CHAPTER 3

MULTI-TENANT DATABASE ARCHITECTURE

Multi-tenancy is an architecture in which a single instance of a software application serves multiple customers. Each customer is called a tenant.[56]

3.1. Multi-Tenant SaaS Architecture

In a multi-tenant architecture which is also known as single instance, data from multiple companies is stored on the same server[12], usually separated by a partition to prevent the data from migrating from one company to another. As all applications are housed on the same server, for handling this server, there must be a standard SaaS architecture that includes the same configuration capabilities for the hardware, network, and operating system for all customers, known as tenants.

Given that multiple customers are running the same instance of software and all data is housed in a multi-tenant database, individual tenants have limited or no ability to make customized modifications to functionality. This does not imply the functionality itself is limited, but rather it is more difficult to customize [13]. As such, a multi-tenant solution is suited to companies with little or no need for minor or major software alteration or reconfiguration.

This kind of standardization is a benefit to companies who use software out of the box. Most reputable SaaS cloud hosting providers are very good at anticipating their customers’ needs, and will offer the most-relevant software applications in the standardized version. Sometimes a multi-tenant application is set up for a chosen group of customers to give these users access to pre-release versions of applications, often in beta versions, for testing purposes.

Some controversy exists pertaining to the necessity of multi-tenant architecture as a component of SaaS. In addition, it is often incorrectly called the only true SaaS
architecture by some providers; however, single-tenant environments have their own distinct benefits [14].

3.2. Single-Tenant SaaS Architecture

Single-tenancy SaaS architecture (also called multi-instance) is where a separate instance of a software application and supporting infrastructure is used by each customer, or tenant. Single-tenant architecture is mainly used by companies who need a customized approach, either because of their geography (or that of their client-base) or their need of a higher level of security. With single-tenant, each company has a distinct database and system that is either placed on an individual server or segregated using extensive security controls to create a virtual server network.

In single-tenancy SaaS environments, each tenant purchases their own copy of the software which can be customized to meet their needs[15]. While the cloud hosting provider’s software acts as the basis for the final application, users are provided with significant capability to make configurations; for example, users can adapt features such as additions to individual modules and channels to various internal databases and external partner databases [18].

3.3. Selection of Architecture

While multi-tenancy environments are highly secure and adequate for the needs of most companies, single-tenancy applications can provide even greater security, as the potential for data migrating from one company to another is removed[17]. For this reason, single-tenancy architecture is sometimes preferred by industries in which data safety is a legal requirement, or for companies that must satisfy specific government protocols. Areas using single-tenancy architecture include manufacturing, retail, professional services such as financial, pharmaceutical, aerospace, defense, and technology industries [18].

Neither multi-tenant nor single-tenant architecture is inherently better; rather, each has distinct differences and advantages depending on our geographical location,
industry, security needs, in-house IT capabilities [19], and other variables. When in
doubt, our cloud hosting provider [56] is usually in the best position to help you decide
which SaaS architecture is right for our organization.

It is used to address the problem of SAAS (Software as a Service (SaaS) is a software
distribution model in which applications are hosted by a vendor or service provider and
made available to customers over a network, typically the Internet) which can serve
multiple clients [20]. Multi-tenant database architecture is very useful when one instance
of database is serving to multiple clients. Only one set of hardware resources is needed to
fulfill the requirements of all users [21]. Multi-tenant is based on subscriber model, so
user has freedom to avail the facility as per business requirement or can turnoff.

There are different approaches to the advantage out of the multi-tenant database. These
are

- **Dedicated database**: Separate databases per tenant.
- **Dedicated table and different schema**: Shared database and separate schema.
- **Share table/schema**: Same database and same table.

Now, it is very important to select appropriate approach for our application depending
upon the following factors [22].

- Size of tenant database
- Number of tenant
- Number of users per tenant
- Growth rate of tenant
- Growth rate of tenant database
- Security
- Cost
3.4. Dedicated database

It is straightforward approach where each tenant has its own database. Each tenant has its own set of data that remains logically isolated from data that belongs to all other tenants [23].

![Dedicated Databases Diagram]

Figure 3.1 Dedicated Databases

Pros:

- More secure data
- Easy to customize for vendor specific needs
- Easy to maintain e.g. backups, restore etc…

Cons:

- Relatively high hardware and maintenance requirements[24]
- This approach tends to lead to higher costs for maintaining equipment and backing up tenant data.

3.4.1. Dedicated table and different schema

Serving multiple tenants under same database, where each tenant has its own sets of tables grouped with schema as required by tenant.
Figure 3.2 Dedicated table and different schema

**Pros:**

- Good for small database application where number of tables per tenant is small.
- Cost is low as compared to dedicated database approach.
- Moderate logical isolation level is there for vendors having security as a concern.

**Cons:**

- Tenant data is harder to restore in case of failure.
- Difficult to manage large database application.

3.4.2. Shared Table/Schema

This approach involves using the same database and the same set of tables to host multiple tenants’ data. A given table can include records from multiple tenants stored in any order; a Tenant ID column associates every record with the appropriate tenant. Any application accessing the row must refer to this column in every query to ensure that one tenant is not able to see another tenant’s data.
<table>
<thead>
<tr>
<th>TenantID</th>
<th>Col1</th>
<th>Col2</th>
<th>Col3</th>
<th>Col4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abc</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>21</td>
<td>Cdw</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>45</td>
<td>Sha</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>72</td>
<td>Wah</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>96</td>
<td>kan</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

**Pros:**

- Lowest hardware cost as compared to other approaches.
- Can serve more tenants per server.
- Ability to update the schema in one place and affect all tenants.

**Cons:**

- More security is required to make sure no one can access cross-tenant data.
- Can affect query performance because of more rows.
- Can only update the schema in one place and thereby affect all tenants.

### 3.4.3. Extension Table

In case, there is need to increase number of fields as per tenant requirement under approach 3 then?

As all tenants will share same table/schema, it is very difficult to customize the number of fields.
One way to avoid these limitations is to allow tenants to extend the data model arbitrarily, storing custom data in a separate table and using metadata to define labels and data types for each tenant’s custom fields.

<table>
<thead>
<tr>
<th>TenantID</th>
<th>FN</th>
<th>Field</th>
<th>TagID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Kim</td>
<td>Trade</td>
<td>221</td>
</tr>
<tr>
<td>202</td>
<td>Tim</td>
<td>HR</td>
<td>433</td>
</tr>
<tr>
<td>……………</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>342</td>
<td>Rim</td>
<td>Fin</td>
<td>510</td>
</tr>
</tbody>
</table>

(Data Table)

<table>
<thead>
<tr>
<th>TenantID</th>
<th>ExtID</th>
<th>Label</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>342</td>
<td>3990</td>
<td>Age</td>
<td>int</td>
</tr>
<tr>
<td>120</td>
<td>3122</td>
<td>Status</td>
<td>bool</td>
</tr>
<tr>
<td>……………</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>1200</td>
<td>LName</td>
<td>string</td>
</tr>
</tbody>
</table>

(Metadata Table)

<table>
<thead>
<tr>
<th>TagID</th>
<th>ExtID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>433</td>
<td>1200</td>
<td>Border</td>
</tr>
<tr>
<td>500</td>
<td>321</td>
<td>abc</td>
</tr>
<tr>
<td>………..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>510</td>
<td>3990</td>
<td>23</td>
</tr>
</tbody>
</table>

(Extension Table)

Table 3.1 Extension Table

Here, a metadata table stores important information about every custom field defined by tenant, including the field’s name (label) and data type. These fields are created dynamically on front end (GUI) with unique id and value entered by end user corresponds to these fields are stored in different table Extension table.
So corresponding to data table we need to create two new tables “Metadata” and “Extension”.

This approach allows each tenant to create as many custom fields as necessary to meet its business need. When the end user retrieves a customer record, it performs a lookup in the extension table[28], selects all rows corresponding to the record ID, and returns a value for each custom field used. To associate these values with the correct custom fields and cast them to the correct data types, the application looks up the custom field information in metadata using the extension IDs associated with each value from the extension table.

This approach adds a level of complexity for database functions, such as indexing, querying, and updating records.

3.5. The multi-tenant architecture in Oracle

The multi-tenant architecture enables an Oracle database to function as a multi-tenant container database (CDB) that includes zero, one, or many customer-created pluggable databases (PDBs). A PDB is a portable collection of schemas, schema objects, and nonschema objects that appears to an Oracle Net client as a non-CDB. All Oracle databases before Oracle Database 12c were non-CDBs.

3.5.1. Containers in a CDB

A container is either a PDB or the root container (also called the root). The root is a collection of schemas, schema objects, and nonschema objects to which all PDBs belongs. Every CDB has the following containers:

- Exactly one root

The root stores Oracle-supplied metadata and common users. An example of metadata is the source code for Oracle-supplied PL/SQL packages. A common user is a database user known in every container. The root container is named CDB$ROOT.
• Exactly one seed PDB[44]

The seed PDB is a system-supplied template that the CDB can use to create new PDBs. The seed PDB is named PDB$SEED. You cannot add or modify objects in PDB$SEED.

• Zero or more user-created PDBs

A PDB is a user-created entity that contains the data and code required for a specific set of features. For example, a PDB can support a specific application, such as a human resources or sales application. No PDBs exist at creation of the CDB. You add PDBs based on our business requirements.

The following graphic shows a CDB with four containers: the root, seed, and two PDBs. Each PDB has its own dedicated application. A different PDB administrator manages each PDB[27]. A common user[51] exists across a CDB with a single identity. In this example, common user SYS can manage the root and every PDB. At the physical level, this CDB has a database instance and database files, just as a non-CDB does[26].
Figure 3.3 A CDB with four containers: the root, seed, and two PDBs.

### 3.5.2 User Interfaces for the Multi-tenant Architecture

You can use the same tools for both CDBs and non-CDBs. For example, you can use:

- SQL*Plus for command-line access
- Oracle Enterprise Manager Cloud Control (Cloud Control)

Cloud Control is an Oracle Database administration tool that provides a graphical user interface (GUI). Cloud Control supports Oracle Database 12c targets, including PDBs, CDBs, and non-CDBs.
• Oracle Enterprise Manager Database Express (EM Express)

EM Express is a web-based management product built into the Oracle database. EM Express enables you to provision and manage PDBs, including the following operations:

- Creating and dropping PDBs
- Plugging in and unplugging PDBs
- Cloning PDBs
- Setting resource limits for PDBs

• Oracle Database Configuration Assistant (DBCA)

DBCA enables you to create CDBs or non-CDBs, and create, plug, and unplug PDBs.

3.6. Benefits of the Multi-tenant Architecture

Large enterprises may use hundreds or thousands of databases. Often these databases run on different platforms on multiple physical servers. Because of improvements in hardware technology, especially the increase in the number of CPUs, servers are able to handle heavier workloads than before. A database may use only a fraction of the server hardware capacity. This approach wastes both hardware and human resources.

For example, 100 servers may have one database each, with each database using 10% of hardware resources and 10% of an administrator's time. A team of DBAs must manage the SGA, database files, accounts, security, and so on of each database separately, while system administrators must maintain 100 different computers.

To show the problem in reduced scale, Figure 3.2 depicts 11 databases, each with its own application and server. A head DBA oversees a team of four DBAs, each of whom is responsible for two or three databases.
A typical response to the management problem is to place multiple databases on each server. The problem is that the multiple database instances do not share background processes, system and process memory, or Oracle metadata. Another response is to logically separate the data into schemas or virtual private databases. The problem is that these virtual entities are difficult to manage, secure, and transport [29].

### 3.6.1. Benefits of the Multi-tenant Architecture for Database Consolidation

The process of consolidating data from multiple databases into one database on one computer is known as database consolidation[44]. Starting in Oracle Database 12c, the Oracle Multi-tenant option enables you to consolidate data and code without altering existing schemas or applications.

The PDB/non-CDB compatibility guarantee[44] means that a PDB behaves the same as a non-CDB as seen from a client connecting with Oracle Net. The installation scheme for an application back end that runs against a non-CDB runs the same against a PDB and produces the same result. Also, the run-time behavior of client code that connects to the PDB containing the application back end is identical to the behavior of client code that connected to the non-CDB containing this back end.
Operations that act on an entire non-CDB act in the same way on an entire CDB, for example, when using Oracle Data Guard and database backup and recovery. Thus, the users, administrators, and developers of a non-CDB have substantially the same experience after the database has been consolidated.

Figure 3.3 depicts the databases in Figure 3.2 after consolidation onto one computer. The DBA team is reduced from five to three, with one CDB administrator managing the CDB while two PDB administrators split management of the PDBs[30].

Figure 3.5 Single CDB
Using the multi-tenant architecture for database consolidation has the following benefits:

- **Cost reduction**

  By consolidating hardware and sharing database memory and files, you reduce costs for hardware, storage, availability, and labor. For example, 100 PDBs on a single server share one database instance and one set of database files, thereby requiring less hardware and fewer personnel.

- **Easier and more rapid movement of data and code**

  By design, you can quickly plug a PDB into a CDB, unplug the PDB from the CDB, and then plug this PDB into a different CDB. The implementation technique for plugging and unplugging is similar to the transportable tablespace technique.

- **Easier management and monitoring of the physical database**

  The CDB administrator can attend to one physical database (one set of files and one set of database instances) rather than split attention among dozens or hundreds of non-CDBs. Backup strategies and disaster recovery are simplified.

- **Separation of data and code**

  Although consolidated into a single physical database, PDBs mimic the behavior of non-CDBs. For example, if user error loses critical data, a PDB administrator can use Oracle Flashback or point-in-time recovery to retrieve the lost data without affecting other PDBs.

- **Secure separation of administrative duties**

  A user account is common, which means that it can connect to any container on which it has privileges, or local, which means that it is restricted to a specific PDB. A CDB administrator can use a common user account to manage the CDB.
A PDB administrator uses a local account to manage an individual PDB. Because a privilege is contained within the container in which it is granted, a local user on one PDB does not have privileges on other PDBs within the same CDB.

- **Ease of performance tuning**

  It is easier to collect performance metrics for a single database than for multiple databases. It is easier to size one SGA than 100 SGAs.

- **Support for Oracle Database Resource Manager**

  In a multi-tenant environment, one concern is contention for system resources among the PDBs running on the same computer. Another concern is limiting resource usage for more consistent, predictable performance. To address such resource contention, usage, and monitoring issues, you can use Oracle Database Resource Manager [31]

### 3.7. Benefits of the Multi-tenant Architecture for manageability

The multi-tenant architecture has benefits beyond database consolidation. These benefits derive from storing the data and data dictionary metadata specific to a PDB in the PDB itself rather than storing all dictionary metadata in one place. By storing its own dictionary metadata, a PDB becomes easier to manage as a distinct unit, even when only one PDB resides in a CDB.

Benefits of data dictionary separation include the following:

- **Easier migration of data and code**

  For example, instead of upgrading a CDB from one database release to another, you can unplug a PDB from the existing CDB, and then plug it into a newly created CDB from a higher release.

- **Easier testing of applications**
You can develop an application on a test PDB and, when it is ready for deployment, plug this PDB into the production CDB.

3.8. Path to Database Consolidation

For the duration of its existence, a database is either a CDB or a non-CDB. You cannot transform a non-CDB into a CDB or vice versa. You must define a database as a CDB at creation, and then create PDBs within this CDB.

The basic path to database consolidation is:

1. **Creation of a CDB** [44]
2. **Creation of a PDB** [44]

3.8.1. Creation of a CDB

The `CREATE DATABASE ... ENABLE PLUGGABLE DATABASE` SQL statement creates a new CDB. If you do not specify the `ENABLE PLUGGABLE DATABASE` clause, then the newly created database is a non-CDB and can never contain PDBs.

Along with the root (CDB$ROOT), Oracle Database automatically creates a seed PDB (PDB$SEED). The following graphic shows a newly created CDB:

![Diagram of a newly created CDB]

**FIGURE 3.6** Shows a newly created CDB
Example 3-1 shows a simple query for determining whether the database to which an administrative user is currently connected is a non-CDB, or a container in a CDB.

Example 3-1 Determining Whether a Database Is a CDB

SQL> SELECT NAME, CDB, CON_ID FROM V$DATABASE;

<table>
<thead>
<tr>
<th>NAME</th>
<th>CDB</th>
<th>CON_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDB1</td>
<td>YES</td>
<td>0</td>
</tr>
</tbody>
</table>

3.8.2. Creation of a PDB

The `CREATE PLUGGABLE DATABASE` SQL statement creates a PDB. This PDB automatically includes a full data dictionary including metadata and internal links to system-supplied objects in the root. You can only create a PDB in a CDB and not within another PDB.

Figure 3.7 Depicts the Options for Creating a PDB
The following graphic shows a CDB that contains six PDBs. hrpdb is a newly created PDB. salespdb was a pre-existing PDB that was unplugged from a different CDB and plugged into this CDB. The remaining four PDBs, each of whose names contains the prefix test, were copied from salespdb.

Figure 3.8 CDB that contains six PDBs. hrpdb is a newly created PDB

The following sections describe the different techniques for creating PDBs.

3.8.3. Creation of a PDB from Seed

You can use the `CREATE PLUGGABLE DATABASE` statement to create a PDB by copying the files from PDB$SEED, which is a template for creating PDBs. The following graphic illustrates creation from the seed:
The following SQL statement creates a PDB named `hrpdb` from the seed using Oracle Managed Files:

```
CREATE PLUGGABLE DATABASE hrpdb
ADMIN USER dba1 IDENTIFIED BY password
```

### 3.8.4. Creation of a PDB by Cloning a PDB or a Non-CDB

You can use the `CREATE PLUGGABLE DATABASE` statement to clone a source PDB or non-CDB and plug the clone into the CDB. The source can be a PDB in a local or remote CDB, or starting in Oracle Database 12c Release 1 (12.1.0.2), it can also be a remote non-CDB. This technique copies the files associated with the source PDB or non-CDB to a new location and associates the copied files with the new PDB.

If you clone from a remote CDB, then you must use a database link[47].

The Figure 3.8 illustrates cloning a PDB from an existing PDB in the same CDB:
If the underlying file system of a PDB supports storage snapshots, then you may specify the `SNAPSHOT COPY` clause to clone a PDB using storage snapshots. In this case, Oracle Database does not make a complete copy of source data files, but creates a storage-level snapshot of the underlying file system, and uses it to create PDB clones. Snapshot copies make cloning almost instantaneous.

The following SQL statement clones a PDB named `salespdb` from the plugged-in PDB named `hrpdb`:

```
CREATE PLUGGABLE DATABASE salespdb FROM hrpdb
```
3.8.5. Creation of a PDB by Plugging in an Unplugged PDB

In its unplugged state, a PDB is a self-contained set of data files and an XML metadata file. This technique uses the XML metadata file that describes the PDB and the files associated with the PDB to associate it with the CDB. The following Figure 3.9 illustrates plugging in an unplugged PDB.

Figure 3.11 plugging in an unplugged PDB

The following SQL statement plugs in a PDB named `financepdb` based on the metadata stored in the named XML file, and specify `NOCOPY` because the files of the unplugged PDB do not need to be renamed:

```
CREATE PLUGGABLE DATABASE salespdb USING '/disk1/usr/financepdb.xml' NOCOPY
```
3.8.6. Creation of a PDB from a Non-CDB

You can use any of the following techniques to create a PDB from an existing non-CDB:

- **Execute `DBMS_PDB.DESCRIBE` on a non-CDB in Oracle Database 12c**

  You place a non-CDB in a transactionally consistent state, and then run the `DBMS_PDB.DESCRIBE` function to generate XML metadata about this database. While connected to the root in the CDB, you execute the `CREATE PLUGGABLE DATABASE` statement to create a PDB from the existing non-CDB.

- **Use Oracle Data Pump with or without transportable tablespaces**

  You can use Oracle Data Pump to define a data set on a non-CDB. This non-CDB can be in the current or a previous Oracle Database release, for example, Oracle Database 10g. You create an empty PDB in an existing CDB, and then use Oracle Data Pump to import the data set into the PDB.

  A Full Transportable Export using Oracle Data Pump exports all objects and data necessary to create a complete copy of the database. Oracle Data Pump exports all objects that can be exported using the transportable option, and then exports the remaining objects using `direct_path INSERT` and external tables. The Full Transportable dump file contains all objects in the database, not only table-related objects. Full Transportable Export is available starting in Oracle Database 11g Release 2 (11.2.0.3) for import into Oracle Database 12c.

  The following Figure 3.10 illustrates the technique of running the `DBMS_PDB.DESCRIBE` function on a non-CDB, and then creating a PDB using the non-CDB files:
Figure 3.12 creating a PDB using the non-CDB files