ORACLE DATABASE 12c

Oracle Multitenant: New Features

In Oracle Database 12*c* Release 2 (12.2)

ORACLE WHITE PAPER | FEBRUARY 2018

ORACLE

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 Database 12*c* Release 2. Available now.

Oracle Database 12*c* Release 2 (12.2), the latest generation of the world's most popular database, is now available in the Oracle Cloud and for download from Oracle Technology Network (OTN).

Table of Contents

Isolation and Agility with Economies of Scale

It's not just about scale. Scale on its own can be viewed as a negative. As organizations scale beyond a single office, communication gets more difficult and you need more layers of management. It's harder to be agile – analogies with turning around an oil tanker come to mind. These problems show how cost can increase *exponentially* with scale.

With sophisticated automation, technologies associated with first-generation clouds can improve this cost model to a nearly linear function. Let's consider a development environment, for example. To support ten developers, you might need ten virtual machines (VMs). To support twenty developers, you'll need 20 VMs. There is an intrinsic cost to each VM, even if it's not carrying any great workload. Hence the linear relationship between cost and scale. However, this was still an important step in the right direction.

Oracle Multitenant is the architecture for the next-generation Database Cloud. Multitenant changes the situation completely. Multitenant delivers true *economies of scale*. The expensive model of a VM containing a database is replaced by a pluggable database (PDB). Because there is negligible intrinsic cost to a PDB, the cost of each developer's PDB is reduced to the actual work they do. All developers' PDBs can be consolidated into a single multitenant container database (CDB) and the costs of running that CDB can be shared among those developers. In terms of compute resources that's because there's a single set of background processes and a single shared memory area (the SGA). In terms of administration, that's a single CDB to be backed up, configured for high availability, patched, etc.

Economies of scale through consolidation are of limited use if that consolidation comes at the expense of isolation and agility.

In the multitenant architecture of Oracle Database 12*c* Release 2 (12.2) we build on what was already a powerful suite of isolation capabilities to deliver a comprehensive model, which can simply be configured to deliver precisely the appropriate level of isolation for a particular use case. With great isolation it becomes realistic to consolidate at great scale because the individual workloads being consolidated do not interfere with each other.

Agility puts in positive terms the avoidance of cumbersome consequences of consolidation conveyed by the analogy of turning the tanker. Scale can reduce unit costs through vast purchasing power but those gains are offset if scale comes at the expense of agility. In database terms, agility is about the speed and efficiency of creating, cloning (and even destroying) databases. It's about moving them

between servers, between data centers, between data centers and the Cloud, and moving them around in the Cloud. This can be for all sorts of reasons; as equipment is retired and replaced with capacity in the Cloud, for load balancing, for changing service levels or for handling peak loads. These are essential capabilities for the modern enterprise and the sophisticated functionality of Multitenant makes them as frictionless as possible.

The key point about economies of scale is that if the savings outweigh the costs of scale, there is true return on investment. The beauty of Multitenant is that the economies of scale are so great that the return on investment is far more readily achieved than with other models. Multitenant delivers true economies of scale, and the two go hand-in-glove.

Multitenant delivers ***isolation and agility with economies of scale***.

Oracle Multitenant in 12.1

With Oracle Multitenant, we are in the paradoxical situation of having a fully functional product in its previous incarnation and at the same time having a huge amount of new functionality in the new release. Before we delve into some of these new capabilities, let's briefly summarize the existing architecture of Multitenant in Oracle Database 12*c* Release 1 (12.1) and its major advantages.

With Oracle Multitenant, multiple pluggable databases (PDBs) may be consolidated into a single multitenant container database (CDB). A pluggable database is a self-contained, fully functional Oracle Database. From an application's point of view it has not changed in any way, and that's very important because it means that no application changes are required to adopt this architecture. So, from an application's point of view, the PDB is the database. However, from an operational point of view it's the CDB that's the database. The CDB represents a single, consolidated operating environment. There is a single set of background processes and a single shared memory area (SGA), shared by all of the PDBs in the CDB. This architecture eliminates replication of overheads, making most efficient use of available resources. What this means is that you can minimize capital expenses (CapEx) because you can consolidate more applications per server. From an operational perspective, you can manage all these consolidated PDBs collectively, greatly reducing operating expenses (OpEx). This applies to things like backups, configuration of high availability, application of patches and upgrades. These CapEx and OpEx reductions are part of how Multitenant delivers on the promise of Cloud computing.

This architecture is particularly well suited to three prominent use cases:

Consolidation

A few years ago an important consideration was whether servers (and the software running on them) were capable of "scaling up" to manage the workload. With the massive power of modern converged infrastructure such as the Oracle Exadata Database Machine, and indeed Cloud environments comprising many such machines, one begins to hear the term "scale down". That is, even with significant applications, it's often quite possible to consolidate several into a single server. The trick, to deliver economies of scale, is to be able to "scale down" these servers to support the various workloads efficiently, without over-provisioning, but retaining the isolation that was the reason to run them standalone in the first place. Multitenant is ideal for this with the intrinsic isolation of PDBs. Resource Manager integrates perfectly with this, allowing for the definition of simple-yet-powerful policies governing the

distribution of resources between PDBs. In 12.1 we can consolidate up to 252 pluggable databases into a single multitenant container database. 252 PDBs per CDB - that's a lot!

Development and Testing

Agility, such as is required in a modern development and testing organization, is another key characteristic of cloud computing. While managing many databases as one, we retain granular control when appropriate and thus we can increase agility. Provisioning of PDBs is extremely fast and simple, and lends itself well to self-service provisioning interfaces – a key characteristic of Database as a Service (DBaaS). Provisioning of PDBs is accomplished by a process known as cloning, of which there are many varieties including full clones, subset clones and even thin provisioning of snapshot clones (leveraging copy-on-write technology). A pluggable database is a portable database – it's very simple and quick to move PDBs between servers using a process known as unplug/plug. Use this for things like load balancing and migration to and from the Cloud.

Seamlessly integrated with Oracle Real Application Clusters (RAC), with the ability to affinitize a PDB to a specific node in the cluster, or to run uniformly across multiple nodes, we have the ability to adjust flexibly to changing workloads. This, along with the simple, self-service provisioning just mentioned, are other key characteristics of a database cloud.

Software as a Service

Software as a Service (SaaS) is a classic example of cloud computing and correspondingly a classic use case of Oracle Multitenant. Multitenant delivers an instant SaaS architecture for what we might characterize as "demoralized Y2k era application vendors". These applications are typically functionally very rich, but, architected for an outmoded, standalone, on-premises deployment model, they can't compete with modern SaaS architectures, even if they're functionally superior. Multitenant dramatically changes the rules of the game. By allowing each tenant to be isolated within a PDB, but with multiple tenants consolidated within a CDB, these applications can be simply redeployed in a SaaS configuration, with no changes required!

New Features in 12.2

This is the extremely strong foundation on which we developed all the new functionality that we're delivering with the new release of Oracle Database 12*c* – Release 2 (12.2). There are significant enhancements in each of these three areas. The remainder of this White Paper is devoted to these various enhancements.

For more detailed information about Oracle Multitenant in 12.1, the reader is recommended to consult the Oracle Multitenant White Paper for 12.1 (2013).

Provisioning Enhancements

Provisioning of databases is a traditional pain point for database administrators (DBAs). The relentless pressure on the modern organization to innovate translates into continuous demand on development teams to deliver new systems that feature the capabilities modern customers demand, including mobile capabilities, social interactions and Cloud ubiquity. All this, while complying with regulatory requirements and the stringent security that is a basic survival skill in this contemporary environment of pervasive internet hacking. Layered on top of all this are fundamental technical requirements of scalability, availability, performance, manageability and productivity – the traditional and unmatched strengths of Oracle Database.

The modern trend of "DevOps", featuring continuous development, continuous integration and continuous delivery, exacerbates this pressure as the fundamental layer of the development environment with all of the attributes

described above is an Oracle Database. As previously discussed, Oracle Multitenant dramatically changed this situation in 12.1. Provisioning of new databases with Multitenant is a simple matter of cloning PDBs. When incorporated in a modern DBaaS model, provisioning databases transforms from a process that may have taken days or weeks (sometimes even months!) and involving staff from multiple teams, into one of a few minutes or even seconds, executed through simple self-service interfaces.

In 12.2 we deliver significant enhancements to these basic provisioning operations by enabling them to be performed on-line.

Clone PDB

In 12.1, to clone a PDB, the source PDB must be quiescent for the duration of the clone operation. Colloquially we refer to this as a "cold clone" operation because it involved an outage (from a transactional perspective) in the source database. It's a simple matter to say that a "hot clone" would be preferable, but it's easier said than done! However, in 12.2, we support hot clone. This on-line cloning capability is available on all storage supported by Oracle Database.

A hot clone can be taken while the source PDB is still open read-write. What that means is that a hot clone can be taken without interrupting operations in the source PDB. No application outage is required for a hot clone.

There follows a brief technical explanation of what happens "under the covers" during a hot clone. These are provided for the curious reader. For those uninterested in these minutiae, you can skip to the next section, *Refreshable PDB*.

Technically what we're doing in a hot clone is known as a "fuzzy read" of all the blocks in the datafiles of the source PDB. What this means is that, if the clone operation begins at time *t0*, by the time the last of the blocks in the source PDB have been read and copied to the target (time *t1*), some changes may have been made to some of the blocks already copied. At this stage, then, the clone may be *physically inconsistent* with the source.

The next step is to copy all the redo that has accumulated for that source PDB between times *t0* and *t1*, and transport this to the target. This redo is then applied to the target PDB. At this stage, the target PDB will be an exact physical copy of the source PDB as of time *t1*. However, this includes both committed transactions and uncommitted transactions, and therefore should be considered to be potentially *transactionally inconsistent*.

What is required now is to rollback the uncommitted transactions. This is achieved by applying the undo for all these uncommitted transactions. This is the final stage of the clone operation. The resultant clone will be a transactionally consistent copy of the source PDB as of time *t1*. That is, all committed transactions in the source PDB as of time *t1* will be present in the clone, and all uncommitted transactions will have been rolled back. It will be seen that a key enabling capability for hot cloning is local undo. This new feature of Oracle Multitenant is described elsewhere in this paper. Archive logging must also be enabled in order to perform hot clones.

All of this is performed *atomically* in the familiar clone operation

```
create pluggable database target from source;
```

The distinction between a hot clone and a cold clone is only relevant for customers familiar with the capabilities of 12.1. As of 12.2 we will refer simply to clones (and all clones are hot clones).

Refreshable PDB

The Refreshable PDB capability builds upon hot clone. A classic use case here would be development and testing, in which we'd want a copy of a production PDB. Production databases tend to be large. The larger they are, the longer they take to copy. A production database in the tens of terabytes scale may take in the order of days to copy. It is undesirable to do that sort of thing very often, and therefore development copies of this sort of production database tend to grow rather stale as they age. The staler the database, the less relevant it is for development purposes, such as debugging. Refreshable PDBs eliminate this problem.

The first step is to take a full clone of the source database. That takes as long as it takes – hours, days, whatever. Remember that now that we have a hot cloning capability, it doesn't really matter how long this clone takes because it doesn't require any downtime in the source database.

Once the initial clone is complete, the database is available for use. The intended use case of Refreshable PDB is as a "golden master" database. That is, a database from which individual developers take clones – typically, and most efficiently snapshot clones (which can be created in seconds or minutes even for very large databases). Snapshot clones are dealt with elsewhere in other White Papers available from Oracle. As a "golden master" database, this Refreshable PDB should remain open read-only between refreshes. The PDB must be closed during the refresh operation itself.

Now, as the clone becomes stale, we can refresh it. We do this by applying all the redo that's accumulated since it was last refreshed. Even if the source database is enormous, the incremental redo will typically be much smaller. It's therefore going to be a much, much quicker process than the initial hot clone. Therefore, it's going to be much simpler to keep production clones refreshed with recent data copied from production.

Refreshable PDBs may be defined to be refreshed either automatically (on a specific schedule) or manually (refreshed on-demand). It is also possible to perform an on-demand refresh of a PDB defined to be refreshed automatically. A PDB defined to be automatically refreshed can be transitioned to being manually refreshed, and vice versa.

To create a PDB for automatic refresh, the syntax is

```
create pluggable database Golden_Master_PDB from Prod_PDB@DBLink
refresh mode every 360; -- (360 minutes)
```

The Refreshable PDB should then be opened read-only as follows:

```
alter pluggable database Golden_Master_PDB read only;
```

For manual refresh, the syntax is

```
create pluggable database Golden_Master_PDB from Prod_PDB@DBLink
refresh mode manual;
```

As before, the Refreshable PDB can be opened read-only when the initial clone operation is complete. Subsequent refresh operations require the PDB to be closed, and can be achieved with the following statement:

```
alter pluggable database Golden_Master_PDB refresh;
```

Note: As described in the Multitenant White Paper for 12.1 (2013), important steps are performed the first time a PDB is opened read-write after creation. In general, then, this first *open read-write* operation can be viewed as an important step to complete the creation of the PDB. It is important to note that this does not apply to Refreshable PDBs. In fact, it is *important not to open a Refreshable PDB read-write, as it may no longer be refreshed thereafter.*

PDB Relocate

A pluggable database is a self-contained, fully-functional Oracle database. A pluggable database is a portable database; hence the term “pluggable”. This portability is achieved in 12.1 by unplugging the PDB from one CDB and plugging it into another. This is a very powerful capability. It makes a PDB the ideal vehicle to the Cloud; customers may achieve this simply by unplugging their PDBs from CDBs in servers in their data centers and plugging them into an Oracle Database Cloud Service, or perhaps other servers in the customer’s estate. Unplug/plug is also a simple way of performing load balancing, perhaps by moving a PDB from a heavily-loaded server to one with more available capacity.

However, moving a PDB between CDBs with the unplug/plug method does involve an outage during the unplug/plug operation. When moving between servers with shared storage, such as is typically the case in Cloud environments when the PDB is being moved for load balancing, this outage can be very brief because it’s only a metadata operation – there’s no need physically to move the datafiles themselves. However, in moving a database between data centers, or from “ground to cloud” (or the reverse), laws of physics apply – all the data must be physically moved. For large databases, depending on the capabilities of the underlying network, this process may take considerable time. It’s possible that the associated outage be problematic.

Service Level Agreements (SLAs) typically feature two conflicting aspects – performance and availability. If a database is in danger of violating its performance SLA, the solution is typically to move it to a server with more available processing power. However, to do so traditionally involves an outage in the application, and this is likely to violate the availability component of the SLA.

PDB Relocate is a new method of moving a PDB between CDBs. This is introduced in 12.2 to allow Oracle Database Cloud Services to meet both performance *and* availability components of SLAs with customers. Because Oracle Database Cloud Services are built on the very same database technology that our customers use, this capability is generally available to customers too. The goal is to eliminate the outage completely.

There follows some more detailed information about PDB Relocate. Readers for whom this information is less relevant may skip to the next section.

Technically, PDB Relocate is built “on top” of Refreshable PDB, which as we’ve seen is in turn built on top of hot clone. When independent listeners are involved, transparent connection forwarding may also be involved.

Operationally, PDB Relocate is achieved in two steps, as follows:

1. Relocate PDB to the new server
2. Open PDB in its new location

Step 1 involves cloning the PDB from its original location to its desired new location. In other words, at the instant when this step is complete, there will be two transactionally consistent copies of this PDB, one in the source server and one in the target server. For the duration of the operation, processing continues uninterrupted on the database in its original location. Users of an application or applications connected to the database will be unaware that a relocation is underway.

Technically, this is essentially the same as creation of a Refreshable PDB, as described above. All existing application connections, and new connections made during this step, continue to be to the PDB in its original location. As we have seen, if Step 1 begins at time *t0* and completes at time *t1*, at time *t1* the contents of the two PDBs will be transactionally consistent. As processing continues it is important to account for the potential for data changes in the PDB in its original location beyond *t1* before connections are switched to the new location.

Step 2 completes the relocation. Again, it's largely transparent to the application user, although at the end of the operation, connections to the PDB will have been switched from its original location to the new location. An important component of this step is to ensure that all transactions committed in the PDB in its source location are preserved in the target location.

Technically, as we've seen, we have to account for any transactions that are committed to the PDB in its source location after time *t1* before processing resumes in its new location. At a high level, all transaction processing must first cease in the PDB in its source location, an instant we can label time *t2*. We must reapply all the committed transactions that have accumulated in the PDB in its source location to the target location before transaction processing can resume there. With these requirements in mind, there are the following sub-steps during this open phase.

2a. Open read-only. We know that the PDB is transactionally consistent in its new location as of time *t1* and therefore it is safe to open it read-only. It is thus immediately available there for query purposes, while connections attempting DML will "spin". Even though it's possible that at this stage some committed transactions are not visible in the target location, they will not be lost (as we shall see). The behavior of any queries against the PDB in its target location – and it's only queries that are allowed at this stage – is exactly as it would have been had they been performed in the PDB in its source location at time *t1*. This may be seen as a special case of read consistency.

2b. Connection forwarding. All connections must be "drained" from the source location and re-established in the target location. New connection requests are forwarded to the target location and made to the PDB there. Modern, well-designed applications typically interact with the underlying database via a connection pool, which can support large numbers of application users via a small number of (relatively expensive – in terms of computing resources) database connections. As transactions complete and these connections are released to the connection pool, the connections are closed. Longer running transactions or poorly written applications may result in connections which are not naturally released in a timely manner. After a brief interval, these connections are automatically terminated and re-established at the destination. This process is performed incrementally, so as to avoid a login "storm" at the destination.

The implication here is that some transactions may be interrupted during the PDB relocation. Note that this is not inconsistent with the statement that the PDB's overall availability is nearly continuous. For this reason, it important for an administrator to be sensitive to the processing windows of these long-running transactions so as to minimize the impact of PDB relocations.

The details of connection forwarding depend on the configuration of listener(s) at the PDB's source and target locations. There are three possibilities:

2.b.i. Shared listener. This situation would typically involve a relocation between CDBs in the same server, such as would apply if the PDB is being relocated to a CDB with a different set of options installed. In this case the PDB is re-registered with the listener in its new location.

2.b.ii. Cross-registered listeners. This situation is typical of relocation between servers within a data center, perhaps for load balancing purposes. In this case the PDB is re-registered in its new location with both listeners.

2.b.iii. Independent listeners with no cross-registration. This will typically be the situation when relocating a PDB between data centers; perhaps from a data center on a customer's premises to an Oracle Database Cloud Service. With no cross-registration, this situation is complicated by the fact that the listener in the target location has no "knowledge" of the source CDB and vice versa. In this case we offer two levels of availability; "high" and "maximum". High availability is aptly named because indeed the PDB is more or less immediately available in its new location for direct connections. The syntax for a high availability PDB relocation is as follows:

```
create pluggable database My_PDB from My_PDB@DBLink relocate;
```

This statement should be issued from the target CDB. (High availability is the default.)

Maximum availability goes a step further by automatically forwarding connections made to the PDB in its original location to its new location. The syntax for a maximum availability PDB relocation is as follows:

```
create pluggable database My_PDB from My_PDB@DBLink relocate availability max;
```

This statement should be issued from the target CDB.

It is now important to guard against the possibility of creating another PDB in the original location with the same name as the PDB that has been relocated. If this were allowed to happen it could result in a very confusing situation indeed! To avoid this, when performing a maximum availability PDB relocation, we create a "tombstone" in the source CDB. This is a metadata entry in the source CDB, which may be seen by querying v$containers from the Root container of the source CDB. The "tombstone" PDB will be visible, with a status of "relocated".

2c. Application of incremental transactions in target location. Any redo data that has accumulated for that source PDB between times *t1* and *t2* is copied to the destination and applied there. At this stage, the target PDB will be an exact copy of the source PDB as of time *t2*. However, this includes both committed transactions and uncommitted transactions. What is required now is to rollback the uncommitted transactions. This is achieved by applying the undo for all these uncommitted transactions. Because we have ensured that no transactions have occurred in the origin, we can see that the destination has now caught up.

2d. Open destination read-write. The datafiles are now deleted from the origin.

The PDB relocation is now complete. All processing resumes in the PDB's new location as if nothing had happened, although as we've seen, a great deal has happened! However, all the operator had to do was issue a couple of SQL statements.

- There was no need to suffer an application outage.
- There was no need to change any applications.
- There was no need to change any connect strings.

It just worked.

Note that, in the case of PDB relocation between independent listeners with no cross-registration, the presence of the "tombstone" PDB at the origin implies that it is not immediately possible to relocate the PDB back to its original container, at least not with its original name (because the tombstone is "occupying" that namespace). It is important to realize, however, that this is not expected to be a permanent situation. As described above, a maximum availability PDB relocation involves automatic connection forwarding. This enables clients to continue to connect to the application database without having to make any connect string changes. However, this does involve an extra network "hop". It is anticipated that, at some convenient juncture, application connect strings can be changed to provide direct connections to the database in its new location. At this point, it is possible simply to drop the tombstone PDB from the origin. Thereafter, there would be no impediment to relocating the PDB back to its original container if so desired.

Limitations of PDB Relocate

In general, PDB Relocate is subject to the same plug compatibility constraints as unplug/plug. This subject is well covered in the Multitenant White Paper for 12.1 (2013). Essentially the plug compatibility of a PDB is checked by evaluating the PDB's xml manifest file against the destination CDB. In the case of PDB Relocate, however, the xml manifest file should not be created by performing an unplug operation (because PDB Relocate is an on-line

operation). Instead, the manifest file should be generated by connecting to the PDB in its original location and issuing the command

```
execute DBMS_PDB.Describe() into PDB_manifest.xml;
```

Enhancements to Unplug/Plug

Migrating Encryption Keys

Modern Cloud deployments should include a comprehensive security strategy, such as is laid out in Oracle's Maximum Security Architecture, which is covered in separate publications from Oracle. Our security philosophy is "defense in depth" with a comprehensive strategy featuring preventive, detective and administrative components. No single technical component provides universal protection; however, a fundamental capability is provided by Oracle Transparent Data Encryption (TDE). In Oracle Database Cloud Services, TDE is not optional – all databases in our Database Cloud Services are protected by TDE. Many of our customers, who embrace a "hybrid cloud" philosophy, want to "build their Cloud like we built our Cloud", and it follows that they have TDE in place too. Accordingly, a methodology is required to migrate per-PDB encryption keys between CDBs during unplug/plug operations.

In 12.1, this export and import of keys was achieved separately from the PDB unplug/plug operation. This procedure is documented in the Multitenant White Paper for 12.1 (2013). In 12.2 the migration of encryption keys is simplified through integration with the unplug/plug operation itself. Technically the encryption keys are transported via the xml manifest file, where they are encrypted by a shared secret. This is achieved with some additional clauses in the unplug and plug statements as follows:

Unplug: Executed as SysDBA in Root container of source CDB:

```
alter pluggable database MyPDB unplug into MyPDB_manifest.xml
encrypt using <shared secret>;
```

Plug: Executed as SysDBA in Root container of destination CDB:

```
create pluggable database MyPDB using MyPDB_manifest.xml
decrypt using <shared secret>;
```

Self-Service Provisioning with Auto-Login Wallets

A key characteristic of Database as a Service (DBaaS) is self-service provisioning. As we've seen, the container database architecture of Oracle Database 12*c* represents by far the most efficient model for deploying DBaaS. PDBs are not only the vehicle to the Database Cloud; they are also the enabler of the Database Cloud. This is why we often refer to *Pluggable* Database as a Service or *P*DBaaS. By definition, PDBaaS does not lend itself to password-based wallets, as these are not designed for unattended operation. For this reason, TDE can also be operated via "auto-login wallets. These operations, while supported in 12.1, are simplified in 12.2.

PDB Archive

Unplug/plug provides a simple, fast and efficient means of moving a PDB between CDBs. The basic syntax of the unplug operation is

```
alter pluggable database MyPDB unplug into PDB_manifest.xml;
```

where PDB_manifest.xml is referred to as the PDB's "xml manifest file". When moving a PDB between CDBs with no shared storage, it is necessary physically to copy this xml manifest file, along with all of the PDB's datafiles, to the destination server.

This syntax continues to be supported in 12.2. In addition, a new capability is introduced, to combine the manifest file and all the datafiles into a single file, known as a PDB archive. This PDB archive always has the extension .pdb, and this can be achieved simply by specifying the archive file (with the .pdb extension) in the familiar unplug statement, thus:

```
alter pluggable database MyPDB unplug into MyPDB_archive.pdb;
```

Correspondingly, when connected as a common SysDBA to the Root container of the destination CDB, plug in the PDB from its archive file by using the command:

```
create pluggable database MyPDB using MyPDB_archive.pdb;
```

When to use PDB Relocate and when to use Unplug/Plug

Unplug/plug remains a powerful operation, fully supported and even enhanced in 12.2. However, as described above, the PDB Relocate facility introduced with 12.2 provides another means by which to achieve the same result. The great advantage of PDB Relocate is that it is an on-line operation – that is, the PDB migration does not require an application outage. This is not the case with unplug/plug. It is anticipated therefore, that the majority of PDB migrations between CDBs will be performed using PDB Relocate. There remain, however, a few circumstances in which unplug/plug is the preferred method. These include:

- Moving a PDB between CDBs of different versions or patch levels of Oracle Database
- Moving a PDB between CDBs with different sets of options installed

Enhancements to Cloning

The publication of this White Paper presents an opportunity to list various enhancements to Multitenant cloning introduced since the initial release of 12.1.

Subset Clones and Metadata-Only Clones

On occasion it is desirable to include a subset of a PDB's tablespaces when creating a clone. This is fully documented elsewhere but in essence is achieved by use of the *user_tablespaces()* clause of the *create pluggable database* statement, thus:

```
create pluggable database NewPDB from OldPDB user_tablespaces(TS1, TS2);
```

Technically the result of this is comparable to having the datafiles associated with any excluded tablespaces being offline.

Customers upgrading to Oracle Multitenant from schema consolidation find this capability very useful, as a typical model is to have dedicated tablespaces for each schema. For example, when upgrading from a database in which three schemas, Schema1, Schema2 and Schema3 are consolidated with dedicated tablespaces Schema1_TS, Schema2_TS and Schema3_TS respectively, the upgrade to Multitenant would be achieved by the following sequence of operations (assuming a starting point of a non-CDB already upgraded to 12.2):

```
-- (In the non-CDB.) Create manifest file.
execute DBMS_PDB.describe(non$cdb@DBLink) into Non_CDB.xml;

-- (In CDB Root as a common SysDBA.) -- Adopt multi-schema DB as a PDB.
create pluggable database MultiSchema using Non_CDB.xml;
alter pluggable database MultiSchema open;
```

```
-- (While connected to PDB MultiSchema.) Complete conversion of PDB's metadata.
@NonCDB_to_PDB.sql

-- (In CDB Root as a common SysDBA.) Create PDBs from schemas.
create pluggable database PDB1 from MultiSchema user_tablespaces(Schema1_TS);
create pluggable database PDB2 from MultiSchema user_tablespaces(Schema2_TS);
create pluggable database PDB3 from MultiSchema user_tablespaces(Schema3_TS);

-- Close and drop multi-schema PDB because it has now served its purpose.
alter pluggable database MultiSchema close;
drop pluggable database MultiSchema including datafiles;
```

In these steps the PDBs for each schema are created. It remains to connect to each PDB in turn and get rid of the metadata for the unused schemas. For example, connect to PDB1 and issue the following statements:

```
drop user Schema2 cascade;
drop user Schema3 cascade;
```

The extreme case of a subset clone is a metadata-only clone. The syntax for creating a metadata-only clone is:

```
create pluggable database DevPDB from ProdPDB@DBLink no data;
```

This capability might be used to create a "quick and dirty" clone of a production database for debugging purposes.

Snapshot Clones

A very powerful variety of PDB clone is the snapshot clone. Snapshot clones behave as full data set clones of a source PDB, while consuming negligible incremental storage. Because so little data need be physically written during the creation of a snapshot clone, the creation process is extremely fast. The basic syntax involves addition of the clause *snapshot copy* to the standard cloning syntax, for example:

```
create pluggable database Snapshot_PDB from Source_PDB snapshot copy;
```

The concept of snapshot clones is well described in the Multitenant White Paper for 12.1 (2013) and it's not productive to repeat that explanation here.

However, the publication of this White Paper presents an opportunity to update the list of storage technologies on which snapshot cloning is supported. These storage technologies fall into three categories, each of which has different characteristics:

Storage-Based Snapshots

Certain storage technologies have native support for storage-efficient copies of files. Oracle Multitenant can take advantage of the full power of these underlying capabilities without the compromises involved in introducing an unnecessary additional proprietary layer. We refer to these as "storage-based snapshots", which are currently supported on the following platforms:

- Oracle ZFS Storage Appliance
- Oracle ASM Cluster File System (ACFS)
- NetApp™
- EMC™

A key characteristic of storage-based snapshots is that the source PDB may remain read-writeable despite being the basis for snapshot clones.

File System Based Snapshots, Enabled by Sparseness

In some cases, databases are deployed on simple file systems rather than specialized storage technology. Most modern file systems support sparseness. In these cases, by setting CDB initialization parameter CloneDB=TRUE, snapshot clones are still possible. In these cases, there is a restriction that the source PDB remain read-only and unchanging for the lifetime of any snapshot clones. In practice, this restriction tends not to be particularly burdensome, because the typical use cases of snapshot clones tend to involve read-only clone sources anyway:

- The use of refreshable PDBs as "golden master" databases in development and testing environments is described elsewhere in this document.
- Another popular use of snapshot clones is in provisioning "sandbox" environments for destructive what-if analysis.

ASM on Exadata Storage

Oracle Automatic Storage Management (ASM) on Exadata Storage also supports creation of snapshot clones of PDBs. These also have the restriction that the source PDB remain read-only and unchanging for the lifetime of any snapshot clones.

Local Undo

In 12.1, there is global or shared undo for the entire CDB. With shared undo, before performing operations such as (cold) clone or unplug, it is necessary for the database to check for any uncommitted transactions in the source PDB. This is to avoid problems with transactional consistency in the PDB after the clone or plug in operation. (If any uncommitted transactions exist, the following error is issued: *ORA-65126 – Pluggable database was not closed cleanly and there are active transactions that need to be recovered.* In such cases it is necessary to open the PDB read-write, and wait until the SMON process can clean them up.)

In 12.2 we introduce per-PDB or local undo. Local undo avoids these difficulties and thereby significantly improves the predictability of these important operations. Additionally, it enables many of the major new capabilities of Multitenant in 12.2, including:

- Hot Clone
- Refreshable PDB
- PDB Relocate
- Flashback PDB

Shared undo is still supported in 12.2, but that is primarily for upgrade transitional purposes only. Arguably there are minor technical and management overheads to having per-PDB undo but these are vastly outweighed by the benefits of having local undo (enabling all the new capabilities described in this White Paper and elsewhere). Simple rules are very often best. In the absence of a really strong case for shared undo in a specific situation, a very good simple rule to follow is "transition to local undo ASAP in all cases". By default, Database Configuration Assistant (DBCA) creates new CDBs with local undo enabled.

Undo mode – whether it's local or shared – is an all-or-nothing property of the entire CDB. There's no half-way situation. Either all the PDBs have local undo or there's shared undo for the entire CDB.

There are capabilities to switch between local and shared undo and back again. This transition requires the CDB to be restarted.

When moving a PDB from a CDB with shared undo to one with local undo, the required local undo tablespace(s) are created automatically. When moving a PDB from a CDB with local undo into one with shared undo, the local undo tablespace will briefly be used for rollback of uncommitted transactions the first time the PDB is opened read-write.

Thereafter it is not required and can be dropped. The same applies when a CDB is transitioned from shared undo to local undo or vice versa.

A local undo tablespace is required for each node in an Oracle Real Application Clusters (RAC) cluster in which the PDB is open. If a PDB is moved from 2-node RAC to 4-node RAC (and opened in all nodes) the additional required undo tablespaces are automatically created. If the PDB is moved back again, two will be redundant and can be dropped.

Notes:

1. It is not necessary to set CDB-level database parameter *compatible* to 12.2 to enable local undo.
2. Undo mode (shared or local) is not considered a plug-compatibility issue.

Enabling Scale – Eliminating Barriers to Consolidation

Much has been said about how Multitenant delivers true economies of scale, and the two go hand-in-glove. The more you consolidate, the more you save. It follows, then, that it is very important to eliminate any barriers to consolidation.

It's fair to say that in 12.1, there were a few capabilities of Oracle Database that were not available with the Multitenant architecture. Collectively these could be viewed as obstacles to large-scale consolidation. In 12.2 these have been eliminated. In this section we'll review some of the important capabilities of Oracle Database that are now available with Multitenant in 12.2.

Flashback PDB

Flashback PDB is now fully supported.

Per-PDB Character Sets

In 12.2 Multitenant supports consolidation of PDBs with diverse character sets.

The requirement is that the CDB be configured with AL32UTF8. However, it is now possible to consolidate PDBs with diverse character sets in a single CDB. Character strings are handled properly in each PDB. Furthermore, when performing cross-container aggregation, perhaps with the *containers()* SQL construct, character strings from these diverse PDBs are still handled properly. In other words, it is no longer necessary to convert non-CDBs to AL32UTF8 (using DMU) before consolidating them with Multitenant.

For example, we could have a CDB with character set AL32UTF8, and consolidate into it:

- PDB_Japan with the Kanji character set;
- PDB_EMEA with the ISO8859_p1 character set;
- Along with any number of other PDBs.

4k PDBs per CDB on Oracle Cloud and Oracle Exadata

252 PDBs per CDB was a decent number in 12.1, but in 12.2 on Oracle Cloud and Oracle Exadata we're increasing this 16-fold to 4,096 (that is, 4k). It's more than a flippant slogan to say that with Multitenant, the more you consolidate, the more you save. We want to support massive scale, for massive savings. This sort of scale is realistic for typical SaaS deployments and is based on experience with implementations of Multitenant with some of Oracle's own SaaS applications running in the Oracle Cloud, such as Oracle Taleo Business Edition (TBE). In other platforms, the limit remains 252 PDBs per CDB.

Automatic Workload Repository (AWR) Data at PDB Level

This allows for granular diagnosis of performance within a particular PDB. It is particularly important in DBaaS environments, where one might expect an autonomous *Pluggable* Database Administrator – a *P*DBA – to have responsibility for performance within that PDB.

Heat Map

Heat Map is an important capability introduced with Oracle Database 12*c*. In 12.2, Heat Map fully supports Multitenant, which means that all of the storage efficiencies enabled by Oracle Automatic Data Optimization (ADO) are now available with the container database architecture.

Isolation

Economies of scale through consolidation are of limited use if that consolidation comes at the expense of isolation. In 12.2 we introduce some very sophisticated capabilities in this area that can ensure great isolation between PDBs and avoid what is colloquially referred to as "the noisy neighbor problem". Importantly, this is configurable so that the level of isolation can be tailored appropriately for the use case. In general when considering the topic of PDB isolation – one that is especially important in a highly consolidated environment such as a Database Cloud – we need to consider all of the potential risks of sharing. These fall into several categories:

- Contention for shared computing resources
- System access
- File system access
- Network access
- Common User or Common Object Access
- Administrative Features

In 12.2 we build on what was already a powerful suite of isolation capabilities to deliver a comprehensive model, which can simply be configured to deliver precisely the appropriate level of isolation for a particular use case.

These capabilities will be explored in more detail shortly, but before then it is important to understand why a "one size fits all" approach to isolation is not appropriate for the Database Cloud, and why the true requirement is configurable isolation. When considering this topic, it's helpful to consider a familiar real-world analogy: residential security. At first thought, one might think that the more security the better, but in reality security is a trade-off with convenience. "Maximum Security" is a phrase associated more with a prison than with a home. A home might be more secure with bars on the windows, surrounded by a high wall topped with barbed wire, armed sentries and triple locks on a steel door, but it probably wouldn't be very nice to live there. On the other hand, leaving all the doors and windows unlocked, while making it easy for the kids to come and go, is likely to result in loss of property. One tries to find the appropriate balance, and that balance will be different in different circumstances. In a dense city environment one is likely to take more precautions than in a suburb where the neighbors are better known. In small towns people sometimes don't bother to lock at all. Everybody knows what everybody is doing. Security in business hotels is interesting to consider. There's typically 24-hour security, with cameras in all common areas, security guards and sophisticated keys providing access to the guest rooms. Isn't it interesting to think how alarming is the prospect that a guest in another room might have access to yours, yet we typically learn to have very little concern that the hotel staff have access to the room literally at countless times during the day without our knowledge (except

that perhaps the beds are made and the bathroom is cleaned) and in general even when we're in the room at night. Somehow in this situation it becomes perfectly acceptable to delegate security to the hotel management.

Similar considerations apply to the Database Cloud in different use cases.

In Database as a Service (DBaaS) on a Public Cloud, it's reasonable to assume that "adjacent" tenants may be competitors. This is a particularly challenging use case because each tenant wants both powerful administrative capabilities within his own PDB, but also that this PDB is fully isolated from all PDBAs in adjacent PDBs. A good residential analogy here is condominium ownership. One wants full sovereignty over one's own space. Everything on the inside of the front door is one's responsibility.

DBaaS on a Private Cloud is a very productive configuration for development teams. Each developer needs to be isolated from the others to the extent that one developer's test does not interfere with another's, but it's typically a collaborative environment in which there is an expectation that everybody will respect everybody else's environments. A good analogy here is sharing a large house with friends. Everybody has a key to the same front door and the individual bedroom doors are usually left unlocked. There are some common areas and common equipment but there's a reasonable expectation of privacy in one's own bedroom.

Software as a Service (SaaS) may be compared to staying in a hotel. For the price of your room you delegate all maintenance and security to the hotel management and within reason expect them to respect the sovereignty of the contents of your room even though they have access more or less at any time. (Perhaps in this case we'd use the in-room safe to secure anything sensitive from the housekeepers.) Everyone understands that there are other guests in the hotel and there is a well-founded expectation that no guest from another room with have access to yours.

It is with these considerations in mind that we designed a configurable isolation model for Multitenant. The following sections cover some of the major features of this model.

Resource Management

As we've seen, tremendous efficiencies can be achieved by sharing resources but we can't afford for one workload – a PDB in our case – to monopolize the resources of the system at the expense of all the others. This is why it is of critical importance to have a powerful, yet simple-to-use, resource management capability.

In 12.1, we can define resource manager policies to control CPU, I/O (but only in Exadata and SuperCluster), sessions and parallel execution (PX) servers. In 12.2 we're significantly enhancing this capability. With 12.2 we add memory management and I/O management on non-engineered systems.

Memory Management

This was a much-requested capability in 12.1. However, it's important to note that many customers have consolidated very successfully without this capability. At first glance the lack of this capability might seem to be a very serious deficiency. On deeper consideration, however, it becomes clearer why in most cases it was not that important after all. There are several reasons for this. First, the efficiencies of Multitenant mean that there will be significantly more memory available to an SGA shared by *n* PDBs than the aggregate of the SGAs if all the applications they supported were running against *n* standalone non-CDBs. Several thorough technical studies have demonstrated and quantified this, but at a high level significant savings are achieved by:

- eliminating program space consumed by replicated background processes;
- sharing over-provisioned SGA in standalone databases;
- sharing parent cursors when common SQL statements are issued across multiple tenants (particularly in SaaS).

Besides having more aggregate memory available, the success may be attributed to the extremely efficient memory management algorithms in the Oracle SGA. Remember, even within a single non-CDB, SGA is shared between users. When multiple PDBs are consolidated it may be viewed simply as sharing more memory among more users. A basic consolidation proposition is that, the more workloads you consolidate, the less likely it becomes that they'll all reach peak demand at the same time. In turn this tends to smooth out the net demand for any resource that can be elastically allocated, and the SGA is a perfect example of this.

All that is said as a matter of fact, not to make a virtue out of a necessity. That is because in 12.2 we do introduce memory management capabilities. These may be used when appropriate, but as just discussed there are many consolidation use cases which will work perfectly well without using memory management. An important use case in which it may be appropriate to use memory management is the relatively low-density consolidation of several mission-critical (or "platinum SLA") application databases into a single server.

The following parameters may now be set at a per-PDB level.

Parameter	Description
SGA_Target	Maximum SGA size for PDB
SGA_Min_Size (New in 12.2)	Guaranteed SGA size (for buffer cache and shared pool) for PDB. Tip: Sum of SGA_Min_Size across all PDBs should be <50% of SGA.
DB_Cache_Size	Guaranteed Buffer Cache size for PDB
DB_Shared_Pool_Size	Guaranteed Shared Pool size for PDB
PGA_Aggregate_Limit	Maximum PGA size for PDB
PGA_Aggregate_Target	Target PGA size for PDB

I/O Management on Commodity Storage

In 12.1 I/O management was only possible with Exadata Storage (available only with Oracle Exadata and Oracle SuperCluster engineered systems). With storage optimized for and integrated with the database, the database "knows" the throughput capabilities of the storage in terms of IOPS and MBPS. In this case, Resource Manager policies to control I/O between PDBs can be defined simply in terms of shares and limits.

In 12.2 two new per-PDB parameters are introduced to impose rate limits for PDBs on non-Exadata storage. These are Max_IOPS (maximum input/output operations per second) and Max_MBPS (maximum megabits per second). Both can be dynamically set and altered. (They may not be set in CDB$Root or on Exadata storage. Error messages are returned if this is attempted.)

CPU Management

In 12.1 CPU limits per PDB were defined as part of the CDB Resource Plan. In 12.2 we introduce CPU_Count as a PDB-level parameter. In a Cloud model, where you get what you pay for, this makes it much easier to impose predictable performance limits on a PDB even as it is moved between servers with different characteristics. A CPU limit of 50% means 12 cores in an X4-2 server and 22 cores in an X6-2 server. With the new model, one could set CPU_Count to, say, 8 for a specific PDB and that limit would apply in whatever server the PDB is hosted. (Note that the percentage limit as defined in a Resource Manager policy is still supported for backward-compatibility reasons. If both are specified, the lower setting will apply.)

System Access

When interacting with the operating system (OS) from within Oracle Database (perhaps using the host command in SQL Plus), operations are performed under the auspices of the Oracle OS user. This is typically a powerful account, and clearly in a Cloud environment this represents a potential risk to be mitigated. In 12.2 we introduce a new PDB-level parameter – PDB OS Credential. Where appropriate, this can be used to identify an OS user (presumably one far less privileged than the Oracle user) which will be used when the OS is accessed.

File Access

Operations that access files or folders in the underlying file system (for example when accessing external tables) present another potential risk to be mitigated. In 12.2 we introduce two new clauses to the *create pluggable database* statement – Path_Prefix and Create_File_Dest. These control the placement of and access to the PDB's files by specifying mount points within the file system so that each PDB's datafiles and directory objects are isolated in a sub-branch of the file system.

Lockdown Profiles

This is an important new capability with Multitenant in 12.2 that enables very fine control over network access, common users and common objects, and administrative features. These are the subject of another White Paper by Oracle, and so will only be briefly summarized here.

The basic philosophy is consistent with our principle of "manage many as one". The expectation is that a central administrator, to whom we might refer as a Container Database Administrator or *C*DBA, would specify a few Lockdown Profiles. These can then be applied to individual PDBs based on their use case. The beauty of this model is that it can support consolidation of multiple PDBs with different profiles into a single container database.

Lockdown profiles are complementary to grants and work by restricting the capabilities typically enabled by granting specific privileges. For example, granting the *alter system* privilege to a user enables various powerful capabilities including, among others:

- cursor_sharing
- ddl_lock_timeout
- optimizer_mode
- parallel_degree_limit
- plsql_code_type
- plsql_debug
- plsql_warnings
- resource_manager_plan

Now, one might imagine wanting to empower a developer with, say, *plsql_code_type, plsql_debug* and *plsql_warnings*, but would be reticent to grant any of those other capabilities. Lockdown profiles can be used to reduce the scope of the *alter system* privilege to just those capabilities that the user needs, while disabling the others. For example, one could issue the following statement to tailor the Dev_PDB lockdown profile for these purposes:

```
alter lockdown profile Dev_PDB
disable statement =
('ALTER SYSTEM')
clause = ('SET')
option = ALL EXCEPT
('plsql_code_type'
,'plsql_debug'
,'plsql_warnings');
```

This would have the effect of restricting the capabilities vested in the developer granted *alter system* privilege to this list:

- plsql_code_type
- plsql_debug
- plsql_warnings

An important technical detail is that these restrictions are evaluated and imposed at run time. This means that no objects are invalidated by the application of a lockdown profile.

Technically a lockdown profile may be applied to a PDB only by a common user, commonly granted either SysDBA or *alter system*. For example, such a user would connect to the appropriate PDB and issue a command such as

```
alter system set pdb_lockdown = Dev_PDB;
```

Data Guard Broker Enhancements

A PDB-level failover capability is enabled in 12.2 with enhancements to Data Guard Broker. In 12.1 we cover both extremes in terms of data replication.

- At one extreme, Oracle Data Guard (or preferably Active Data Guard) can be configured at the CDB level. This is a great example of managing many as one (and thus reducing operating costs). Configure this once at the CDB level and (Active) Data Guard will replicate transactions from primary to standby for all PDBs in a single stream.
- At the other extreme, for replication of data at the individual PDB level, Oracle GoldenGate is the perfect choice. This can even be used in an active-active configuration, which is a key capability for supporting a truly zero-downtime upgrade.

However, we acknowledge that some customers would like a capability between these two extremes. In 12.2 Data Guard Broker has been enhanced to provide what may be viewed as a per-PDB failover capability. To enable this, we have two pairs of CDBs in two servers, one primary and one standby in each, with replication in opposite "directions". In the event of a PDB-level failure, its standby counterpart may be moved to the other CDB in the same server, where it becomes a new primary. Because this involves shared storage, no physical movement or copying of datafiles is required and therefore this can be achieved with minimal downtime. In due course a standby can be created for this new primary PDB.

This is illustrated in Figure 1.

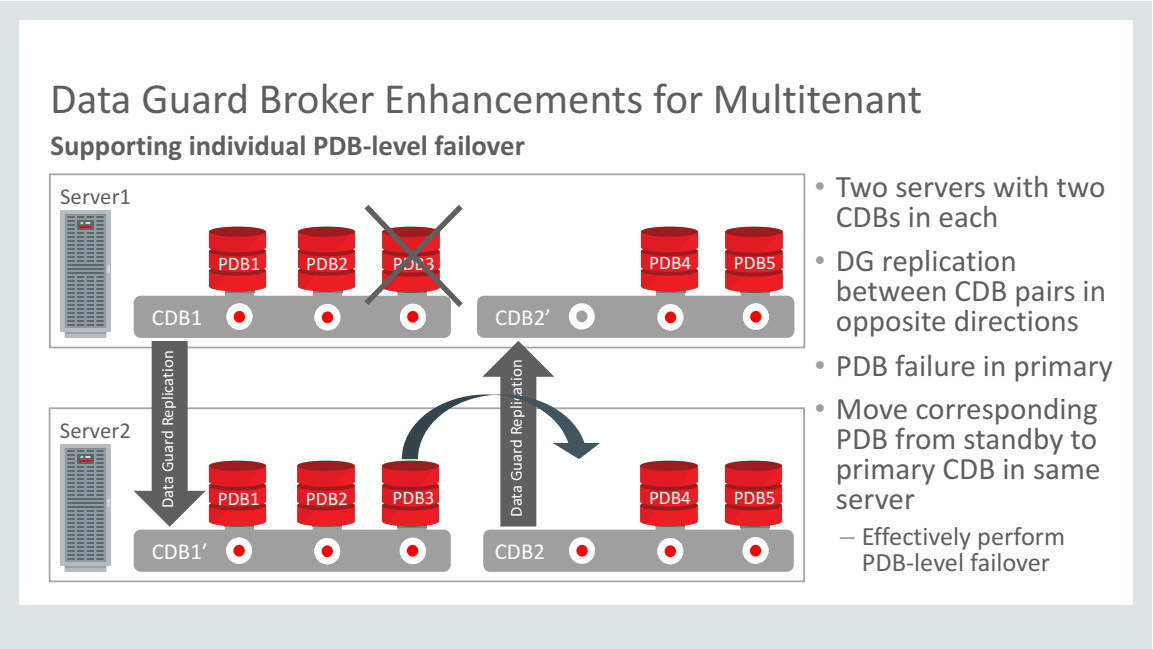

Figure 1. Data Guard Broker Enhancements for Multitenant

Software as a Service

We've already seen that a very important use case for Multitenant in 12.1 was Software as a Service (SaaS). After all, this is the use case for which the product is named! The phrase "isolation and agility with economies of scale" applies perfectly to SaaS. In 12.2 we dramatically expand our capabilities in this area with some very important enhancements. These are covered in the following section.

Elsewhere in this White Paper we've compared SaaS with hotel management. That's a very useful analogy for understanding the isolation requirements in a consolidated environment in which significant functionality is delegated to a central administrator. Another excellent analogy, which illustrates the economies of scale of SaaS, is gym membership. Let's say you set up a new gym, and sell 1000 memberships. How many treadmills do you buy? Maybe twenty (and even those are unlikely all to be in constant use – except perhaps in the first week of January).

This is a very important point to understand. The efficiency gains of the SaaS model powered by Oracle Multitenant are not merely incremental. The gains are measured in orders of magnitude. These are changes in kind, not of degree. (Incidentally, the whole point of DBaaS is to bring the economies of scale of the SaaS business model to the general database management space, but that's a topic for another discussion.) What is relevant to this topic is that SaaS is not just for ISVs. While perfect for them, it's also very well suited to any business running a franchise model or perhaps a global operation where common business practices are implemented in subsidiaries around the world. For the purposes of this discussion we will use the term *tenant* to indicate either the end customer of a SaaS ISV, a franchisee, a subsidiary in a global operation, or for that matter any self-contained business unit managed in this way. Similarly, we'll use the term *ISV* to designate the central service provider for those various tenants.

When one considers the benefits of the SaaS use case, it's important to understand that there are two different parties to consider. Each has different interests, requirements and in turn the SaaS model delivers different benefits:

To the End-Customer	To the ISV – Economies of Scale
A rich application experience tailored to their individual requirements	Massively scalable infrastructure
Isolation from other tenants	Highly efficient use of all computing resources – CPU, memory, storage, I/O
Simple self-service capabilities throughout the customer lifecycle	Ability to manage many tenants as one with granular control where appropriate
Services delivered at a competitive price	Agility in operations including provisioning, load balancing, elasticity and maintenance
	Ease of adoption and ease of use

The benefits to the end-customer are very important, but can be reviewed briefly. It is the ISV's responsibility to develop a rich application experience. Here of course Oracle Database is the perfect development environment, with our rich legacy of all of the power, flexibility and productivity of the world's most advanced relational database, SQL & PL/SQL and countless development tools, which are beyond the scope of this White Paper. Multitenant helps significantly here because it is not necessary to redesign a standalone application for multitenancy – it will run unchanged in a PDB. In this way, Multitenant delivers an instant cloud architecture. Tenant isolation is discussed in some depth elsewhere in this White Paper, and the intrinsic isolation of a PDB is complemented by the full set of Oracle Security technologies, all compatible with Multitenant. Self-service lifecycle management can simply be configured by the ISV according to their requirements because the underlying operations, while very sophisticated, are neatly encapsulated in simple SQL statements such as *alter pluggable database…* In modern Database Cloud environments these will typically be wrapped in RESTful APIs and orchestrated either by Oracle Enterprise Manager Cloud Control (EMCC) or by other industry-standard automation frameworks such as OpenStack / Chef or Puppet. Services may be delivered at a competitive price because the economies of scale of Multitenant enable the ISV to run operations extremely efficiently. The competitive price is a result of the ISV's being able to pass on these economies to their end-customer while still running a profitable business.

For the remainder of this section we'll concentrate on the benefits to the ISV. ISV applications may be classified into two groups: those formerly architected for on-premises deployment, and those "cloud-born" – that is, architected for a SaaS deployment model.

Applications formerly architected for standalone on-premises deployment are typically characterized by rich functionality built up through years of development, but hampered by an outdated deployment model that just can't compete with a modern SaaS application, despite often being far superior from a functional perspective. Without Multitenant, these ISVs are faced with two bad options. Either they could re-architect their application to retrofit multitenancy, which will typically involve very dramatic re-engineering and testing efforts and lots of unsatisfactory compromises of performance, isolation and functionality. Or, they can move to a hosted model, running each tenant standalone in a VM in some first-generation Cloud environment. As we've already seen, this model of hosting tenants in Public Cloud Infrastructure as a Service (IaaS), while typically well automated and instrumented, has a linear cost model. That is, to support twice as many tenants requires twice as many VMs. This does not represent true economies of scale. For this class of applications, Multitenant delivers an instant cloud architecture with true economies of scale.

More modern, “cloud-born” applications typically employ a model sometimes described as “row-based consolidation”. Essentially this model hosts multiple tenants in a single database, with each tenant’s data distinguished by a column (such as Tenant_ID) in each table, designating to whom each row belongs. With data for all tenants intermingled in the same tables it is thus up to the application to segregate the data. For example, if a particular tenant were to consult an Orders web page, the “show me my orders” query would be satisfied by the application code constructing a SQL statement similar to

```
select * from orders
where Tenant_ID = :Tenant_ID;
```

The first line (in black) may be viewed as the business logic and the second line (in red) as the application functionality which ensures isolation between tenants. In general, this works very well, as far as it goes, but there are some important limitations. For example, there is no scope for a tenant to develop custom screens or reports using popular application development tools, since there’s no guarantee that the required Tenant_ID predicate will be added in all cases (if at all). It’s certainly out of the question to allow the use of modern BI tools, such as Oracle Business Intelligence Enterprise Edition (OBIEE), or whatever may be your tool of choice.

Tenant mobility also becomes a major challenge. In general, all tenants are “born equal”, but it’s notoriously hard to predict their growth rates. Tenants that grow very rapidly can easily outgrow the capacity of the servers on which they’re hosted. This results in reduction of quality of service both to the large tenant and to the other tenants with which it’s consolidated. An obvious solution, of course, is load balancing, which typically means moving either the large tenant or its current bedfellows to a less heavily loaded server. The challenge with row-based consolidation is that this is far more easily said than done. Because the data for all tenants resides in the very same tables, this sort of segregation requires row-by-row operations, which are likely to be extremely time consuming and in turn involve significant application downtime. We return to the great dilemma of how to maintain performance SLAs without violating availability SLAs.

Both of these challenges can be summarized by saying that cloud-born applications that implement multitenancy in the application layer (typically by row-consolidation techniques) have traded isolation and agility for economies of scale. For this class of application, Multitenant delivers isolation and agility, without compromising economies of scale.

Multitenant for SaaS in 12.1

Even in 12.1, Multitenant delivered great functionality for the SaaS use case and is illustrated in Figure 2.

The PDB represents an ideal container for each tenant. A formerly standalone application can be installed in the PDB and run unchanged there. Each tenant’s data is beautifully isolated in its own container. New tenants can be rapidly “on-boarded” by provisioning for them a fresh PDB with the initialized application pre-installed. To enable this, the ISV simply runs the basic application installation script in an empty PDB – just once – and designates this as the “Tenant Seed”. The PDB for a new tenant is provisioned very simply and rapidly as a clone of this Tenant Seed.

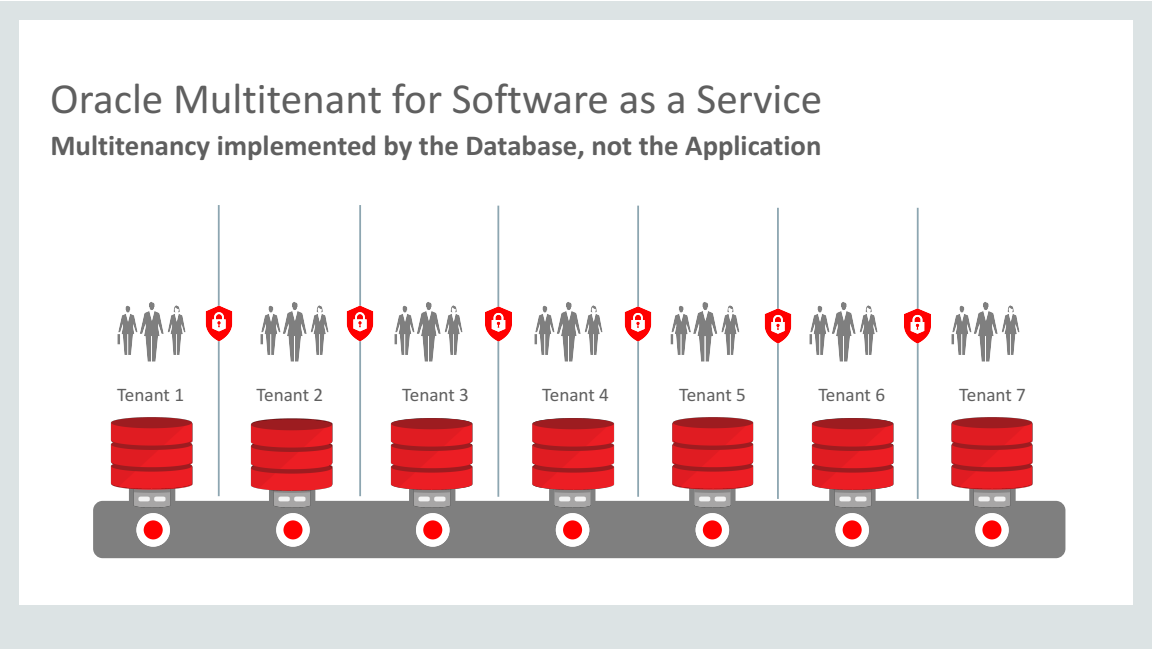

Figure 2. Oracle Multitenant for Software as a Service

There are multiple advantages to supporting SaaS with Multitenant in 12.1:

- Great isolation between tenants;
- Setup of new tenants is very quick;
- Capital expenses are dramatically reduced because many tenants can be supported on a relatively small server footprint;
- Operating expenses are dramatically reduced because many tenants can be managed collectively for typical DBA operations (such as backup and patching).

However, although many PDBs can be managed collectively from a *Database* Administration perspective, this is not true for *Application* Administration. Because there is a full copy of the database components of the application in each PDB, an application upgrade script must be run on each tenant PDB individually. What is needed is an ability to extend the benefits of managing many as one from the database administrator to the application administrator.

Multitenant for SaaS in 12.2

In 12.2, to address these important SaaS requirements, we introduce a new concept, Application Container. This topic is covered in greater detail in other White Papers by Oracle, but the major functionality is summarized below.

An Application Container comprises an Application Root, optionally a Tenant Seed and zero or many Tenant PDBs. The application definition is installed just once, in Application Root. From there the Tenant Seed can be created and thereafter new tenants provisioned simply by cloning the Tenant Seed. This is illustrated in Figure 3.

Technically, only skeletal metadata is stored in the data dictionaries in Tenant Seed and the Tenant PDBs. This comprises a set of pointers, known as “stubs”, which are references to the full metadata definitions stored in Application Root. This is precisely the same mechanism used in Multitenant 12.1 to share full definitions of Oracle-supplied metadata from the CDB Root to the various PDBs (which themselves have only skeletal sets of system metadata). In 12.2, with Application Container, this capability is now made available for customers’ use.

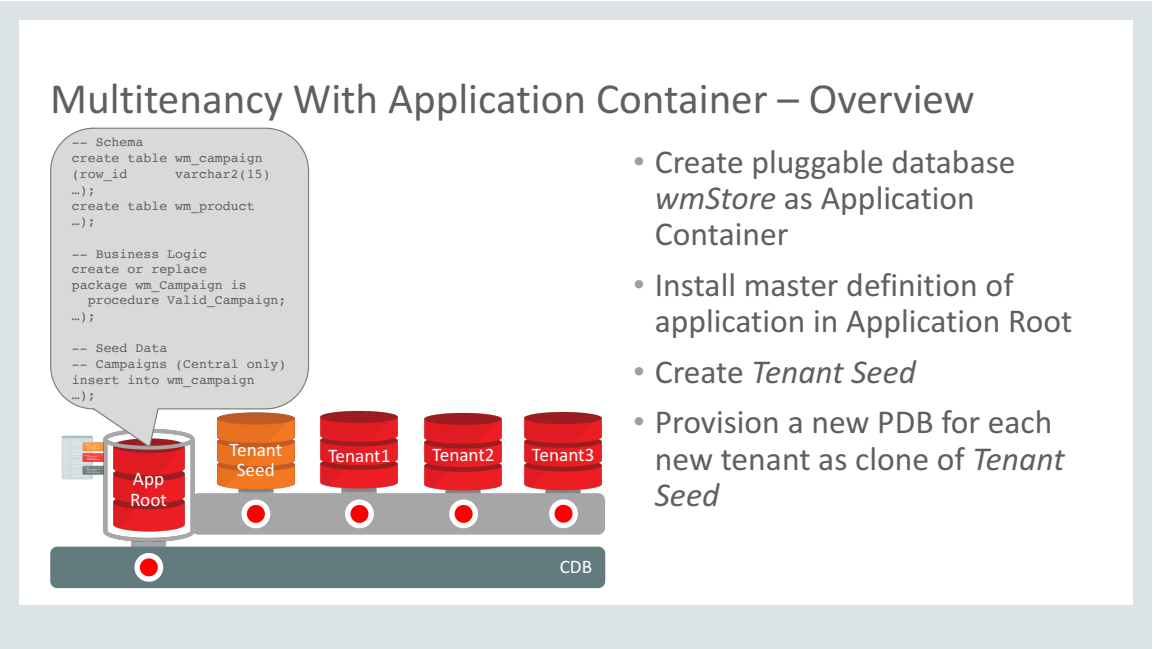

Figure 3. Application Container Overview

Application Maintenance

The beauty of this model is that, since there is only one master copy of the application (installed in Application Root), it follows that an application patch or upgrade script need only be executed on Application Root. Thereafter individual tenants can simply synchronize with the new master application definition. Importantly, this does not have to be done all at once. Individual tenants can choose to synchronize on their own schedule. This eliminates the cumbersome requirement to negotiate a convenient time to perform a business-wide application upgrade (or the alternative, which is simply – and typically unpopularly – to enforce one by diktat). In turn, this is likely to enable a much more agile application development process, because the inconvenience of delivering application upgrades is so significantly reduced.

An important consideration for application patching and upgrade is to restrict the impact of performing the upgrade on any individual tenant. In general, an application upgrade may involve a schema change, perhaps with associated data changes. A simplistic example would be the introduction of a new column, Full_Name to a Person table. In each tenant this would involve adding this column to the table and then populating it with a concatenation of First_Name and Last_Name. Of course, this involves a full scan of the affected table and may take significant time. With Application Container technology, this impact is restricted merely to the tenant being upgraded. This includes the isolation of each tenant from the execution of the application upgrade to Application Root. All of this is available when performing an application upgrade with Multitenant alone. Beyond this, Oracle provides various complementary technologies, such as Edition-Based Redefinition (EBR), GoldenGate, etc., all of which are compatible with Multitenant in general and Application Containers in particular. These technologies are described in great detail in other publications from Oracle. They are mentioned here as a reminder that if it is necessary to reduce the outage associated with an application upgrade even beyond the basic capabilities of Application Container, these existing capabilities can still be leveraged.

Cross-Tenant Aggregation

It is important for an ISV to be able to monitor the performance of the SaaS application across all tenants. For example, the ISV may use certain key metrics such as the number of orders created in the current quarter. In first-generation cloud architectures, this would have involved a cumbersome and error-prone process of going to each tenant in turn, repeatedly issuing the SQL statement to count the tenants' orders, writing down the answers in a spreadsheet and having the spreadsheet add them all up. A simplistic example of this query might be as follows:

```
select :Tenant_Name
,      count(*)
from orders
where current_quarter = 'Y';
```

With Multitenant in 12.2, it is possible to do this with a single SQL statement, by slightly rewriting this query to use the new *containers()* construct as follows:

```
select con$name
,      count(*)
from containers(orders)
where current_quarter = 'Y'
group by con$name;
```

This query can now be run just once, in Application Root. Technically, the *containers()* construct means that the SQL statement is to be executed recursively (and in parallel) in each Tenant PDB. Results are returned to Application Root, where they are aggregated. This is illustrated in Figure 4.

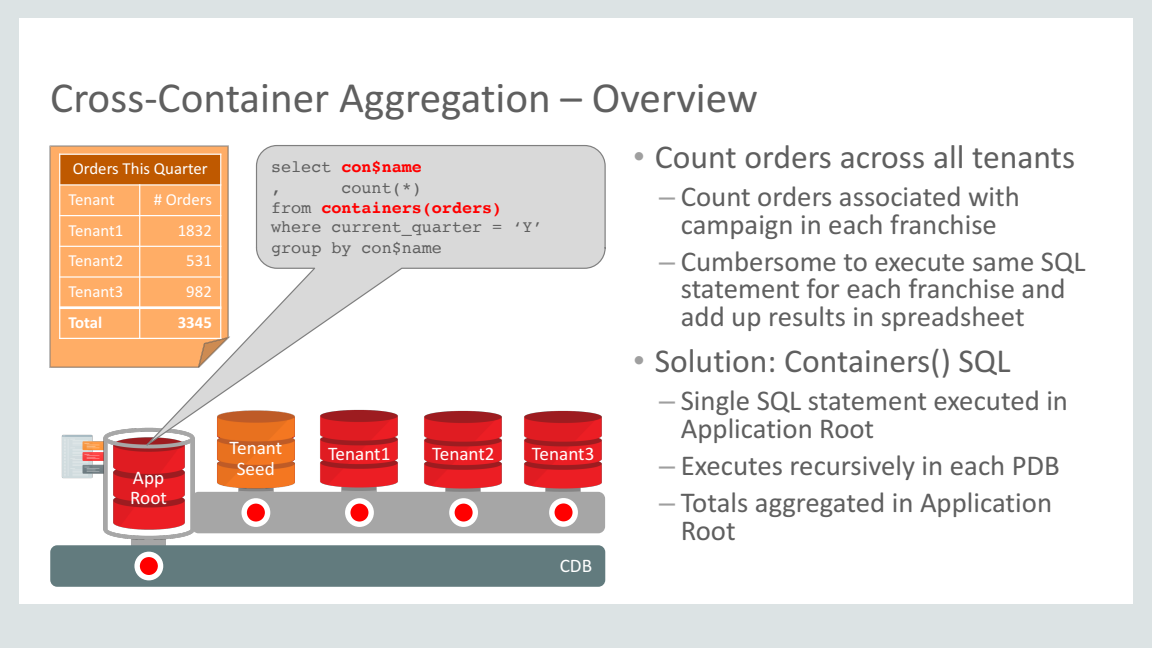

Figure 4. Cross-Container Aggregation

Application Container – Assessment

We have seen how the new Application Container capabilities introduced with Multitenant in 12.2 can deliver to the SaaS use case all the benefits of Multitenant in 12.1, namely:

- Great isolation between tenants;
- Setup of new tenants is very quick;
- Capital expenses are dramatically reduced because many tenants can be supported on a relatively small server footprint;
- Operating expenses are dramatically reduced because many tenants can be managed collectively for typical DBA operations (such as backup and patching).

Furthermore, major enhancements are provided, which allow many tenants to be managed as one both in terms of Application Administration as well as Database Administration.

Review of Requirements of a SaaS Solution

There are multiple pieces to this puzzle, and Multitenant delivers powerful features to support each requirement.

- Tenant isolation
  - Each tenant's data is isolated in a self-contained PDB
- Central Administration
  - Application Root contains a central master definition of the application, which serves all tenants.
  - Cross-container aggregation, using *containers()* queries in Application Root, allows global metrics to be gathered
  - Application Containers may span multiple servers
- Agility
  - All the capabilities of PDB-based provisioning are available, including on-line cloning, refreshable clones and PDB relocate
  - Application upgrades can be performed on a per-tenant basis, negating the requirement for system-wide outages
- Security – The intrinsic isolation of PDBs is enhanced with
  - Lockdown profiles
  - New parameters path_prefix and create_file_dest
  - PDB OS Credential
  - Additionally, all major Oracle Security features are fully compatible with Multitenant
- Location Transparency is an important capability in a flexible cloud deployment, where workloads may be relocated for various reasons. These requirements are supported by
  - Proxy PDB
  - Container Map
- True Economies of Scale are possible with Multitenant
  - 4k PDBs per CDB
  - Multitenant resource efficiency

Conclusion

Oracle Multitenant in 12.2 enables ***isolation and agility with economies of scale*** whether deployed in the Oracle Cloud or in your Data Center. Although a highly capable product in 12.1, with great support for a wide range of use cases including Development and Testing, Consolidation of Disparate Applications and Software as a Service (SaaS), significant enhancements are delivered in each of these areas.

Modern, innovative and highly productive development and testing organizations demand great agility. For database components of an application this requires highly efficient capabilities to provision databases. Multitenant in 12.2 supports several powerful on-line provisioning capabilities. Pluggable databases (PDBs) may be cloned without requiring any downtime in the source. Incremental refresh capabilities are supported to allow “development master” clones of even huge production databases to be topped up with fresh data simply and efficiently with negligible impact on the source. On-line relocation of PDBs realizes the dream of simple migration to the Cloud without requiring an application outage. Likewise, this on-line relocation capability enables a Cloud provider flexibly to adjust to changing workloads by performing load balancing with minimal impact on running applications.

Unlike first generation cloud architectures, in which individual databases are hosted in dedicated VMs and therefore the relationship between number of databases and cost is linear, Multitenant delivers true economies of scale to the Database Cloud. The greater the scale, the greater the economies, and therefore the key requirement for this potential to be fulfilled is to eliminate any barriers to consolidation. Tenant isolation is a major component of this. Significant enhancements are delivered in 12.2 including memory management between PDBs, I/O management on commodity storage, and lockdown profiles, all of which allow the level of isolation appropriate for any particular use case to be simply configured and applied. Important features of Oracle Database are now fully supported in a PDB, including Flashback PDB, Continuous Query Notification (CQN) and Heat Map. Automatic Workload Repository (AWR) reports at a per-PDB basis are now supported, PDBs with diverse character sets can be consolidated into a single Multitenant Container Database (CDB) and Data Guard broker enhancements enable per-PDB failover. The maximum number of PDBs in a CDB has been increased to 4,096 on Oracle Cloud and Oracle Exadata, thus enabling truly cloud-scale consolidation.

For Software as a Service (SaaS), Multitenant in 12.2 delivers a powerful new Application Container capability. This extends the benefits of managing many tenants as one from the DBA to the Application Administrator. For applications formerly architected for standalone deployment on premises, Application Container provides an instant cloud architecture. For cloud-born applications, the SaaS characteristic of great economy of scale has typically been achieved at the expense of agility. Application Container delivers all the agility of PDBs without compromising on these great economies of scale. A master application definition can now be installed in a single container, known as Application Root. Individual tenants in this Application Container are served by this master definition. This enables application maintenance operations (such as applying an application patch) to be performed in a central location, and individual tenants can simply synchronize with this new version on their own schedule. Powerful cross-container aggregation capabilities are possible, allowing for the simple creation of dashboards from which an ISV can assess the overall performance of the application using key metrics such as number or orders or sum of revenue in the current quarter. These capabilities are not just for ISVs, but apply equally well to many other business models, such as franchises or global organizations that replicate business practices across multiple subsidiaries around the globe.

Oracle Multitenant in 12.2: Welcome to the Cloud.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

ORACLE

CONNECT WITH US

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

Integrated Cloud Applications & Platform Services

Oracle Multitenant: New Features in Oracle Database 12*c* Release 2
February 2018
Author: Patrick Wheeler, Senior Director, Product Management, Oracle Database