

Oracle Multitenant: New Features

In Oracle Database 18c

ORACLE WHITE PAPER | MARCH 2018





Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



ORACLE®



Table of Contents

Disclaimer	1
Introduction	2
Refreshable PDB Switchover	3
Enhanced Integration with Data Guard	7
Snapshot Carousel	10
Dynamic Lockdown Profiles	12
CDB Fleet Management	14
Summary	16



Introduction

This White Paper covers various new features of Oracle Multitenant, introduced with Oracle Database Release 18c. In this fourth major release of Oracle Multitenant, we deliver a number of important enhancements:

- » **Refreshable PDB Switchover**

The refreshable PDB switchover capability enables the creation and maintenance of replicas on a per-PDB basis with only two CDBs to manage.

- » **Enhanced Integration with Data Guard**

simplifies PDB provisioning operations in high availability configurations, protected by Data Guard.

- » **Snapshot Carousel**

This is a repository for periodic point-in-time copies of a PDB. The Snapshot Carousel is ideally suited to development environments that typically require multiple copies of databases at different points in time, or to augment a non-mission critical backup and recovery process.

- » **Dynamic Lockdown Profiles**

Changes to Lockdown Profiles are now dynamically propagated to all applicable PDBs, without the need to restart either PDB or CDB.

- » **CDB Fleet Management**

A CDB Fleet is a group of CDBs managed collectively. This further extends Multitenant's operational efficiency advantage: In a CDB, we can manage many PDBs as one. Now, with CDB Fleet, we can manage many CDBs as one!

Each of these is discussed in a section of this White Paper.

Refreshable PDB Switchover

The refreshable PDB switchover capability, new in 18c, allows us to create and maintain replicas on a per-PDB basis with only two CDBs to manage. We might start by creating a PDB named Red in CDB1 and then simply creating a replica of this in CDB2 using the familiar refreshable PDB syntax. (This was introduced with 12.2.) We might add a few more PDBs: - Gold in CDB2; Brown in CDB1; Grey in CDB2. No replicas are required for Gold and Brown, but we want one for Grey. As simply as that, we have two pairs of replicas being refreshed in “opposite directions” between these two CDBs. This is illustrated in Figure 1.

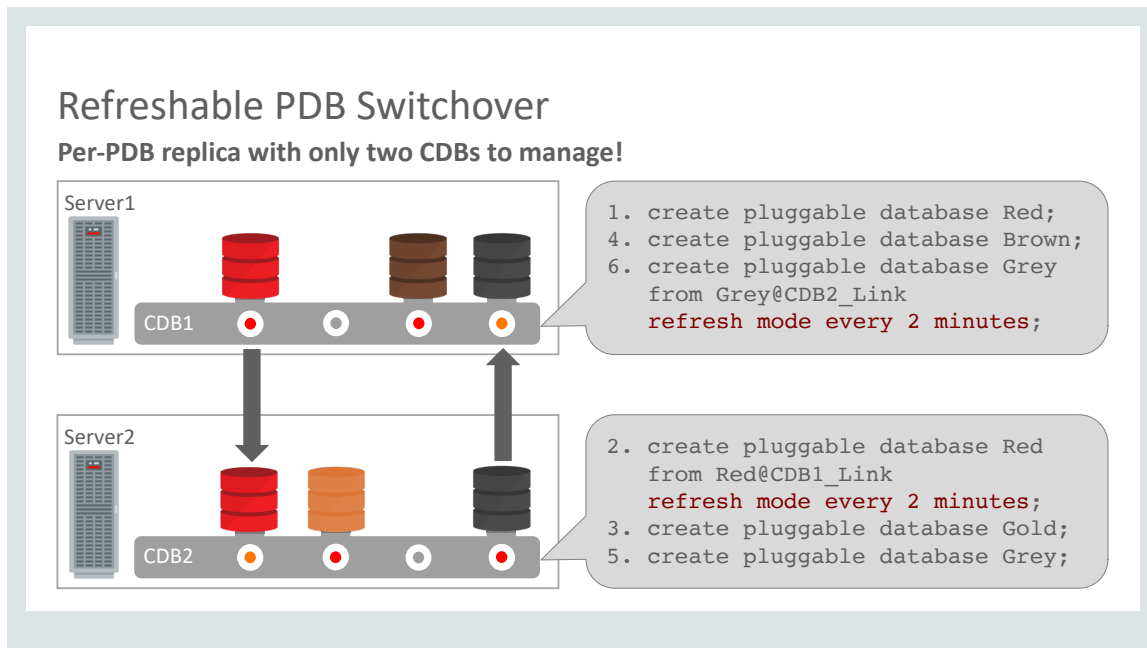


Figure 1. Using RSeRefreshable PDBs to maintain replicas

Everything discussed so far is possible in 12.2, but now we come to the new switchover capability. There are several ways in which this can be very useful. For example, planned switchovers can be very helpful for load balancing. This role transition is executed by connecting to the current primary - You may find it helpful to think in terms of connecting to the “future replica” - and issuing this statement

```
alter pluggable database Grey  
refresh mode every 2 minutes  
from Grey@dblink switchover;
```

The syntax to setup the refreshable clone is familiar, but you’ll also notice the new keyword “switchover”. The result of this operation is that PDB Grey in CDB1 assumes the primary role. Its replica is now maintained in CDB2. User connections to Grey will now be to the new primary – now in CDB1. Figure 2 illustrates this switchover capability.

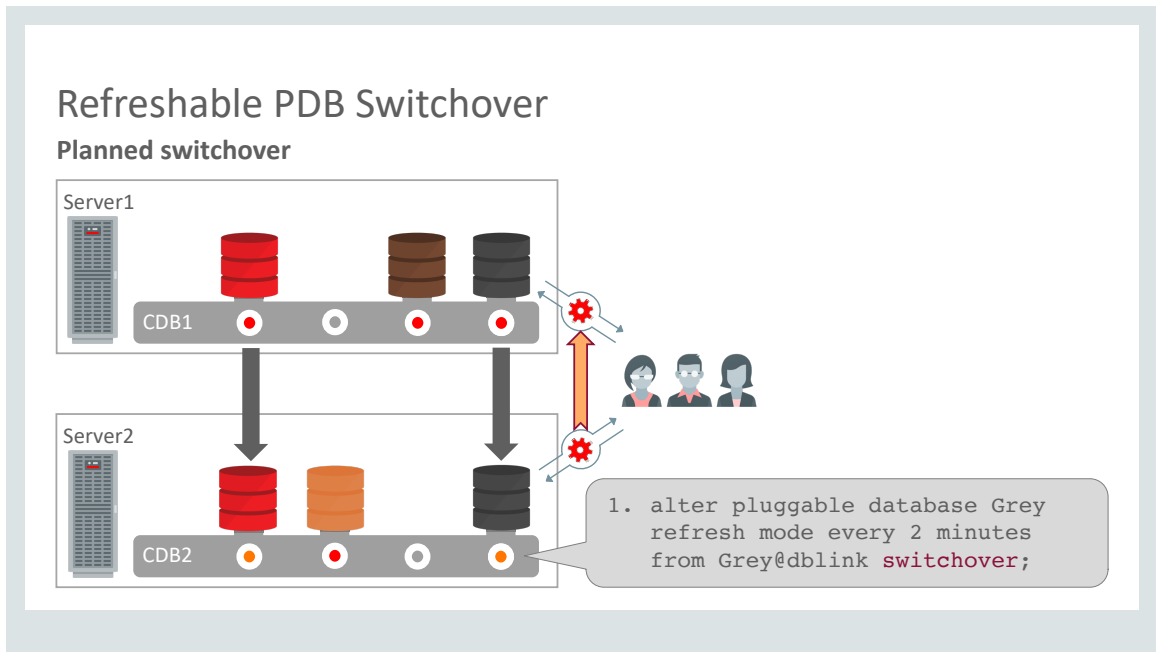


Figure 2. Refreshable PDB Switchover

In this example, we've seen how trivial it is to perform load balancing with this capability, while maintaining a replica on a remote server just in case. In situations where the primary PDB fails for some reason, we can switchover to the replica and resume operations there. We refer to this as an unplanned switchover. The three steps to perform the role transition from replica to primary, illustrated in Figure 3, are as follows:

1. Perform one more refresh of the replica.
Remember that, although the source *PDB* is no longer available, the refresh process involves reading redo from the primary *CDB*'s redo stream – online redo log and archive redo log files. For the purposes of this discussion, we assume that the source *CDB*, including its redo stream, is intact and that the failure is confined to the source *PDB* alone. Therefore, it is reasonable to assume this final refresh will succeed. What it achieves is to transport and apply any transactions that were performed against the primary between the time of the previous refresh and the *PDB*'s failure.
2. Alter the surviving replica so that it is no longer a refreshable *PDB*.
3. Open the former replica read-write so that it can be used as the new primary.

Refreshable PDB Switchover

Unplanned switchover

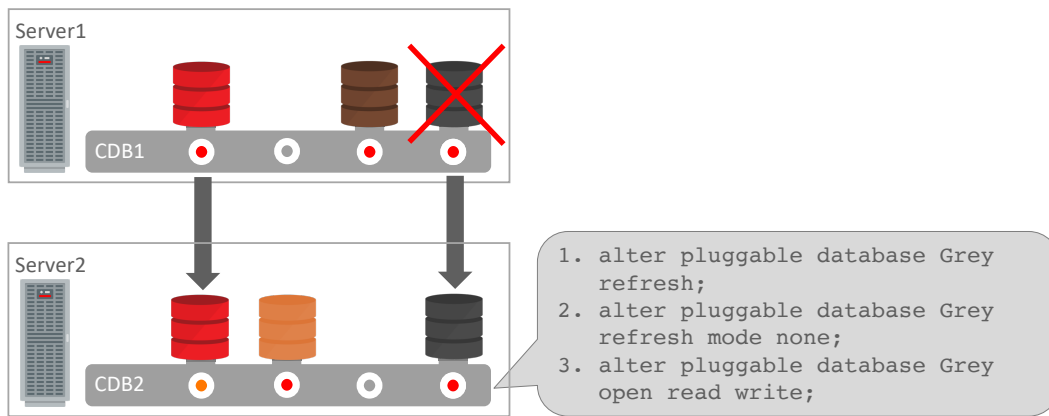


Figure 3. Unplanned switchover with refreshable PDBs

It's important to note that this capability should not be considered as a replacement for Data Guard from a High Availability point of view. However, in the sense that it can be used to maintain a replica in another server, we've seen how refreshable PDBs may be used as replicas on which operations of certain low-load, non-critical applications may be resumed, whether the switchover is a planned or an unplanned event. In that sense, it's worthwhile to consider the characteristics of a switchover from the point of view of *Recovery Time Objectives* (RTOs), and *Recovery Point Objectives* (RPOs).

RTOs consider how long it takes before operations can resume in the event of a switchover. In other words, how long is the outage? RTO is affected by the amount of redo to be applied. In turn, that is a function of the rate of redo generation in the source PDB and the frequency of refresh of the replica. The worst case is a failure of the source just before the next refresh was to have occurred. Assuming a relatively steady rate of redo generation at the source, this situation is likely to result in the maximum amount of redo to be applied to the replica before a switchover can be completed. In general, a relatively high frequency of refreshes – of the order of every couple of minutes rather than every few hours – is likely to leave relatively little redo to be applied before a switchover can be completed. It is important to test thoroughly with realistic transaction volumes to ensure that the process of refreshing the replica can keep up with the rate of redo generation.

RPOs apply to how much data is lost in the event of failure of the primary. In many cases the failure of the source *PDB* alone will not cause any data loss. As long as the source *CDB* remains operational, all committed transactions should be protected in the online redo logs and archive redo logs of the source. It's therefore likely that even transactions that had not been applied to the replica at the time of failure of the source *PDB* can be recovered and applied to the replica before switching over to that. The worst case is a total failure of the source *CDB* or its host server. In this case the online redo logs and potentially even the archive redo logs on that server are inaccessible.

With the basic refreshable PDB capability described so far, there is in this and similar situations the possibility that all transactions that happened on the source since the last refresh will be lost. For example, if the refreshable PDB had been created with this statement:

```

create pluggable database myPDB
from myPDB@DBLink
refresh mode every 2 minutes;

```

The maximum data loss is two minutes of transactions. (This assumes that the refreshes had been able to keep up with the rate of generation of redo from the source.)

However, another strategy exists, through integration with a Data Guard remote redo repository (a component of Data Guard Far Sync). This possibility arises because the refreshable PDB can read data:

- » Either over database link (from online and archive redo logs on the primary – the situation described so far)
- » Or from a remote redo repository (maintained via redo transport)

This second option – refreshing the replica PDB from a remote redo repository – has several important advantages:

- » It minimizes impact of scanning redo logs on primary
- » It minimizes amount of redo transport between CDBs
- » It enables near-zero data loss in failover situation

This is illustrated in Figure 4.

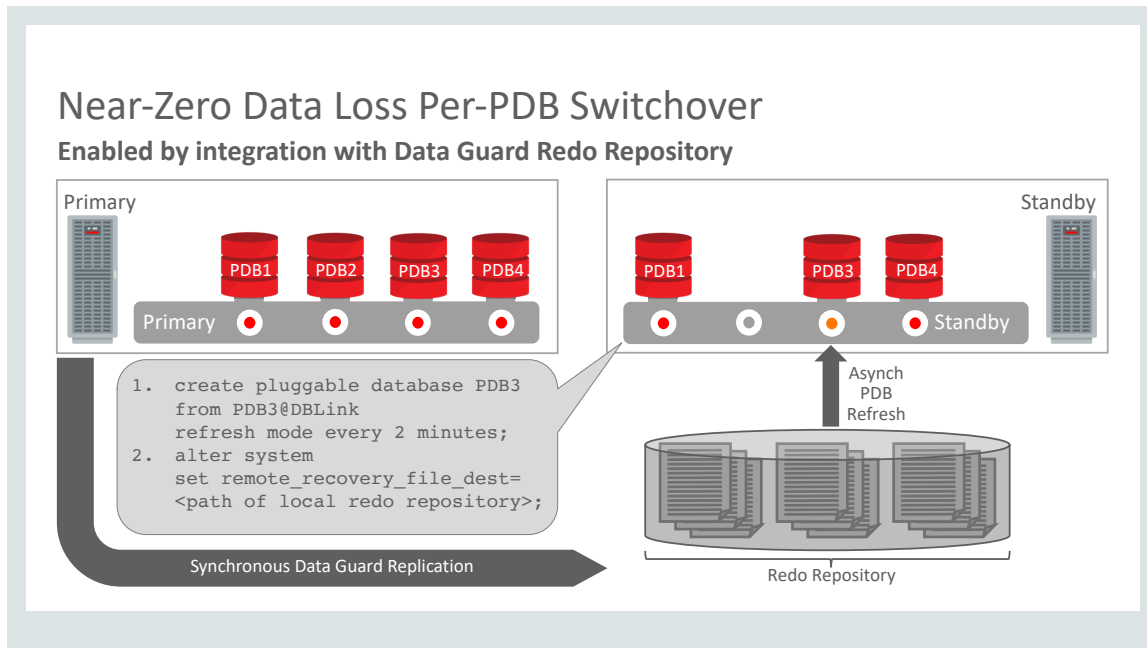


Figure 4: Integration with Data Guard Redo Repository

Enhanced Integration with Data Guard

Certain forms of PDB creation could become a little complicated in Data Guard configurations prior to 18c. Two of the most common provisioning operations – creating a new PDB from seed and creating a local cold clone work absolutely fine with Data Guard. Let’s walk through an illustration of this to understand what’s going on, as illustrated in Figure 5.

Here we see a high availability configuration with a primary CDB at the top, and a standby CDB at the bottom, with replication managed by Data Guard. We begin with a “new” CDB, with no customer-created PDBs. Only PDB\$Seed is present in this initial state.

When we create a new PDB (from Seed), the *create pluggable database* operation replays successfully on the standby. This is possible because PDB\$Seed – including its data files – is already present on the standby. The new PDB is created successfully on both the primary and the standby.

Similarly, a local cold clone (that is, a clone of a PDB already present in the primary and open read-only) will succeed – and replay successfully on the standby – because it is already present in the standby including all the data files necessary to process the operation.

However, for various reasons, some other provisioning operations are more complicated.

For example, a hot clone – that is, a clone of a PDB performed while the source PDB remains online (or in other words open read-write) – is complicated by the fact that there are two “recovery” operations involved at the same time. One is to ensure that the hot clone itself is *transactionally consistent*. The other is the application of redo on the standby, performed by Data Guard. These two “recovery” operations would interfere with each other, and to avoid that, a hot clone to a Data Guard primary will have no standby replica. This may be viewed as the functional equivalent to including the clause “standbys=none” in the hot cloning operation on the primary.

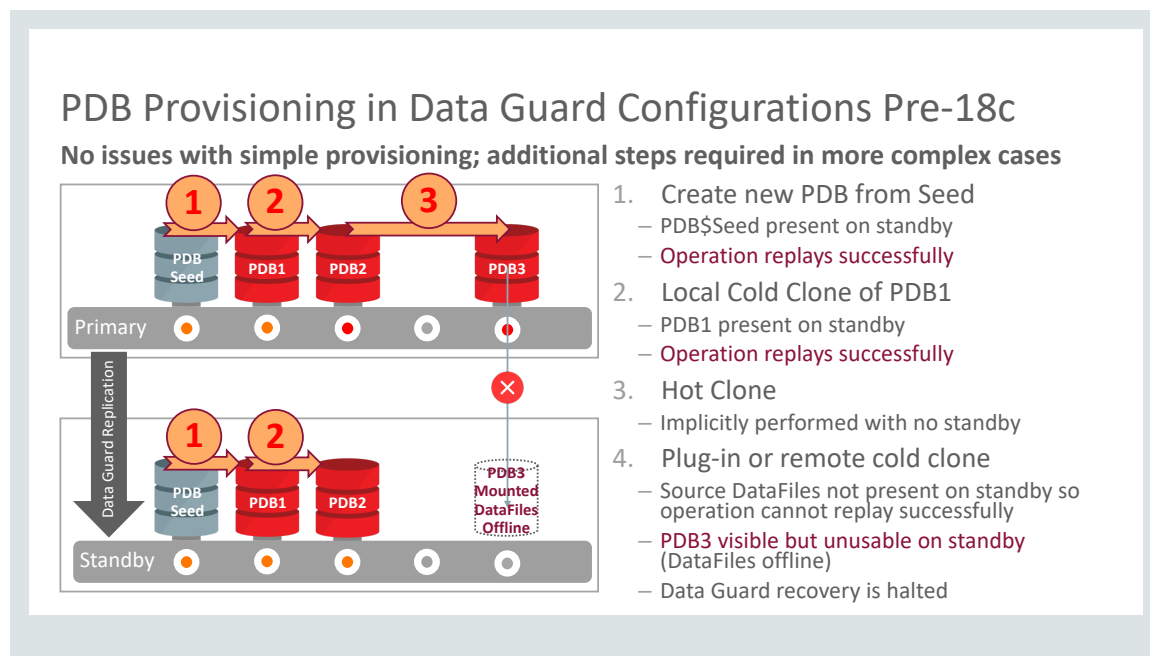


Figure 5. Provisioning operations in Data Guard configurations prior to 18c

For the purposes of this discussion, a remote cold clone, that is, a cold clone of a PDB in another CDB, is comparable to a plug-in of an unplugged PDB. In both cases, unlike creating a new PDB from Seed, or performing a local cold clone, by definition the data files for the newly created PDB are *not* already in the CDB. Therefore the complication in these cases is that, not only do we need to copy the data files for the new PDB into the primary CDB, we also need to copy them on to the remote CDB. Prior to 18c, there was no simple way to automate this transport of these data files to the standby. Therefore, prior to 18c, unless the data files are carefully pre-positioned on the standby, manually, although the PDB is created successfully on the primary, it is not successfully created on the standby and Data Guard recovery was halted until the situation could be rectified by manual intervention.

The complications just described are greatly simplified in 18c. This involves some additional one-time steps in the process of setting up the Data Guard environment and a simple procedure to follow in each of these formerly somewhat more complicated provisioning cases.

First, let's discuss the additional one-time steps involved in Data Guard setup, as illustrated in Figure 6.

1. We start by creating a self-referencing database link on the primary. This replays successfully on the standby to create a database link referencing the primary.
2. Next, we set a parameter on the standby. This is named *standby_PDB_source_file_DBLink* and it should reference the database link created in the previous step.
3. Finally, CDB\$Root of the standby should be opened read-only. (Note that no Active Data Guard license is required if only CDB\$Root is opened read-only on the standby.) This allows Data Guard to read the value of the parameter just specified. When a *create pluggable database* operation is replayed on the standby, but the required data files are not already present on the standby, those data files may be copied *from the primary* via this database link.

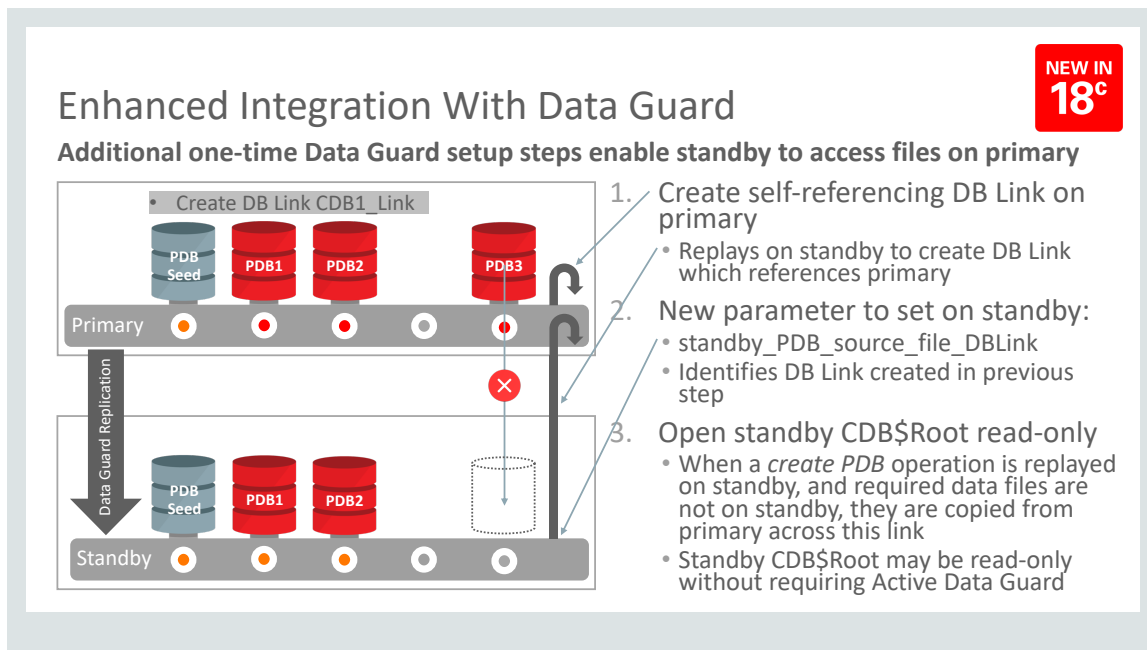


Figure 6. Additional one-time steps in Data Guard Setup

Now we introduce a simple new procedure employing what we refer to as a “transient no-standby PDB”. Following this procedure reduces what were formerly more complex provisioning operations to a few trivial steps.

For example, let's say that our intention is to plug-in an unplugged PDB or to perform a remote cold clone. We want this new PDB to be named PDB3.

1. We start by creating what is referred to as a "transient no-standby PDB". To do this, simply include the clause `standbys=none` in the `create pluggable database` statement. The result of this is that on the standby, the transient PDB will be visible in mounted state but its data files will show as unnamed or offline.
2. Next, we create the desired PDB – PDB3 – as a local cold clone of the transient no-standby PDB. This replays successfully on the standby by copying the PDB's data files across the database link identified by the parameter `standby_PDB_source_file_DBLink`.
3. The transient no-standby PDB may now be dropped.

So, what was formerly a relatively complicated procedure is reduced to this simple 1-2-3, illustrated in Figure 7.

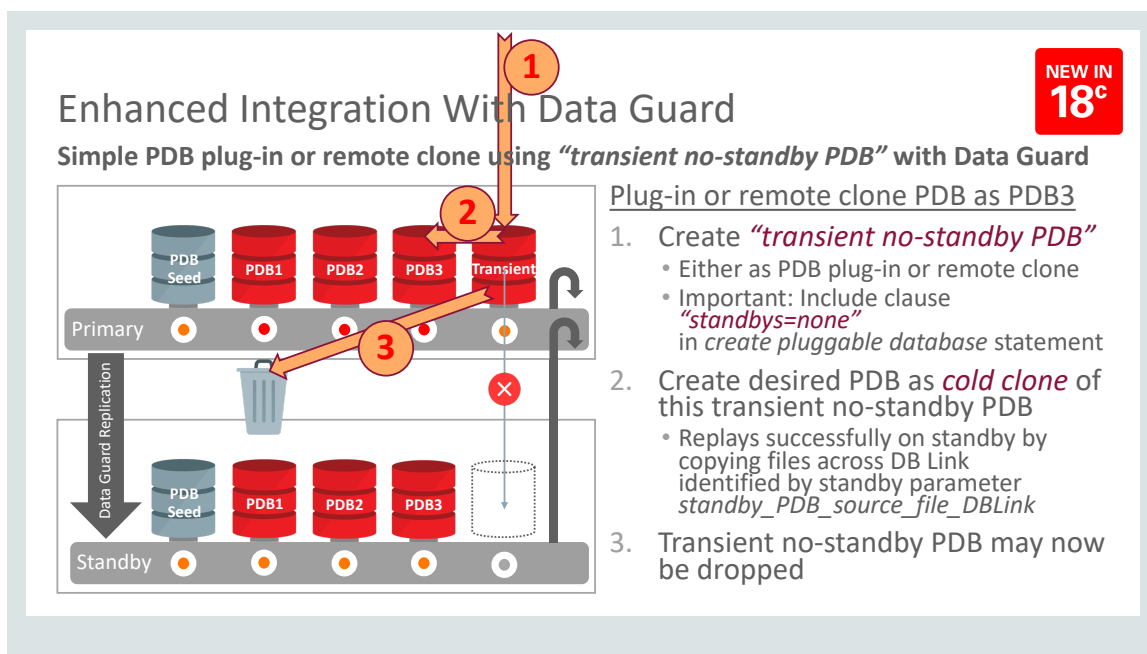


Figure 7. Simple PDB plug-in or remote clone using "transient no-standby PDB" with Data Guard

Follow this same, simple, 1-2-3 procedure for hot clones in high availability configurations protected by Data Guard.

1. As before, the first step is to create the "transient no-standby PDB", this time as a hot clone. Again, it's important to include the clause `standbys=none` in the `create pluggable database` statement. Remember, the intention of a hot clone is to take the clone while the source remains online. Clearly, this requirement is met in this case.
2. Next, we create the desired PDB as a local cold clone of the transient no-standby PDB. This replays successfully on the standby by copying the PDB's data files across the database link identified by the parameter `standby_PDB_source_file_DBLink`.
3. The transient no-standby PDB may now be dropped.

As you see, now in 18c we can use this same, simple procedure, employing the "transient no-standby PDB" technique for provisioning operations which were formerly relatively complicated in Data Guard configurations.

Snapshot Carousel

The snapshot carousel is a repository for periodic point-in-time copies of a PDB. It makes sense to limit the number of historical snapshots we take. The maximum number of snapshots to be retained is configurable. The default is eight, which is also the highest value that is currently supported.

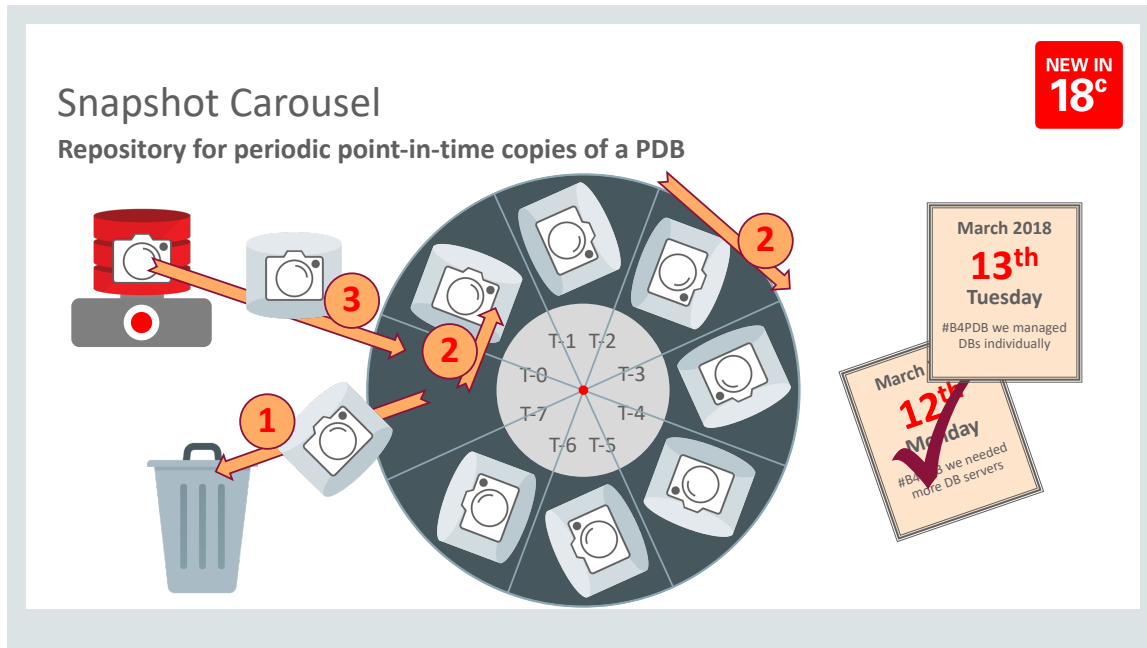


Figure 8. Snapshot Carousel

For example, if we are taking nightly snapshots of a particular PDB, perhaps we only need snapshots for the past week. This is illustrated in Figure 8. Every 24 hours, at midnight, we drop the oldest snapshot. You can think in terms of the carousel rotating by one slot. The newly vacated slot now becomes the slot for $T-0$. A fresh snapshot is taken, and this is dropped into slot $T-0$.

A great use case for Snapshot Carousel is as a source for debugging date-specific problems. For example, we might have a problem reported on Friday. It's a new problem, which seems to be data-related, which arose earlier in the week. However, the report is not specific about the exact date on which it was first encountered.

We might try a "bracketing" approach: Take clones from the snapshots from Monday, Tuesday and Wednesday and run tests to see whether the problem shows up. The problem was not reproducible on Monday, so we can eliminate that date. Further testing shows that the problem shows up already on Tuesday. Therefore, we can eliminate Wednesday too. We now drill into Tuesday in more depth. This is illustrated in Figure 9.

You'll notice how agile this process can be. These are the sorts of capabilities that modern, cloud-based development organizations require so that they can keep up with the demands of the modern business climate. Technically, snapshots can be materialized as either full clones, or as storage-efficient snapshot clones.

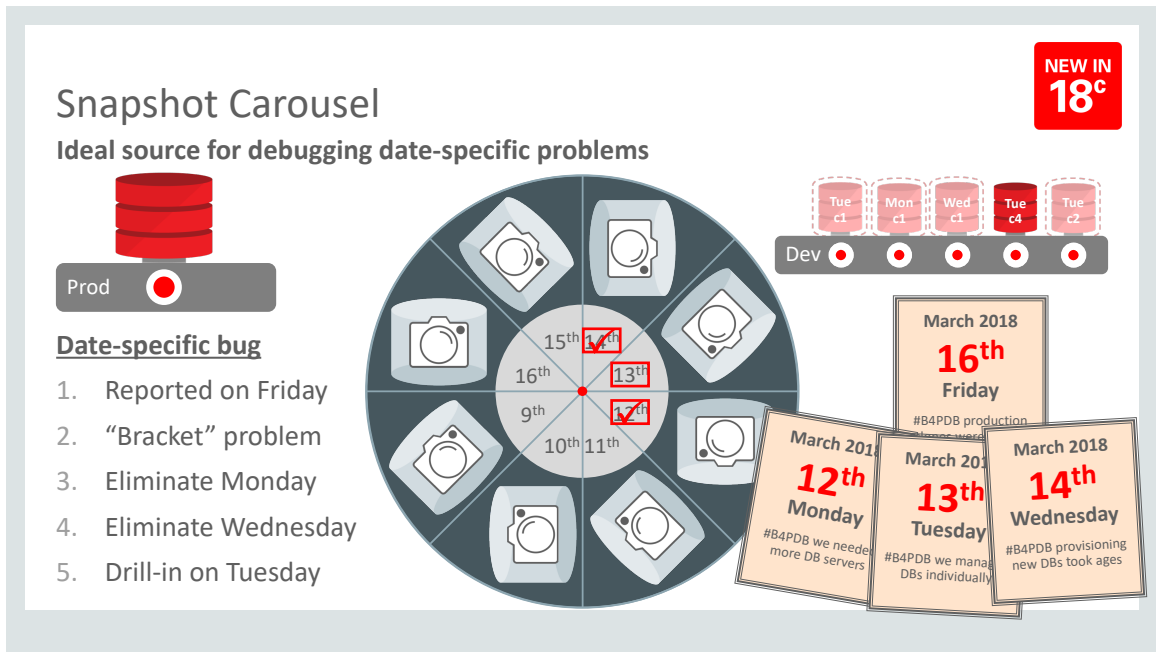


Figure 9. Using Snapshot Carousel to debug date-specific problems

Another great use case for Snapshot Carousel is as a convenient source for simple point-in-time recovery of a PDB. Let’s say a few days go by uneventfully. Business proceeds smoothly and snapshots are taken automatically every midnight. But then we discover that a serious data error was introduced on Wednesday 21st. It is determined that the simplest remedy is to recover from Tuesday’s snapshot – taken before the data error was introduced. We simply drop the current PDB and restore a clone of Tuesday’s snapshot, before the data error was made. See Figure 10.

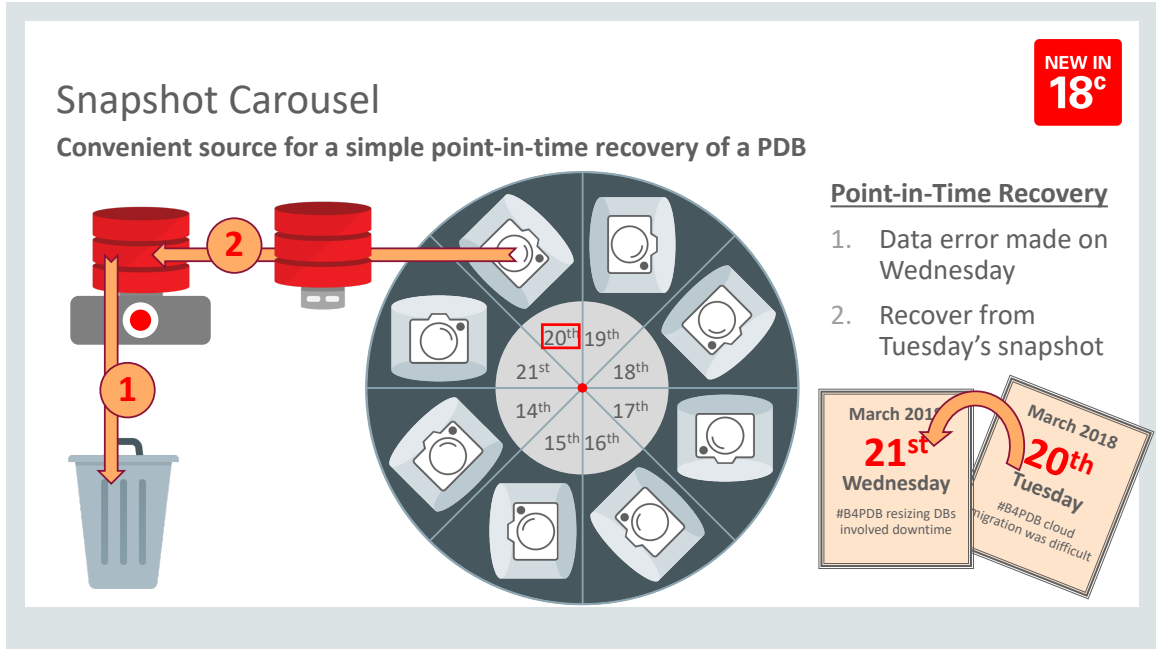


Figure 10. Snapshot Carousel as a source for point-in-time recovery of a PDB

Dynamic Lockdown Profiles

Unlike various competitive offerings, which may deliver some of the requirements of the Cloud at the expense of major compromises on other requirements, Multitenant delivers on the full promise of the Cloud, which from the point of view of a database may be summarized as **isolation and agility with economies of scale**. Focusing on this *isolation* requirement, our philosophy is *configurable isolation*. In 12.2 we introduced a very powerful capability – *Lockdown Profiles* to address this requirement. (Incidentally, we use these very extensively in various Oracle Cloud services, such as Autonomous Data Warehouse Cloud.)

Lockdown Profiles are significantly enhanced in 18c. New in this release, changes to a Lockdown Profile take effect immediately in all associated PDBs. There is no need to restart either CDB or PDBs.

Consistent with our philosophy of managing many as one, the concept is to create a few “isolation stencils” centrally. There may just be one, which would apply to all PDBs. Perhaps you’d have two or three, one of which would be the default. (Some people think of these in terms of “tee shirt sizes.”) As new PDBs are added to the CDB, they inherit the default Lockdown Profile, or a specific profile can be applied to them.

There are four major aspects to a Lockdown Profile:

- » Database Options and Features
- » Parameter Settings
- » Security Capabilities
- » Resource Manager Plans

The table at the top of Figure 11 illustrates how three Lockdown Profiles might be defined. These Lockdown Profiles are shown in the column on the left. Let’s call them Red, Mustard and Purple. At a glance, the degree of lockdown is relatively low in Red, medium in Mustard and high in Purple,

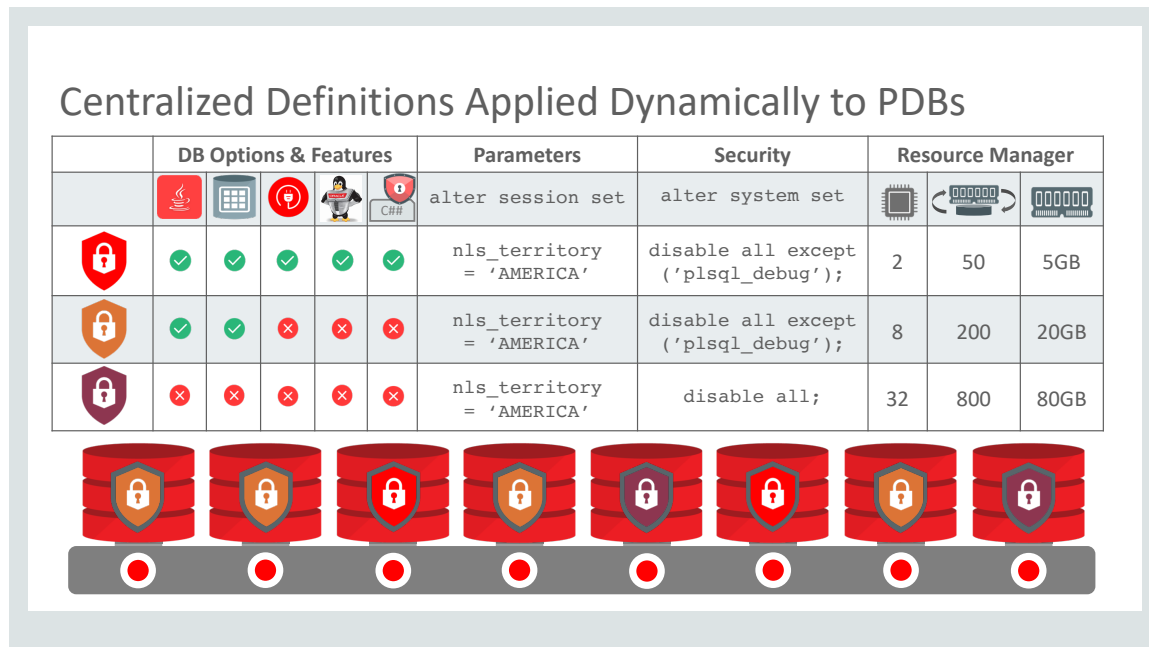



Figure 11. Simplified Illustration of Lockdown Profiles



Examining this in a little more detail, the four major categories are listed in the header row:

- » Database Options and Features
- » Parameter Settings
- » Security Capabilities
- » Resource Manager Plans

In the next row we have greater detail in each category. For example, In the category of DB Options and Features, we see: Java in the database, Partitioning, Network Access, OS Access, Common Schema Access. The Red Lockdown Profile enables all of these. Mustard enables Java and Partitioning. All are disabled in Purple. (In reality there is much finer grained control within some of these categories, which is well documented in the White Paper [“Isolation in Oracle Multitenant Database 12c Release 2 \(12.2\)”](#).)

In the next column we have the category of database Parameters. Here we see the setting for parameter NLS_Territory. It looks like this CDB is US-based, so the applicable value should be “AMERICA” for all Lockdown Profiles. This is the only parameter shown in this illustration, although there’s no limit to the number of parameter settings that can be specified.

The next column shows the category of Security settings. In this illustration we’re severely restricting the scope of “alter system”. It’s completely locked down in Purple. In Red and Mustard, all clauses are disabled except for “PLSQL_Debug”. This means that we can grant alter system to any user in an PDB subject to these Lockdown Profiles and the only aspect of “alter system” that would be effective is to set “PLSQL_Debug”. In this way Lockdown Profiles can be viewed as complementary to grants. Where a grant of a powerful privilege such as *alter system* may be seen as too broad, the Lockdown Profile can be defined in this way to restrict the scope of *alter system* to just the few capabilities that are really required. This is consistent with a best practices philosophy of granting *minimum privileges*. Another important point to make here is that these restrictions are applied at run time. Since they’re not considered during compilation time, there is no invalidation of objects involved when these restrictions are put in place.

In the final category we have settings for various parameters which drive the behavior of Resource Manager. Here we see settings for CPU Count, IOPS and SGA_Max_Size. Red is a small Lockdown Profile, with 2 CPUs, 50 IOPS and 5GB SGA. Mustard is medium, with 8, 200 and 20 respectively. Purple is large, with 32, 800 and 80 respectively.

Across the bottom, we see a CDB with several PDBs plugged into it. The various colored shields on these PDBs represent the application of the appropriate lockdown profile to each of these PDBs.

CDB Fleet Management

Ever since its introduction in 12.1, Multitenant has helped organizations greatly reduce operating costs by managing many *PDBs* as one. CDB Fleet extends this advantage by allowing many *CDBs* to be managed as one.

Organizations tend to have large estates of *PDBs* deployed across several *CDBs*, for several good reasons. The physical capacity of individual servers may require several of them to support the entire estate. Data sovereignty or latency requirements may dictate that the servers be physically distributed around the world. Occasionally the technical limit of the number of *PDBs* per *CDB* (4,096 *PDBs* per *CDB* in Exadata and Oracle Cloud; 252 *PDBs* per *CDB* on other platforms) may require multiple *CDBs* to be created within a single server. In these situations, CDB Fleet allows the entire Fleet of *CDBs* to be managed as if it were a single *CDB*.

A CDB Fleet is a group of *CDBs* managed collectively. There are two possible roles within the CDB Fleet:

» Lead CDB

Only one *CDB* in the Fleet may be designated as the Lead *CDB*. This role is designated by setting the *LEAD_CDB* database property to TRUE.

» Member CDB

A *CDB* becomes a member of a CDB Fleet by setting its *LEAD_CDB_URI* database property to identify the appropriate Lead *CDB*. This is expressed in terms of a database link. (Note that this database link must use “fixed user” semantics, which means that the user name and password are in the link definition. Database links with “connected user semantics” may not be used.)

As Member *CDBs* are added to the CDB Fleet, a Proxy *PDB* for each Member *CDB* is automatically created in the Lead *CDB*. Figure 12 illustrates how a CDB Fleet may be set up.

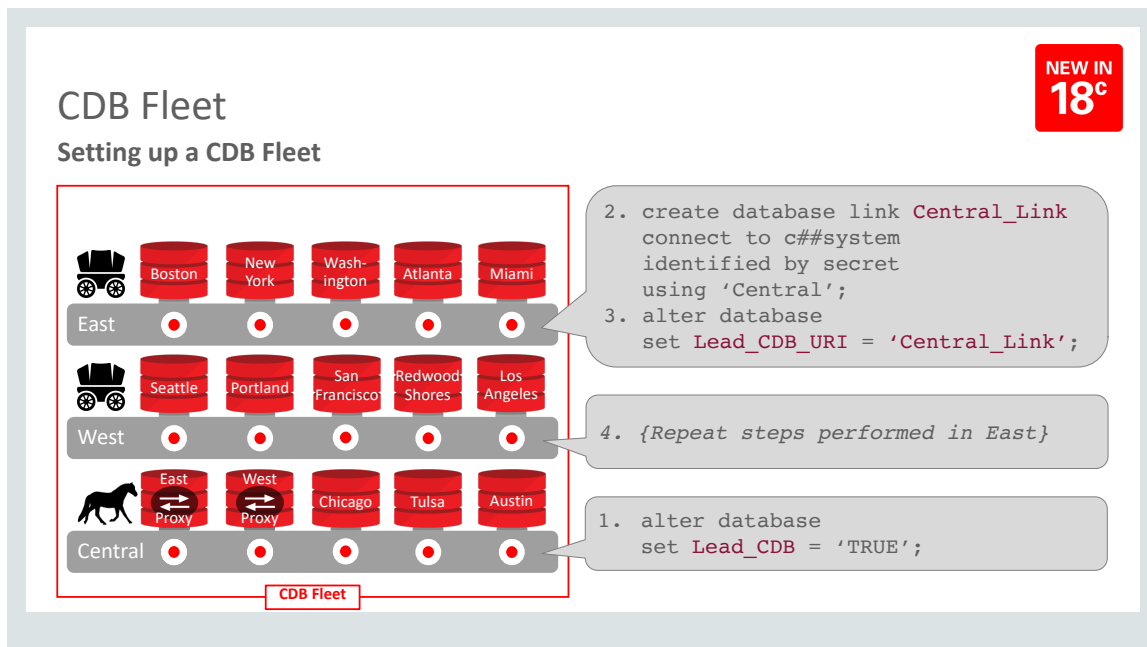


Figure 12. Setting up a CDB Fleet

These two database properties – LEAD_CDB & LEAD_CDB_URI – are mutually exclusive. Only two-level CDB Fleets may be configured; it is not possible to nest sub-fleets within larger fleets.

The CDB Fleet allows the entire estate of PDBs, physically distributed across various CDBs, to be treated as if they were all in a single CDB. There are some important corollaries to this.

1. All PDB names must be unique within the entire CDB Fleet. In other words, the CDB Fleet represents a single logical namespace for all PDBs.
2. As mentioned above, fleet membership is determined by the setting of the database property LEAD_CDB_URI. This specifies a database link that must be defined in “fixed user semantics”. This user – the *Fleet Common User* – must be defined identically in every CDB in the CDB Fleet.

Some important fleet-wide operations enabled by CDB Fleet include:

- » Queries against CDB views
- » Queries against GV\$ views
- » Containers() queries

In general, in each of these situations, these queries, executed against CDB\$Root of the Lead CDB in the CDB Fleet, will aggregate data from all PDBs in the CDB Fleet as if they were in a single CDB. The precise requirement is that data will be returned for all PDBs in which the Fleet Common User has container_data privilege for all objects or for the specific object being queried. Figure 13 shows a CDB-Fleet-wide query against a CDB View.

CDB Fleet

Managing many CDBs as one

NEW IN
18^c

```
SQL> select PDB_Name
2 from CDB_PDBs
3 order by PDB_Name;
```

PDB_NAME
ATLANTA
AUSTIN
BOSTON
CHICAGO
EAST_PROXY
LOS_ANGELES
MIAMI
NEW_YORK
PORTLAND
REDWOOD_SHORES
SAN_FRANCISCO
SEATTLE
TULSA
WASHINGTON
WEST_PROXY

15 rows selected.

```
SQL>
```

Figure 13. CDB-Fleet-wide query against a CDB View



Summary

Since its introduction in 2013, Oracle Multitenant has been widely adopted by ISVs and customers alike and deployed both on-premises and in the Cloud. Indeed, many Oracle Cloud Database Services, including Exadata Express and Autonomous Data Warehouse Cloud, rely on Multitenant for tenant isolation, agility and scalability. Oracle Database 18c introduces a number of enhancements to existing multitenant functionality including:

- » Refreshable PDB Switchover
enables extremely simple management of per-PDB replicas
- » Enhanced Integration with Data Guard
simplifies PDB provisioning operations in high availability configurations
- » Snapshot Carousel
a repository for periodic point-in-time copies of a PDB
- » Dynamic Lockdown Profiles
enable changes to isolation configurations to be applied immediately to PDBs without needing to restart either the CDB or PDBs
- » CDB Fleet Management
allows many CDBs to be managed as one

The multitenant architecture of Oracle Database 18c enables tenant **isolation and agility with economies of scale**, whether deployed in the Oracle Cloud, on-premises or hybrid cloud environments.



Oracle Corporation, World Headquarters

500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries

Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

-  blogs.oracle.com/multitenant
-  facebook.com/oracle
-  twitter.com/OraclePDB
-  oracle.com/goto/multitenant

Integrated Cloud Applications & Platform Services

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. This document is provided *for* information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0116

Oracle Multitenant: New Features in Oracle Database 18c
March 2018
Author: Patrick Wheeler, Senior Director, Product Management, Oracle Database