard, RMAN backup, redo, and undo"* on *page 39*.

7. We shall see in *"The SGA is logically virtualized"* on *page 34* that each PDB can have a different *Open_Mode* in each RAC instance.

8. The full understanding of this depends on notions explained later, notably the *root* (in *"The static aspects of the multitenant architecture: the horizontally partitioned data dictionary and pluggability"* on *page 12*), the *common user* (in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*), and *container_data views* (in *"Data dictionary views and performance views"* on *page 36*).

The multitenant architecture extends Resource Manager to allow a CDB-level plan to manage the competition of resources between the PDBs.

Oracle Active Data Guard is conducted at CDB-level, as is the regime of scheduled RMAN backup. Point-in-time-recovery is supported at PDB-level.

Finally, you can unplug a PDB from one CDB and plug it into another CDB. It is this, of course, that gives the pluggable database its name. You can also create a new PDB as a clone of an existing one. When the underlying filesystem supports thin provisioning, many terabytes can be cloned almost instantaneously. All the operations on PDBs as opaque entities — creating a brand new one, unplugging one, plugging one in, cloning one, and dropping one — are exposed as SQL statements. You request cloning using thin provisioning simply by using the keyword *snapshot* in the SQL command. Unplug/plug is implemented by a natural extension of the transportable tablespace technology, and is possible because of within-database virtualization that characterizes the multitenant architecture. Unplug/plug is even supported across a difference in the Oracle Database software version[9].

The new multitenant architecture is available in Standard Edition, in Standard Edition One, and in Enterprise Edition. However, without licensing the Oracle Multitenant option, you are limited to a maximum of one PDB[10]. The significance of this model is discussed in *"Lone-PDB in CDB versus non-CDB"* on *page 47*.

9. The detail of all this is explained in *"The static aspects of the multitenant architecture: the horizontally partitioned data dictionary and pluggability"* on *page 12*.

10. The Oracle Database 12*c* Licensing Guide sets out these terms formally.

The static aspects of the multitenant architecture: the horizontally partitioned data dictionary and pluggability

Customers who have implemented schema-based consolidation have, in effect, made an attempt to implement within-database virtualization without any intrinsic support from Oracle Database. This section attributes the difficulties of this approach to the historical architecture of the data dictionary. It then introduces the basic notion of the multitenant architecture's intrinsic virtualization: the horizontally partitioned data dictionary.

Tables: the ultimate logical reality

A quiesced Oracle Database consists only of its files. Besides the bootstrap files (the control file, *spfile*, and so on), the overwhelming bulk is the files that implement the tablespaces. Tablespaces, in turn, contain only tables and various structures to speed up access and to ensure their recoverability. It is the tables, of course, that are a tablespace's *raison d'être*. Other database artifacts, of all kinds, like constraints, views, PL/SQL objects, and so on, are ultimately represented as rows in tables. Tables hold only three kinds of data: metadata that describes the Oracle system, metadata that describes customer-created application backends, and quota-consuming data in an application backend's tables.

The non-CDB architecture's monolithic data dictionary

Through 11.2, both the metadata that describes the Oracle system, and the metadata that describes customer-created application backends, are represented in a single set of tables, known collectively as the data dictionary. The tables are famously implicitly maintained by the executables that implement the Oracle instance as side-effects of DDL statements. Customers are not allowed to do direct insert, update, and delete to these tables, and their structure is not documented so that the information they represent must be queried via the data dictionary views. Nevertheless, customers have come to know the names of these tables (*Obj$*, *Tab$*, *Col$*, and so on) and to understand their significance. These data dictionary tables are stored in dedicated tablespaces, most notably the *System* tablespace. *Figure 1* shows the full content of a freshly created non-CDB, before any customer artifacts have been created.

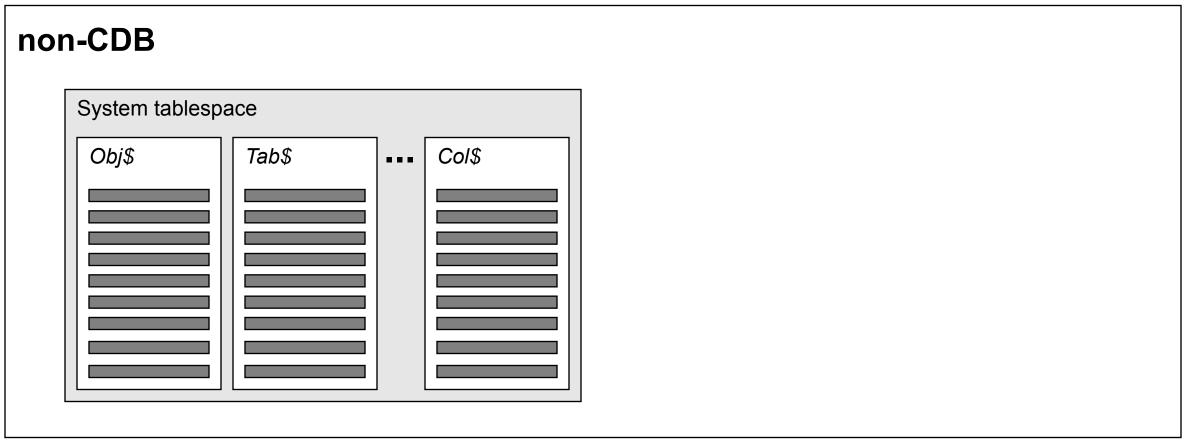

Figure 1 A pristine non-CDB

The first effect of installing an application backend into a non-CDB is that rows are inserted into the data dictionary tables to represent its metadata. Because the ultimate purpose of an application backend is to be the application's persistence mechanism, this metadata will, of course, include the description of application tables. Following conventional best practice, these tables will be stored on dedicated application data tablespaces. The application tables will soon contain quota-consuming data, so that total set of tables that define the non-CDB will be as shown in *Figure 2*.

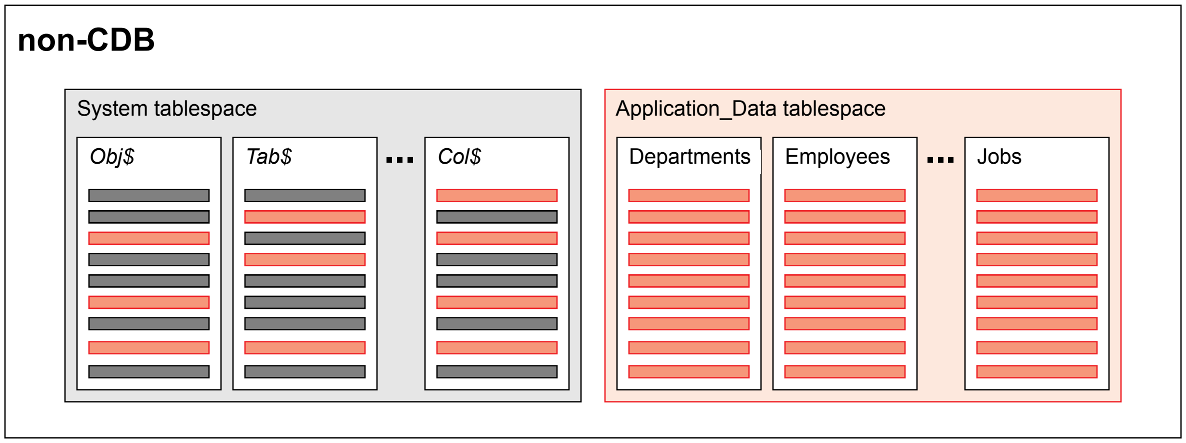

Figure 2   A non-CDB after installing application backend artifacts

In *Figure 2*, and in *Figure 3* and *Figure 4* in the next section, we use red to denote customer-created artifacts (both metadata rows and quota-consuming data rows) and black to denote the artifacts that represent the Oracle system (only metadata rows).

Of course, the namespaces for global artifacts like users, roles, editions, tablespaces, and the like are defined by the unique indexes on tables like *User$*, *Edition$*, *TS$*, and so on. Correspondingly, global artifacts like the grants made to users and to roles, including the *public* role, are represented in the *SysAuth$* and *ObjAuth$* tables. This immediately explains the name collisions that characterize the attempt to install two or more application backends into the same non-CDB, and the security problems that follow.

Figure 2 lets us understand, too, the leap forward that was made by the second generation Data Pump — by using transportable tablespaces. The first generation Data Pump used entirely slow-by-slow[11] construction, at export time, and replay, at import time, of SQL — both DDL statements and DML statements. The second generation replaced the use of DML statements by treating the set of tablespaces that hold the application's quota-consuming data (shown on the right, and entirely in red, in the diagram) as self-contained units holding the complete account of the table data and the indexes on them removing the need for using DML statements. Notably, at import time, a transportable tablespace is plugged in using only metadata operations that, therefore, are very fast.

Figure 2 also lets us understand the reason that there was, pre-12.1, no third generation Data Pump: because each data dictionary table contains both customer metadata and Oracle system metadata, the only recourse for moving this part of the application backend is still slow-by-slow DDL statements.

In summary, it is the non-CDB architecture's monolithic data dictionary that causes the difficulty of schema-based consolidation, the poor security of the resulting regime, and the

11. I attribute the phrase *slow-by-slow* to Oracle's Tom Kyte. You can find it in frequent use in *AskTom* and in his articles for Oracle Magazine.

limits to the mobility of an application backend. To put this another way, the non-CDB architecture's data dictionary is not virtualized.

The multitenant architecture's horizontally partitioned data dictionary

The solution, then, is obvious: the multitenant architecture's data dictionary *is* virtualized.

The approach in broad outline

The textbook approach to virtualization is to identify every phenomenon that implements a system and to tag it with the identifier of whom it belongs to. The simplest way, conceptually, to have done this would have been to add a new column to every data dictionary table to denote the application backend that it describes — and to use a special value for the rows that describe the Oracle system. But this approach would not have addressed the mobility of the application backend. Therefore, the logical tagging scheme was implemented physically, by horizontally partitioning the data dictionary. (Don't confuse the implementation of this with the implementation of Oracle Partitioning, as this feature is exposed for customers to use. The implementation is quite different. The data dictionary is partitioned in the ordinary, abstract, sense of the term.) *Figure 3* shows the scheme for a CDB holding a single application backend.

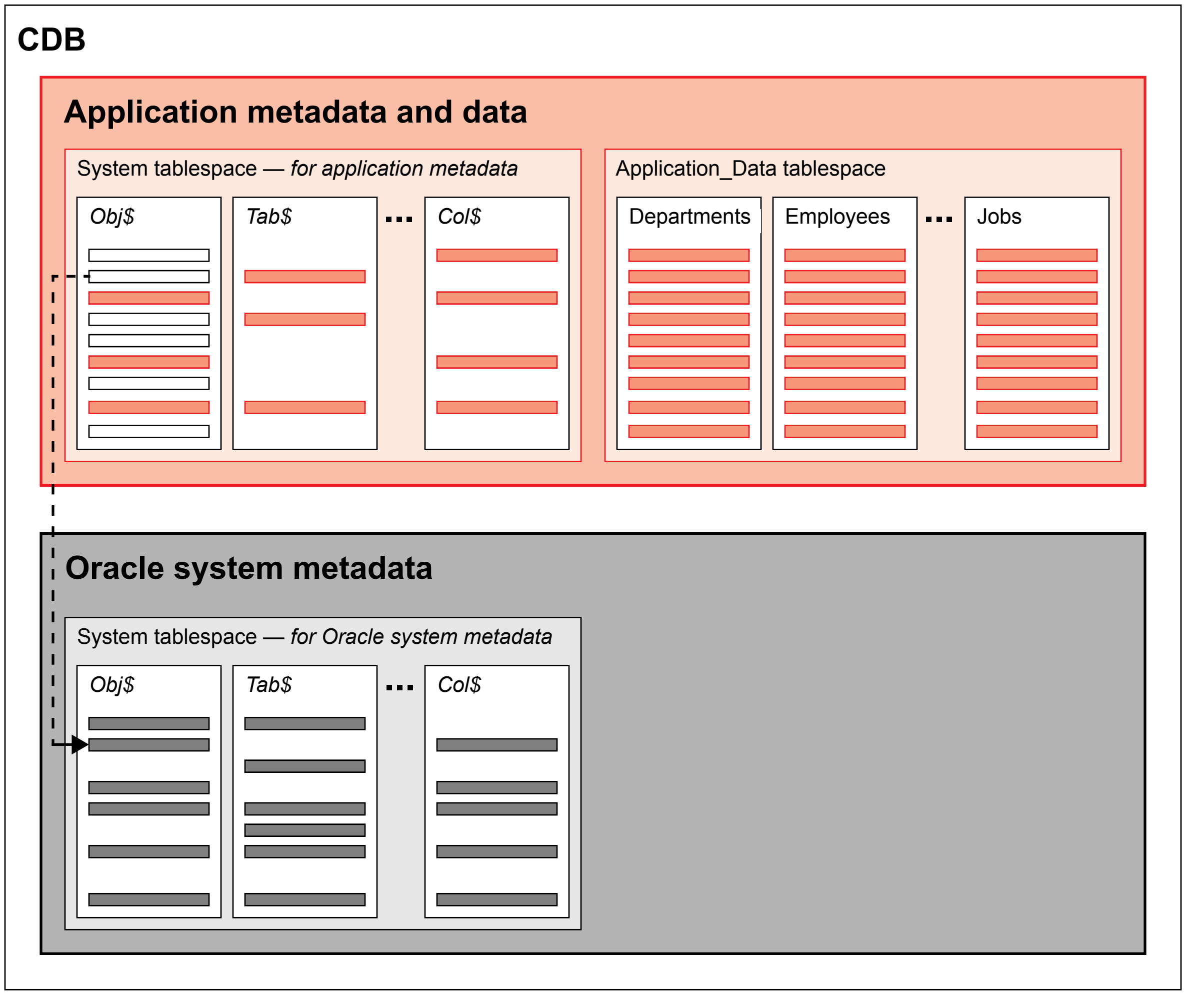
CDB
Application metadata and data
System tablespace — *for application metadata*
Obj$
Tab$
...
Col$
Application_Data tablespace
Departments
Employees
...
Jobs
Oracle system metadata
System tablespace — *for Oracle system metadata*
Obj$
Tab$
...
Col$

Figure 3 The multitenant architecture introduces the horizontally partitioned data dictionary

The "lower" half, as we choose to show it in this paper, holds the metadata for the Oracle system — and nothing else. And the "upper" half holds the metadata for the application backend — and nothing else. In other words, each data dictionary table now occurs twice, once for the Oracle system, and once for the application backend. Conceptually, every query against a data dictionary table is now a union between the two occurrences of that table.

Notice that each set of data dictionary tables is in its own tablespace(s) and therefore in its own datafiles. The paths of the datafiles must, of course, be unique within the filesystem. But the names of the tablespaces, and the segments they contain, need to be unique only within the data dictionary table in either the "upper" or the "lower" partition. It is natural, therefore to use the same names for the data dictionary tables, and the same names for the tablespaces that contain them, in both partitions.This point is explored in the next section.

The practical definition of PDB and *root*

The set of tablespaces, and therefore the set of datafiles, that implement the data dictionary tables that hold the metadata for the Oracle system is called the *root*. (Its name is *CDB$Root*.) And the set of tablespaces (and therefore their datafiles) that implement the data dictionary tables that hold the metadata for the application backend, together with the set of tablespaces that hold the application backend's quota-consuming data (and therefore the datafiles for all of these), is the *PDB*. The PDB is therefore a full database while the *root* is just a meta-database. Because the data dictionary is now virtualized, the same CDB can, of course, contain many PDBs, where each is, ultimately, its own set of self-contained datafiles.

This is shown in *Figure 4*.

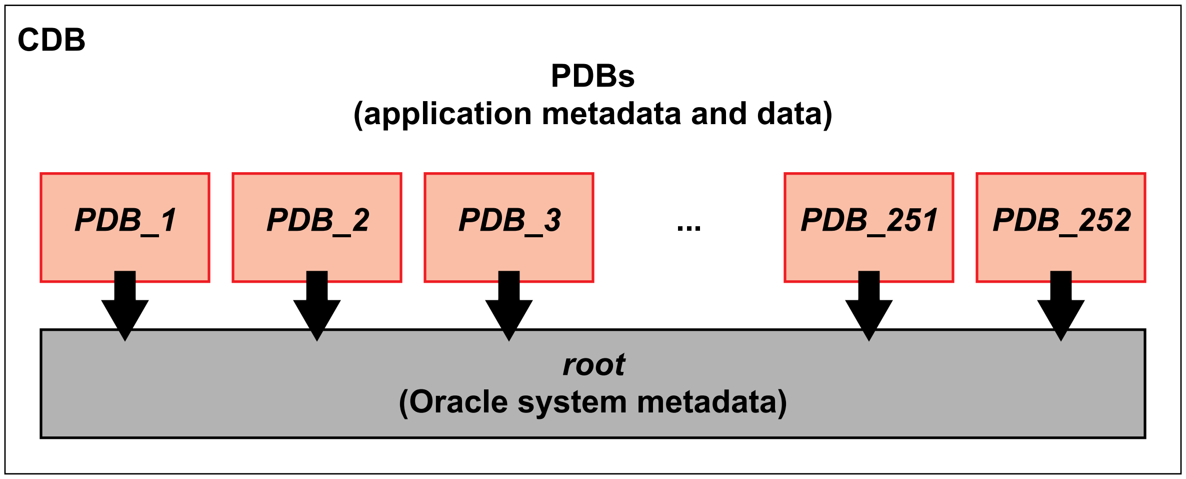

Figure 4 In the CDB architecture, up to 252 PDBs plug into the root

Significantly, then, each PDB defines a private namespace for all the phenomena, like users, roles, public synonyms, and so on, that in a non-CDB must be unique within the whole database. Each PDB in a particular CDB can have a user called *Scott*. It follows, then, that the result of granting privileges and roles, represented in the *SysAuth$* and *ObjAuth$* data dictionary tables are also contained within a PDB.

Though the *root* differs significantly from a PDB in that it never holds quota-consuming data, there is a strong similarity between these CDB components because each has a data dictionary and can be the "focus" of a foreground process. We therefore use the term *container* as the superclass term for the *root* or a PDB. A foreground process, and therefore a session, at every moment of its lifetime, has a uniquely defined *current container*.

The PDB is so-called because it can be unplugged from the *root* of one CDB and plugged into the *root* of another CDB. The big black arrow in *Figure 4* denotes this. Of course, the *root*'s metadata describes each PDB that is plugged into it. On unplug, the metadata for the unplugged PDB is deleted; and on plugging in, metadata is created to describe the plugged-in PDB. This is a natural extension of the pre-12.1 transportable tablespace technology. We can see, therefore, that unplug and plug of a PDB can be considered to be third generation Data Pump: both the quota-consuming data and the metadata that jointly completely represent an application backend can be moved as an opaque, self-contained unit, removing the need for constructing and replaying slow-by-slow DDL statements.

A sketch of the internals of the approach

You might wonder how data dictionary queries are implemented — but if you don't, you can skip this section. We don't refer again to this explanation, or rely on it for understanding what follows.

All queries are implemented, ultimately, by bringing blocks into the buffer cache. A block must therefore be addressed, and part of its address is the number of the file where it is stored. The multitenant architecture uses a relative scheme for datafile numbering based on the identity of session's current container. In normal operation, the current container is a PDB, and therefore the session has access only to blocks from that PDB's datafiles. This relative datafile numbering scheme is implemented at a low level in the layered modules that implement Oracle Database: the session's PDB identifier is passed via a side-channel to the low-level module; and the various higher-level APIs through which the SQL and PL/SQL compilation and execution subsystems communicate downwards to the data layer are unchanged with respect to the non-CDB architecture. This means, in turn, that very large amounts of the code that implements Oracle Database — the whole of SQL (including, therefore the optimizer), the whole of PL/SQL, and everything (like, for example, the Scheduler) that builds upon these — needed no change in 12.1 to accommodate the new multitenant architecture. It was therefore relatively easy to honor the PDB/non-CDB compatibility guarantee.

When a data dictionary query, issued in the context of a PDB, needs to access the rows that describe the Oracle system, the current container switches to the *root* and then switches back. The mechanism that signals this need, and that brings efficiency, is indicated by the dotted line arrow in *Figure 3* from a row in the PDB's *Obj$* table to a row in the *root*'s *Obj$* table. An Oracle-supplied object is represented fully in the *root* by the closure of all the various detail tables that refer directly or indirectly to a row in the *root*'s *Obj$* table. And it is represented sparsely in a PDB by just one row in its *Obj$* that points to the corresponding row in the *root*.

The operations on PDBs as entities: unplug/plug, clone, create, drop

We next see how these operations, implemented in the context of the multitenant architecture, bring new paradigms for provisioning and patching the Oracle version.

Unplug/plug from machine to machine

Figure 5 shows *PDB_1* plugged in to the *root* of *CDB_1* running on *machine 1*. By design, *PDB_1* holds exactly one application backend. The aim now is to move this application backend from *machine 1* to *machine 2*.

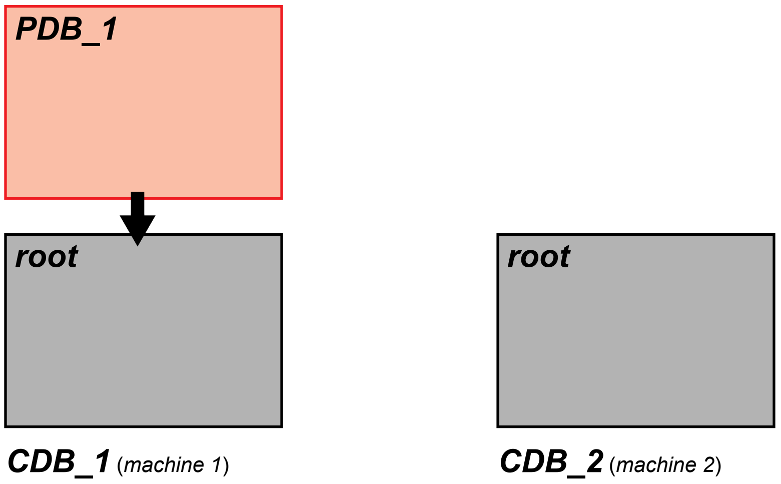

Figure 5 Unplug/plug between two machines: PDB_1 starts in CDB_1

While a PDB is contained in a CDB, the CDB holds metadata for it, like the name it has in this CDB and the paths to its datafiles. (Some of this is held in the *root*, and some is held in the CDB's control file. This distinction is unimportant for the present discussion.) On unplug, this metadata is written to an external manifest that accompanies the PDB's datafiles. Unplug is implemented as a single SQL statement, illustrated in *Code_1*.

```
-- Code_1
-- The PDB must be closed before unplugging it.
alter pluggable database PDB_1
unplug into '/u01/app/oracle/oradata/…/pdb_1.xml'
```

The semantics are "give the name of the PDB", "say that you want to unplug it", and "specify the path for the external manifest".

Correspondingly, plugging in is implemented as a single SQL statement, illustrated in *Code_2*. (The SQL statement for plugging in an unplugged PDB is a variant of the *create pluggable database* statement. The other variants are *clone PDB* and creating a brand-new PDB. The examples shown in *Code_2*, *Code_3 on page 25*, *Code_4 on page 25*, *Code_6 on page 25*, and *Code_7 on page 26* assume that Oracle-Managed Files is turned on.)

```
-- Code_2
-- The PDB must be opened after plugging it in.
create pluggable database PDB_2
using '/u01/app/oracle/oradata/…/pdb_1.xml'
path_prefix = '/u01/app/oracle/oradata/pdb_2_Dir'
```

The *path_prefix* clause is optional. It may be specified only with the *create pluggable database* SQL statement. It may not be changed for an existing PDB. It affects the outcome of the *create directory* SQL statement when it is issued from a PDB that was created using this clause. The string supplied in *create directory*'s *path* clause is appended to the string supplied in *create pluggable database*'s *path_prefix* clause to form the actual path that the new directory object will denote. This is an important measure for controlling between-PDB isolation.

Code_2 produces the result shown in *Figure 6*.

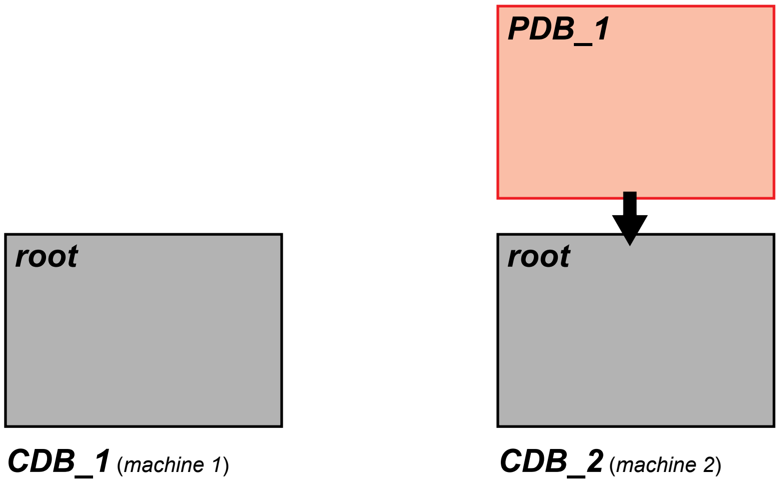

Figure 6 Unplug/plug between two machines: PDB_1 finishes up in CDB_2

The semantics are "say that you want a new PDB", "say that it is created by plugging in an unplugged PDB by specifying the path for the external manifest", "give the plugged in PDB a name in its new CDB, and ensure that any directory objects created within the PDB are constrained to the specified subtree in the filesystem that the CDB uses".

Unplug/plug across different operating systems, chipsets, or endiannesses

Similar considerations apply for cross-platform unplug/plug as apply for the cross-platform use of transportable tablespaces. When a customer-created tablespace is converted from one endianness to the other, using the RMAN *convert* command, it is only block headers that are changed (because these are encoded in an endianness-sensitive way). Data in columns in customer-created tables is left untouched. Most of these (we hope) will use primitive scalar datatypes like *varchar2* and *number* — and these are represented, endianness-insensitively, in network byte order. On the other hand (we hope, again) when columns of datatype *raw* or *blob* are used, then these will be used to hold data (like, for example, *.pdf* or *.jpg* files) that is consumed only by client-side code that knows how to interpret it. However, customers might encode data in an endianness-sensitive way, and in such a case it is their responsibility to know where this has been done and to apply their own conversion processing after a transportable tablespace containing such data has been plugged in across an endianness difference.

The discussion is made more complicated because an unplugged PDB contains data dictionary tables, and some of the columns in these encode information in an endianness-sensitive way. There is no supported way to handle the conversion of such columns automatically. This means, quite simply, that an unplugged PDB cannot be moved across an endianness difference.

Meanwhile (while the data dictionary remains endianness-sensitive), if the functional goal is cross-endianness unplug/plug, then the only viable approach is to use Data Pump to migrate the content into a new, empty PDB in the target CDB. If transportable tablespaces are used, then they must be processed using RMAN *convert*. If the data volume is small, it may be faster to use "classic" Data Pump without transportable tablespaces.

Moreover, moving on from the endianness discussion, PL/SQL native compilation might have been used; and if it has, then a particular data dictionary table will hold machine code that is sensitive to the chipset of the platform where it was compiled. If an unplugged PDB holding such machine code is plugged in to a CDB running on a platform with a different chipset from the platform where the CDB from which the PDB was unplugged was running, then it is no longer viable. In this case, the remedy is simple: all natively compiled PL/SQL units in the incoming PDB must be recompiled before the PDB can be made available for general use.

Unplug/plug for patching the Oracle version

The multitenant architecture supports plugging a PDB into a CDB whose Oracle Database software version differs from that of the CDB from which it was unplugged. (While 12.1.0.1 is the only available 12.1 version, this capability, of course, cannot be demonstrated.)

The least aggressive patches change only the Oracle binaries — in other words, by definition, this kind of patch makes no changes inside the datafiles. This implies that when the source and destination CDBs differ by this kind of patch, the incoming PDB needs no change whatsoever; it is sufficient (and of course necessary) only to check, using information in the PDB's manifest, that the Oracle Database software version difference is of this kind.

More aggressive patches make changes to the definition of the Oracle system in the data dictionary. (These typically make changes to the Oracle binaries as well.) But to deserve the name *patch* rather than *upgrade*, these data dictionary changes usually cause no knock-on invalidation. For example, if an Oracle-supplied package body is recompiled using a new implementation to fix a bug, then the package spec remains valid, and so, therefore, do all the objects in the closure of its dependants. With the advent of fine-grained dependency tracking in Oracle Database 11*g*, even if an Oracle-supplied package spec is changed by adding a new subprogram that doesn't overload an existing subprogram name, then there is no knock-on invalidation. When this kind of patch is applied to a non-CDB, then no customer-created artifact is changed. This implies that, even for this kind of patch, the incoming PDB needs no change[12]. We shall see, presently, that even for a patch that does cause some knock-on invalidation, the compensating work needed in the PDB will be small.

To exploit the new unplug/plug paradigm for patching the Oracle version most effectively, the source and destination CDBs should share a filesystem so that the PDB's datafiles can remain in place. This is shown in *Figure 7*.

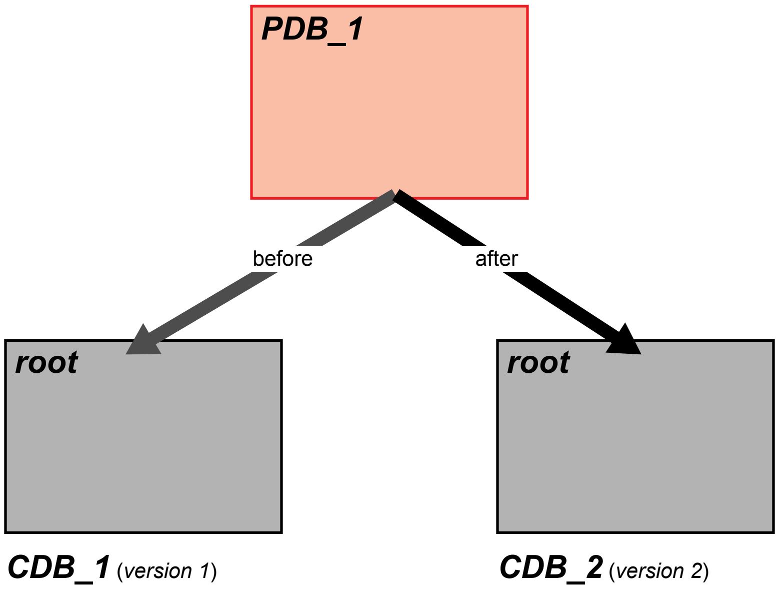

Figure 7 Unplug/plug between two CDBs across an Oracle Database software version difference while the datafiles remain in place on a shared filesystem

Because the unplug/plug operations affect only the metadata in the *root* that describes the PDB, they are very fast. The total time to close a PDB, unplug it from the source CDB, plug it into the destination CDB, and open it, when the datafiles remain in place, is measured in seconds on a typical machine — when there is no difference in the Oracle Database software version. This implies that the time will be the same when the difference is due only to changes in the Oracle binaries, and only marginally more when the difference is due to data dictionary changes that don't cause invalidation of any customer-created artifacts. Even for patches that do cause some invalidation among customer-created artifacts, fine-grained dependency tracking will minimize the amount of recompilation that needs to be done in the incoming PDB. Notice that our self-imposed compatibility rules guarantee that changes made by patching the Oracle version (or by upgrading it) will never leave a customer-created object, that was valid before these changes, in an irrevocably invalid state. Recompilation will always return these cascadingly-invalidated objects to validity.

12. Because of the fact that the implementation uses an *Obj$* row in the PDB to point to the corresponding *Obj$* row in the *root*, as described in *"A sketch of the internals of the approach"* on *page 16*, and because the row in the PDB encodes a signature of its partner row in the *root*, then the PDB row needs to be re-computed. Further, the *Dependency$* table in the PDB is fully populated. Its semantic content, for Oracle system artifacts, is the same as its partner row in the *root*, but it uses local values for the object numbers for the dependency parent and dependant. It records the details of fine-grained dependencies, like the timestamp of the compilation of the dependency parent, and an encoding of what attributes of this the dependant depends upon. You can guess that a PDB cannot express its dependencies on objects in the *root* using the object numbers there because these can be different in different CDBs. The connection between the *Obj$* row in the PDB and its partner in the *root* implements the mapping — and without this, then fast unplug/plug wouldn't work. Therefore, following plugging in the PDB, when source and destination CDBs differ by this kind of patch, it will be necessary also to recompute the details of fine-grained dependencies, and, possibly, some rows in the PDB's *Dependency$* table. This is a very quick operation.

Upgrades to the Oracle Database software version are allowed to make drastic data dictionary changes with knock-on effects for customer-created artifacts. (We use *upgrade* to denote going from a *point-one release* to a *point-two release* like 12.1 to 12.2, or from a *point-two release* to the next *point-one release*, like 12.2 to 13.1.) When this kind of upgrade is applied to a non-CDB, then the scripts attend first to the Oracle system and then to the customer-created artifacts. This implies that, when source and destination CDBs differ by this kind of upgrade, the incoming PDB needs to suffer the same scripted changes that, in a non-CDB, follow the scripted changes to the Oracle system. The time needed to make the required changes in the incoming PDB is, therefore, inevitably less than that taken, in a non-CDB, to do exactly this *as well as* first making the required changes to the Oracle system.

In other words, when an application backend is hosted in a PDB, the outage time for the application caused by patching the Oracle version, using the unplug/plug approach, is *always* less than that for applying the same patch to a non-CDB — and sometimes it can be measured in seconds, even for patches that cause limited cascading invalidation within the PDB. When the processing inside the incoming PDB is guaranteed not to cause invalidation among the customer-created artifacts, then this might be done implicitly. Otherwise, the user will be notified that an Oracle-supplied script must be run.

Notice that, further, the cost of setting up the patched destination CDB is highly amortized. It is paid once, but benefits patching the Oracle version for each of many PDBs, as each is unplugged/plugged across the Oracle Database software version difference.

The possibility of unplug/plug, when the destination CDB is at an *earlier* Oracle Database software version than the source CDB follows the exact same rules as for backing out a patch, or downgrading, in a non-CDB. (The "law" says the value of the *compatible* parameter must be equal to that of the earlier version. And practical reality adds the requirement that not one of the customer-created artifacts relies, semantically, on functionality that's unique to the newer version.) We shall use *backing out a patch to the Oracle version* as the collective term for both backing out a patch and downgrading the Oracle Database software version. The account above shows that the considerations for backing out a patch to the Oracle version are identical to those for patching the Oracle version.

Finally, in this section, it's useful to consider patching the Oracle version for an entire CDB with its many PDBs as a single exercise. At the least aggressive end of the spectrum, all the changes are limited to just to the *root*. Therefore, the time to patch the environment for *n* application backends, when each is housed in its own PDB and the entire CDB is patched, will be about the same as that to patch the environment for *one* application backend when this is housed in its own non-CDB. In other words, the multitenant architecture allows patching the Oracle version for *n* application backends for the cost of patching the Oracle version for *one* application backend in the non-CDB architecture. We expect this approach to be preferred in development and test environments where the users are required all to move to a new version together, and thereafter to use only that. And we expect the unplug/plug approach to be preferred in production environments where certification of each consolidated application backend for the new version will follow its own schedule.

The section *"Remote cloning with GoldenGate replication as an alternative to unplug/plug"* on page 24 discusses an approach which effectively achieves *hot unplug/plug*.

Unplug/plug for responding to SLA changes

The new unplug/plug paradigm is also very useful for responding quickly to SLA changes. Again, the source and destination CDBs should share a filesystem so that the PDB's datafiles can remain in place.

We expect, therefore, that customers will usually choose to have several CDBs on the same platform, at different Oracle Database software versions, and configured differently to meet a range of SLAs.

Cloning a PDB

An unplugged PDB is a complete, but latent, representation of everything that defines an application backend, that typically has been in use and therefore holds quota-consuming data. If an identical copy — in other words a *clone* — of such an application backend is needed, while it is in use, then you could unplug the PDB, use operating system methods to copy all the files, plug the original files back in, and plug in the cloned files, using a different name, and possibly into a different CDB[13]. This thought experiment gives the clue to how the *clone PDB* SQL statement works. Without unplugging the source PDB, the foreground process copies its files and plugs these in as the clone. This is shown in *Figure 8*.

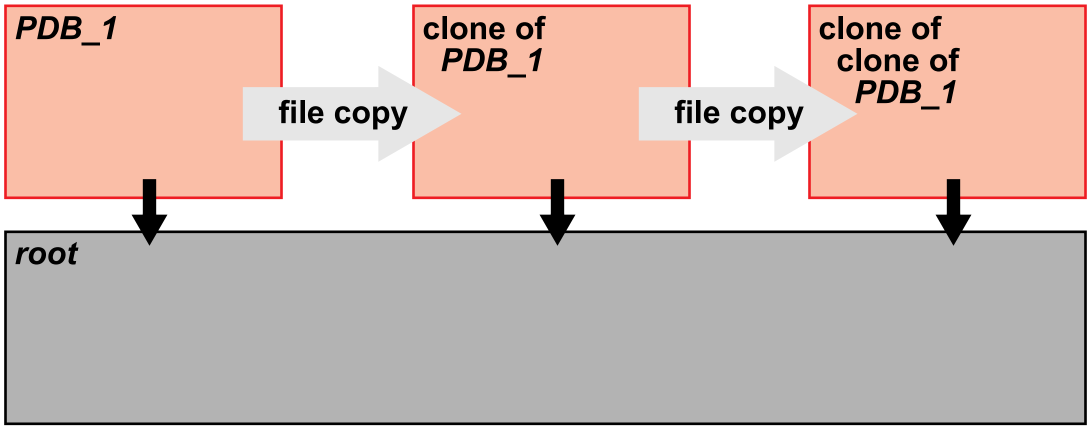

Figure 8 Using the foreground process to copy a PDB's datafiles to create cloned

The foreground process allocates a new globally unique identifier for the cloned PDB. The clone records the globally unique identifier of the PDB from which it was copied, and this lineage can, of course, be an arbitrarily long chain.

The multitenant architecture allows creating the cloned PDB in the same CDB as the source PDB or in a different CDB, as long as there is Oracle Net connectivity between the two CDBs. We will use the terms *local cloning* and *remote cloning* to distinguish between these choices.

13. Strictly speaking, this is unsound. The *create PDB* SQL statement allocates it a globally unique identifier, and this travels with it across unplug/plug operations. Clearly two PDBs ought not to have the same globally unique identifier. Therefore, you must not manually copy the files of an unplugged PDB. Rather, if you want to clone, use the *clone PDB* SQL statement. (This also allows cloning from an unplugged PDB — see *"Cloning from an unplugged PDB"* on *page 24*.) And if you want an "insurance" copy, use RMAN backup or Data Guard.

Cloning a PDB using full copy

Because Oracle Database is doing the file copy, Oracle parallel server processes can be recruited to reduce the total time for the task. Moreover, the datafiles that are to be copied contain Oracle data blocks, and the Oracle binaries understand what these are. Therefore, the degree of parallelism is not limited by the number of datafiles, nor even by coarse-grained allocation units within these understood by the operating system; rather, it is limited by the number of Oracle data blocks within the set of datafiles. This is so large that, in practice, the degree of parallelism is limited only by the number of Oracle parallel server processes that can be dedicated to the *clone PDB* operation, given other concurrent activity on the machine.

The multitenant architecture allows the *clone PDB* operation to be done without quiescing the source PDB — but this is not supported in 12.1.0.1. (The attempt causes *ORA-65081: database or pluggable database is not open in read only mode*.) The challenge is identical to the one that is met by RMAN online backup: copy a set of datafiles, starting at time *t*, when during the time it takes to make the copy, some of the Oracle data blocks in these files are changing. The solution is to notice if the to-be-copied block has changed since time *t* and, if it has, reconstruct it as it was at time *t* using undo information. Clearly the RMAN approach can be re-purposed for the *clone PDB* clone operation. We shall refer to *clone PDB*, without quiescing the source PDB, as *hot cloning*[14].

Cloning a PDB using snapshot copy

For some time, filesystem vendors, such as Oracle Corporation (with Sun ZFS), or NetApp, have exposed a method to make a copy of an arbitrarily large file in times measured in a few seconds or less. The approach takes advantage of a generic notion, relevant for many other kinds of structures besides files, called *copy-on-write*.[15]. Briefly, when the notion is applied to copying a file, advantage is taken of the fact that the blocks are accessed through a separate record that bears the file's name and that lists pointers to the file's blocks. Rather than copying all the blocks, just the list of pointers is copied, and this is given the name of the new file. So far, each block in the file is pointed to both from the source file's pointer list and from the copy file's pointer list. No substantive storage has been consumed, and so the copy is near instantaneous. Only when an attempt is made to write to a block in one of the files is a second copy of the block made to allow the two files to differ in this block. It's this idea that gives the approach its name.

When a large file is copied this way, and then changes are made to only small fraction of its blocks, the net saving in time and machine resource is enormous. For this reason, the approach is very popular when a clone is made for a development or testing purpose. Typically, in this scenario, the file is deleted after a relatively short life when most of its blocks remain unchanged.

Filesystem vendors prefer to use a term like snapshot copy. And the business benefit is usually referred to as thin provisioning.

14. The discussion of hot cloning is included in this whitepaper because, were it not, the reader would point out that the challenge of supporting hot cloning is the same as the challenge of supporting RMAN online backup! As of the whitepaper's timestamp, it is unknown what Oracle Database release might support hot cloning.

15. The generic ideas are described in this Wikepedia article *http://en.wikipedia.org/wiki/Copy-on-write*.

Customers have used snapshot copy to make a thinly provisioned copy of the RMAN backup set for a non-CDB, restoring the cloned non-CDB from this copy. (A new globally unique identifier has to be created for the cloned non-CDB.) But the whole process involves various separate steps, executed in various environments, and requiring different forms of authority. Usually, because of a traditional division of labor between database administrators and storage administrators, more than one person, each knowing different passwords, have to cooperate and synchronize their efforts. This is not always easy to arrange!

In the multitenant architecture, the foreground process sends a request to the storage system to copy the source PDB's datafiles using a dedicated secure protocol. The particular CDB records the username/password credentials for the storage system's administrative user (for the CDB's datafiles) in a wallet.

Snapshot PDB cloning is supported only when the new PDB is created in the same CDB as the PDB from which it is cloned. Moreover, while there exists a thinly provisioned PDB, then the PDB from which it was cloned cannot be dropped. A PDB created using snapshot copy cannot be unplugged. Both source PDB and clone PDB can be opened in read-write mode.

In 12.1.0.1, these filesystems support snapshot PDB cloning: Sun ZFS, NetApp, and Oracle ACFS. An error is reported if snapshot PDB cloning is attempted on any other filesystem. (The error is *ORA-17517: Database cloning using storage snapshot failed*.)

Cloning from an unplugged PDB

An unplugged PDB can be used as a very convenient delivery vehicle for a complete application backend, entirely replacing earlier methods that involve running various tools (for example SQL*Plus, Data Pump, and the like). In this use case, it would not be sound simply to plug in the unplugged PDB into many different CDBs because each PDB so created would have the same globally unique identifier. The *plug in PDB* command therefore allows the new PDB to be created as a clone of the unplugged PDB so that it is allocated its own globally unique identifier. *Code_6* shows the SQL statement.

In a variant of this usage, a set of "gold master" unplugged PDBs, representing useful starting points for, for example, stress testing or debugging of application logic, might be established at a central location at a customer site.

Remote cloning with GoldenGate replication as an alternative to unplug/plug

Should hot cloning be supported, then it would be possible to use remote cloning to establish a new PDB in the CDB where otherwise it would have been established using unplug/plug, without yet causing any downtime for the application whose application backend is housed in the to-be-moved PDB. Then GoldenGate replication could be used, having noted the SCN at which the hot cloning started, to catch up the cloned PDB and then keep it tracking. The steady tracking state is analogous to that which characterizes the use of transient logical standby for rolling upgrade to the Oracle Database software version. Of course, therefore, the procedure to cutover the client-side application would be the same. In other words, remote cloning across a difference in Oracle Database software version between source and destination CDBs would be the natural counterpart, for a PDB, to the established transient logical standby method for a non-CDB.

The syntax for the *clone PDB* SQL statement

Code_3 shows the basic *clone PDB* SQL statement.

```
-- Code_3
create pluggable database PDB_2 from PDB_1
```

The semantics are “create a clone of *PDB_1* and call the resulting new PDB *PDB_2*”.

Code_4 shows the *clone PDB* SQL statement for remote cloning.

```
-- Code_4
create pluggable database PDB_2 from PDB_1@CDB_02_link
```

The semantics are “create a clone of *PDB_1*, to be found in the CDB whose *root* is denoted by *CDB_02_link*, and call the resulting new PDB *PDB_2*”. (Alternatively, the database link may denote the to-be-cloned PDB.)

The database link encapsulates the specification of the location of the source CDB (it specifies listener name, listener port, and service name, which leads to the database and its files). It also specifies the authorization to start a session whose current container is the *root* of the source CDB. Notice that, once location and authorization are established, the transport of the files uses a suitable low-level protocol and is parallelized, using Oracle parallel server processes as described. The files are not transported over the database link.

Code_5 shows the *clone PDB* SQL statement using snapshot copy.

```
-- Code_5
create pluggable database PDB_2 from PDB_1 snapshot copy
```

The semantics are “create a clone of *PDB_1* using underlying the filesystem’s thin provisioning approach, and call the resulting new PDB *PDB_2*”.

Code_6 shows the *clone PDB* SQL statement cloning from an unplugged PDB.

```
-- Code_6
create pluggable database PDB_2 as clone
using '/u01/app/oracle/oradata/…/pdb_1.xml'
copy
```

The semantics are “create a clone of the unplugged PDB *PDB_1*, and call the resulting new PDB *PDB_2*, copying the unplugged PDB’s datafiles to a new set to that they may be changed independently of the source”.

The three operations *create PDB*, *plug in PDB*, and *clone PDB* are all variants of the *create pluggable database* SQL statement. As such, each allows the *path_prefix* clause.

Creating a PDB

Rather than constructing the data dictionary tables that define an empty PDB from scratch, and then populating its *Obj$* and *Dependency$* tables, the empty PDB is created when the CDB is created. (Here, we use *empty* to mean containing no customer-created artifacts.) It is referred to as the *seed PDB* and has the name *PDB$Seed*. Every CDB non-negotiably contains a *seed PDB*; it is non-negotiably always open in read-only mode. This has no conceptual significance; rather, it is just an optimization device. The *create PDB* operation is implemented as a special case of the *clone PDB* operation. The size of the *seed PDB* is only about 1 gigabyte and it takes only a few seconds on a typical machine to copy it. The choice to use snapshot PDB cloning is not available for *create PDB*.

The syntax for the *create PDB* SQL statement

Code_7 shows the basic *create PDB* SQL statement.

```
-- Code_7
create pluggable database PDB_1
path_prefix = '/u01/app/oracle/oradata/pdb_2_Dir'
admin user PDB_1_Admin identified by p
roles = (DBA, Sysoper)
```

The semantics are "create a brand-new PDB called *PDB_1*, ensure that any directory objects created within the PDB are constrained to the specified subtree in the filesystem that the CDB uses, create a local user[16] in the new PDB called *PDB_1_Admin* to be its administrator, and grant it the given list of roles".

Dropping a PDB

A PDB that is to be dropped must be closed[17].

There are two use cases for the *drop PDB* operation. The first is the obvious one, when the PDB simply is not needed any more. This would be common in a non-production environment where a PDB that represents a starting point for a specific development task, or is under investigation, is repeatedly cloned and dropped. We refer to this as a destructive drop. This is shown in *Code_8*.

The second use case is less obvious. The *unplug PDB* operation leaves the PDB still known in the the *root*'s metadata; and its datafiles are still listed in the CDB's control file. This is so that RMAN backup can be used to record the state of just this one PDB as it was at the moment it was unplugged. Therefore, whether or not a backup is to be made, the *unplug PDB* operation must always be followed by the *drop PDB* operation — either immediately, if no backup is to be taken, or after the backup is completed successfully. Of course, the intention of the *plug in PDB* operation implies that the datafiles must be retained. We refer to this as a non-destructive drop. This is shown in *Code_9*.

The syntax for the *drop PDB* SQL statement

Code_8 shows the destructive *drop PDB* SQL statement.

```
-- Code_8
drop pluggable database PDB_1 including datafiles
```

The semantics are "drop the PDB called *PDB_1*, and remove all of its datafiles without trace. Obviously, this should be used with caution.

16. We shall define the term *local user* in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*.

17. The column *gv$PDBs.Open_Mode* has these three allowed values: `MOUNTED`, `READ ONLY`, and `READ WRITE`. This is explained in *"The SGA is logically virtualized"* on *page 34*.

Code_9 shows the non-destructive *drop PDB* SQL statement.

```
-- Code_9
drop pluggable database PDB_1 keep datafiles
```

The semantics are "drop the PDB called *PDB_1*, but leave all of its datafiles in place.

Why it's important that *create PDB*, *clone PDB*, *drop PDB*, and unplug/plug are SQL statements

The *create PDB* and *drop PDB* operations are functionally equivalent to using *dbca* to create, or to delete, a non-CDB — but *dbca* can be used only by logging on to the machine where the relevant *Oracle Home* is installed, as the operating system user that owns the software installation, and therefore owns the files of any other databases that have been created within that software installation. Therefore, only very trusted people are able to do these provisioning tasks. The *clone PDB* operation is functionally equivalent to using, for a non-CDB, a rather complex sequence of steps that typically involve cloning an RMAN backup set, and then restoring the non-CDB from the resulting set. This, too, requires logging on as the operating system user that owns the *Oracle Home*. The nearest functional equivalents, for a non-CDB, to the *unplug PDB* and *plug in PDB* operations are Data Pump's full database export and full database import. These, too, require operating system authorization as the *Oracle Home* owner to copy or move the dump files.

Recall, too, that the *unplug PDB* and *plug in PDB* operations can be used to achieve the effect of patching the Oracle version of a PDB. This operation, too, for a non-CDB requires operating system authorization as the *Oracle Home* owner.

In contrast, the SQL statements that implement *create PDB*, *clone PDB*, *unplug PDB*, *plug in PDB*, and *drop PDB* can be done from a client machine and need only authorization as suitably privileged Oracle Database users. And the statements are direct atomic expressions of the semantic requirements of the intended operation. Moreover, except when physical full copy, or transport, of datafiles is involved, the SQL statements can complete in times measured in seconds — and this includes the case of snapshot PDB cloning.

PL/SQL can be used simply for automation, for example to encapsulate closing a PDB, *unplug PDB*, and the subsequent non-destructive *drop PDB*; and *plug in PDB* and then opening it. Tools such as SQL Developer and Enterprise Manager[18] expose the functionality of the PDB provisioning SQL statements directly.

The net result is a very powerful new paradigm for database provisioning.

Unplug/plug and *clone PDB* for CDBs protected by Data Guard

The standby database will respond automatically to the *create PDB* and *clone PDB* operations in the primary database. But for the *plug in PDB* operation in the primary database, its datafiles will need to be made available at the standby database in just the same way that the datafiles for transportable tablespaces need to be handled. Similarly, after the *drop PDB* operation with

18. Enterprise Manager exposes provisioning functionality for non-CDBs for pre-12.1 non-CDBs; but this relies on installing and using server-side agents and a scheme for operating system authorization. The implementation of the corresponding functionality for PDBs is much more straightforward and, as has been explained, the operations execute much faster on PDBs than on non-CDBs.

the *keep datafiles* choice, which follows the *unplug PDB* operation, the datafiles of the unplugged PDB, must be removed manually from the standby environment.

How to adopt a non-CDB as a PDB

There are two approaches: direct adoption of a 12.1 non-CDB as a PDB; and adoption of the content of a non-CDB, using for example Data Pump, into an empty PDB.

Direct adoption of a 12.1 non-CDB as a PDB

Oracle Database 12*c* supports both the new multitenant architecture and the old non-CDB architecture. A pre-12.1 non-CDB cannot be directly adopted as a PDB, and so an it must be upgraded in place, using the usual approach. The actual direct adoption is very simple — both conceptually and practically. The non-CDB is simply shut down and then treated as if it were an unplugged PDB. Recall that an unplugged PDB has an external manifest that is generated by the *unplug PDB* operation. Therefore, before shutting down the to-be-adopted non-CDB, it must be put into `READ ONLY` mode and then the procedure *DBMS_PDB.Describe()* must be executed, nominating the path for the manifest as its only argument — just as a path must be nominated when using the *unplug PDB* operation.

After shutting down the to-be-adopted non-CDB, it would be natural to do an *ad hoc* RMAN backup.

The non-CDB's datafiles, together with the manually created manifest, can now be treated as if they were an ordinarily unplugged PDB and simply plugged in to the target CDB and opened as a PDB. However, as will be explained immediately, it must be opened with the *Restricted* status set to `YES`. The tablespaces holding quota-consuming data are immediately viable, just as if this had been a Data Pump import using transportable tablespaces. However, the former non-CDB's data dictionary so far has a full description of the Oracle system. This is now superfluous, and so it must simply be removed. While still with the *Restricted* status set to `YES`, the *noncdb_to_pdb.sql* script (found on the *admin_directory* under *Oracle Home*) must be run. Now the PDB may be closed and then opened ordinarily.

We shall refer to this adoption method as the *upgrade-and-plug-in adoption approach*.

Adoption of the content of a non-CDB

The approaches described in this section are inevitably possible by virtue of the PDB/non-CDB compatibility guarantee. Data Pump's full database export and import can be used to migrate most of the customer-created artifacts from the non-CDB to the new PDB. New in 12.1, Data Pump supports full database export, and full database import, making maximum possible use of transportable tablespaces, using single commands. The capability is called *Full Transportable Export/Import*. This enhancement has been backported to 11.2.0.3. We shall refer to this adoption method as the *Data Pump adoption approach*. However, database artifacts of some kinds, for example XML schemas, directories and grants made to *public*, cannot be moved using Data Pump. These will need to be migrated using *ad hoc* methods.

Notice that when the size of the to-be-adopted non-CDB is relatively small, and when Data Pump is able to handle all the content, it might be quicker to use the Data Pump adoption approach than to use the upgrade-and-plug-in adoption approach. Moreover, a consolidation exercise often involves moving application backends from old equipment to modern equipment — and this might mean cross-endianness migration. As explained in *"Unplug/plug across different operating systems, chipsets, or endiannesses"* on page 18, the Data Pump adoption approach is the only option in this scenario.

Alternatively, GoldenGate replication can be used to populate to new PDB and then to keep it tracking ongoing changes in the source non-CDB. Then the client-side application code can be moved from the old non-CDB to the new PDB in a brief brownout using the same cutover approach as is used in connection with the transient logical standby method for patching the Oracle version for a non-CDB.

The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions

This section explores the consequences of the fact that it is the entire CDB that is opened by the RAC instance.

Commonality of users, roles, and granting

In a CDB, we distinguish between *local users* and a *common users*. The fact that there are two kinds flows inevitably from the PDB/non-CDB compatibility guarantee. This requires two things:

- that each PDB in the same CDB can have a customer-created user called, say, *Scott* — where these are independent and just happen, coincidentally, to have the same name
- that Oracle-supplied users like *Sys* and *System* must be present in every container with a uniquely defined, documented purpose and characterization.

Local users and local roles

As was explained in *"The multitenant architecture's horizontally partitioned data dictionary"* on *page 14*, each PDB in a particular CDB can have a user called *Scott*. Such a user, then, is a *local user* and is defined exactly and only in the PDB's *User$* data dictionary table and is understood as such, therefore, only in that PDB. (Obviously, then, *Scott* in each PDB can have a different password from each *Scott* in every other PDB.) Without this, the PDB/non-CDB compatibility guarantee would not be honored. Correspondingly, a *local role* is defined exactly and only in a particular PDB.

Neither a local user, nor a local role, can be created in the *root*.

Common users and common roles

On the other hand, a session that uses a PDB in a particular CDB must see the identical set of Oracle-supplied artifacts as a session that uses any other PDB in that CDB. These artifacts, of course, include users like *Sys* and *System* and roles like *DBA* and *Select_Catalog_Role*. *Sys* and *System* are *common users* and *DBA* and *Select_Catalog_Role* are *common roles*. The name and password of a common user is defined in the *root* but the invariant is maintained that the identity is known in every PDB in that CDB — present and future — with the same name and password. Correspondingly, a *common role* is defined in the *root* and is known everywhere.

Objects like *Sys.DBMS_Output* and *System.Sqlplus_Product_Profile* must also, for the same reason, be present in every container with a unique definition. Such objects are called *common objects* and are defined in the *root*[19]. Customers cannot create common objects.

19. Common objects are represented in a PDB's dictionary by a single row in *Obj$*. This matches the corresponding row in *CDB$Root*'s *Obj$* by name (matching *Owner*, *Object_Name*, and *Namespace*). It is this device that allows a PDB to be immediately viable after unplug/plug between CDBs even though its dependency graph, in *Dependency$*, spans both *CDB$Root*'s common objects and the PDB's local objects and the facts in *Dependency$* are recorded using *Object_ID* values.

Commonly granting privileges and common roles

The effect of granting a privilege or role when the current container is a PDB is represented in the *SysAuth$* and *ObjAuth$* data dictionary tables within that PDB. The result is that the effect of a grant is contained within the PDB where it was made. This is true, even when the grantee is a common user or a common role. Therefore, a particular common user or common role might have received different grants in each PDB. However, the PDB/non-CDB compatibility guarantee requires that Oracle-supplied common users like *Sys* and *System*, and Oracle-supplied common roles like *DBA* and *Select_Catalog_Role*, have exactly the same set of grants in every PDB in that CDB — present and future. (We assume, here, that customers follow the recommended best practice and neither grant, nor revoke, privileges or roles to/from Oracle-supplied users. This best practice does not apply to customer-created common users and common roles.) This must be supported formally; otherwise a privilege or role might be revoked just in a particular PDB and then the guarantee would be violated. The formal support takes the form of a special flavor of the *grant* SQL statement. *Code_10* shows an example.

```
-- Code_10
grant Create Session to A_Common_User container = all
```

The semantics are "ensure that this item remains granted to the grantee in every container — present and future". When commonly granting, or revoking, a privilege or a common role, the current container must be the *root*. Logic demands that the grantee must be known everywhere — so it must be a common user or a common role. Further, the granted item must also be known everywhere — so it must be either a system privilege or object privilege (known everywhere by virtue of being Oracle-supplied) or a common role.

Customer-created common users and common roles

The use case for a customer-created common user is as the authorization vehicle for a person who should be able to do administrative tasks for the CDB as a whole, or in two or more PDBs, where, with respect to name, password, and auditing, the user should be a single entity.

This follows from the established pre-12.1 best practice of locking and expiring all Oracle-supplied users, and instead relying on customer-created users to allow people to perform all required administrative tasks. In a CDB, this best practice would rely on customer-created common users. Such users, pre-12.1, typically don't own objects. However, they might do so — but the objects would be, for example, procedures that implement certain administrative tasks in a safe way. Such objects would not be considered to be part of the implementation of an application backend. This understanding suggests that, in a CDB, a customer-created common user might own local objects in the *root*. A customer-created common user is not prevented from owning local objects in a PDB. But we recommend against it. Local objects in PDBs should be owned by local users.

Similar considerations establish the case for customer-created common roles.

The name of a customer-created common user or common role must start with *C##* or *c##*. *Code_11* shows the syntax to create a common user.

```
-- Code_11
create user c##u1 container = all identified by p
```

The *container = all* clause is also used to create a common role.

Notice that a local user cannot exercise system privileges in the schema of a common user.

Services and sessions

To create a session against a non-CDB using Oracle Net, you must present these five facts: listener location, listener port, service name, username, and password. As long as the first two identify a running listener, and it presently supports the specified service, then a foreground process is started against one of the RAC instances in which the non-CDB of interest is open. So far, the foreground process is running as *Sys*. Then the *User$* table is queried to see if the given username and password are valid and the denoted user has the *Create Session* privilege. If they are valid, then the given user becomes the session user, and the client can start to issue database calls; otherwise, then the foreground process terminates and the client is, of course, left unconnected.

This same account, with one small addition, describes how a session is created against a chosen PDB in a CDB. The key point is that the same five facts are presented in the same way. When the foreground process, running as *Sys*, starts, the current container is the *root*. It can see the properties of the service that got the session creation attempt this far. Newly in 12.1, a service has a property that specifies the PDB that, when authorization is complete, will be the session's current container. (When this is *null*, it means that the new session's current container should be the *root*.) The session now switches to the designated container and only then queries the *local User$* table to see if the given username and password are valid and the denoted user has the *Create Session* privilege. Thereafter, the success/failure story is the same as for a non-CDB.

Notice that the requirement is just that the nominated user must be known in the container specified by the service's property. It doesn't matter whether the user is a local user or a common user.

Changing the session's current container for an established session

The *alter session set container* SQL statement changes the current container to the specified target. *Code_12* shows an example.

```
-- Code_12
alter session set container = PDB_2
```

The user issuing this must be known in the target container and must have the *Set Container* system privilege there. Of course, this user must also be known in the container from which this SQL statement is issued. This implies that only a common user can use this statement without error. The result of the statement's successful use is a single container-to-container transition; history is irrelevant and isn't maintained.

You cannot start a transaction in a container while there exists an unfinalized transaction in another container.

The *alter session set container* SQL statement is legal only as a top-level server call.

An overload of *DBMS_Sql.Parse()*, new in 12.1, brings the effect of *alter session set container*, within PL/SQL code, in a strict "trampoline" fashion. For this reason, we refer to the approach that uses this as the *DBMS_Sql.Parse trampoline*.The *Container* formal parameter specifies the location for the single SQL statement that is parsed. When the statement terminates, the current container automatically, and non-negotiably, is reset to what it was when *DBMS_Sql.Execute()* was invoked. This is true also if the remotely executed SQL statement causes an error. *Code_13* shows an example.

```
-- Code_13
begin
  ...surrounding PL/SQL context...
  -- For example
  Stmt := 'create table t(PK...)';
  declare
    Cur integer := DBMS_Sql.Open_Cursor(Security_Level=>2);
    Dummy integer;
  begin
    Dbms_Sql.Parse(
      c=>Cur,
      Statement=>Stmt,
      Container=>Con_Name,
      Language_Flag=>DBMS_Sql.Native);
    Dummy := DBMS_Sql.Execute(Cur);
    DBMS_Sql.Close_Cursor(Cur);
  end;
  ...surrounding PL/SQL context...
end;
```

The *DBMS_Sql.Parse* trampoline is legal only when the starting container is *CDB$Root*[20].

These rules guarantee that a session that is created by a local user cannot escape the PDB that was specified by the service name that was used at connect time. This implies that, for local users, the between-PDB isolation model, for PDBs in the same CDB, is the same as it is between non-CDBs on the same platform, owned by the same operating system user[21].

It can be convenient to use *alter session set container* in *ad hoc* SQL*Plus scripts for CDB administration, to avoid repeated use of the `CONNECT` command, and the concern this brings for password safety. The approach using *DBMS_Sql.Parse()* can be useful for CDB administration tasks that are orchestrated using PL/SQL.

The SGA is logically virtualized

Within-database consolidation delivers its consolidation density benefit because all the application backends share, per RAC instance, the same SGA. Using schema-based consolidation, the SGA simply ends up with a population whose items, though they reflect all of the application backends, and the Oracle system, are not attributed as such. Only the human, who thinks about it, realizes that this is the case. Using PDB-based consolidation, each of the items (most notably data blocks and library cache structures like child cursors) is annotated with its provenance. In other words, the SGA is logically virtualized. This is significant. The data dictionary is *physically* virtualized, to bring pluggability; and the SGA is *logically* virtualized to retain the physics of within-database consolidation. This is shown in *Figure 9 on page 35*.

20. Otherwise, the normal PL/SQL rules apply: the code may be any of the three kinds (anonymous block, invoker's rights unit, or definer's rights unit) and the privilege test is based on the current user, and only that.

21. To reinforce this principle, there is no direct way to reference an object in one PDB from a SQL statement issued, or part of the code that defines an object, in another PDB. We deliberately decided not to invent an three-part naming scheme (PDB-name, schema-name, object-name) that would allow cross-PDB name resolution. if you want, from one PDB, to reference an object in another PDB, the you must use a database link.

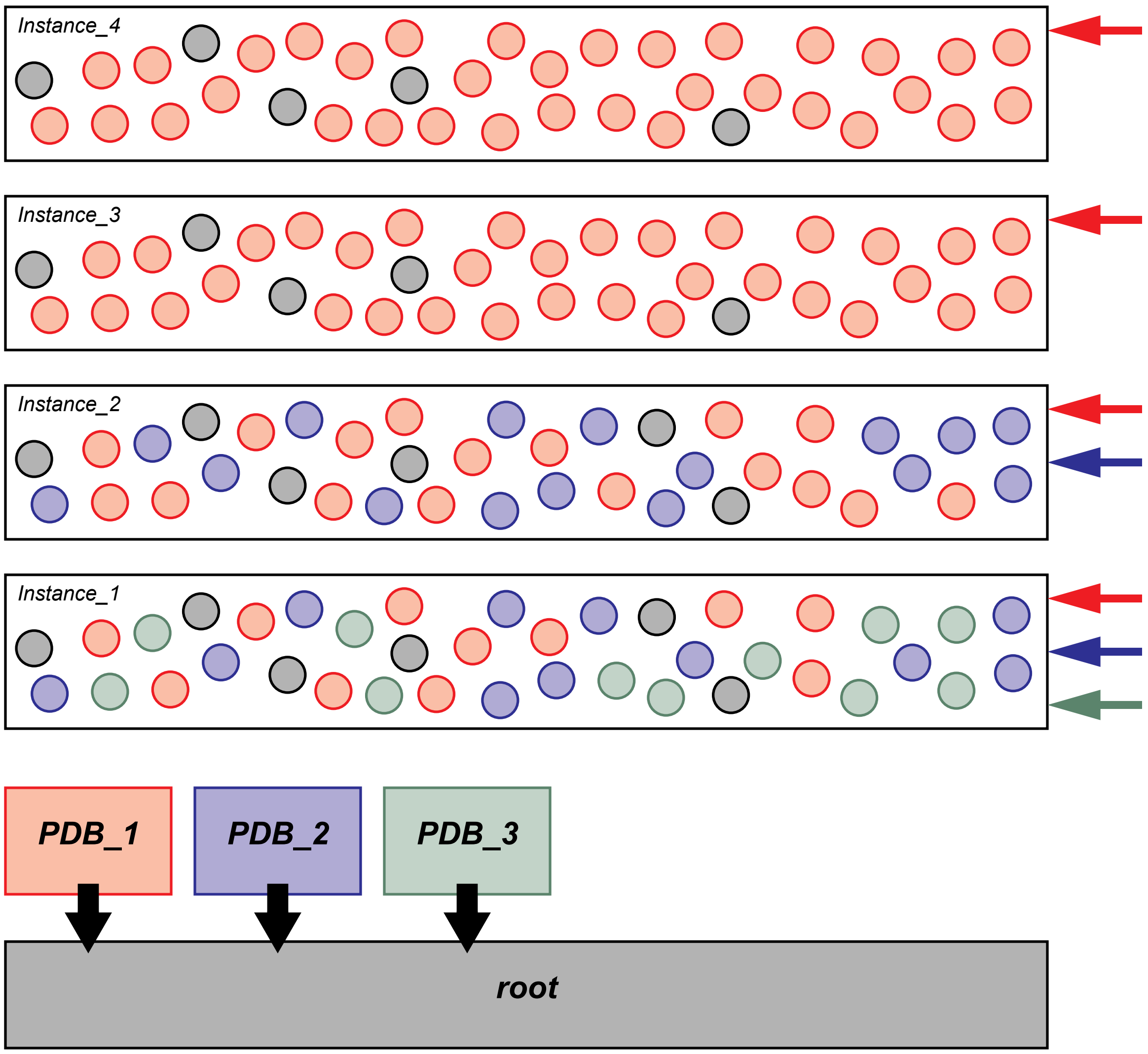

Figure 9 *PDB-to-RAC-instance affinitization. The arrows on the right are color-coded to show the affinitization between each PDB and each of the RAC instances.*

Some customers have followed the practice, using schema-based consolidation, of affinitizing the sessions that support users working with a particular application backend to a particular set of RAC instances. They have done this by following a self-imposed discipline of using specific service(s) to start sessions dedicated to a particular application backend.

The convention has formal support in the multitenant architecture. A PDB can be in `READ WRITE` mode, in `READ ONLY` mode, or in `MOUNTED` mode. The mode is shown in the performance view column *gv$PDBs.Open_Mode* — and it can be set explicitly, and differently, for each PDB in each RAC instance. Notice that the `MOUNTED` mode is the closest a PDB can get to idle. This is precisely because the RAC instance deals with the CDB as a whole, so it is meaningless to want to shut it down for just a specific PDB. The *Open_Mode* can be set, using the *alter pluggable database* SQL statement, for any PDB when the current container is the *root*; and can be set for a particular PDB when the current container is that PDB. For compatibility, the SQL*Plus commands `STARTUP` and `SHUTDOWN` can be issued when the current container is a PDB. They are translated into the corresponding *alter pluggable database* SQL statements.

Orthogonally to the *Open_Mode*, the *Restricted* status can also be set to NO or to YES. There are five distinct combinations of *Open_Mode* and the *Restricted* status for a PDB. (When the *Open_Mode* is MOUNTED, then the *Restricted* status can be only *null*.) The *alter pluggable database* SQL statement can be used to go directly from any such combination to any other one by using the keyword *force*, as illustrated in *Code_14*.

```
-- Code_14
-- PDB_1 starts in READ WRITE mode
alter pluggable database PDB_1 open read only force
```

In this example, all sessions connected to *PDB_1* and involved in non-read-only transactions will be terminated, and their transactions will be rolled back. This is a nice improvement in usability compared to the rules for the corresponding mode changes for a non-CDB.

The notifications that are sent from a RAC instance where an SGA change takes place to the other RAC instances, to avoid disagreement on information that is cached in several SGAs, are pruned at a coarse level in the multitenant architecture. There is no point in informing RAC instances where a particular PDB is not open of changes that have occurred to its items cached in the RAC instances where it is open.

The multitenant architecture provides, therefore, a simple declarative model for application-backend-to-RAC-instance affinitization that more reliably implements the intention, and provides better performance, than the hand orchestrated scheme to meet this goal using schema-based consolidation. This idea is discussed more in *"PDB-to-RAC-instance affinitization"* on *page 45*.

Data dictionary views and performance views

The PDB/non-CDB compatibility guarantee implies that the data dictionary views and performance views, queried when the current container is a PDB, must show only information about artifacts defined within that PDB together with those defined within the *root*. For example, when the Sample Schemas are installed in *PDB_1*, and when *PDB_1* is the current container, then the query shown in *Code_15*

```
-- Code_15
select Owner, Object_Name
from DBA_Objects
where Owner = 'SYS'
and Object_Type like 'VIEW'
and Object_Name like 'ALL%'
union
select Owner, Object_Name
from DBA_Objects
where Object_Type like 'VIEW'
and Owner in (select Username from DBA_Users where Common = 'NO')
```

will include these results:

```
SYS   ALL_ARGUMENTS
SYS   ALL_DIRECTORIES
SYS   ALL_ERRORS
SYS   ALL_INDEXES
SYS   ALL_OBJECTS
SYS   ALL_TABLES
SYS   ALL_VIEWS
HR    EMP_DETAILS_VIEW
OE    PRODUCTS
SH    PROFITS
```

However, it will not include any objects defined within *PDB_2* or any other PDB in the CDB.

In the same way, when *PDB_1* is the current container and when *v$Sql* is queried, information will be seen only about SQL statements issued when *PDB_1* is the current container.

The multitenant architecture brings new rules for the results when data dictionary views and performance views are queried when the current container is the *root*. Potentially, the results will be the union over the occurrences of the view in question across all presently open containers. The rows are distinguished because the views have a new *Con_ID* column in 12.1.

This meets the business goal of supporting a single system image across the whole CDB. For example, violations in customer-defined practices for naming conventions can easily be policed. (Some customers insist that names never include special characters like single-quote, double-quote, semicolon, or question-mark.) Another use might focus on finding the longest-running SQL statements, CDB-wide.

The performance views simply acquire the *Con_ID* column. But, for the data dictionary views, the *Con_ID* column is exposed only in a new family whose names start with *CDB_*. These extend the concept of the *DBA_* data dictionary views; the *All_* and *User_* flavors are not extended this way.

The views can return rows for more than one container (that is, those acquired the new *Con_ID* column) are collectively called *container_data views*. When the current container is a PDB, then every *container_data view* shows only rows whose *Con_ID* value denotes this PDB, or denotes the CDB as a whole. But when the current container is the *root*, then the *container_data views* might show rows from many containers. Recall that a local user cannot be created in the *root*, and so only a common user might see results from a *container_data view* with more than one distinct value of *Con_ID*. The set of containers for which a particular common user might see results in a *container_data view* is determined by an attribute of the user (set using the *Alter User* SQL statement).

This attribute can specify, at one end of the spectrum, illustrated in *Code_16*, just one particular *container_data view* for, say, just two named PDBs.

```
-- Code_16
-- Allow c##u1 to see data for just CDB_Objects
-- in CDB$Root, PDB_1 and PDB_2
alter user c##u2
set container_data = (cdb$root, pdb_001, pdb_002)
for Sys.CDB_Objects
container = current
```

Notice that this SQL statement is allowed only when the current container is the *root*, and that the attribute can be set only locally in the *root*. Notice, too, that *CDB$Root* must always be included in the list.

At the other end of the spectrum, illustrated in *Code_17*, it can specify all *container_data views*, present and future (across an Oracle Database release boundary) for all containers, present and future. The Oracle-supplied users like *Sys* and *System* are configured this way.

```
-- Code_17
-- Allow c##u2 to see data for every container, present and future
-- in every container_data view
alter user c##u2
set container_data = all
container = current
```

Per CDB choices *versus* per PDB choices

Consolidation inevitably implies trading out some of the autonomy that was enjoyed by the previously independently managed application backends in order to win the sought-after reduction in capital expense and operating expense. This section sets out the choices that can be made only for the CDB as a whole, and gives some examples of the remaining choices — those that can be made differently for each PDB.

Choices that can be made only for the CDB as a whole

Oracle Database software version, and platform specifics

Just as with schema-based consolidation, the central consolidation notion is to house many distinct application backends within the same database, sharing (per RAC instance) the same SGA and the same set of backround processes. This means, of course that every PDB within the same CDB must be at the same Oracle Database software version. It is no more meaningful to ask if *different PDBs in the same CDB* can be at different Oracle Database software versions than it is to ask if *different schemas in the same non-CDB* can be at different Oracle Database software versions. Both within-database consolidation models imply, of course, a single choice of platform type and operating system type and version for all the consolidated application backends.

The unplug/plug paradigm for patching the Oracle version removes the discomfort that otherwise would be caused by the fact that CDB as a whole is at a particular Oracle Database software version. Further, the mobility that this brings loosens the tie that schema-based consolidation establishes between an application backend and the platform, and operating system type and version, on which the non-CDB that houses it runs.

The *spfile*, control files, and password file

These files are common for the CDB as a whole. This implies that a PDB cannot *literally* have its own *spfile*. We shall see in *"PDB-settable initialization parameters and database properties"* on *page 42* that some parameter values can be set persistently with *alter system* within the scope of a PDB. These are persisted appropriately — but not actually in the *spfile*. By extension of this you cannot name a *pfile* when you open a PDB. Therefore, if you want to dump the values of parameters that have been set specifically with *alter system* within the scope of a PDB, then you must write a SQL query to do this. Correspondingly, to impose these values, you must program a sequence of *alter system* SQL statements.

Data Guard, RMAN backup, redo, and undo

Customers who have used schema-based consolidation have shown significant reduction in operating expense because they can set up, and operate, Data Guard for a single non-CDB that houses many different application backends — rather than, as they used to before consolidation, managing Data Guard for each individual application backend as a separate task. This is the canonical example of the manage-as-one principle. In the same way, they have derived operating expense savings by operating a single scheduled RMAN backup regime to benefit many different application backends. However, as mentioned in *"Striving to achieve maximum consolidation density"* on *page 5*, this doesn't give a generally workable solution for application backend point-in-time-recovery. We shall return to this point in *"Ad hoc RMAN backup for a PDB"* on *page 42*.

Both Data Guard and RMAN backup rely on the redo logs: Data Guard applies redo continuously at the standby site; and RMAN backup uses redo when restoring to roll forward from the time of the backup to the required restore time. Therefore, to deliver the same manage-as-one savings for PDB-based consolidation as schema-based consolidation delivers, the CDB as a whole has a single redo stream. This implies that, just as is the case with the SGA, the redo log is *logically* virtualized: each redo log entry is annotated with the container identifier for where the change that it records occurred. Redo and undo go hand in hand, and so the CDB as a whole has a single undo tablespace per RAC instance. The undo, then, is also *logically* virtualized: here too, each undo record is annotated with origin container identifier. These two points inform the discussion about the performance implications:

- Firstly, the situation in this regard, when using PDB-based consolidation is the same as when using schema-based consolidation. And customers have demonstrated that schema-based consolidation doesn't harm performance. In the bigger picture, other factors are at work — most notably that the set of application backends is served by far fewer backround processes in total, when either of the two within-database consolidation models is used, and that the application backends jointly make much more effective use of memory when a single SGA is shared between them all than when each has its own individually sized, and individually constraining, SGA.

- Secondly, new within-redo and within-undo indexing schemes can speed up access to the records for the relevant PDB. Moreover, because of the functional independence of the individual logically virtualized "partitions" with the redo and undo, it is practical for concurrent processes to access these without waiting for each other.

Customers who aim to increase return on investment by consolidation usually do their own empirical investigations using representative models of their own consolidation candidates.

Character set

The multitenant architecture does allow the possibility the each PDB might have its own character set. However, allowing this would detract from the manage-as-one benefit.

Further, a trend has emerged over recent years where a huge number of applications have an international, and multilingual, user base. And this is reflected by the fact that an ever-increasing proportion of client-side processing is done using Unicode. This implies inefficiencies as data is converted backwards and forwards between client and database if the data isn't store in Unicode in the database.

It was therefore decided to allow the character set to be determined only for the CDB as a whole. We recommend, therefore, that Unicode (*AL32UTF8*) be used as the CDB's character set.

A consideration remains, of course, for legacy applications where the application backend uses a specific character set and cannot economically be re-engineered. This is where advantage would be taken of the fact that you can have several CDBs on the same platform, each created with its own character set. (This is just one example of a more general principle. The most obvious reason to have more than one CDB on the same platform is to allow each to have a different Oracle Database software version.)

CDB-wide initialization parameters and database properties

The *v$System_Parameter* view family has a new column in 12.1, *IsPDB_Modifiable*, with allowed values `FALSE` and `TRUE`. (We shall consider those with *IsPDB_Modifiable* `TRUE` in *"PDB-settable initialization parameters and database properties"* on *page 42*). About 200 have *IsPDB_Modifiable* `FALSE`. These are some illustrative examples:

```
audit_file_dest
audit_trail
background_core_dump
background_dump_dest
core_dump_dest
cpu_count
db_block_size
db_name
db_recovery_file_dest
db_recovery_file_dest_size
db_unique_name
instance_name
local_listener
log_archive_dest
log_archive_duplex_dest
memory_max_target
memory_target
parallel_degree_policy
pga_aggregate_limit
pga_aggregate_target
processes
result_cache_max_size
sga_max_size
sga_target
shared_pool_reserved_size
shared_pool_size
undo_management
undo_retention
undo_tablespace
user_dump_dest
```

The list speaks to the manage-as-one theme.

Apart from the already discussed `NLS_CHARACTERSET`, all the other properties listed in the *Database_Properties* view can be set using the *alter database* SQL statement when the current container is a PDB.

AWR Reports

You can create AWR Reports for the CDB as a whole, and we expect this to be the level at which scheduled reports will be run. To support *ad hoc* performance investigations within the scope of a single application backend, you can also create AWR Reports for a particular PDB.

Choices that can be made differently for each PDB

PDB point-in-time-recovery

The most notable per PDB freedom is the ability to perform PDB point-in-time-recovery without interfering with the availability of peer PDBs in the same CDB. And this ability is one of the most significant differentiators between schema-based consolidation and PDB-based consolidation. The former simply has no comfortable method to meet the business goal of per application backend point-in-time-recovery. The *flashback pluggable database* command is not supported in 12.1.

Ad hoc RMAN backup for a PDB

The manage-as-one principle is generally best served by performing scheduled RMAN backup at the CDB level, as has been explained. However, occasions arise when it is useful to take an *ad hoc* backup for a particular PDB. This was discussed in *"Dropping a PDB"* on *page 26* in connection with the joint use of the *unplug PDB* command and then the *drop PDB* command (keeping the datafiles).

alter system flush Shared_Pool

As was explained in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*, the data blocks in the block buffer and the library cache structures are annotated with the container identifier. This gives immediate meaning to the effect of *alter system flush Shared_Pool* when the current container is a PDB.

PDB-settable initialization parameters and database properties

Every initialization parameter that, in the *v$System_Parameter* view has both *IsSes_Modifiable* and *IsSys_Modifiable* not equal to `FALSE`, can be set using *alter system* when the current container is a PDB. In other words, all such parameters have *IsPDB_Modifiable* equal to `TRUE`. Further, some other parameters with *IsSys_Modifiable* not equal to `FALSE`, and with *IsSes_Modifiable* equal to `FALSE`, can also can be set using *alter system* with PDB scope. This is the relatively small list in 12.1:

```
cell_offload_decryption
fixed_date
listener_networks
max_string_size
open_cursors
optimizer_secure_view_merging
resource_limit
resource_manager_plan
sessions
```

Reflecting the other allowed values, `IMMEDIATE` and `DEFERRED`, for *IsSys_Modifiable*, the *scope* clause (*scope=memory*, *scope=spfile*, or *scope=both*) has the expected effect: that the value will be persisted if requested. To honor the PDB/non-CDB compatibility guarantee, the established *alter system* syntax was retained — but the persistence mechanism is not, in fact, the *spfile*. Rather, it is held in appropriate data dictionary tables so that *both* such PDB-specific values can take effect when it is opened *and* the values will travel with the PDB on unplug/plug.

The *ORA-65040* error

Oracle Database supports a very wide range of SQL statements. Some of these would be used only by the administrator who controls the overall database, under special circumstances. In a CDB, a small subset of these administrator-only SQL statements cannot be issued in a PDB. The attempt causes *ORA-65040 operation not allowed from within a pluggable database*. This is very much by design. The PDB/non-CDB compatibility guarantee should be read in this light. Notice that all SQL statements can be automatically issued from client-side code. But client code that's worthy of the name *application* — rather than, for example, something like *administration tool*, like Enterprise Manager — doesn't issue the statements that cause *ORA-65040*. Even client code that would be called *installation automation* should not, if it follows best practice, issue such statements.

Within-CDB, between-PDBs, resource management

The sessions using different application backends that are hosted on the same platform compete for these computing resources:

- the number of concurrent sessions
- CPU
- ability to use Oracle parallel server processes
- file i/o
- use of SGA memory and ability to allocate PGA
- network i/o

Some customers decide that they want full control over the competition for every one of these resources and therefore host each application backend in its own non-CDB in its own virtual machine, taking advantage of operating system virtualization. However, this scheme utterly defeats manage-as-one operating expense savings. Many customers have found that return on investment is maximized by prioritizing the manage-as-one savings, and have therefore used schema-based consolidation.

It is possible to use Resource Manager within the context of schema-based consolidation. But this requires a carefully planned, and humanly policed, discipline where each application backend is accessed only using services created for that purpose. The problem, of course, is that with schema-based consolidation, only the human being knows how to draw the boundaries around each distinct application backend; Oracle Database has no understanding of this. With PDB-based consolidation, the PDB provides a powerful, declarative mechanism that enables the human to tell Oracle Database where the boundaries are: each application backend is installed in its own PDB, and no PDB houses more than one application backend.

Resource Manager is enhanced, in the multitenant architecture, by the ability to create a CDB-level plan to govern the resource competition between the PDBs contained in the CDB.

The computing resources controlled by the CDB-level plan in 12.1

The Resource Manager CDB-level plan, in 12.1, controls these:

- the number of concurrent sessions
- CPU
- ability to use Oracle parallel server processes
- file i/o — but only on Exadata

But it does not control these:

- use of SGA memory and ability to allocate PGA

- network i/o

PDB-to-RAC-instance affinitization[22] can be used as a coarse-grained approach controlling the competition for SGA and PGA memory.

The shares and caps model

Resource Manager has implemented an industry-standard model based on two notions: the *share*, and the *cap*[23]. In this scheme, each competitor is allocated a number of shares, from *one* to any positive integer (but anything more than about *ten* might turn out to be unhelpful). At a moment, typically only some of the competitors are active. Each of the active competitors gets a fraction *n/t* of the managed resource, where *n* is the number of shares allocated to a particular competitor and *t* is the total of the allocated shares over all the currently active competitors. Optionally, any competitor can be given a cap; this is a fraction in the range *zero* through *one* (often expressed as a percentage). A competitor with a cap of *c* never gets more than that fraction of the managed resource, no matter what the share calculation might indicate. Of course, such a capped competitor might get less, depending on the instantaneous result of the share calculation.

The value of the share notion is obvious; it is the *sine qua non* of resource management. The value of the cap notion is realized when the number of competitors gradually increases up to some planned number. It ensures headroom remains for the ultimate plan, without setting false expectations along the way. (Users are pleased by a performance improvement — as would be seen when an application is moved from one machine to a more powerful one, uncapped, as the only application it initially hosts. But they have a notoriously short memory. So as more and more applications are moved onto the consolidation machine, they notice, and remember, only performance getting steadily worse. It is better, therefore, to cap the performance for an application, that is among the first to be moved to the consolidation machine, to the level that is expected when the consolidation exercise is complete.)

For between-PDBs resource management, when a PDB is created, it is allocated one share by default and is not capped.

How the CDB-level plan in 12.1 manages sessions, CPU, Oracle parallel server processes, and file i/o

When a plan is set, or modified, the effect is felt immediately by all current sessions[24].

22. This is described in *"PDB-to-RAC-instance affinitization"* on *page 45*.

23. Here is an academic paper on the topic, published in 1995:
http://people.cs.umass.edu/~mcorner/courses/691J/papers/PS/waldspurger_stride/waldspurger95stride.pdf
And here is a developer forum posting from a vendor of operating system virtualization software:
http://communities.vmware.com/docs/DOC-7272
The notion *cap* is sometimes called a *resource utilization limit*.

24. The plan is set using *alter system set resource_manager_plan = My_Plan*. As noted in *"PDB-settable initialization parameters and database properties"* on *page 42*, the *resource_manager_plan* initialization parameter can be set with container scope. So to set the CDB-level plan, *alter system* will be issued when the current container is the *root*.

The number of concurrent sessions is subject only to capping; the shares notion is meaningless here. In fact, the sessions cap is set using the *sessions* initialization parameter[25].

CPU and file i/o are managed using the shares and caps mechanisms described above. Fine-grained scheduling of sessions is performed every 100 ms.

Code_18 shows how a CDB-level plan is configured to allocate one share to *PDB_1* and. three shares to *PDB_2*.

```
-- Code_18
DBMS_Resource_Manager.Create_CDB_Plan(
  'My_Plan', 'some comment');

DBMS_Resource_Manager.Create_CDB_Plan_Directive(
  'My_Plan', 'PDB_1', Shares => 1);

DBMS_Resource_Manager.Create_CDB_Plan_Directive(
  'My_Plan', 'PDB_2', Shares => 3);
```

And *Code_19* shows how a CDB-level plan is modified to allocate two shares to *PDB_1* and. five shares to *PDB_2*, and to set caps for these at 20% and 50% respectively.

```
-- Code_19
DBMS_Resource_Manager.Update_CDB_Plan_Directive(
  'My_Plan', 'PDB_1', New_Shares => 2);

DBMS_Resource_Manager.Update_CDB_Plan_Directive(
  'My_Plan', 'PDB_1', New_Utilization_Limit => 20);

DBMS_Resource_Manager.Update_CDB_Plan_Directive(
  'My_Plan', 'PDB_2', New_Shares => 5);

DBMS_Resource_Manager.Update_CDB_Plan_Directive(
  'My_Plan', 'PDB_2', New_Utilization_Limit => 50);
```

The ability to use Oracle parallel server processes is governed by the same shares model. The *parallel_degree_policy* initialization paramter must be set to `AUTO`. Then enough SQL statements are run in parallel to keep the machine busy, but the queuing of SQL statements avoids downgrade of the degree-of-parallelism. Here, the cap notion has its own separate parameterization using the *parallel_server_limit* parameter in *DBMS_Resource_Manager*'s *Create_CDB_Plan_Directive* subprogram and the *new_parallel_server_limit* parameter in its *Update_CDB_Plan_Directive* subprogram.

PDB-to-RAC-instance affinitization

As explained in *"The dynamic aspects of the multitenant architecture: the Oracle instance, users, and sessions"* on *page 31*, the *Open_Mode* of each PDB can be set to `MOUNTED`, `READ ONLY`, or `READ WRITE` specifically, and therefore differently, in each RAC instance. For example, when a CDB with eight PDBs is configured as a RAC database with eight Oracle instances, then each PDB can be opened in exactly one RAC instance, so that each RAC instance opens exactly one PDB. This scheme delivers the maximum possible resource isolation. But it also compromises the sharing of SGA and backround processes that brings the high consolidation density that is the hallmark of PDB-based consolidation (and of schema-based consolidation).

25. As mentioned in *"PDB-settable initialization parameters and database properties"* on *page 42*, the *sessions* initialization parameter can be set using the *alter system* SQL statement when the current container is a PDB.

Perhaps a more sensible use of PDB-to-RAC-instance affinitization would be, with a CDB with a couple of hundred PDBs or more, to affinitize, say, the top quarter (with respect to requirement for processing power) to 75% of the RAC instances, and the remaining three quarters of the PDBs to the remaining 25% of the RAC instances.

Lone-PDB in CDB *versus* non-CDB

Suppose that an application backend has such a high throughput requirement that a particular platform must be dedicated exclusively to it. It could be installed either in a non-CDB or in a PDB that (apart from the *seed PDB*) is the *only* PDB in its CDB. We shall refer to the latter configuration as a *lone-PDB*. It is also referred to as the *single-tenant configuration*.

Is the non-CDB choice in any way better than the single-tenant configuration?

We promise that the answer is a resounding *no*. The PDB/non-CDB compatibility guarantee shows that there is no functional difference. Careful testing by Oracle Corporation engineers has proved this claim and has shown, further, that there is no performance difference.

The more interesting question is the other way round: is the single-tenant configuration better than the non-CDB choice?

Here, the answer is *yes*. Even when you never exceed one PDB, the new multitenant architecture brings significant benefits:

- Unplug/plug brings you, in terms of functionality, Data Pump Generation Three.
- Unplug/plug, into an empty, newly created CDB, brings you a new paradigm for patching the Oracle version[26].

This realization lead to the decision to specify the Oracle Multitenant option as allowing between *two* and 252 PDBs in a particular CDB. (The *seed PDB* is not counted among the 252 allowed PDBs.)

The single-tenant configuration, and therefore the choice to use the multitenant architecture, is available, for no extra cost, in each of Standard Edition, Standard Edition One, and Enterprise Edition[27].

26. Hot cloning (should a future release bring support for this) suggests this following approach: hot cloning into a "staging" CDB, at the same Oracle Database software version, noting the SCN at which it was created; followed by unplug/plug into the newer version CDB, with, then, catch-up to the source PDB, and tracking, using GoldenGate replication. This would bring the functional equivalent of the use of transient logical standby with greatly improved usability.

27. The Oracle Database 12*c* Licensing Guide sets out these terms formally.

Summary

We have seen that the essential difference between the old non-CDB architecture and the new multitenant architecture is that the latter brings true within-database virtualization. This is implemented physically in the data dictionary by virtue of the horizontal partitioning that separates the Oracle system, in the *root*, from the customer system, in the PDB — and that this basic separation allows, in turn many PDBs to be contained in the same CDB. The PDB is a declarative mechanism to delineate a consolidated application backend.

True within-database virtualization removes the principle drawbacks of schema-based consolidation: the collision of global names, which means that expensive and risky changes have to be made to existing application backends before they can co-exist in the same non-CDB; and the fact that *any* privileges, and similar powerful privileges, and grants to *public*, span all application backends in a non-CDB.

It is the physical separation between *root* and PDB that brings pluggability. And pluggability brings, effectively, the third generation of Data Pump, via unplug/plug. Further, unplug/plug between CDBs at different Oracle Database software versions brings a new paradigm for patching the Oracle version. Pluggability, then, removes these two problems brought by schema-based consolidation: difficulty of provisioning (i.e. moving an application backend from place to place, and cloning it) and the fact that patching the Oracle version for a single application backend, when circumstances mandate this, is impossible without, at the same time, affecting other application backends that are not ready for this change in environment.

We have seen, too, that the sharing model for the SGA and the backround processes is essentially the same for PDB-based consolidation as for schema-based consolidation — and so the high consolidation density benefit of the older approach is retained in full. Moreover, the logical virtualization, within the SGA (carried via the data block into the datafiles), the redo, and the undo brings new powers for within-CDB, between-PDBs, resource management and allows PDB point-in-time-recovery.

Finally, it's tempting to say that *CDB is to PDB as operating system is to non-CDB*. And the PDB/non-CDB compatibility guarantee means that the new phenomenon that characterizes the multitenant architecture is the *root*. Just as various operating system primitives orchestrate provisioning tasks, and other maintenance tasks for non-CDBs, SQL statements, executed against the *root* implement the corresponding tasks for PDBs. This, in turn, means that PL/SQL, famously platform independent and portable, available, of course, wherever Oracle Database is available, and known by all database administrators, is the language of automation for operations on PDBs.

Oracle Multitenant represents, therefore, the next generation of consolidation for application backends. It delivers the manage-as-one benefits that adopters of schema-based consolidation had hoped to win. Adopting it is a pure deployment choice: neither the application backend, nor the client code, needs to be changed.

Appendix A: The treatment of the multitenant architecture in the Oracle Database Documentation Library

For an introduction and overview, start with the Concepts book. This has a new main section, *Multitenant Architecture*.

The main coverage is in the Administrator's Guide. This, too, has a new main section, *Managing a Multitenant Environment*. Of course, because of the PDB/non-CDB compatibility guarantee, the Database Development Guide needs no more than a passing mention of the multitenant architecture. (The scope of an *edition* is the PDB in which it is defined. This means that many concurrent EBR exercises can be conducted in a single CDB. This fact removes a challenge that customers face, who have used schema-based consolidation to implement several application backends in the same pre-12.1 database, necessarily a non-CDB, when more than one of these backends needs to undergo online patching at the same time.)

The treatment of the new distinction between local users and common users, and of the scoping of privileges to the current container, and how this leads to new notions for formal control of the separation of duties between those of the über database administrator (the CDB admin) and those of the of application administrator (the PDB admin) is in the Security Guide. The book also covers another essential concept in this space: *container_data views*.

The multitenant architecture introduces remarkably few new SQL constructs. They are limited mainly to implementing the operations on PDBs as entities. The formal definitions of the syntax and semantics for these is in the SQL Language Reference.

There are a few brand new data dictionary views (for example, *DBA_PDBs*) and performance views (for example, *v$PDBs*). More noticeably, each performance view has a new *Con_ID* column — the container identifier for the provenance of the facts represented in such a view. Correspondingly, but using a slightly different model, the set of flavors of data dictionary view (*DBA_*, *All_*, and *User_*) is extended with the new *CDB_* flavor. Each *CDB_* view has a new *Con_ID* column. The presence of a *CDB_* column in a view reflects its status as a *container_data view*. The descriptions of all the views are in the Database Reference.

Oracle Multitenant
June 2013
Author: Bryn Llewellyn

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com