# Migration Guide

## Oracle Database 11g/12c
## To GCP Cloud SQL For PostgreSQL Compatibility (9.6.x)

Date: March 2019

Presented by:
Troposphere Technologies
[www.Troposphere.tech](http://www.Troposphere.tech)

Google Cloud

## Notices

# Contents

# 1. Oracle to Cloud SQL PostgreSQL Migration Guide

## 1.1 Preface

The purpose of this guide is to assist users with migrating Oracle database to Cloud SQL for PostgreSQL as part of application modernization or cloud migration initiatives. This guide covers the following topics:

- Automatic Schema Conversion: How to use automated schema conversion tools, such as Ora2PG, to convert source Oracle schemas to a Cloud SQL for PostgreSQL target.

- Operational Migration Aspects: Common Oracle DBA tasks and their alternatives when managing a Cloud SQL for PostgreSQL instance.

- Database Backup and Recovery: Concepts and examples on how to create and execute a Cloud SQL For PostgreSQL backup and recovery strategy.

- Data replication aspects: Concepts behind real time data replication across heterogeneous databases and the technologies which can be used for these purposes.

- Application Migration Aspects: Detailed overview of specific Oracle schema-level objects and commonly used features and their best equivalents in Cloud SQL for PostgreSQL. The specifics include blueprints on how to convert:
    a. Data Types.
    b. Database Hints.
    c. Database Links.
    d. Oracle Data Pump.
    e. DBMS_packages.
    f. Execute Immediate.
    g. Execution Plan.
    h. Indexes.
    i. Information Views (Data Dictionary, V$ views).
    j. IOTs.
    k. Json Document Storage and Processing.
    l. Partitioning.
    m. Procedures, Packages and Functions.
    n. Security Mode: Users and Roles.

o. LOBs and CLOBs.
p. SQL*Loader.
q. Table statistics.
r. Constraints.
s. Temp Tables.
t. Transaction Model.
u. Triggers.
v. UDFs.
w. UTL_Packages.
x. Views.
y. Virtual Columns.
z. Synonyms

- Copying Data from Oracle Source Databases to Cloud SQL PostgreSQL Targets
- Testing and Validating Data Migrations

**Note:** Not all source Oracle features are fully compatible with PostgreSQL or have simple workarounds. Recommended alternatives are provided, where applicable.

## 1.2 Planning for Successful Database Migrations

Database migrations are a team effort and usually require a mix of skills from both IT and development backgrounds, including, but not limited to:

- Project manager / primary stakeholder.
- At least one Database Administrator who is familiar with the source database technology and at least one Database Administrator who is familiar with the target database engine. For example, one Oracle DBA and one PostgreSQL DBA. Since heterogeneous database migration projects involve a mix of operational and application centric DBA duties, more than one DBA is often needed, depending on their availability expertise and skill set. For example, an ops Database Administrator might be responsible for designing the high availability architecture of the target database while an application-centric DBA might be responsible for the performance tuning aspects of the new database.
- Cloud architect with GCP, security and networking expertise.
- Representative from each affected application team that currently leverages the source database technology and will require transitioning to the target database engine.
- Consultant or SME familiar with any 3rd party software that utilizes the source database.

The Database Administrator should have intimate knowledge of all of the source databases that are planned to be migrated. The Database Administrator probably holds the most critical role in the process. They will map the source databases in terms of size, connecting applications, proprietary features being used, integrated third party software, and features that are being used.

Due to the complexity involved in heterogeneous conversion of source database schema, large scale data movement and porting applications, heterogeneous database migration projects tend to be classified into "Small, Medium, and Large" categories based on multiple selection criteria:

| Category | Database size | Number of stored procedures | Type of connecting applications | Total number of non-table/index schema objects |
|---|---|---|---|---|
| Small | <2TB | <100 | Modern, <20 | <500 |
| Medium | 2TB-10TB | 100-500 | Mix, 20-100 | 500-1000 |
| Large | 10TB+ | 500+ | Legacy, 100+ | 1000+ |

The category of the migration can dramatically affect the migration process and the tools which will be used.

## 1.3 Determining the complexity of source Oracle databases

Most of the information required to accurately assess the complexity of a source Oracle database for migration to Cloud SQL for PostgreSQL can be obtained by querying the Oracle data dictionary. The Oracle database dictionary is a set of system catalog views which contain metadata about the database. This metadata includes the number and type of different schema objects used, the source code for stored procedures, the server configuration and more.
Querying the Oracle data dictionary is straightforward. Some of the queries that can be used for this purpose include:

Find all index types:

```
SELECT DISTINCT INDEX_TYPE
 FROM DBA_INDEXES
 WHERE OWNER NOT IN ('SYS', 'SYSTEM', 'DBSNMP', 'OUTLN', 'APPQOSSYS', 'DIP');
```

Find all data types:

```
SELECT DISTINCT DATA_TYPE
 FROM DBA_TAB_COLS
 WHERE OWNER NOT IN ('SYS', 'SYSTEM', 'DBSNMP', 'OUTLN', 'APPQOSSYS', 'DIP');
```

Find all object types:

```
SELECT DISTINCT OBJECT_TYPE
 FROM DBA_OBJECTS
 WHERE OWNER NOT IN ('SYS', 'SYSTEM', 'DBSNMP', 'OUTLN', 'APPQOSSYS', 'DIP');
```

Find procedure and functions that use UTL_* or DBMS_* Oracle packages:

```
SELECT DISTINCT OWNER, NAME
 FROM DBA_SOURCE
 WHERE OWNER NOT IN ('SYS', 'SYSTEM', 'DBSNMP', 'OUTLN', 'APPQOSSYS', 'DIP') AND
 (TEXT LIKE 'UTL_%' OR TEXT LIKE 'DBMS_%');
```

Find if Oracle Text is being used:

```
SELECT IDX_OWNER, IDX_NAME
 FROM CTXSYS.CTX_INDEXES
 WHERE IDX_OWNER <> 'CTXSYS';
```

Find if Oracle Spatial is being used:

```
SELECT OWNER, TABLE_NAME, COLUMN_NAME
 FROM DBA_TAB_COLUMNS
 WHERE DATA_TYPE = 'SDO_GEOMETRY' AND OWNER != 'MDSYS';
```

Find additional Oracle features that are enabled:

```
SELECT COMP_NAME
 FROM DBA_REGISTRY
 WHERE STATUS = 'VALID';
```

**Note:** Querying the Oracle data dictionary to detect Oracle-specific features is the first place to start. However, not all Oracle-specific features can be found without also analyzing the application code and/or the queries being executed against the Oracle database.

Using automatic schema conversion tools, such as Ora2PG (discussed in the following chapter), can help automate the detection and conversion of many proprietary Oracle schema objects.

- After all objects and proprietary Oracle features have been detected, the Database Administrator must verify that there are alternatives or equivalents in Cloud SQL for PostgreSQL. The specific blueprints for implementing many alternatives to Oracle features in PostgreSQL are detailed later in this guide, as part of the “Application Migration” chapter.
- For migration testing, QA and performance tuning, the Database Administrator will have to start the Cloud SQL for PostgreSQL service to have the target database ready. Database Administrators will often consult with the organization Cloud Architect to validate the target database network configuration and security policies. Sizing the target Cloud SQL should be done based on the actual utilization of the source Oracle database (in terms of workload) and not just based on the hardware. Often, on-premise databases are over-provisioned to accommodate for shifting workloads. When running a managed database service in the cloud, resizing a database instance is much easier and overall a faster operation.
- Once the target Cloud SQL for PostgreSQL instance is up and running, the Database Administrator will perform a “dry run” migration test. This includes converting the source database schema to the target PostgreSQL schema using a combination of automatic and manual tools as well as

copying parts of the source database data to the target. Populating the target database with data ensures that application QA and performance tuning can commence.
- Extensive performance tuning should be done on the target database instance. Since Oracle and PostgreSQL utilize different database optimizers, expect queries to behave differently. The Database Administrator should compare the execution plan and query statistics of the most frequently executed queries from the source database to the target database engine.
- In the event the actual migration fails and a rollback a required, the DBA must plan in advance on how to achieve a complete rollback procedure. The rollback procedure must be tested in advanced to simulate the extreme case it will be required.
- In the final stages of the migration, the Database Administrator should implement the process required to copy the entire data from the source database to the target. This can be done as a one-off batch process for smaller databases or require real time continuous data replication solution for larger databases. Switching over from the source Oracle database to the target PostgreSQL database should be done in an orchestrated and managed approach.

### Additional resources:

1. Google provides a checklist that can help you validate your production launch readiness. For more the checklist, see Launch Checklist for Google Cloud Platform.
2. Quick-start guide for Cloud SQL for PostgreSQL. See Quickstart for Cloud SQL for PostgreSQL
3. Security aspect definitions and guides can be found under GCP documentation:
    a. Instance and database levels access control. See Instance Access Control.
    b. Project level access control. See Project Access Control. Network and firewall security. See Firewall Rules Overview.

    c. You can monitor and view information about your instance using the GCP Console or with the gcloud command-line tool, or the API. See Viewing Information About Your Cloud SQL Instance.

# 2. Introduction to Heterogeneous Database Migrations

When discussing “heterogeneous database migrations” from a technical perspective, the intent is usually to describe two aspects of the process: schema conversion and data copy. Because heterogeneous database migrations involve two different database technologies (such as Oracle as source and PostgreSQL as a target), both converting the existing Oracle schema objects (stored procedures, triggers, materialized views, etc.) and copying the data can be challenging.  Schema conversion and data migration are described in greater detail in the following sections of the migration guide.  Both aspects can be achieved using manual scripts, automatic tools, or a combination of both. For schema conversion, our guide will focus primarily on Ora2PG, which is an open-source product for converting Oracle schemas to PostgreSQL databases. While it is often recommended to attempt automatic conversion, manual database migration scripts can also be used for the following purposes:

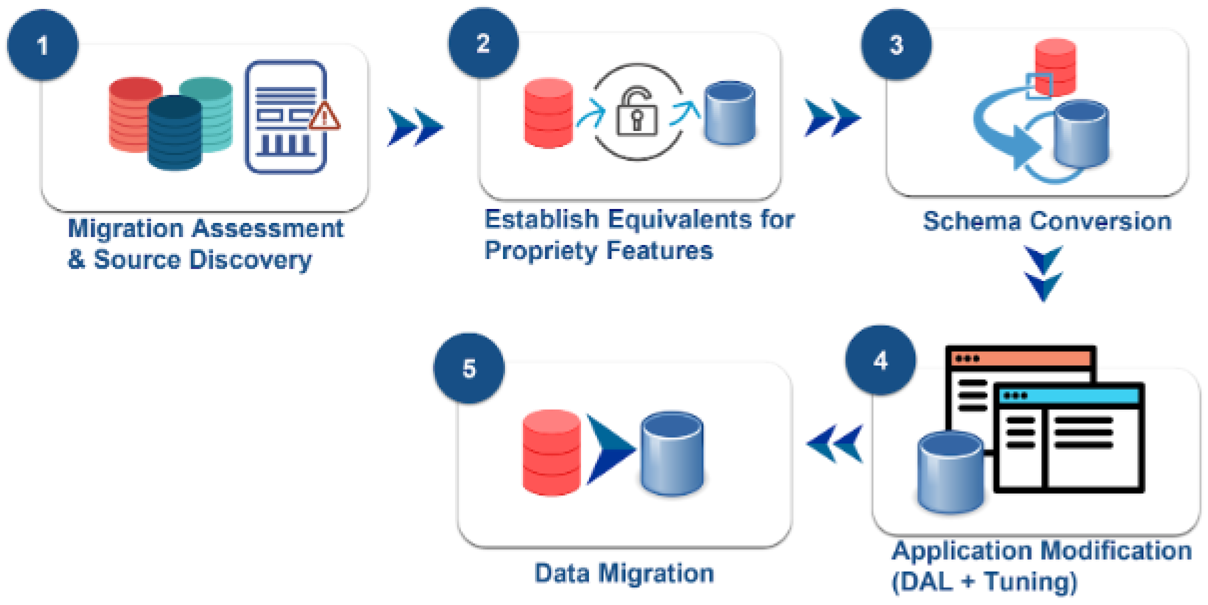
- Manual schema object conversion:
- Extract the DDL commands for all objects in the source database, make modifications to change the source object syntax to be compatible with the target, and execute updated DDL script on the target.
- Freeware tools, such as DBeaver, allow for easy extraction of source database object DDL code and support both PostgreSQL and Oracle databases.
- Manual data copy:
- Use scripts to extract data from the source database and generate INSERT commands which can be used to insert the data into the target database.
- This approach for data copy may require application downtime—preventing applications from changing data in the database while the data copy process is executing. For databases where downtime is not allowed, using real time (CDC - Change Data Capture) replication tools is advisable. More information regarding this topic can be found here.
- Manually comparing the source and target tables row count to verify integrity is recommended.

Using manual methods for heterogeneous database migrations is only suited for very small and low complexity databases. The use of automated tools is encouraged whenever possible.

# 3. The Steps for Heterogeneous Database Migrations

This chapter describes the actions that should be used when running a heterogeneous database migration. The chart below diagrams the different migration steps: assess and source discovery, establishing equivalent options, converting the schema, implementing required changes in the application, and migrating the data.

There are many ways to run a database migration and these will vary based on use cases and project requirements. This overview is intended as a general guidance to help you determine where you should start.


1
Migration Assessment & Source Discovery
2
Establish Equivalents for Propriety Features
3
Schema Conversion
4
Application Modification (DAL + Tuning)
5
Data Migration

1. Assessment and discovery

Before you execute the actual database migration, proper planning and discovery of source databases is required.

- Catalog the source fleet of Oracle databases which are potential candidates for migration to Cloud SQL PostgreSQL.
- Identify the Oracle database versions that are being used.
- Review hardware specifications for source databases paying specific focus on the number of CPU cores, Oracle SGA (memory) size and storage performance characteristics. You can use the Oracle v$osstat data dictionary view for retrieving this information.
- Pay special attention to the type of high-availability and DR configuration of the source Oracle database. Such as if Oracle Real Application Clusters or Oracle Active Data Guard are used.
- Identify proprietary database features currently in use by querying the Oracle dba_feature_usage_statistics data dictionary view.
- Identify the list of applications which have connected to the source Oracle database by querying the Oracle dba_hist_active_sess_history data dictionary view.
- Review database storage size in-use by querying the Oracle dba_segments data dictionary view.
- Identify the application stakeholders inside the organization. Migrating an Oracle database to PostgreSQL will always require some degree of changes to the applications which connect to the database.

Even large-scale database migration projects always start with a single database, choosing the initial Oracle database that will be migrated to PostgreSQL is crucial. It is important to identify the primary drivers for migration as these can help with proper planning of the target Cloud SQL database architecture. Some of the most common drivers for heterogeneous database migrations include:

1. Reducing commercial database licensing costs by migrating to open-source database engines such as PostgreSQL.

2. Reducing operational/IT services cost by moving to managed database instances.
3. Improving the database system's high availability, disaster recovery capabilities, or scalability.
4. Modernizing legacy systems by embracing next-generation and cloud-native database technologies.
5. Reducing vendor lock-in risk.

Using the criteria above, determine your migration plan. For example:

- If the driver for database migrations is reducing Oracle licensing costs, you might want to consider starting with your most "expensive" Oracle database.
- If the driver for database migrations is modernizing legacy applications, consider migrating your oldest version Oracle databases first, before they go out of support.

Create a dedicated migration plan for each source database:

1. Size the target database based on usage of the source database, and not just hardware specifications. Due to the nature of on-premise vs. cloud economics, many on-premise Oracle databases might be over-provisioned. Consider actual CPU, memory, I/O, and network utilization when planning the target database architecture size.
2. Certain source Oracle databases are large and monolithic. These databases might be serving dozens of different applications. It is a good strategy to break up these large databases into smaller target databases, usually by migrating individual schema(s) instead of the entire database at once.
3. Prototyping and testing each source database are essential. Treat each source database as a "snowflake" and assume the data volumes, workload characteristics, connecting applications, and schema objects will be different.
4. Dedicate time in your migration plan for both functional testing as well as performance testing. Remember, SQL queries that were tuned to run efficiently on Oracle might perform inefficiently when switched to a new database engine. This is especially true for queries that use database hints extensively to influence database optimizer behaviour.
5. Avoid applying last minute/untested changes/sudden deployments to your source database before switching over to the target database.
6. Plan your post-migration fallback and rollback procedures. When using real-time data replication products, ensure you can switch the replication direction from PostgreSQL back to Oracle, to eliminate data loss just in case a rollback is required.

Additionally, consider using products such as migVisor for Google Cloud, which can automate the source database discovery and assessment process and help accelerate the creation of a migration plan.

migVisor helps organizations migrate from a traditional database to next-generation and cloud-native database platforms and solutions. migVisor assess the cost savings and ease of migration from traditional source databases to a variety of next-generation database platforms such as Google Cloud SQL.

migVisor allows users to perform rapid cloud database migration assessments which can help accelerate your database migrations including adoption of next generation and cloud-native database technologies.

## 3.1 Schema conversion planning

You will need to convert your source Oracle schema including table structure, objects and stored procedures into the PostgreSQL dialect. This is one of the more complicated steps in the migration process and can be done using a combination of automated and manual tools.

- Prepare a list of the schema(s) in the source database that you plan to migrate. Note that in some cases, not all schemas in the source Oracle database will be part of the migration.

- Identify usage of proprietary features in the source Oracle database which might be incompatible with PostgreSQL. These are the features that can cause “stickiness” to the source database platform. For example—certain table partitioning methods, Oracle Advanced Queuing, Oracle Text, Oracle Spatial, Oracle Advanced Security, etc.

- Identify the size of each source Oracle schema. Be sure to pay attention to the size that is actually being used by database, not just the size that is allocated on disk. The Oracle dba_segments view can be used for this purpose. The size of the source schemas will also affect the migration time.

You can use an existing open source tools like ora2pg or 3rd party tool like migVisor for automating as much of the schema conversion process as possible. Expect around 50%-70% automation during conversion. The rest will have to be done manually.
For additional information about using ora2pg, see Ora2PG.

## 3.3 Modify your applications and data access layer

Always pay specific focus to your applications:

1. Identify applications that were developed in-house compared to applications which are owned by a 3rd party. Making changes to applications that were developed in-house is usually easier.
2. Identify legacy applications using out-of-date frameworks or deprecated programming languages. The migration effort for these applications will be much higher.
3. Applications which are tightly coupled to the Oracle database (OCI/TNS) could be more difficult to migrate,
4. Third party software—verify with application vendor if PostgreSQL is supported and if not, verify if a newer version of the applications does provide the required support for PostgreSQL.
5. In-house developed applications—check the willingness of the application team to make the changes required for supporting PostgreSQL. Try to identify and resolve any friction within the organization over implementing  the required changes.

6. Remember that when switching database engines from Oracle to PostgreSQL, the database optimizer (responsible for parsing and executing SQL queries) changes as well. Significant performance tuning efforts will be required for most typical workloads to ensure performance stability post-migration.

## 3.4 . Copying database between source and target databases

Data can either be replicated in real-time—also known as CDC (Changed Data Capture replication)—or as a one-off batch process. You can use 3rd party tools to copy the data between your source Oracle database and your target PostgreSQL instance. There are different products and solutions available for each approach. Real-time data replication products usually require a commercial license for tools like Oracle GoldenGate or DBSync by DBConvert.
Ideally, in order to minimize downtime a real-time replication method is suggested. The following approach should be used to keep source and target databases in-sync during the migration process and until the switchover to the target is performed.

- Create the target database schema using a schema conversion tool.
- Ensure that both source and target databases meet the schema prerequisites for real-time data replication.
- Validate the character sets between the databases are compatible.
- Install the CDC product on the source and target databases.
- Enable capturing of changes on the source database.
- Perform an "initial load" of the data, copying the existing data contained in the source database to the target database. This can be done using either the CDC product or 3rd party ETL tools.
- Enable real-time replication of all changes that have occurred on the source database after the "initial load" (previous step) took place. This will ensure that no new changes to data are lost and that a real-time data replication pipeline is established between the source and target database.
- Measure replication lag to ensure databases can be kept in sync.
- Test applications with target database.
- Stress test the target database to ensure it meets the required performance characteristics.
- Perform high-availability and recovery from failure tests for the target database.
- Ensure target database is configured for backups and monitoring.
- Monitor your production database to ensure real-time replication does not affect normal workload performance.
- Before the switchover, run data validations comparing the number of rows between source and target databases match.
- Setup, do not enable, a reverse-replication pipeline so that changes applied on the target database can also be sent to the source database. This will be used after the switchover in case rollback to the source database is required.
- Perform the switchover to the target database. A very small amount of application downtime will be required.

- Reverse the replication direction so that new changes which are applied to the target database will also be sent to the source database.

Additional information regarding data replication across heterogeneous database engines, as well as a detailed overview of the process can be found here.

## 4. Deciding between PostgreSQL and MySQL

Many users looking to modernize their relational databases choose to migrate to open-source and cloud-native database technologies. The most popular open-source relational database engines are PostgreSQL and MySQL. Both are available as variants of Google's managed Cloud SQL database service. The decision of choosing between PostgreSQL and MySQL as the database target is complex and involves several factors, including:

1. Support of certain advanced source Oracle database features varies between PostgreSQL and MySQL.
2. Compatibility of specific source Oracle schema objects with PostgreSQL or MySQL. For example, declarative table partitioning support in PostgreSQL 9.6 is lacking compared to MySQL.
3. Compatibility of Oracle PL/SQL programming language is greater with PostgreSQL compared to MySQL.
4. Application support—some applications that may be using the existing Oracle database can also be compatible with just PostgreSQL or MySQL.
5. How familiar the IT staff within the organization is with either PostgreSQL or MySQL.

For more about Google's manage Cloud SQL database service, see Cloud SQL.

For schema-level object support, below are some of the most common incompatibilities between Oracle, PostgreSQL, and MySQL:

- SEQUENCES are supported only in PostgreSQL
- USER DEFINED TYPES are supported only in PostgreSQL
- VIRTUAL COLUMNS have equivalent option only in MySQL
- TRIGGERS statements and system event triggers are only supported in PostgreSQL
- DBMS_SCHEDULER is supported only in MySQL
- MATERIALIZED VIEWS are supported only in PostgreSQL
- DML operation on VIEWS are supported only in PostgreSQL
- PARTITIONS are more like Oracle's partition in MySQL

| Oracle Feature | Cloud SQL for MySQL | Cloud SQL for PostgreSQL |
|---|---|---|
| Aggregate Functions | Not all functions are supported and may require to create manually | Not all functions are supported and may require to create manually |
| Create Table As select (CTAS) | | |

| | | |
|---|---|---|
| Common Table Expression (CTE) | | |
| Insert As Select | ERROR LOG and sub-query options are not supported | ERROR LOG and sub-partition options are not supported |
| Locking | | Auto commit is ON by default (can be changed) |
| MERGE | MERGE is not supported, workaround available | MERGE is not supported, workaround available |
| OLAP Functions | GREATEST and LEAST functions might get different results<br>CONNECT BY is not supported by MySQL, workaround available | GREATEST and LEAST functions might get different results<br>CONNECT BY is not supported by PostgreSQL, workaround available |
| Sequences and Identity | MySQL does not support sequences, Identity column has different syntax and options | Different syntax<br>Need to maintain the auto generated sequence manually in order to have the same functionality as IDENTITY options |
| Transactions | MySQL default isolation level is REPEATABLE READ<br>Nested transactions are not supported in MySQL | PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions |
| Data Types | BFILE, ROWID, UROWID are not supported | BFILE, ROWID, UROWID are not supported |
| Read Only | READ ONLY mode is not supported | READ ONLY mode is not supported |
| Constraints | CHECK, REF, DEFERRABLE, DISABLE constraints are not supported in MySQL<br>Constraints on Views are not supported | REF, ENABLE/DISABLE are not supported<br>Constraints on views are not supported |
| Temp Table | GLOBAL temporary table is not supported<br>Can't read from multiple sessions<br>Table dropped after session ends | GLOBAL temporary table is not supported<br>Can't read from multiple sessions<br>Table dropped after session ends |
| Triggers | Statement and System event triggers are not supported in MySQL<br>CREATE OR REPLACE is not supported in MySQL | Different paradigm and syntax<br>System triggers are not supported |
| User Defined Types | User Defined Types are not supported in MySQL | FORALL statement and DEFAULT option are not supported<br>PostgreSQL doesn't support constructors of the "collection" type |

| | | |
|---|---|---|
| Unused Columns | MySQL does not support unused columns | PostgreSQL does not support unused columns |
| Virtual Columns | Different paradigm and syntax | PostgreSQL does not support Virtual Columns |
| Alerting | Use GCP console and API | Use GCP console and API |
| Cache and Pools | Different cache names, similar usage | Different cache names, similar usage |
| Database Parameters | Use GCP console and API | Use GCP console and API |
| Session Parameters | SET options are significantly different | SET options are significantly different |
| Advanced Queues | Use GCP Task Queue | Use GCP Task Queue |
| Character Set | Different syntax<br>Can have different collation for each database in the same instance | UTF16 character and NCHAR/NVARCHAR data types are not supported |
| Database Links | MySQL does not support database links | Extension is not supported on GCP Cloud SQL |
| DBMS_SCHEDULER | Different paradigm and syntax (EVENTS) | PostgreSQL does not support schedule tasks |
| External Tables | MySQL doesn't support EXTERNAL TABLEs | PostgreSQL doesn't support EXTERNAL TABLEs |
| Inline Views | | |
| JSON | Different paradigm and syntax will require application/drivers rewrite | Different paradigm and syntax will require application/drivers rewrite |
| Materialized Views | MySQL doesn't support a materialized VIEW | Does not support automatic or incremental REFRESH |
| Multi-tenant | Distribute load/applications/ users across multiple instances | Distribute load/applications/ users across multiple instances |
| Resource Manager | Distribute load/applications/ users across multiple instances | Distribute load/applications/ users across multiple instances |
| Secure File and LOBs | SecureFiles are not supported, automation and compatibility refer only to LOBs | SecureFiles are not supported, automation and compatibility refer only to LOBs |
| Views | MySQL does not support View with READ ONLY option<br>MySQL does not support DML on views | PostgreSQL does not support View with READ ONLY option |
| XML | Different paradigm and syntax will require application/drivers rewrite | Different paradigm and syntax will require application/drivers rewrite |
| Active Data Guard | Distribute load/applications/ users across multiple instances | Distribute load/applications/ users across multiple instances |

| | | |
|---|---|---|
| RAC | Distribute load/applications/ users across multiple instances | Distribute load/applications/ users across multiple instances |
| BITMAP Indexes | MySQL does not support BITMAP index | PostgreSQL does not support BITMAP index - BRIN index can be used in some cases |
| BTree Indexes | | |
| Composite Indexes | | |
| Function Based Index (FBI) | MySQL doesn't support functional indexes, workaround available | PostgreSQL doesn't support functional indexes that aren't single-column |
| Invisible Indexes | MySQL does not support Invisible Indexes | PostgreSQL does not support Invisible Indexes |
| Index-Orginized Table (IOT) | MySQL does not support Index-Organized Tables object but this is the default behavior for InnoDB | PostgreSQL does not support Index-Organized Tables |
| Local and Global Indexes | MySQL does not support domain index | PostgreSQL does not support domain index |
| Anonymous Block | Different syntax may require code rewrite | Different syntax may require code rewrite |
| Conversion Functions | Not all functions are supported by MySQL and may require to create manually | Not all functions are supported by PostgreSQL and may require to create manually |
| Cursors | Minor different in syntax may require some code rewrite<br>%ISOPEN, %ROWTYPE, and %BULK_ROWCOUNT are not supported | TYPE ... IS REF CURSOR is not supported<br>Minor different in syntax may require some code rewrite<br>%ISOPEN, %BULK_EXCEPTIONS, and %BULK_ROWCOUNT are not supported |
| DBMS_DATAPUMP | No equivalent option | No equivalent option |
| DBMS_OUTPUT | Different paradigm and syntax will require application/drivers rewrite | Different paradigm and syntax will require application/drivers rewrite |
| DBMS_RANDOM | Different syntax and missing options may require code rewrite | Different syntax and missing options may require code rewrite |
| DBMS_REDEFENITION | MySQL does not support DBMS_REDEFENITION | PostgreSQL does not support DBMS_REDEFENITION |
| DBMS_SQL | Different paradigm and syntax will require application/drivers rewrite | Different paradigm and syntax will require application/drivers rewrite |
| EXECUTE IMMEDIATE | Must use PREPARE command in MySQL | Different syntax may require code rewrite. |

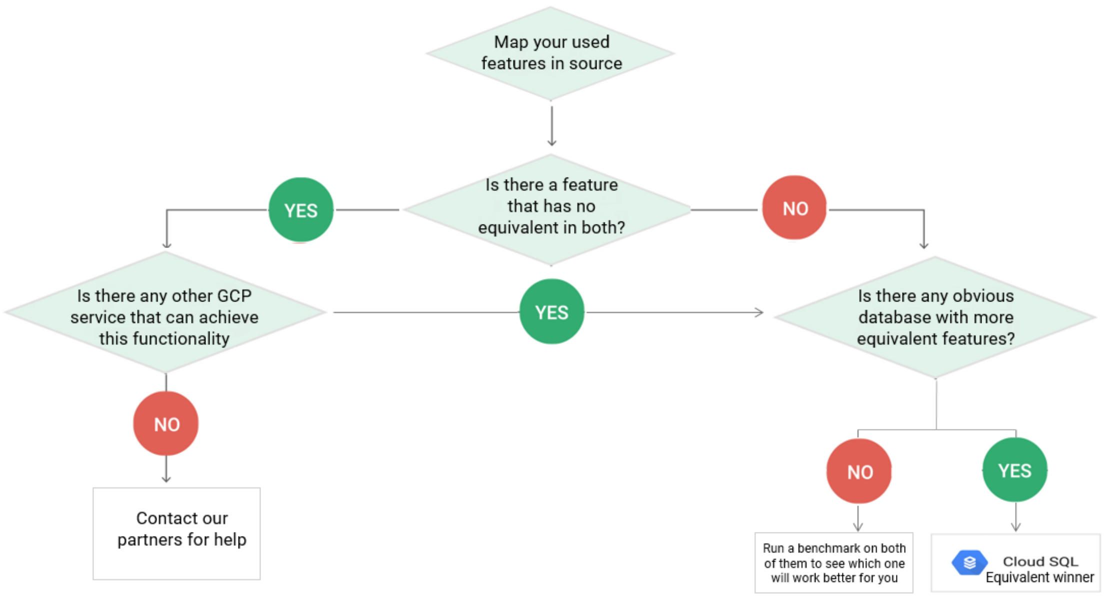
| | | |
|---|---|---|
| | Execute SQL with results and bind variables or Anonymous block execution using EXECUTE are not supported in MySQL | Remove IMMEDIATE keyword |
| Procedures and Functions | Syntax and options differences | Syntax and option differences |
| Database Hints | Very limited set of hints - Index hints and optimizer hints as comments<br>Syntax differences | Very limited set of hints - Index hints and optimizer hints as comments<br>Syntax differences |
| Execution Plan | Syntax differences<br>Completely different optimizer with different operators and rules | Syntax differences<br>Completely different optimizer with different operators and rules |
| Statistics | Syntax and option differences, similar functionality | Syntax and option differences, similar functionality |
| Partitions | Interval Partitioning, Partition Advisor, Preference Partitioning, Virtual Column Based Partitioning, and Automatic List Partitioning are not supported | Partition with DEFAULT or by TIMESTAMP are not supported<br>Foreign keys referencing to/from partitioned tables are not supported<br>Some partition types are not supported |
| Encryption | Data is encrypted by default | Data is encrypted by default |
| Roles | There are no roles - only privileges | Syntax and option differences, similar functionality<br>There are no users - only roles |
| Users | Syntax and option differences, similar functionality | Syntax and option differences, similar functionality<br>There are no users—only roles |
| Compression | Syntax and option differences, similar functionality<br>Compress a partition are not supported | PostgreSQL does not support compression options |
| LogMiner | MySQL does not support LogMiner | PostgreSQL does not support LogMiner |
| Result Cache | Syntax and option differences, similar functionality<br>Off the MySQL roadmap suggested not to be used | Syntax and option differences, similar functionality |
| DataPump | Non-compatible tool | Non-compatible tool |
| Flashback Database | GCP Cloud SQL restore mechanism | GCP Cloud SQL restore mechanism |
| Flashback Table | GCP Cloud SQL restore mechanism | GCP Cloud SQL restore mechanism |
| RMAN | GCP Cloud SQL restore mechanism | GCP Cloud SQL restore mechanism |
| SQL*Loader | Non-compatible tool | Non-compatible tool |

| | | |
|---|---|---|
| Information Views | Table names in queries need to be changed | Table names in queries need to be changed |

GCP Cloud SQL is the suggested option to use for your fully managed database. If you have one or more of the constraints below you should consider using Google Compute Engine and install your own database version:

- Specific software is required to be installed on the database server
- Missing extensions in GCP Cloud SQL to your database needs
- Required database version which is not supported by Cloud SQL
- You want to upgrade the database software yourself. This can be relevant if your organization wants to have the ability to install specific patches and upgrades at a custom schedule.

Use the chart below for better understanding:


Map your used features in source
Is there a feature that has no equivalent in both?
YES
NO
Is there any other GCP service that can achieve this functionality
YES
Is there any obvious database with more equivalent features?
NO
Contact our partners for help
NO
YES
Run a benchmark on both of them to see which one will work better for you
Cloud SQL Equivalent winner

## 5. Automatic Schema Conversion Between Oracle and PostgreSQL Databases

Converting the source Oracle schema to PostgreSQL is usually one of the more challenging aspects of heterogeneous database migrations. The source Oracle schema contains various objects which require conversion to PostgreSQL native schema objects including:

- Table structure, including partitioned tables and data types.

- Indexes,
- PL/SQL stored procedures, packages and functions.
- Triggers,
- Sequences.
- Views.

Any many others... Note that while many Oracle schema objects are compatible with PostgreSQL, not all schema objects can be converted, and workarounds might be required in such cases. Heterogeneous database schema conversion can be done using either automatic tools, manual re-coding or a combination of both. This chapter focuses primarily on using the Ora2PG tool for automatic schema conversion.

## 5.1 Schema object conversion support matric from Oracle to PostgreSQL

The summary table below contains a list of primary Oracle-level schema objects and their supportability in PostgreSQL. Note that even where full PostgreSQL support exist, recreating the specific schema object in PostgreSQL using native PostgreSQL dialect will be required.

| Oracle Feature | PostgreSQL Differences |
|---|---|
| Aggregate Functions | Not all functions are supported and may need to be created manually |
| Create Table As Select (CTAS) | - |
| Common Table Expression (CTE) | - |
| Insert As Select | ERROR LOG and sub-partition options are not supported |
| Locking | Auto commit is ON by default<br>(can be changed) |
| MERGE | MERGE is not supported, workaround available |
| OLAP Functions | GREATEST and LEAST functions might get different results<br>CONNECT BY is not supported by PostgreSQL, workaround available |
| Sequences and Identity | Different syntax<br>Need to maintain the auto generated sequence manually in order to have the same functionality as IDENTITY options |
| Transactions | PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions |
| Data Types | BFILE, ROWID, UROWID are not supported |
| Read Only | READ ONLY mode is not supported |
| Constraints | REF, ENABLE/DISABLE are not supported<br>Constraints on views are not supported |
| Temp Table | GLOBAL temporary table is not supported<br>Can't read from multiple sessions |

| | |
|---|---|
| | Table dropped after session ends |
| Triggers | Different paradigm and syntax<br>System triggers are not supported |
| User Defined Types | FORALL statement and DEFAULT option are not supported<br>PostgreSQL doesn't support constructors of the "collection" type |
| Unused Columns | PostgreSQL does not support Unused Columns |
| Virtual Columns | PostgreSQL does not support Virtual Columns |
| Alerting | Use GCP console and API |
| Cache and Pools | Different cache names, similar usage |
| Database Parameters | Use GCP console and API |
| Session Parameters | SET options are significantly different |
| Advanced Queues | Use GCP Task Queue |
| Character Set | UTF16 character and NCHAR/NVARCHAR data types are not supported |
| Database Links | Extension is not supported on GCP Cloud SQL |
| DBMS_SCHEDULER | PostgreSQL does not support schedule tasks |
| External Tables | PostgreSQL doesn't support EXTERNAL TABLEs |
| Inline Views | - |
| JSON | Different paradigm and syntax will require application/drivers rewrite. |
| Materialized Views | Does not support automatic or incremental REFRESH |
| Multi-tenant | Distribute load/applications/ users across multiple instances |
| Resource Manager | Distribute load/applications/ users across multiple instances |
| Secure File and LOBs | SecureFiles are not supported, automation and compatibility refer only to LOBs |
| Views | PostgreSQL does not support View with READ ONLY option |
| XML | Different paradigm and syntax will require application/drivers rewrite |
| Active Data Guard | Distribute load/applications/ users across multiple instances |
| RAC | Distribute load/applications/ users across multiple instances |
| BITMAP Indexes | PostgreSQL does not support BITMAP index - BRIN index can be used in some cases |
| BTree Indexes | - |
| Composite Indexes | - |
| Function Based Index (FBI) | PostgreSQL doesn't support functional indexes that aren't single-column |
| Invisible Indexes | PostgreSQL does not support Invisible Indexes |
| Index-Organized Table (IOT) | PostgreSQL does not support Index-Organized Tables |
| Local and Global Indexes | PostgreSQL does not support domain index |
| Anonymous Block | Different syntax may require code rewrite |
| Conversion Functions | Not all functions are supported by PostgreSQL and may require to create manually |
| Cursors | TYPE ... IS REF CURSOR is not supported<br>Minor different in syntax may require some code rewrite.<br>%ISOPEN, %BULK_EXCEPTIONS, and %BULK_ROWCOUNT are not supported |

| | |
|---|---|
| DBMS_DATAPUMP | No equivalent option |
| DBMS_OUTPUT | Different paradigm and syntax will require application/drivers rewrite. |
| DBMS_RANDOM | Different syntax and missing options may require code rewrite |
| DBMS_REDEFENITION | PostgreSQL does not support DBMS_REDEFENITION |
| DBMS_SQL | Different paradigm and syntax will require application/drivers rewrite |
| EXECUTE IMMEDIATE | Different syntax may require code rewrite<br>Remove IMMEDIATE keyword |
| Procedures and Functions | Syntax and option differences |
| Database Hints | Very limited set of hints - Index hints and optimizer hints as comments<br>Syntax differences |
| Execution Plan | Syntax differences<br>Completely different optimizer with different operators and rules |
| Statistics | Syntax and option differences, similar functionality |
| Partitions | Partition with DEFAULT or by TIMESTAMP are not supported<br>foreign keys referencing to/from partitioned tables are not supported<br>Some partition types are not supported |
| Encryption | Data is encrypted by default |
| Roles | Syntax and option differences, similar functionality<br>There are no users—only roles |
| Users | Syntax and option differences, similar functionality<br>There are no users—only roles |
| Compression | PostgreSQL does not support compression options |
| LogMiner | PostgreSQL does not support LogMiner |
| Result Cache | Syntax and option differences, similar functionality |
| DataPump | No compatible tool |
| Flashback Database | GCP Cloud SQL restore mechanism |
| Flashback Table | GCP Cloud SQL restore mechanism |
| RMAN | GCP Cloud SQL restore mechanism |
| SQL*Loader | No compatible tool |
| Information Views | Table names in queries need to be changed |

Some of the more complicated migrations of Oracle database schemas to PostgreSQL involve objects with very low target compatibility. For example, expect schema conversion to be more challenging if the following schema objects exist in your source Oracle database:

- MERGE sql command
- IDENTITY, UNUSED, and VIRTUAL columns
- Database Links
- DBMS_* and UTL_* functions
- BITMAP, INVISIBLE, FUNCTION-BASED, INDEX-ORGANIZED TABLE, and GLOBAL indexes

- TRIGGERs, FUNCTIONs, and PARTITIONs

For the above list of schema objects, manual review on a per-case basis is usually required even if automatic conversion was done using Ora2PG.

## 5.2 The automatic migration process steps should be:

Production Oracle databases usually contain a large amount of schema objects which will require conversion to PostgreSQL. Manual conversion of all schema objects is not advisable as it will complicate the migration process and will take considerable time. Instead, the recommended approach is an automatic schema conversion tool (such as Ora2PG) to convert the bulk of the source Oracle schema objects to PostgreSQL and have the Database Administrator manually fix and convert objects where the automatic process failed. The exact percentage of Oracle schema objects which can be automatically converted varies by the source database but expect between 50%–70% automatic conversion with manual workarounds required for the rest.

Using an automated tool for bulk conversion of Oracle schema objects usually involves multiple steps:

1. Installation and configuration of the schema migration tool.
2. Execute a schema complexity analysis which will provide information on the complexity of the source Oracle schema and highlight any incompatibilities with PostgreSQL which were detected. You will need to carefully review the schema conversion analysis report and determine the amount of manual coding which will be required to resolve and compatibility gaps between the source Oracle schema and what is supported in PostgreSQL.
3. Execute the schema conversion which will usually either generate a DDL (Data Definition Language) script containing the command needed to create the converted PostgreSQL schema or create the target schema in PostgreSQL automatically.
4. Copy some source Oracle data into the target schema to ensure tables are compatible and to commence application testing. Some schema conversion tools (Ora2PG included) provide basic capabilities of loading source data into the target schema, but in many cases 3rd party tools are advisable as they can load larger amounts of data faster and even enable real-time data replication between source and target databases. For additional information, review the Data Replication chapter included in this guide.

# 6. Quickstart Guide for Using Ora2PG for Automatic Schema Conversion

## 6.1 Background

Ora2PG is an open-source CLI tool for converting Oracle schemas to PostgreSQL. Ora2PG provides extensive migration support for the following Oracle schema objects:

- Full database schema (tables, views, sequences, indexes), with unique, primary, foreign key and check constraints.
- Provides basic automatic code conversion of PL/SQL to PL/pgSQL.
- Grants/privileges for users and groups.
- Range/list partitions and subpartitions.
- Table selection (by specifying the table names).
- Oracle spatial geometries into PostGis.
- Full support of Oracle BLOB object as PostgreSQL BYTEA.
- Oracle Synonyms as views.
- Oracle views as PG tables.
- Oracle Database Links to PostgreSQL FDW (Foreign Data Wrapper) extension).
- User-defined types.
- Predefined functions, triggers, procedures, packages and package bodies.
- Materialized view.
- Oracle DIRECTORY as external table or directory for external_file extension.

For more about Ora2PG, see Ora2PG.

In addition, Ora2PG provides the following pre-migration and post-migration capabilities:

- Generates a detailed report of an Oracle database schema including identification of problematic schema object for conversion.
- Bulk load of source Oracle data to a target PostgreSQL database. Either complete tables or filter using a WHERE clause.
- Cost estimations for how long it will take to migration and convert the queries and PL/SQL code—you can use it by adding the estimate_cost and cost_unit_value flags (described later in this section).
- Generates XML *.ktr files to be used with Pentaho Data Integration (Kettle) as the data replication tool between the Source Oracle database and the target PostgreSQL schema.
- Performs a diff check between Oracle and PostgreSQL schemas.

**Note:** For more on Ora2PG, see Ora2PG Documentation.

For more about Ora2PG installation, see Installing Ora2PG.

## 6.2 Configure Ora2PG for first-use

1. Configure the Oracle database connection properties in the ora2pg.conf configuration file. By default, this file can be found under the /etc/ora2pg path. In the example below, ORACLE_DSN (DataSource Name) is specified as well as the ORACLE_USER And ORACLE_PWD which correlate to the Oracle database system or sysdba username and password.

```
# Set Oracle database connection (datasource, user, password)
ORACLE_DSN     dbi:Oracle:host=localhost;sid=ORCL;port=1521
ORACLE_USER    system
ORACLE_PWD     manager
```

2. Test your connection properties by running ora2pg with the SHOW_VERSION argument.

```
$ ora2pg -t SHOW_VERSION -c /etc/ora2pg/ora2pg.conf
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
```

## 6.3 Generating a schema conversion report

Before executing the actual conversion of the source Oracle Schema to a PostgreSQL target, it is recommended to start by generating a “conversion cost estimation” report using the “SHOW_REPORT” argument and specifying a report name.

```
ora2pg -t SHOW_REPORT -u c##test -w test -c /etc/ora2pg/ora2pg.conf --estimate_cost --dump_as_html >
report.html
```

It is recommended to use the -u and -w flags (specifying username and password) to specify the database user/schema on which the conversion report will be executed. For more about the report created by Ora2PG, see Ora2Pg - Database Migration Report.

## 6.4 Generating a migration template

migration template is a set of folders and script, you should create the directory structure by the template, this is needed because when running the export script it will create *.sql files with all the DDL command in these directories and also will put the data in particular directory.

```
ora2pg --project_base /home/oracle/ --init_project migration_project
```

The directory structure will be created using the following format:

```
Creating project migration_project.
  /home/oracle//migration_project/
      schema/
            dblinks/
            directories/
            functions/
            grants/
            mviews/
            packages/
            partitions/
            procedures/
            sequences/
            synonyms/
            tables/
            tablespaces/
            triggers/
            types/
            views/
      sources/
            functions/
            mviews/
            packages/
            partitions/
            procedures/
            triggers/
            types/
            views/
      data/
      config/
      reports/

 Generating generic configuration file
 Creating script export_schema.sh to automate all exports.
 Creating script import_all.sh to automate all imports.
```

Ora2PG will generate the template scripts for exporting the source Oracle schema (export_schema.sh) and import the schema to PostgreSQL (import_all.sh).

In the "config" folder, you can find the file that holds the connection string to the database and all the properties that should be used when running the export (described later in this chapter), the file’s name is “ora2pg.conf” and you should edit it before running the export.sh script

## 6.5 Exporting an Oracle schema

1. After the migration template has been created and after validating the source Oracle connection config file (ora2pg.conf), run the export_schema.sh script to create all of the DDL (Data Definition Language) scripts which will be used later to create the schema in the PostgreSQL database . The output from running the export_schema.sh script will be similar to the following example:

```
./export_schema.sh
[========================>] 8/8 tables (100.0%) end of scanning.
 [========================>] 10/10 objects types (100.0%) end of objects auditing.
 Running: ora2pg -p -t TABLE -o table.sql -b ./schema/tables -c ./config/ora2pg.conf
 [========================>] 8/8 tables (100.0%) end of scanning.
 [========================>] 8/8 tables (100.0%) end of table export.
 Running: ora2pg -p -t PACKAGE -o package.sql -b ./schema/packages -c ./config/ora2pg.conf
 [========================>] 0/0 packages (100.0%) end of output.
 Running: ora2pg -p -t VIEW -o view.sql -b ./schema/views -c ./config/ora2pg.conf
 [========================>] 2/2 views (100.0%) end of output.
 Running: ora2pg -p -t GRANT -o grant.sql -b ./schema/grants -c ./config/ora2pg.conf
 WARNING: Exporting privilege as non DBA user is not allowed, see USER_GRANT
 Running: ora2pg -p -t SEQUENCE -o sequence.sql -b ./schema/sequences -c ./config/ora2pg.conf
 [========================>] 3/3 sequences (100.0%) end of output.
 Running: ora2pg -p -t TRIGGER -o trigger.sql -b ./schema/triggers -c ./config/ora2pg.conf
 [========================>] 1/1 triggers (100.0%) end of output.
 Running: ora2pg -p -t FUNCTION -o function.sql -b ./schema/functions -c ./config/ora2pg.conf
 [========================>] 0/0 functions (100.0%) end of functions export.
 Running: ora2pg -p -t PROCEDURE -o procedure.sql -b ./schema/procedures -c ./config/ora2pg.conf
 [========================>] 2/2 procedures (100.0%) end of procedures export.
 Running: ora2pg -p -t TABLESPACE -o tablespace.sql -b ./schema/tablespaces -c ./config/ora2pg.conf
 WARNING: Exporting tablespace as non DBA user is not allowed, see USER_GRANT
 Running: ora2pg -p -t PARTITION -o partition.sql -b ./schema/partitions -c ./config/ora2pg.conf
 [========================>] 0/0 partitions (100.0%) end of output.
 Running: ora2pg -p -t TYPE -o type.sql -b ./schema/types -c ./config/ora2pg.conf
 [========================>] 0/0 types (100.0%) end of output.
 Running: ora2pg -p -t MVIEW -o mview.sql -b ./schema/mviews -c ./config/ora2pg.conf
 [========================>] 0/0 materialized views (100.0%) end of output.
 Running: ora2pg -p -t DBLINK -o dblink.sql -b ./schema/dblinks -c ./config/ora2pg.conf
 [========================>] 0/0 dblink (100.0%) end of output.
 Running: ora2pg -p -t SYNONYM -o synonym.sql -b ./schema/synonyms -c ./config/ora2pg.conf
 [========================>] 0/0 synonyms (100.0%) end of output.
 Running: ora2pg -p -t DIRECTORY -o directorie.sql -b ./schema/directories -c ./config/ora2pg.conf
 [========================>] 0/0 directory (100.0%) end of output.
 Running: ora2pg -t PACKAGE -o package.sql -b ./sources/packages -c ./config/ora2pg.conf
 [========================>] 0/0 packages (100.0%) end of output.
 Running: ora2pg -t VIEW -o view.sql -b ./sources/views -c ./config/ora2pg.conf
 [========================>] 2/2 views (100.0%) end of output.
 Running: ora2pg -t TRIGGER -o trigger.sql -b ./sources/triggers -c ./config/ora2pg.conf
 [========================>] 1/1 triggers (100.0%) end of output.
 Running: ora2pg -t FUNCTION -o function.sql -b ./sources/functions -c ./config/ora2pg.conf
 [========================>] 0/0 functions (100.0%) end of functions export.
 Running: ora2pg -t PROCEDURE -o procedure.sql -b ./sources/procedures -c ./config/ora2pg.conf
```

```
[========================>] 2/2 procedures (100.0%) end of procedures export.
Running: ora2pg -t PARTITION -o partition.sql -b ./sources/partitions -c ./config/ora2pg.conf
[========================>] 0/0 partitions (100.0%) end of output.
Running: ora2pg -t TYPE -o type.sql -b ./sources/types -c ./config/ora2pg.conf
[========================>] 0/0 types (100.0%) end of output.
Running: ora2pg -t MVIEW -o mview.sql -b ./sources/mviews -c ./config/ora2pg.conf
[========================>] 0/0 materialized views (100.0%) end of output.


To extract data use the following command:

ora2pg -t COPY -o data.sql -b ./data -c ./config/ora2pg.conf
```

After the export.sh the folder and files structure will be similar to the example below, the files are dependent on the object types that will be exported:

```
.
|-- config
|   `-- ora2pg.conf
|-- data
|   |-- A_data.sql
|   |-- COUNTRIES_data.sql
|   |-- DEPARTMENTS_data.sql
|   |-- EMPLOYEES_data.sql
|   |-- JOBS_data.sql
|   |-- JOB_HISTORY_data.sql
|   |-- LOCATIONS_data.sql
|   |-- REGIONS_data.sql
|   `-- data.sql
|-- export_schema.sh
|-- import_all.sh
|-- reports
|   |-- columns.txt
|   |-- report.html
|   `-- tables.txt
|-- schema
|   |-- dblinks
|   |-- directories
|   |-- functions
|   |-- grants
|   |-- mviews
|   |-- packages
|   |-- partitions
|   |-- procedures
|   |   |-- ADD_JOB_HISTORY_procedure.sql
|   |   |-- SECURE_DML_procedure.sql
|   |   `-- procedure.sql
|   |-- sequences
|   |   `-- sequence.sql
|   |-- synonyms
|   |-- tables
|   |   |-- CONSTRAINTS_table.sql
|   |   |-- FKEYS_table.sql
|   |   |-- INDEXES_table.sql
|   |   `-- table.sql
|   |-- tablespaces
|   |-- triggers
|   |   |-- UPDATE_JOB_HISTORY_trigger.sql
|   |   `-- trigger.sql
|   |-- types
|   `-- views
|       |-- EMP_DETAILS_VIEW_view.sql
|       |-- VW_A_view.sql
|       `-- view.sql
`-- sources
    |-- functions
    |-- mviews
    |-- packages
    |-- partitions
    |-- procedures
    |   |-- ADD_JOB_HISTORY_procedure.sql
    |   |-- SECURE_DML_procedure.sql
    |   `-- procedure.sql
    |-- triggers
    |   |-- UPDATE_JOB_HISTORY_trigger.sql
    |   `-- trigger.sql
    |-- types
    `-- views
        |-- EMP_DETAILS_VIEW_view.sql
        |-- VW_A_view.sql
        `-- view.sql
```

2. After the DDL scripts were created in the previous step, you can also optionally use the COPY argument to extract data from the source Oracle tables and create the SQL script with the insert command that can be used to populate the target tables in PostgreSQL. Note that for very large source Oracle database, it is often preferable to use a 3rd party ETL or real-time data replication product for copying the data from Oracle to PostgreSQL. The output will be similar to the following example:

```
ora2pg -t COPY -o data.sql -b ./data -c ./config/ora2pg.conf

[========================>] 8/8 tables (100.0%) end of scanning.
 [========================>] 0/0 rows (100.0%) Table A (0 recs/sec)
 [>                        ] 0/7 total rows (0.0%) - (0 sec., avg: 0 recs/sec).
 [========================>] 25/1 rows (2500.0%) Table COUNTRIES (25 recs/sec)
 [========================>] 25/7 total rows (357.1%) - (1 sec., avg: 25 recs/sec).
 [========================>] 27/1 rows (2700.0%) Table DEPARTMENTS (27 recs/sec)
 [========================>] 52/7 total rows (742.9%) - (1 sec., avg: 52 recs/sec).
 [========================>] 107/1 rows (10700.0%) Table EMPLOYEES (107 recs/sec)
 [========================>] 159/7 total rows (2271.4%) - (1 sec., avg: 159 recs/sec).
 [========================>] 19/1 rows (1900.0%) Table JOBS (19 recs/sec)
 [========================>] 178/7 total rows (2542.9%) - (2 sec., avg: 89 recs/sec).
```

```
[========================>] 10/1 rows (1000.0%) Table JOB_HISTORY (10 recs/sec)
[========================>] 188/7 total rows (2685.7%) - (2 sec., avg: 94 recs/sec).
[========================>] 23/1 rows (2300.0%) Table LOCATIONS (23 recs/sec)
[========================>] 211/7 total rows (3014.3%) - (2 sec., avg: 105 recs/sec).
[========================>] 4/1 rows (400.0%) Table REGIONS (4 recs/sec)
[========================>] 215/7 total rows (3071.4%) - (3 sec., avg: 71 recs/sec).
[========================>] 7/7 rows (100.0%) on total estimated data (3 sec., avg: 2 recs/sec)
```

## 6.6 Advance Ora2PG configuration and conversion customization

The following properties can be configured to control or change the structure of migrated source Oracle schema objects.

- MODIFY_STRUCT: allow granular selection of a specific columns from a given table, for example, only export the ID and COUNTRY columns of the EMP_TBL table and the ID and DEP_NAME columns from DEP_TBL table. All other tables will be exported as-is (all columns) without specific selection

  MODIFY_STRUCT EMP_TBL(ID,COUNTRY) DEP_TBL(ID,DEP_NAME)

- REPLACE_TABLES: rename source tables in the target PostgreSQL database. For example, create the source EMPLOYEES table as EMPLOYEES_TBL and the source ORA_DEP_CODES table as DEP_CODES.

  REPLACE_TABLES EMPLOYEES:EMPLOYEES_TBL ORA_DEP_CODES:DEP_CODES

  REPLACE_COLS: specify new names for table columns. For example, rename the ID column in the EMPS table to EMP_ID and the ADDRESS column to EMP_ADDRESS.

  REPLACE_COLS EMPS(ID:EMP_ID,ADDRESS:EMP_ADDRESS)

- INDEXES_SUFFIX: allows to automatically add a suffix to the names of target indexes. For example, adding the “_idx” suffix.

INDEXES_SUFFIX _idx

- INDEXES_RENAMING: if this option is enabled (set to 1), all indexes’ names are renamed by the following convention: 'tablename_columns_names'.
  For example, an index named “sales_idx” on column named “sales” in the “retailers” table will be

renamed to "retailers_sales_sales_idx" - this can be helpful to prevent the object already exists errors.
- By default, CONTEXT indexes will be exported as PostgreSQL FTS indexes and CTXCAT indexes will be exported as indexes using the pg_trgm extension.

To handle most of the use cases, install the pg_trgm extension in your database. For information about all supported extensions, see PostgreSQL Extensions.

For specific options, see Controm of Full Text Search Export.
- When exporting Oracle Spatial objects, no special actions are needed. There are five options which can be used to configure:
- CONVERT_SRID: If enabled (default), Ora2Pg will use the Oracle sdo_cs.map_oracle_srid_to_epsg function to convert all SRID.
- GEOMETRY_EXTRACT_TYPE: can be set to WKT, WKB or INTERNAL.
- WKT will use SDO_UTIL.TO_WKTGEOMETRY() to extract the geometry data.
- WKB will use SDO_UTIL.TO_WKBGEOMETRY() to extract the geometry data.
- INTERNAL will use a Pure Perl library to convert the SDO_GEOMETRY data into a WKT representation, the conversion is done in the Ora2Pg itself.

POSTGIS_SCHEMA: allows to set the right path for the PostGis functions, map to target
- There are several options that can be set for more granular control over data types during export. Two of the most common options are:

DATA_TYPE: controls how exported data types can be mapped to target data types. Arguments will be provided in a comma-separated list. For example:

```
DATA_TYPE
INT:numeric,INTEGER:numericXMLTYPE:xml,BINARY_INTEGER:integer,PLS_INTEGER:integer,TIMESTAMP WITH
TIME ZONE:timestamp with time zone
```

MODIFY_TYPE: similar to the DATA_TYPE parameter with one major difference: when using the MODIFY_TYPE parameter a cast is used between the source data type and the target data type. The MODIFY_TYPE option should be used with TABLE and COLUMN names before the type such as:

```
MODIFY_TYPE     EMPS:DEP:varchar,EMPS:SALARY:integer
```

In the above example, the DEP column in EMPS table will be cast to VARCHAR and SALARY column in EMPS table will be converted to an integer.
Ora2PG fails to export LONG RAW columns data type. In order to be able to export this data type, a workaround can be used.

Create a staging table with the same structure, replace the LONG RAW data type with BLOB and use the INSERT AS SELECT command to move the data to the staging table - export the staging table instead of the source table.

## 6.7 PostgreSQL Import

The script created by Ora2PG will be used to import the converted schema and data into the target PostgreSQL database. The Ora2PG-generated output script should be used the same way as a regular psql import command:

```
        psql pg_db < output.sql
```

## 6.8 Test the migration

The TYPE action will create objects count to ensure that all objects were created in the PostgreSQL database.

```
ora2pg -t TEST -c config/ora2pg.conf > migration_results.log
```

An example of this report is:

```
 [TEST ROWS COUNT]
        ORACLEDB:COUNTRIES:42
       POSTGRES:countries:42
        ORACLEDB:CUSTOMERS:12
        POSTGRES:customers:12
        ORACLEDB:DEPARTMENTS:10
        POSTGRES:departments:10
        ORACLEDB:EMPLOYEES:45312
        POSTGRES:employees:45312
        [ERRORS ROWS COUNT]
        Table projects does not exist in PostgreSQL database.

        [TEST INDEXES COUNT]
        ORACLEDB:COUNTRIES:1
       POSTGRES:countries:1
        ORACLEDB:CUSTOMERS:3
        POSTGRES:customers:2
        ORACLEDB:DEPARTMENTS:1
        POSTGRES:departments:1
        ORACLEDB:EMPLOYEES:2
        POSTGRES:employees:2
        [ERRORS INDEXES COUNT]
        Table customers doesn't have the same number of indexes in Oracle (3) and in PostgreSQL (2).

        [TEST VIEW COUNT]
        ORACLEDB:VIEW:5
        POSTGRES:VIEW:5
        [ERRORS VIEW COUNT]
        OK, Oracle and PostgreSQL have the same number of VIEW.
```

```
[TEST MVIEW COUNT]
ORACLEDB:MVIEW:0
POSTGRES:MVIEW:0
[ERRORS MVIEW COUNT]
OK, Oracle and PostgreSQL have the same number of MVIEW.

[TEST SEQUENCE COUNT]
ORACLEDB:SEQUENCE:5
POSTGRES:SEQUENCE:4
[ERRORS SEQUENCE COUNT]
SEQUENCE does not have the same count in Oracle (5) and in PostgreSQL (4).

[TEST TYPE COUNT]
ORACLEDB:TYPE:1
POSTGRES:TYPE:0
[ERRORS TYPE COUNT]
OK, Oracle and PostgreSQL have the same number of TYPE

[TEST FDW COUNT]
ORACLEDB:FDW:0
POSTGRES:FDW:0
[ERRORS FDW COUNT]
OK, Oracle and PostgreSQL have the same number of FDW.
```

### 6.9 Troubleshooting common Ora2PG issues

Most common errors which can prevent the successful completion of an Ora2PG job include:

1. The user account that Ora2PG uses to connect to the source Oracle database is missing certain privileges, preventing it from accessing the required data or metadata. Be sure to use the SYS/SYSDBA/SYSTEM Oracle user to prevent these types of errors.
2. Incorrect connection string specification for the source Oracle database. The DSN or login information should be validated before running Ora2PG.
3. Oracle returned an error during the execution of the schema conversion process. Be sure to check the output file for any "ORA-" errors which originate from the Oracle database and consult with your Oracle Database Administrator on how to solve them.

# 7. Operational Migration Aspect

There are operational differences when migrating an Oracle database to a Google Cloud SQL instance. Certain DBA tasks are done differently, such as backup and recovery, data export/import and daily database administration. This section covers some of the basics to get you started with transitioning DBA tasks from Oracle to Cloud SQL PostgreSQL.

Note that all database administration operations can be done for Cloud SQL instances using CLI or the console. This section focuses mainly on using CLI commands as the CLI allows for easier migration of source Oracle scripts and often the method of choice for Oracle database administrators to manage the database.
CLI commands for managing Google Cloud SQL instances is done via the gcloud utility. For more information about installing gcloud , please see Installing Google Cloud SDK. .

## 7.1 Migrating Oracle RMAN Backups and Data Pump Exports

Oracle provides RMAN and Data Pump as the primary tools for database backup and data export/import, respectively. The following section identifies the common use cases for RMAN and DataPump, while demonstrating the equivalent options in GCP Cloud SQL for PostgreSQL.
Oracle RMAN is the primary tool for online Oracle database backups supporting both full, and incremental backup methods. Oracle supports other backup options, such as copying data files or using storage level snapshots. However, RMAN backup scripts can do more and are platform agnostic.

The main use cases for RMAN are performing backups, restoring backups and moving data to other destinations.

RMAN backups can be taken as:

- Full backups
  Create a standalone copy all data, control, and redo log files of the source Oracle database.

- Incremental backups
  Create a backup of the "delta" (changes) which have been performed in the database after an existing full backup was taken. DBAs can restore a full backup to establish a baseline, and one or more incremental backups to apply additional (more recent) changes to database data. These are the types of incremental RMAN backups:
  - Cumulative
    Backup all changes done in the database since the last full backup.
  - Differential
    Backup all changes done in the database since the previous backup, which can be either full or incremental

Another common use case for the RMAN is to duplicate an entire Oracle database. This is accomplished using the RMAN duplicate command and can be very useful for "lift and shift" migrations. The Oracle DBA can prepare the target host for migration with some RMAN prerequisites, connect the source Oracle server to the target server and use RMAN to duplicate the database. After completion of the RMAN duplicate command, the DBA can use Oracle Data Guard to keep the source and target databases in sync.

Existing RMAN backup and restore scripts will need to be converted to use Cloud SQLs native APIs. Since Cloud SQL is a managed database service, Cloud SQL backups are done differently. Unlike an on-premises Oracle database, there is no storage-level access to Cloud SQL storage. The database storage is managed by Cloud SQL automatically.

## 7.2 Converting Oracle RMAN to Cloud SQL for PostgreSQL Backup and Recovery

Cloud SQL backups are incremental; they contain only data that has changed since the last previous backup. This means that your oldest backup is similar in size to your database, but the sizes of subsequent backups depend on the rate of change of your data. When the oldest backup is deleted, the size of the next oldest backup increases so that a full backup still exists. Seven automated backups are retained for each Cloud SQL instance. On-demand CloudSQL backups can also be performed. It is considered a best practice to create a manual backup of your CloudSQL instance prior to big changes in the database schema or data.

- Backups are being kept in two regions for redundancy.
- Write operations on the database are unaffected by backup operations.
- Point-in-time restore is currently available only for Cloud SQL for MySQL.

For full documentation on Cloud SQL backups and recovery can be found here.

**Note:** All **gcloud** commands related to backups can be found **here**

Perform a FULL backup of the database including the database archived redo logs.
Oracle Example:

```
BACKUP DATABASE PLUS ARCHIVELOG;
```

Cloud SQL for PostgreSQL Example:

Use the following example to call the API or use the console to explicitly create a backup on your Cloud SQL instance.

```
gcloud sql backups create --instance=INSTANCE
```

Perform an INCREMENTAL level 0, level 1, and CUMULATIVE backup of the database.

Oracle Example:

Use the example below to perform an incremental and cumulative backup of the database:

```
BACKUP INCREMENTAL LEVEL 0 DATABASE;
BACKUP INCREMENTAL LEVEL 1 DATABASE;
BACKUP INCREMENTAL LEVEL 1 CUMULATIVE DATABASE;
```

Cloud SQL for PostgreSQL Example:

Use the following example to call the API or use the console to explicitly create a backup on your Cloud SQL instance, all backups in GCP are incrementals so this will be the equivalent.

```
gcloud sql backups create --instance=INSTANCE
```

Use the example below to restore a database.

Oracle Example:

```
RUN {
 RESTORE DATABASE;
 RECOVER DATABASE;
 ALTER DATABASE OPEN;
 }
```

Cloud SQL for PostgreSQL Example:

Use the following example to call the API or use the console to restore, to restore to the same instance:

```
gcloud sql backups restore BACKUP_ID --restore-instance=RESTORE_INSTANCE
```

The command above will take backup BACKUP_ID and restore it to RESTORE_INSTANCE ,Cloud SQL instance ID. Note that this instance will be overridden . The DBA decides whether to use the current instance or to create another new one.

To restore a database to a specific point in time, use the example below:

Oracle Example:

```
RUN {
 SHUTDOWN IMMEDIATE;
 STARTUP MOUNT;
 SET UNTIL TIME "TO_DATE('20-SEP-2017 21:30:00','DD-MON-YYYY HH24:MI:SS')";
 RESTORE DATABASE;
 RECOVER DATABASE;
 ALTER DATABASE OPEN RESETLOGS;
 }
```

Cloud SQL for PostgreSQL Example:

Currently, this operation is not supported for Cloud SQL for PostgreSQL, only for MySQL.

In order to achieve similar functionality, some applications will be able to restore the database and then reinsert the data using queues or other mechanism.

Oracle Example:

To list all current database backups created with RMAN, use the following example:

```
LIST BACKUP OF DATABASE;
```

Cloud SQL for PostgreSQL Example:

The following example shows how to list all current database backups:

```
gcloud sql backups list --instance=INSTANCE
```

The above examples include basic operations and scenarios. For more complex examples, see gcloud SQL Backups.

## 7.3 Converting Oracle Data Pump to PostgreSQL pg_dump and pg_restore

Exporting data from an Oracle database is usually done using the Oracle Data Pump utilities (expdp and impdp). Oracle export and import commands should be converted to the PostgreSQL native pg_dump utility or pgsql. There is no intercompatibility between the Oracle expdp and impdp utilites and PostgreSQL pg_dump/pg_restore. These tools can only be used with the corresponding source database engine.

To export data from Oracle and import into PostgreSQL, 3rd party programs will be required, such as ETL tools or CDC real-time data replication tools.This section provides examples on how to use the free open-source Ora2Pg tool for copying data between Oracle and PostgreSQL databases.

## 7.4 Migrating Oracle expdp and impdp scripts to PostgreSQL pg_dump and pg_restore

Cloud SQL for PostgreSQL instances support the PostgreSQL native pg_dump and pg_restore utilities. However, since Cloud SQL is a managed database instance, these tools must be executed from a desktop, server or laptop computer. Additional information is available in GCP documentation. For more about these tools, see Exporting Data.

**Note:** Oracle exp creates binary dump files that contains the exported database data compared to pg_dump which stores export data as plain-text files.

Oracle Example:
Use the following example to export a schema:

```
expdp system directory=expdp_dir schemas=hr dumpfile=hr.dmp logfile=hr.log
```

Import a schema using the example below:

```
impdp system directory=expdp_dir schemas=hr dumpfile=hr.dmp logfile=hr.log
```

**Note:** The import command should be executed against a database where the HR schema does not already exists.

Cloud SQL for PostgreSQL Example: Use the example below to export a database:

```
pg_dump -h cloud.sql.postgres.service -U user_name -d hr_db -f hr_dump.sql
```

If a full-path was not specified with the -f flag, then the file is created under the current location where the pg_dump utility was executed. Use the gsutil command to upload your database export file to cloud storage. For more about the gsutil command, see gsutil Tool.
To upload the export file to the gcloud-pg-backups bucket, use the following example:

```
pg_dump -h cloud.sql.postgres.service -U user_name -d hr_db -f hr_dump.sql | gsutil cp -I gs://gcloud-pg-backups
```

Use the following command to import a database:

```
pg_restore -h cloud.sql.postgres.service -U user_name -d hr_db -f hr_dump.sql
```

For use-cases where you wish to duplicate a database within the same instance, use the CREATE DATABASE command specifying the new (copy) database name and the source database as the template.

```
CREATE DATABASE hr_copy TEMPLATE hr;
```

## 7.5 Migrating Oracle SQL*Loader to the PostgreSQL COPY command

Oracle SQL*Loader can be used to load external files into database tables. SQL*Loader can use a configuration file (called a control file) which holds the meta-data used by SQL*Loader to determine how data should be parsed and loaded into the Oracle database. SQL*Loader supports both fixed and variable source files.
Oracle Example:
Load a data file into an Oracle database specifying the source file formatting:

```
control file: load data
 infile 'example3.dat'  "str '\n'"
 into table example
 fields terminated by ',' optionally enclosed by '"'
 (id int,
  name char(20))
```

```
example.dat: 396,Amber
142,Greg
```

PostgreSQL Example:
The equivalents in PostgreSQL include the COPY command, Cloud Dataflow, or Cloud Dataproc. This section focuses primarily on the PostgreSQL COPY command which is a more direct equivalent to Oracle's SQL*Loader utility.

The PostgreSQL COPY command can be executed from within the database and supports any one-character delimiter or fixed-sized columns in the input files. The source file must reside on the same machine from where the COPY command is executed and cannot reside inside a cloud storage bucket.

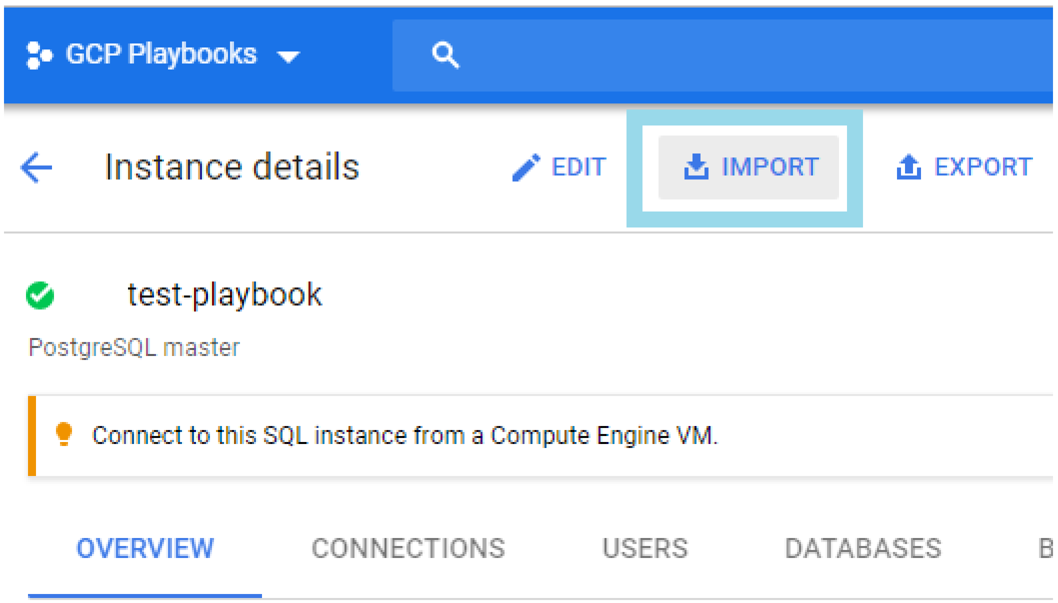
Example of loading the COPYtest.csv file into the emps table. The source file and target table should have identical structure.

```
psql -p 5432 -U username -h public_ip -d db_name -c "\copy emps from '/opt/files/COPYtest.csv' WITH csv;" -W
```

If the source file is located on a cloud storage bucket, you will have to perform one of these options:

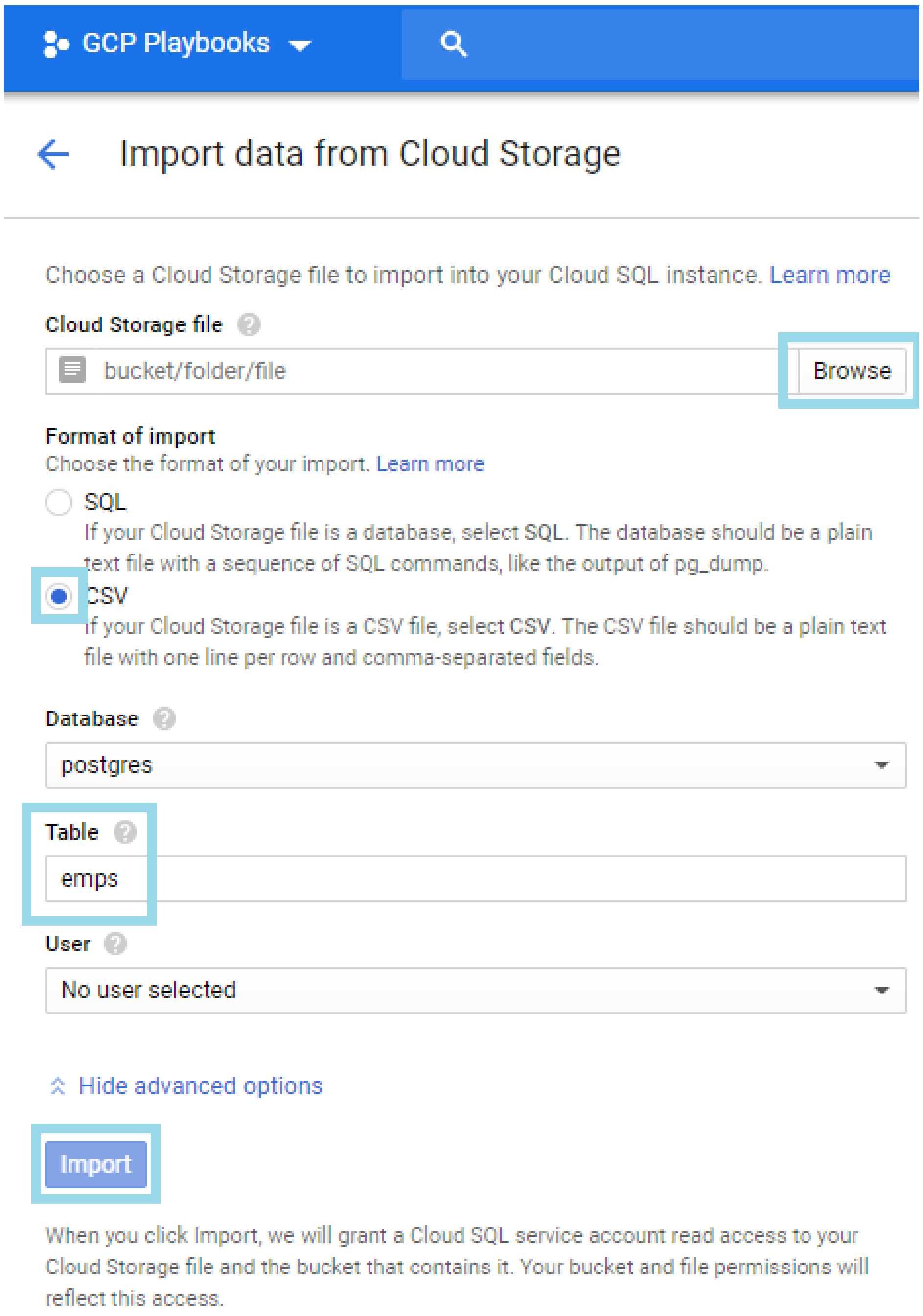
1. Download it locally before running the COPY command.
2. Use the GCP console to import the source file into the database. In the instance dashboard page, click Import. You will be able to Browse, then choose the CSV file to upload and select the target table.

In order to use the GCP console, from within the instance dashboard page, click Import.



On the new page you will be able to Browse, then select the CSV file to upload and point to the target table.

**Note:** You can use this page also for running sql scripts which also can be pg_dump files.



Import data from Cloud Storage

Choose a Cloud Storage file to import into your Cloud SQL instance. Learn more

Cloud Storage file: bucket/folder/file [Browse]

Format of import
Choose the format of your import. Learn more

SQL
If your Cloud Storage file is a database, select SQL. The database should be a plain text file with a sequence of SQL commands, like the output of pg_dump.

[✓ CSV]
If your Cloud Storage file is a CSV file, select CSV. The CSV file should be a plain text file with one line per row and comma-separated fields.

Database: postgres

Table: emps

User: No user selected

Hide advanced options

[Import]

When you click Import, we will grant a Cloud SQL service account read access to your Cloud Storage file and the bucket that contains it. Your bucket and file permissions will reflect this access.

For more complex data loads into your Cloud SQL PostgreSQL database, consider using Google Cloud Dataflow or Dataproc. This involves creating an ETL process.
For more about Google Cloud Dataflow, see [Cloud Dataflow Documentation](#).
For more about Google Cloud Dataproc, see [Cloud Dataproc Documentation](#).

1. Create your pipelines using the Apache Beam SDK (currently, only JAVA has Built in JDBC connector). Run them on the Cloud Dataflow service. For more about the Apache Beam SDK, see Apache Beam.
2. After setting up the framework you should implement your Python code for querying and transforming the data.

For more information on how to use this method with Python, see Quickstart Using Python.
For additional code samples using Python, see Apache Beam Python Examples.
For additional code samples using Java, see Apache Beam Java Examples.

Migrating scripts using Oracle Flashback technologies.
Oracle Flashback provides the option to revert either the entire database or a specific table to a previous point in time. The previous point in time can be specified as either the database SCN number, a timestamp value, or a specific user-created restore point. One of the main use cases for Oracle Flashback is recovering from human-errors, such as accidentally deleting a table or logical data corruption.

Oracle Example:
To create a flashback database restore point, use the following example:

```
CREATE RESTORE POINT before_deploy GUARANTEE FLASHBACK DATABASE;
```

The example below reverts the database back to the previously created flashback restore point:

```
FLASHBACK DATABASE TO RESTORE POINT before_deploy
```

To revert the database to a previous point in time, use the example below:

```
FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2019','MM/DD/YY')";
```

PostgreSQL Example:
Some of the functionality of Oracle Flashback use-cases can be achieved by taking a snapshot of the database and then restoring it as needed. This action will override the existing instance.
Create a Cloud SQL database snapshot using the example below:

```
gcloud sql backups create --instance=INSTANCE
```

To restore a snapshot backup of the database instance (overwriting that instance), use the following example:

```
gcloud sql backups restore BACKUP_ID --restore-instance=RESTORE_INSTANCE
```

In order to flashback a table, just restore the backup without overriding the instance, then export/import the specific table to the active instance. This is the same restore command with different instance id.

For export and import commands, use the examples from the Data Pump section.
There is no built-in capability to restore a table to a previous point in time. You can restore the backup without overriding the instance and export/import the specific table to the active instance

For more information about this topic, see the Backup and Recovery section.

## 7.6 Migrating the Oracle Security Model to PostgreSQL

In the Oracle database, there is the concept of USERS and ROLES. Users are used to authenticate with the database and roles provide grouping of permissions which can be granted as a whole.

Oracle Example:
Create a new user in the database. User is a login entity that can be used for application/users to connect the database. Users in the Oracle database are also schemas, there is no difference. You cannot create a schema without creating a user and vice-versa.
Use the following example to create a user:

```
CREATE USER user_name IDENTIFIED BY password;
```

PostgreSQL Example:
In order to create a database user in PostgreSQL, you can use either of the following examples:

```
CREATE USER user WITH PASSWORD 'password';
 CREATE ROLE user WITH LOGIN PASSWORD 'password';
```

The CREATE USER command in PostgreSQL is an alias to CREATE ROLE command with a single difference—the created role has the LOGIN privilege granted by default.
Oracle Example:
Use the following example to create a role:

```
CREATE ROLE dba_team_role;
```

To associate (GRANT) the relevant privileges to it, use the example below:

```
GRANT CONNECT, RESOURCE, ALTER SYSTEM, ALTER DATABASE TO dba_team_role;
 dba_team_role
 {
     CONNECT,
     RESOURCE,
     ALTER SYSTEM,
     ALTER DATABASE
 };
```

Oracle Example:
Use the following example to LOCK or UNLOCK a user account:

```
ALTER USER user ACCOUNT LOCK;
ALTER USER user ACCOUNT UNLOCK;
```

PostgreSQL Example:
Use the example below to grant or revoke the LOGIN privileges from a role:

```
ALTER ROLE user WITH NOLOGIN;
ALTER ROLE user WITH LOGIN;
```

Use the following example to grant permissions on a specific schema to a PostgreSQL database user:

```
GRANT ALL ON ALL TABLES IN SCHEMA source_schema TO user_role;
```

To limit the number of connections for a role in PostgreSQL, use the example below:

```
ALTER ROLE user WITH CONNECTION LIMIT 5;
```

To retrieve the permissions granted for each database, use the following example:

```
SELECT datname as "Relation", datacl as "Access permissions" FROM pg_database;
```

This access permissions query returns a string of characters in the datacl column. Each character represents a granted privilege. The meaning for each character and privilege can be found in the PostgreSQL documentation:

```
rolename=xxxx -- privileges granted to a role
        =xxxx -- privileges granted to PUBLIC

            r -- SELECT ("read")
            w -- UPDATE ("write")
            a -- INSERT ("append")
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
            X -- EXECUTE
            U -- USAGE
            C -- CREATE
            c -- CONNECT
            T -- TEMPORARY
      arwdDxt -- ALL PRIVILEGES (for tables, varies for other objects)
            * -- grant option for preceding privilege

        /yyyy -- role that granted this privilege
```

The output for the access permissions query is similar to the following:

```
Relation              |Access permissions
 ----------------------|------------------------------------------------------------
 cloudsqladmin |
 template0       |{=c/cloudsqladmin,cloudsqladmin=CTc/cloudsqladmin}
 postgres          |
 template1       |{=c/cloudsqlsuperuser,cloudsqlsuperuser=CTc/cloudsqlsuperuser}
```

If you refer to the template0 database as an example, the access permissions are as follows: {=c/cloudsqladmin,cloudsqladmin=CTc/cloudsqladmin}.

The meaning of this string is:

- cloudsqladmin role grant CONNECT access to PUBLIC
- cloudsqladmin role grant CONNECT, CREATE, and TEMPORARY for itself

Use the example below to view the permissions granted for specific database tables:

```
SELECT * FROM information_schema.table_privileges;
```

Use the following example to view the users that have the CONNECT permission granted on a specific database:

```
SELECT rolename FROM (SELECT rol.rolename, db.datacl::text,substring (substring(db.datacl::text from position(','
|| rol.rolname || '=' in db.datacl::text)+character_length(',' || rol.rolename || '=')) from 0 for 5) as "permissions"
FROM pg_roles rol, pg_database db
WHERE db.dataname = 'emp_db') as priv
WHERE position(',' || priv.rolname || '=' in priv.datacl) >0 AND permissions LIKE '%c/%';
```

**Note:** In the example above, replace the database name with the name of your specific database.

## 7.7 Migrating Oracle Encryption to PostgreSQL

Oracle provides TDE (Transparent Data Encryption) to handle encryption of database storage (also known as "encryption at rest") and Advanced Security for handling encryption of data over the network (also known as "encryption in transit").

Similarly, Cloud SQL for PostgreSQL data is encrypted when stored in database tables, temporary files, and backups. External connections can be encrypted by using SSL, or by using the Cloud SQL Proxy.

Encryption at rest
Data that is not moving through networks (stored) is known as "data at rest." This data is encrypted using 256-bit Advanced Encryption Standard (AES-256), or better. The same key is used (called

symmetric keys) to encrypt the data when it is stored, and to decrypt it when it is used. These data keys are encrypted using a master key stored in a secure keystore and changed regularly. For more information about Encryption at rest, see Encryption at Rest in Google Cloud Platform.

Encryption in transit
Google encrypts and authenticates all data in transit at one or more network layers when data moves outside physical boundaries not controlled by Google or on behalf of Google. Data in transit inside a physical boundary controlled by or on behalf of Google is generally authenticated but might not be encrypted by default. You can choose which additional security measures to apply based on your threat model. For example, you can configure SSL for intra-zone connections to Cloud SQL. For information about Encryption in transit, see Encryption in Transit in Google Cloud.

## 7.8 Migrating Administrative (DBA) Queries from Oracle to PostgreSQL

Oracle provides dba_ and V$ views which contain useful information about the database, object metadata, and performance information. These are views which are updated dynamically and are frequently used by Database Administrators for database operations, management, and monitoring. Examples of the kind of information provided by the Oracle V$ and DBA_ views include: session data, memory, usage, progress of jobs, SQL executions, statistics and various other details about the instance.

PostgreSQL provides equivalent views which can be used to query information regarding the database and object metadata.

| Purpose | Oracle V$ view | PostgreSQL table |
|---|---|---|
| Details on connected sessions | V$SESSION | PG_STAT_ACTIVITY |
| Instance information | V$INSTANCE | No direct equivalent |
| Memory advisor | V$MEMORY_TARGET_ADVICE | No direct equivalent, find example below |
| Database information | V$DATABASE | PG DATABASE |
| The parameters the are being used and their values | V$PARAMETER | PG_SETTINGS |
| Long operations progress details | V$SESSION_LONG_OPS | pg_stat_progress_vacuum (for VACUUM operations only) |
| Locked objects information | V$LOCKED_OBJECT | PG_LOCKS |
| System statistics | V$SYSSTAT | PG_STAT_DATABASE |
| IO operations statistics | V$SEGSTAT | PG_STATIO_ALL_TABLES |
| Used nls parameters | V$NLS_PARAMETERS | PG_SETTINGS |
| All open cursors | V$OPEN_CURSOR | PG_CURSORS |
| IO operations statistics | V$FILESTAT | PG_STATIO_ALL_TABLES |

Oracle Example:
To query the size of the PGA and SGA (Oracle memory cache), these queries can be used:

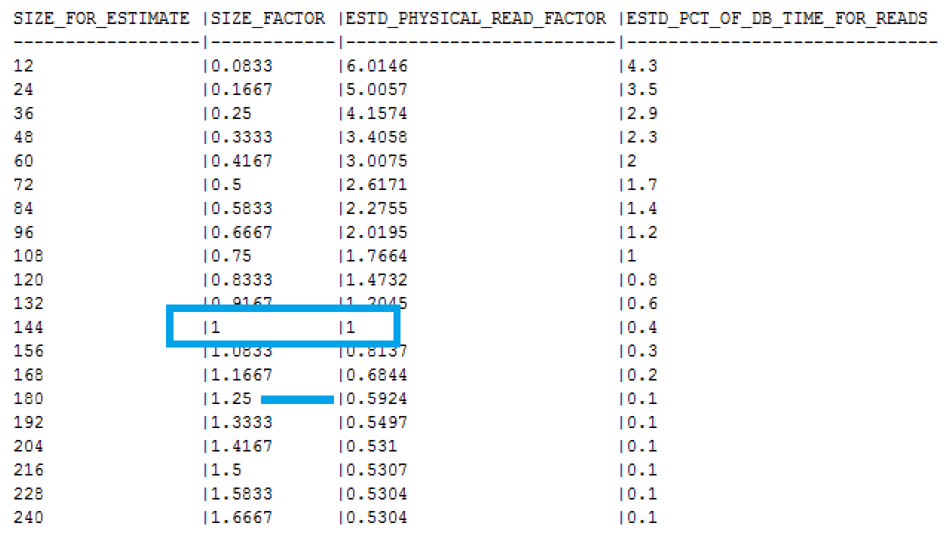
```
select sum(value)/1073741824 "SGA allocated in GB" from v$sga;
```

```
select value/1073741824 "PGA allocated in GB" from v$pgastat where name = 'total PGA allocated';
```

Oracle also provides many other views that can help you understand what the best cache size for you is and what will be the impact where changing the cache sizes.
For example, one cache can be the buffer cache that holds the data.
Run the query:

```
SELECT SIZE_FOR_ESTIMATE, SIZE_FACTOR, ESTD_PHYSICAL_READ_FACTOR,
ESTD_PCT_OF_DB_TIME_FOR_READS FROM V$DB_CACHE_ADVICE;
```

The results are similar to the following example. Notice where the SIZE_FACTOR is "1" and then check what happens if you will increase or decrease the cache size:



```
SIZE_FOR_ESTIMATE |SIZE_FACTOR |ESTD_PHYSICAL_READ_FACTOR |ESTD_PCT_OF_DB_TIME_FOR_READS
------------------|------------|--------------------------|------------------------------
12                |0.0833      |6.0146                    |4.3
24                |0.1667      |5.0057                    |3.5
36                |0.25        |4.1574                    |2.9
48                |0.3333      |3.4058                    |2.3
60                |0.4167      |3.0075                    |2
72                |0.5         |2.6171                    |1.7
84                |0.5833      |2.2755                    |1.4
96                |0.6667      |2.0195                    |1.2
108               |0.75        |1.7664                    |1
120               |0.8333      |1.4732                    |0.8
132               |0.9167      |1.2045                    |0.6
144               |1           |1                         |0.4
156               |1.0833      |0.8137                    |0.3
168               |1.1667      |0.6844                    |0.2
180               |1.25        |0.5924                    |0.1
192               |1.3333      |0.5497                    |0.1
204               |1.4167      |0.531                     |0.1
216               |1.5         |0.5307                    |0.1
228               |1.5833      |0.5304                    |0.1
240               |1.6667      |0.5304                    |0.1
```

You can see in the example above, that if the buffer cache is increased by 25%, the physical reads will be reduced by 40%.

PostgreSQL Example:
Memory advisors do not exist natively in PostgreSQL. However, the pg_buffercache extension is supported in Cloud SQL for PostgreSQL, which can provide useful information regarding the PostgreSQL buffer cache.

Use the following example to run the CREATE EXTENSION command from within the PostgreSQL database:

```
CREATE EXTENSION pg_buffercache;
```

Use the query below to determine per-object usage of the buffer cache:

```
SELECT  c.relname,
       pg_size_pretty(count(*) * 8192) as buffered, round(100.0 * count(*) / (SELECT setting FROM pg_settings
WHERE name='shared_buffers')::integer,1) AS buffers_percent,
       round(100.0 * count(*) * 8192 / pg_relation_size(c.oid),1) AS percent_of_relation,
       round(100.0 * count(*) * 8192 / pg_table_size(c.oid),1) AS percent_of_table
 FROM    pg_class c
       INNER JOIN pg_buffercache b
          ON b.relfilenode = c.relfilenode
       INNER JOIN pg_database d
          ON (b.reldatabase = d.oid AND d.datname = current_database())
 GROUP BY c.oid,c.relname
 ORDER BY 3 DESC;
```

The following example displays the output returned by the query above, showing a list of all tables currently occupying space in the database buffer cache:

```
relname                              |buffered    |buffers_percent  |percent_of_relation  |percent_of_table
-------------------------------------------------|---------------|------------------------|----------------------------|-----------------
 pg_default_acl_role_nsp_obj_index |8192 bytes |0.0              |100.0                |100.0
 pg_default_acl_oid_index          |8192 bytes |0.0              |100.0                |100.0
 pg_aggregate                      |16 kB      |0.0              |100.0                |33.3
 pg_am                             |8192 bytes |0.0              |100.0                |20.0
 pg_amop                           |48 kB      |0.0              |100.0                |60.0
 pg_amproc                         |40 kB      |0.0              |100.0                |55.6
 pg_cast                           |16 kB      |0.0              |100.0                |33.3
 pg_depend                         |80 kB      |0.0              |18.2                 |16.9
 pg_description                    |48 kB      |0.0              |17.6                 |15.4
 pg_index                          |24 kB      |0.0              |100.0                |42.9
 pg_language                       |8192 bytes |0.0              |100.0                |20.0
 pg_namespace                      |8192 bytes |0.0              |100.0                |20.0
```

Use the output of the query to determine if you need to increase the size of the database buffer cache.

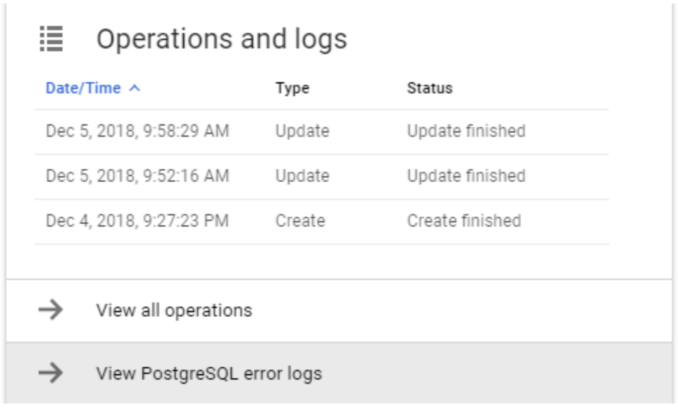
## 7. 9 Migrating Oracle Troubleshooting Methodologies using Log Files and Trace Files to PostgreSQL

Oracle's central error log is known as the alert log file. The alert log file (alert.log) contains details about database events, operations, and error messages. It is usually the first-place database administrators look when attempting to troubleshoot or investigate database issues.
Oracle Example, the following example displays sample output from the Oracle alert.log:

```
Thu Nov 29 23:48:11 2018
Thread 1 advanced to log sequence 74669 (LGWR switch)
Current log# 3 seq# 74669 mem# 0:
/db/ORCL_A/onlinelog/o1_mf_3_f71kq2xd_.log
Thu Nov 29 23:48:11 2018
Archived Log entry 74662 added for thread 1 sequence 74668 ID 0x5914db01 dest 1:
```

```
Thu Nov 29 23:53:12 2018
Thread 1 advanced to log sequence 74670 (LGWR switch)
Current log# 4 seq# 74670 mem# 0:
/db/ORCL_A/onlinelog/o1_mf_4_f71kq8cj_.log
Thu Nov 29 23:53:12 2018
Archived Log entry 74663 added for thread 1 sequence 74669 ID 0x5914db01 dest 1:
Thu Nov 29 23:58:12 2018
Thread 1 advanced to log sequence 74671 (LGWR switch)
Current log# 1 seq# 74671 mem# 0:
/db/ORCL_A/onlinelog/o1_mf_1_f71kq701_.log
Thu Nov 29 23:58:12 2018
Archived Log entry 74664 added for thread 1 sequence 74670 ID 0x5914db01 dest 1:
```

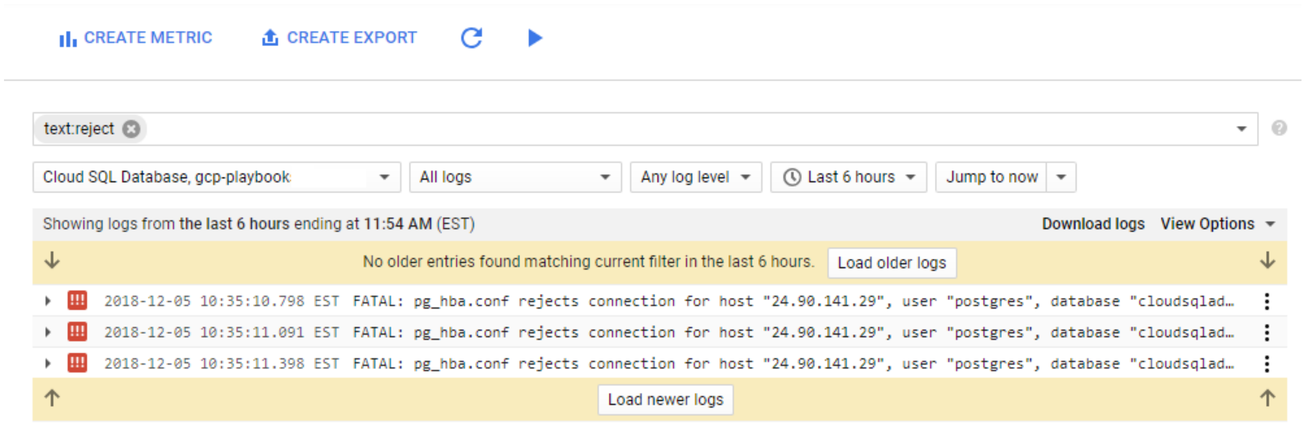
PostgreSQL Example:
When running Cloud SQL PostgreSQL databases in Google Cloud, there is no access to the underlying database filesystem. The PostgreSQL error log is accessible through the console.
In the dashboard page of the instance, you can scroll down to the Operations and logs section. Here you'll find the View PostgreSQL error logs button to open the PostgreSQL error logs.



| Date/Time | Type | Status |
|---|---|---|
| Dec 5, 2018, 9:58:29 AM | Update | Update finished |
| Dec 5, 2018, 9:52:16 AM | Update | Update finished |
| Dec 4, 2018, 9:27:23 PM | Create | Create finished |

After clicking on the View PostgreSQL error logs button, you will be redirected to the PostgreSQL error log page where you can filter the logs to find the information you need.

You can filter the output by any keyword. In the example below, use the word "reject" as a filter. You could filter for log type "activity" for GCP operations, such as starting, stopping the instance, or changing the resources. You could also filter the output by using "Postgres log" for the database error log (the equivalent of alert.log in Oracle).



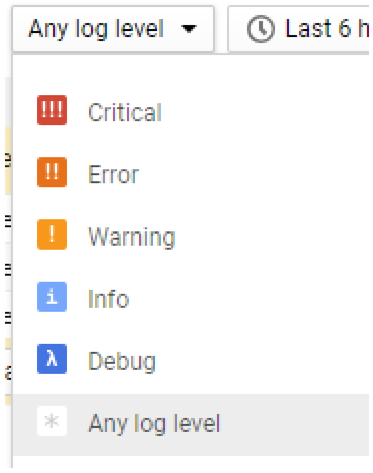
Showing logs from the last 6 hours ending at 11:54 AM (EST)

No older entries found matching current filter in the last 6 hours.

```
2018-12-05 10:35:10.798 EST  FATAL: pg_hba.conf rejects connection for host "24.90.141.29", user "postgres", database "cloudsqlad…
2018-12-05 10:35:11.091 EST  FATAL: pg_hba.conf rejects connection for host "24.90.141.29", user "postgres", database "cloudsqlad…
2018-12-05 10:35:11.398 EST  FATAL: pg_hba.conf rejects connection for host "24.90.141.29", user "postgres", database "cloudsqlad…
```

Additionally, you can filter based on time frame of the database events or errors. In the example above, the last six hours are selected as the time frame for displaying the logs. You can also choose to download the logs, if needed.

You can configure the logging level, such as capturing user-defined events generated inside PostgreSQL functions. The following graphic displays the available log levels:



| Log level | Details that can be found in this level |
|---|---|
| Debug1 to Debug5 | Selective detailed messages - often used by developers |
| Info | Messages implicitly requested by the user |
| Warning | Messages most raised by issues happened in the database |
| Error | Message about an issue that caused command to abort |
| Critical | Error message caused a session to disconnect or might harm the database |

Database Administrators can use the PostgreSQL RAISE command to send messages to the PostgreSQL Error log.
Use the following example to send messages to the PostgreSQL Error log:

```
RAISE DEBUG 'This is the message from your developers';
```

**Note:** Cloud SQL PostgreSQL does not currently provide the equivalent to Oracle's session tracing capabilities.

## 7.10 Migrating Oracle Troubleshooting Methodologies for Database Locks from Oracle to PostgreSQL

DIagnosing transaction locks in the database is a common daily task for most Database Administrators. Both Oracle and PostgreSQL provide database metadata views allowing you to troubleshoot locking scenarios between sessions, applications, or conflicting SQL statements.
Oracle Example:
To find locks, use the following example:

```
SELECT * FROM V$LOCKED_OBJECT;
```

Terminate a session using the following command:

```
ALTER SYSTEM KILL SESSION 'SID,#SERIAL' IMMEDIATE;
```

PostgreSQL Example:
Use the following example to retrieve most of the locking related information in PostgreSQL using pg_locks:

```
SELECT relation::regclass, * FROM pg_locks WHERE NOT GRANTED;
```

The following example displays a SQL query that will JOIN that view with the pg_stat_activity view for the sessions list:

```
 SELECT blocked_locks.pid     AS blocked_pid,
        blocked_activity.usename  AS blocked_user,
        blocking_locks.pid     AS blocking_pid,
        blocking_activity.usename AS blocking_user,
        blocked_activity.query    AS blocked_statement,
        blocking_activity.query   AS current_statement_in_blocking_process
  FROM  pg_catalog.pg_locks         blocked_locks
   JOIN pg_catalog.pg_stat_activity blocked_activity  ON blocked_activity.pid = blocked_locks.pid
   JOIN pg_catalog.pg_locks         blocking_locks
      ON blocking_locks.locktype = blocked_locks.locktype
      AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
      AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
      AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
      AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
      AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
      AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
      AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
      AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
```

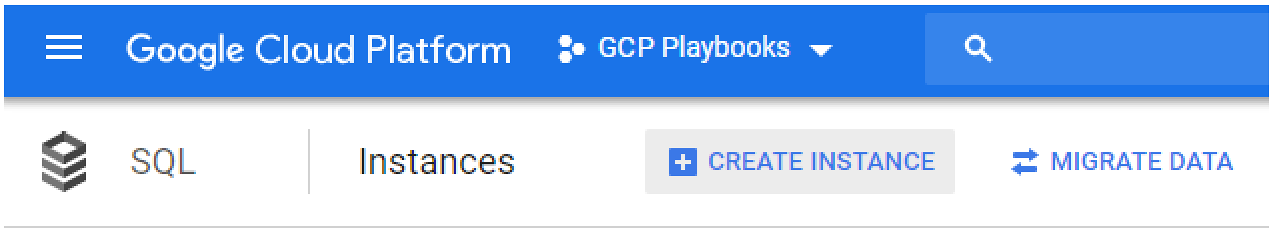
```
    AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
    AND blocking_locks.pid != blocked_locks.pid

  JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid = blocking_locks.pid
 WHERE NOT blocked_locks.GRANTED;
```

After you will find the pid of the session causing the lock, the following command can be used to terminate that session:

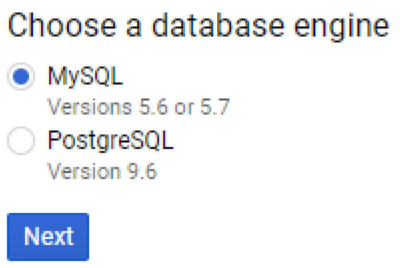
```
select pg_terminate_backend(PID);
```

## 7.11 Common PostgreSQL DBA Tasks for Oracle Database Administrators

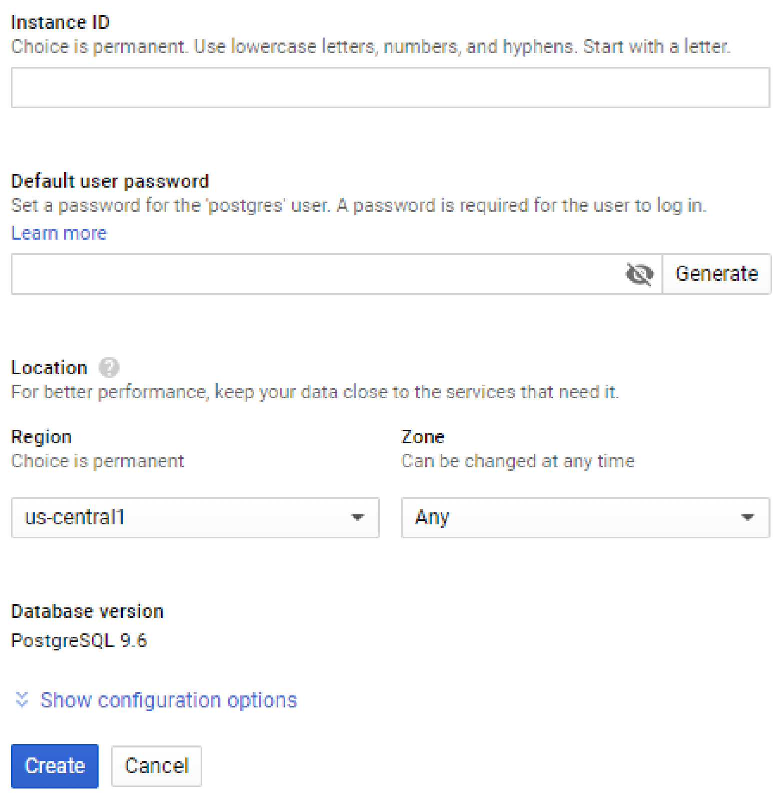
This section covers some of the most common Database Administrator tasks relevant for Oracle DBAs transitioning to PostgreSQL DBA roles.
PostgreSQL Example:
To create a Cloud SQL for PostgreSQL database, navigate to the Cloud SQL dashboard and click Create Instance.



Choose the relevant database engine for your database and click Next.


Choose a database engine
- [✓ MySQL] Versions 5.6 or 5.7
- PostgreSQL Version 9.6

Specify the instance name, the password for the default user, region, and zone.


Instance ID
Choice is permanent. Use lowercase letters, numbers, and hyphens. Start with a letter.

Default user password
Set a password for the 'postgres' user. A password is required for the user to log in.
Learn more

Location
For better performance, keep your data close to the services that need it.

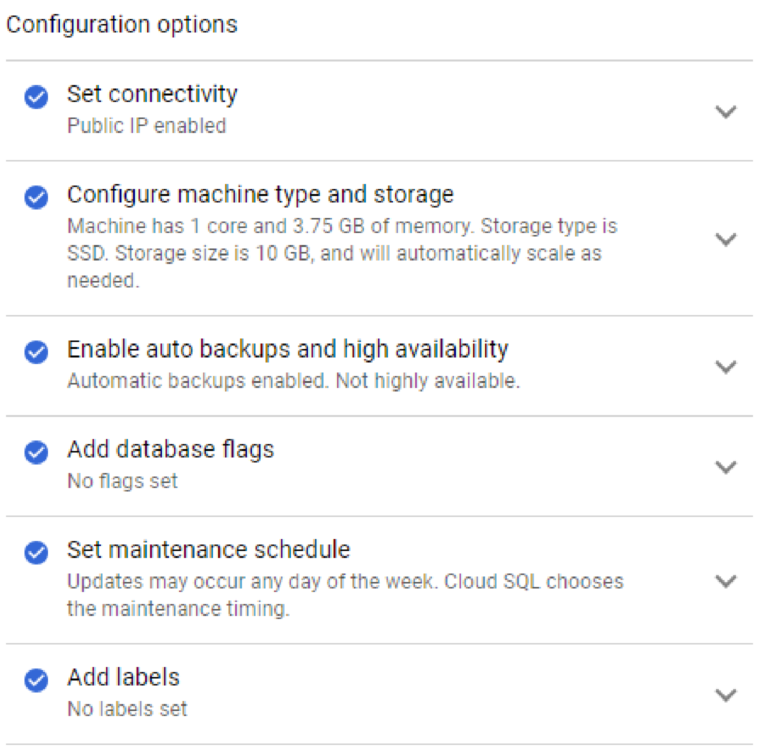
Region
Choice is permanent
us-central1

Zone
Can be changed at any time
Any

Database version
PostgreSQL 9.6

Certain Compute Engine resources live in regions or zones. A region is a specific geographical location where you can run your resources. Each region has one or more zones; most regions have three or more zones. For example, the us-central1 region denotes a region in the Central United States that has zones us-central1-a, us-central1-b, us-central1-c, and us-central1-f.
Resources that live in a zone, such as instances or persistent disks, are referred to as zonal resources. Other resources, like static external IP addresses, are regional. Regional resources can be used by any resources in that region, regardless of zone, while zonal resources can only be used by other resources in the same zone. For more about Region and Zone terms, see Regions and Zones.

In the Configuration Options dialog, you can specify a variety of  options for your Cloud SQL PostgreSQL instance.



Configuration options

Set connectivity
Public IP enabled

Configure machine type and storage
Machine has 1 core and 3.75 GB of memory. Storage type is SSD. Storage size is 10 GB, and will automatically scale as needed.

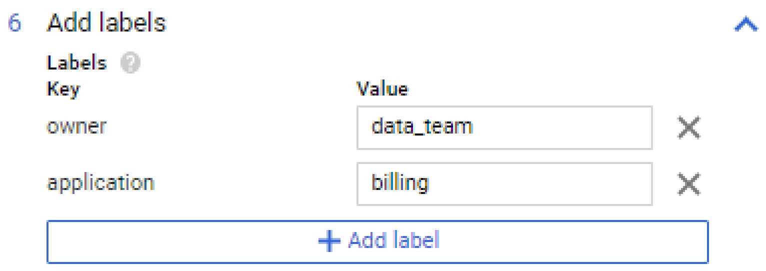
Enable auto backups and high availability
Automatic backups enabled. Not highly available.

Add database flags
No flags set

Set maintenance schedule
Updates may occur any day of the week. Cloud SQL chooses the maintenance timing.

Add labels
No labels set

Database Flags
Parameters that can be changed so it will take effect after every restart. For a list of supported flags, see Configuring Database Flags. You can mark your Cloud SQL databases using Labels. For example, mark your database with environment labels such as QA, DEV, PROD, or the application name. These labels will help you with your administration of resources in the cloud as well as creating reports or running actions which will affect GCP resources with a specific label.

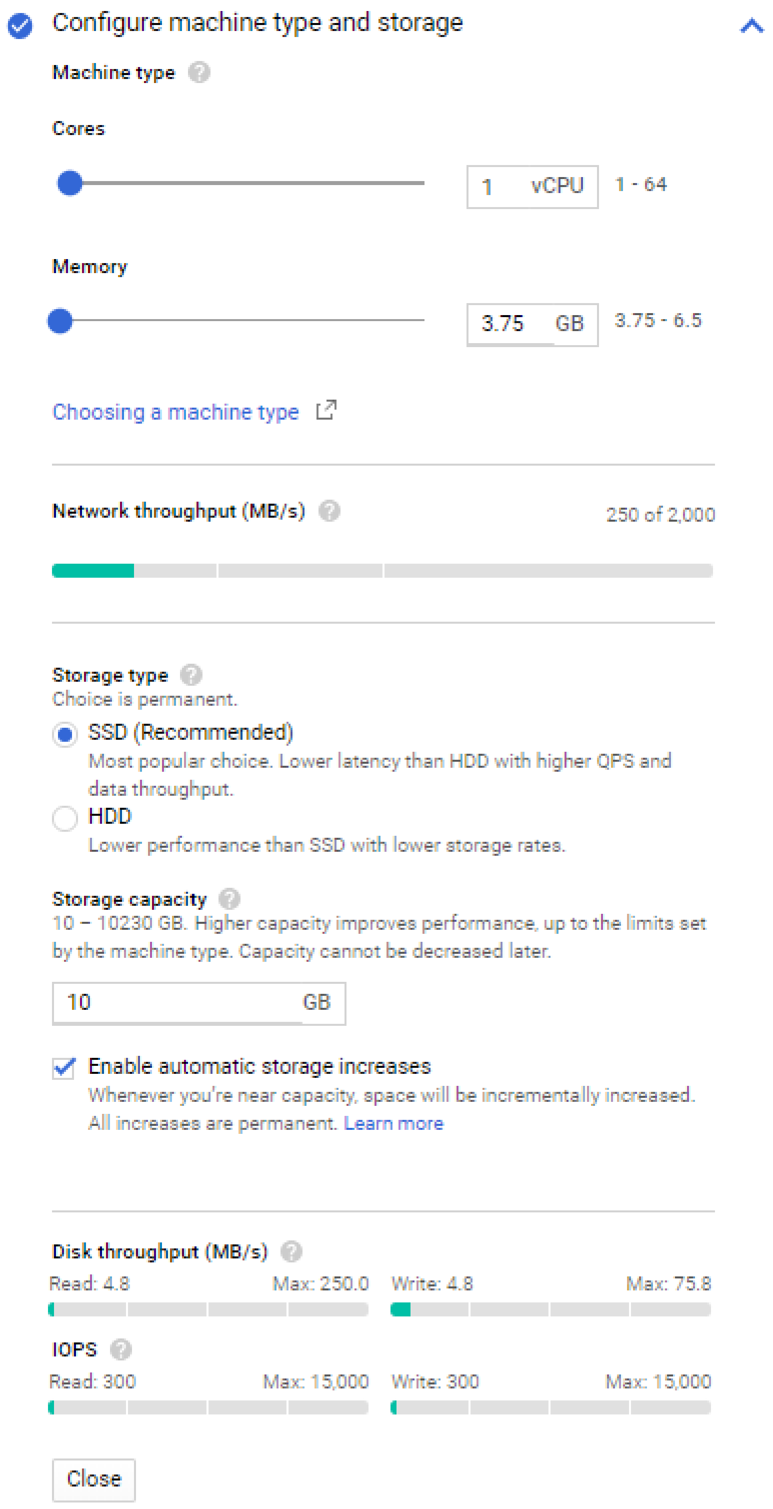


6 Add labels

Labels

| Key | Value |
|---|---|
| owner | data_team |
| application | billing |

If you navigate to the Configuration machine type and storage dialog.



Configure machine type and storage

Machine type

Cores

1 vCPU 1 - 64

Memory

3.75 GB 3.75 - 6.5

Choosing a machine type

Network throughput (MB/s) 250 of 2,000

Storage type
Choice is permanent.

[✓ SSD (Recommended)]
Most popular choice. Lower latency than HDD with higher QPS and data throughput.

HDD
Lower performance than SSD with lower storage rates.

Storage capacity
10 – 10230 GB. Higher capacity improves performance, up to the limits set by the machine type. Capacity cannot be decreased later.

10 GB

[✓] Enable automatic storage increases
Whenever you're near capacity, space will be incrementally increased. All increases are permanent. Learn more

Disk throughput (MB/s)
Read: 4.8 Max: 250.0 Write: 4.8 Max: 75.8

IOPS
Read: 300 Max: 15,000 Write: 300 Max: 15,000

Close

The Configuration machine type and storage dialog provides all the definitions for the hardware that you should set, based on requirements. GCP creates the relevant instance for you.
Click Create. After a few minutes your database will be up and running.

Starting, stopping and restarting instances can be done from the console or using the gcloud component using the commands below:
Start your instance by using the following command:

```
gcloud sql instances patch [INSTANCE_NAME] --activation-policy ALWAYS
```

Stop your instance by using the following command:

```
gcloud sql instances patch [INSTANCE_NAME] --activation-policy NEVER
```

Restart your instance by using the following command:

```
gcloud sql instances restart [INSTANCE_NAME]
```

Delete your instance by using the following command:

**Warning:** All data on an instance, including backups, is permanently lost when that instance is deleted. To preserve your data, export the instance before you delete it.

```
gcloud sql instances delete [INSTANCE_NAME]
```

If you want to query your instances for scripting purposes without connecting the console:

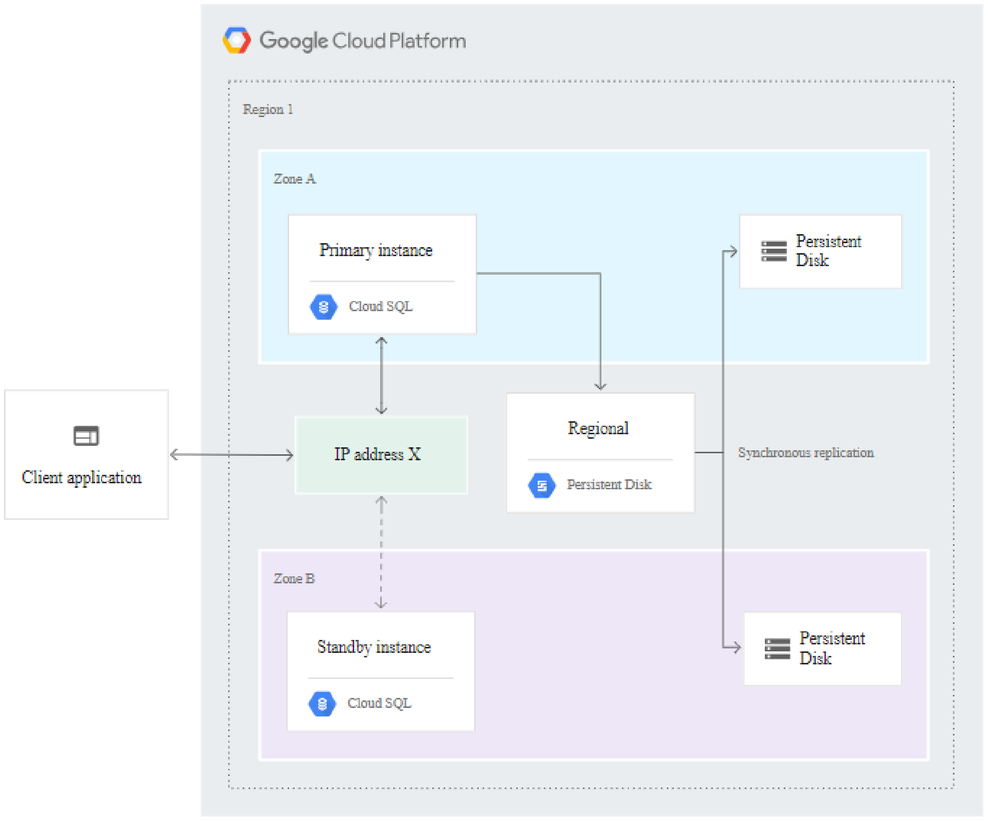
```
gcloud sql instances describe [INSTANCE_NAME]
```

If you want to add another instance for scripting purposes, use the following command:

```
gcloud sql instances create [INSTANCE_NAME] --database-version=POSTGRES_9_6 --cpu=[NUMBER_CPUS] --
memory=[MEMORY_SIZE]
```

For more about these commands, see Starting, Stopping and Restarting Instances.
A complex task for the DBA is adding another server to use for high availability purposes. This can be done with Cloud SQL for PostgreSQL by configuring the setting for the instance to high availability. This operation can be done when the instance is being created or after the instance is up and running.
Use the example below to configure an instance for high availability:

```
gcloud beta sql instances patch [INSTANCE_NAME] --availability-type REGIONAL
```



For more about configuring an instance for high availability, see High Availability.
To initiate a failover that will take a few minutes from one instance to another, use the following example:

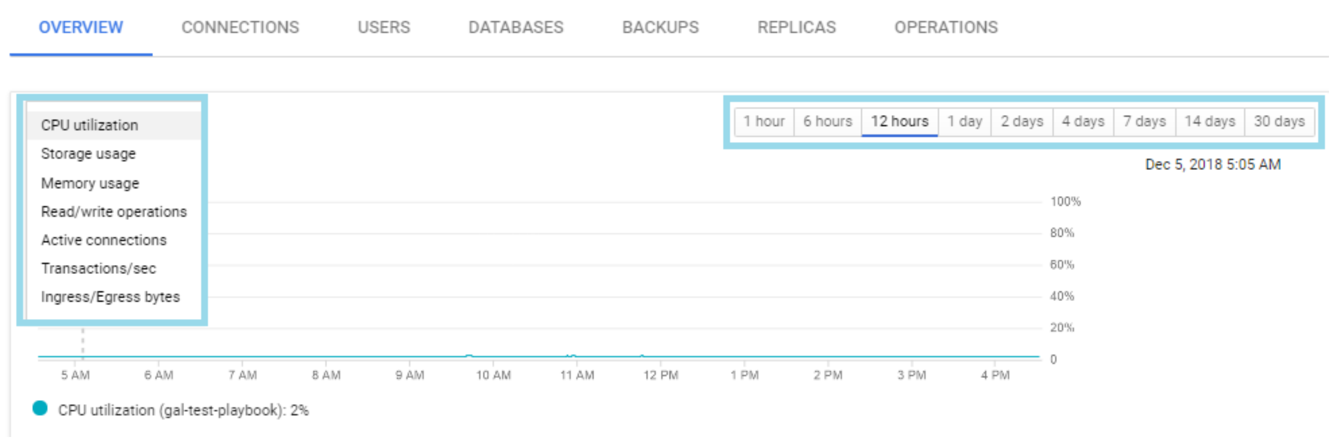
```
gcloud sql instances failover [MASTER_INSTANCE_NAME]
```

Use the following example to add read scaling to your instance:

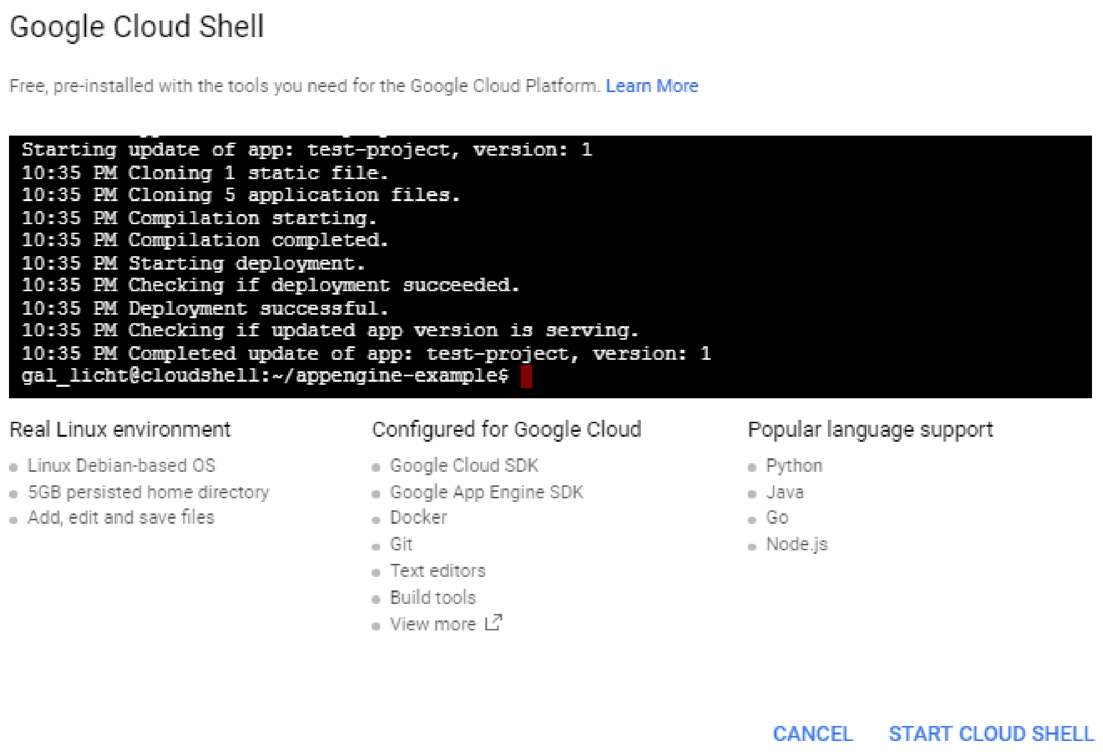
```
gcloud sql instances create [REPLICA_NAME] --master-instance-name=[MASTER_INSTANCE_NAME]
```

**Note:** You can have both read replicas and setting for high availability

For more about read scaling, see Read Replicas.
For monitoring, you are probably using Oracle Cloud Control. After creating the Cloud SQL instance you can use the instance page dashboard instead.
The instance page dashboard displays the metrics you can view and provides various levels of granularity that you can set.



This information can be very helpful to investigate resource consumption during an event or an issue. You can also use the Connect Using Cloud Shell button under the Connect to This Instance section in the instance page dashboard. This is a great option for DBAs who like using the CLI for their daily operations.



Google Cloud Shell

Free, pre-installed with the tools you need for the Google Cloud Platform. Learn More

```
Starting update of app: test-project, version: 1
10:35 PM Cloning 1 static file.
10:35 PM Cloning 5 application files.
10:35 PM Compilation starting.
10:35 PM Compilation completed.
10:35 PM Starting deployment.
10:35 PM Checking if deployment succeeded.
10:35 PM Deployment successful.
10:35 PM Checking if updated app version is serving.
10:35 PM Completed update of app: test-project, version: 1
gal_licht@cloudshell:~/appengine-example$
```

Real Linux environment
- Linux Debian-based OS
- 5GB persisted home directory
- Add, edit and save files

Configured for Google Cloud
- Google Cloud SDK
- Google App Engine SDK
- Docker
- Git
- Text editors
- Build tools
- View more

Popular language support
- Python
- Java
- Go
- Node.js

In the table below, you can find some common Oracle commands and the corresponding command in PostgreSQL.

| Purpose | Oracle command | Postgres command |
|---|---|---|
| Interacting with log files | alter database force logging<br>alter system switch logfile<br>alter database add logfile group.... | Not needed in PostgreSQL |
| Terminate session | alter system kill session 'SID, SERIAL#' | SELECT pg_terminate_backend(pid) |
| Flushing the Buffer Cache | alter system flush buffer_cache | No equivalent, use pg_buffercache extension to determine what's inside the database |
| Create restricted session | alter system enable restricted session | Revoke connect privileges from roles and<br>terminate all sessions:<br>select pg_terminate_backend(procpid) from pg_stat_activity where datname = 'emps_db' and procpid <> pg_backend_pid(); |
| GRANT/REVOKE privileges | GRANT/REVOKE ... TO/FROM | GRANT/REVOKE ... TO/FROM |
| Managing jobs | Using scheduler or Oracle jobs | Using Cloud Scheduler and Functions, Cloud Composer. |
| Monitoring | Using OEM | Using instance dashboard and metrics |
| Starting, stopping, or restart the database | Using SQL*Plus | Using gcloud or the console |
| Export/Import | Using Oracle Data Pump | GCP console or pgdump/psql |
| Run scripts | Using Oracle SQL*Plus | GCP console or psql |
| Managing backups | Using RMAN and scripts | No need to manage backups, only set it in the console or using the gcloud . See manage backups. |
| Manage high availability solution | Setup and config Oracle Data Guard or Oracle RAC | Enable high availability for your instance using the console or gcloud |
| Manage reading scale solutions | Setup and config Oracle RAC | Add Read Replicas to your instance using the console or gcloud |

# 8. Migration Topics

## 8.1 Application Aspects

Oracle has many features that can achieve almost each required functionality from the database.

When migrating from this database type to another, the DBA will need to map these features and determine how to provide the same functionality from their new database.
The next chapter describes some of the most commonly used features in Oracle and what should be done to get the same results in Cloud SQL for PostgreSQL.

Moving between heterogeneous database types can require some out-of-the-box thinking. When you plan your migration, you should also consider using other GCP services like GCP Functions, Pub/Sub, Dataproc, or Dataflow to develop a more complete understanding of available features and functionality.

## 8.2 Data Types - Key Differences

Most of the common Oracle data types have comparable options in PostgreSQL. The data types can be used as the column's data types or in the PL/SQL / PG/SQL code as variables.
The data type enforces the integrity of a specific value and allows other functions or operators to be used with those values.

### All comparable data type

In **bold** you'll find all the data types which are not fully compatible in terms of syntax between Oracle RDBMS and PostgreSQL.

| Oracle | PostgreSQL Equivalent | Comments |
|---|---|---|
| CHAR(n) | CHAR(n) | Maximum size of 2000 bytes in Oracle<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| CHARACTER(n) | CHARACTER(n) | Maximum size of 2000 bytes in Oracle<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| NCHAR(n) | CHAR(n) | Maximum size of 2000 bytes in Oracle<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |

| | | |
|---|---|---|
| VARCHAR(n) | VARCHAR(n) | Maximum size of 2000 bytes in Oracle<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| NCHAR VARYING(n) | CHARACTER VARYING(n) | Varying-length UTF-8 string Maximum size of 4000 bytes in Oracle<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| VARCHAR2(n) 11g | VARCHAR(n) | Maximum size of 4000 bytes Maximum size of 32KB in PL/SQL<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| VARCHAR2(n) 12g | VARCHAR(n) | Maximum size of 32767 bytes MAX_STRING_SIZE= EXTENDED<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| NVARCHAR2(n) | VARCHAR(n) | Maximum size of 4000 bytes<br>"n" is for bytes in Oracle and for number of characters in PostgreSQL |
| LONG | TEXT | Maximum size of 2GB in Oracle or 1G in PostgreSQL |
| RAW(n) | BYTEA | Maximum size of 2000 bytes in Oracle or 1G in PostgreSQL |
| LONG RAW | BYTEA | Maximum size of 2GB in Oracle or 1G in PostgreSQL |
| NUMBER | NUMERIC(n,m) | Floating-point number<br>If SMALLINT, INT or BIGINT will be used in PostgreSQL then the performance will improve |
| NUMBER(*) | DOUBLE PRECISION | Floating-point number |
| NUMBER(p,s) | DECIMAL(p,s) | Precision can range from 1 to 38 Scale can range from -84 to 127 |
| NUMERIC(p,s) | NUMERIC(p,s) | Precision can range from 1 to 38 |
| FLOAT(p) | DOUBLE PRECISION | Floating-point number |
| DEC(p,s) | DEC(p,s) | Fixed-point number |
| DECIMAL(p,s) | DECIMAL(p,s) | Fixed-point number |
| INT | INTEGER / NUMERIC(38,0) | 38 digits |
| INTEGER | INTEGER / NUMERIC(38,0) | 38 digits |
| SMALLINT | SMALLINT | 38 digits |
| REAL | DOUBLE PRECISION | Floating-point number |
| DOUBLE PRECISION | DOUBLE PRECISION | Floating-point number |
| DATE | TIMESTAMP(0) | In Oracle, stores date and time, there is also DATE data type in PostgreSQL but it won't return time |
| TIMESTAMP(p) | TIMESTAMP(p) | Date and time with fraction |
| TIMESTAMP(p) WITH TIME ZONE | TIMESTAMP(p) WITH TIME ZONE | Date and time with fraction and time zone |
| INTERVAL YEAR(p) TO MONTH | INTERVAL YEAR TO MONTH | Date interval |

| | | |
|---|---|---|
| INTERVAL DAY(p) TO SECOND(s) | INTERVAL DAY TO SECOND(s) | Day and time interval |
| BFILE | VARCHAR (255) / CHARACTER VARYING (255) | Pointer to binary file Maximum file size of 4G in Oracle |
| BLOB | BYTEA | Binary large object Maximum file size of 4G in Oracle |
| CLOB | TEXT | Character large object Maximum file size of 4G in Oracle or 1G in PostgreSQL |
| NCLOB | TEXT | Unicode string maximum size of 4G or 1G in PostgreSQL |
| ROWID | CHARACTER (255) | Physical row address |
| UROWID(n) | CHARACTER VARYING | Universal row id logical row addresses |
| XMLTYPE | XML | XML data |
| BOOLEAN | BOOLEAN | TRUE or FALSE values |
| SDO_GEOMETRY | N/A | Geometric description of a spatial object |
| SDO_TOPO_GEOMETRY | N/A | Describes a topology geometry |
| SDO_GEORASTER | N/A | A raster grid or image object is stored in a single row |
| ORDDicom | N/A | Audio data |
| ORDDicom | N/A | Digital Imaging and Communications in Medicine (DICOM), |
| ORDDoc | N/A | Media data |
| ORDImage | N/A | Image data |
| ORDVideo | N/A | Video data |

If your Oracle database has data types with no comparable equivalent in PostgreSQL, the database can still be migrated. You will need to investigate if the same functionality may be available by changing the application or using other GCP components such as Cloud Dataflow to translate between those data types.

## 8.3 Backup and Recovery Features - Key Differences

Oracle provides the Recovery Manager (RMAN) for the backup and recovery of databases. Using RMAN, you can specify different levels for incremental backups.If you have the prerequisites you can also use FLASHBACK DATABASE and FLASHBACK TABLE to recover from logical errors. The FLASHBACK DATABASE command returns the database to a target time, a System Change Number (SCN) or a log sequence number.

A FLASHBACK DATABASE recovery time is dependent on the number of transactions from the selected target time, SCN, or log sequence number. A FLASHBACK TABLE command restores an existing table to earlier versions using a timestamp, restore point, or SCN. You may also be able to recover from a DROP TABLE or TRUNCATE TABLE command. The table is restored to a former version and may undo needed updates or deletes.

Cloud SQL provides backup capabilities, but does not support different incremental levels, or a point-in-time backup.

In Cloud SQL, backup and recovery tasks are handled using on-demand backups. You can use a backup to recover lost data or repair a problem with your instance. To reduce the risk of lost or damaged data, any instance should have automated backups.

Cloud SQL creates a special database user, cloudsqladmin, for each instance, and generates a unique instance-specific password for this account. Cloud SQL logs in as the cloudsqladmin user to perform automated backups.

Backup data is stored in two regions for redundancy. If there are two regions on the continent, the backup data remains within the same continent. If there is only one GCP region in a continent, the backup data will be stored in the next nearest region.

Database operations (such as writes) are unaffected by backup operations.
Cloud SQL backups are incremental; they only contain data that has changed since the previous backup. This means that your oldest backup is similar in size to your database, but the sizes of subsequent backups depend on the rate of change of your data. When the oldest backup is deleted, the size of the next oldest backup increases so that a full backup always exists.

You can create on-demand backups for any instance at any time, whether the instance has automatic backups enabled or not. This option could be useful if you are about to perform a risky operation on your database or if you need a backup and you do not want to wait for the backup window. On-demand backups are not automatically deleted the way automated backups are. On-demand backups persist until you delete them or until their instance is deleted. n-demand backups may have a long-term effect on your billing charges if you do not delete them in a timely manner.

## 8.4 Backup and Recovery Features

Backup and recovery are essential subject in each data store solution.
This topic reviews the equivalent options for the Oracle features listed below:

- RMAN
- Flashback Database
- Flashback Table

In Cloud SQL, backup and recovery tasks are handled using on-demand backup taken on the database.

Backups provide a way to restore your Cloud SQL instance to recover lost data or recover from a problem with your instance. You should enable automated backups for any instance that contains data that you need to protect from loss or damage.

You can create a backup at any time. This could be useful if you are about to perform a risky operation on your database, or if you need a backup and you do not want to wait for the backup window. You can create on-demand backups for any instance, whether the instance has automatic backups enabled or not.

On-demand backups are not automatically deleted the way automated backups are. They persist until you delete them or until their instance is deleted. Because they are not automatically deleted, on-demand backups can have a long-term effect on your billing charges if you do not delete them

When you enable automated backups, you specify a 4-hour backup window. The backup will start during the backup window. When possible, schedule backups when your instance has the least activity and if your data has not changed since your last backup, no backup is taken.
Cloud SQL retains up to 7 automated backups for each instance. The storage used by backups is charged at a reduced rate, it depends on the region that the backups will be stored. For more about the pricing list, see Cloud SQL for PostgreSQL Pricing.

For more on how to create or manage on-demand and automatic backups, see Creating and Managing On-Demand and Automatic Backups..
For scripting purposes, this topic focuses on the gcloud commands. For more about these commands, see gcloud SQL Backups.

To take on-demand backup, use the following steps:

1. Go to the Cloud SQL Instances page in the Google Cloud Platform Console.
2. Click the instance to open its Overview page.
3. Click the Backups tab.
4. Click Create backup.
5. On the Create backup page, add a description if needed and click Create.
6.

or use the following gcloud command:

```
gcloud sql backups create --async --instance [INSTANCE_NAME]
```

**Note:** You can provide a description of the backup using the --description parameter.

To enable automatic backups just schedule the start time of the backups using the following steps:

1. Go to the Cloud SQL Instances page in the Google Cloud Platform Console.
2. Select the instance for which you want to configure backups.
3. Click Edit.
4. In the Backups section, check Enable automated backups, and choose a backup window.
5. Click Save.

or use the following gcloud command:

```
gcloud sql instances patch [INSTANCE_NAME] --backup-start-time [HH:MM]
```

**Note:** The `backup-start-time` parameter is specified in 24-hour time, in the UTC±00 time zone, and specifies the start of a 4-hour backup window. Backups can start any time during the backup window.

You can check if the schedule command was already set by using the following gcloud command:

```
gcloud sql instances describe [INSTANCE_NAME]
```

**Note:** Check that under backup Configuration you should see enabled: true and the time that you specified.

In order to disable automatic backups, use the following steps:

1. Go to the [Cloud SQL Instances page]() in the Google Cloud Platform Console.
2. Select the instance for which you want to disable backups.
3. Click Edit.
4. In the Backups section, uncheck Enable daily automated backups.
5. Click Save.

or use the gcloud command:

```
gcloud sql instances patch [INSTANCE_NAME] --no-backup
```

To query all the available backups, use the following steps:

1. Go to the [Cloud SQL Instances page]() in the Google Cloud Platform Console.
2. Click the instance to open its Overview page.
3. At the bottom right corner, the list of Recent backups is displayed.

or use the gcloud to list all the backups :

```
gcloud sql backups list --instance [INSTANCE_NAME]
```

A possible output will be:

```
ID                 WINDOW_START_TIME             ERROR  STATUS
1544639623034  2018-12-12T18:33:43.034+00:00        -     SUCCESSFUL
```

In order to describe a specific backup use the following:

```
gcloud sql backups describe [BACKUP_ID] --instance [INSTANCE_NAME]
```

To delete backups, use the following steps:

1. Go to the [Cloud SQL Instances page]() in the Google Cloud Platform Console.
2. Click the instance to open its Overview page.
3. Click the Backups tab, the list of existing backups is displayed.
4. Click the More actions menu icon for the backup you want to delete.
5. Select Delete.

6. In the Delete backup window, type 'Delete' in the text box and click Delete.
   or use the following gcloud command:

```
gcloud sql backups delete [BACKUP_ID] --instance [INSTANCE_NAME]
```

**Note:** Deleting a backup might not free up as much space as the size of the backup. This is because backups are incremental, so deleting an older backup might transfer some of its contents to a more recent backup to preserve the integrity of the newer backup.

When you restore an instance from a backup, whether to the same instance or to a different instance, you should be aware of the following:

- The restore operation overwrites all data on the target instance.
- The target instance is unavailable for connections during the restore operation; existing connections are lost.
- If you are restoring to an instance with read replicas, you must delete all replicas and recreate them after the restore operation completes.

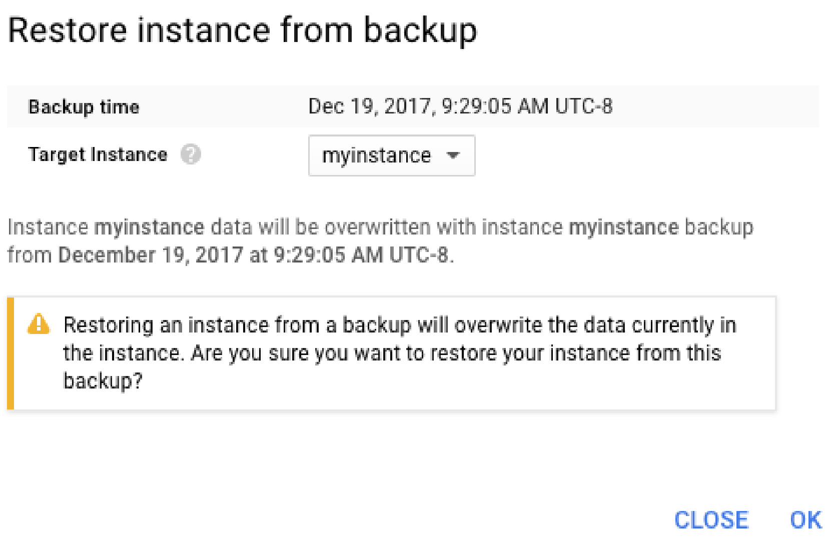
When you are restoring a backup to a different instance, you should be aware of the following restrictions and best practices:

- You cannot restore an instance using a backup taken in a different GCP project.
- The target instance should have the same database version as the instance from which the backup was taken.
- The storage capacity of the target instance must be at least as large as the capacity of the instance being backed up. The amount of storage being used does not matter.
- The target instance must be in the Runnable state.
- The target instance can have a different number of cores or amount of memory than the instance from which the backup was taken.

For more about restoring your instance, see Backup Recovery.
For scripting purposes, this topic focuses on the gcloud commands. For more about these commands, see gcloud SQL Instances Restore-Backup.
In order to restore to your current instance, follow these steps:

**Warning:** The restore process overwrites all the current data on the instance.

1. Go to the Cloud SQL Instances page in the Google Cloud Platform Console.
2. If the target instance has any read replicas, use the More actions menu icon for the instance you want to delete.
3. Click the instance you want to restore to open its Instance details page.
4. Click the Backups tab.
5. Find the backup you want to use and select Restore from its More actions menu icon.
6. In the Restore instance from backup dialog box, click OK to start the restore process.
   The default target instance is the same instance from which the backup was created.


Restore instance from backup

| | |
|---|---|
| Backup time | Dec 19, 2017, 9:29:05 AM UTC-8 |
| Target Instance | myinstance |

Instance **myinstance** data will be overwritten with instance **myinstance** backup from **December 19, 2017 at 9:29:05 AM UTC-8**.

Restoring an instance from a backup will overwrite the data currently in the instance. Are you sure you want to restore your instance from this backup?

7. You can check the status of the restore operation by going to the Operations page of the instance.
8. After the restore operation completes, recreate any replicas you deleted in the first step.

You cannot reuse instance names for up to a week after an instance is deleted.
or use the following gcloud commands:

1. Describe the instance to see whether it has any replicas:

```
gcloud sql instances describe [INSTANCE_NAME]
```

2. Delete all replicas (repeat for all replicas):

```
gcloud sql instances delete [REPLICA_NAME]
```

3. List the backups for the instance:

```
gcloud sql backups list --instance [INSTANCE_NAME]
```

4. Find the backup you want to use and record its ID value.
   Be sure to select a backup that is marked as Successful.
5. Restore the instance from the specified backup using the following command:

```
gcloud sql backups restore [BACKUP_ID] --restore-instance=[INSTANCE_NAME]
```

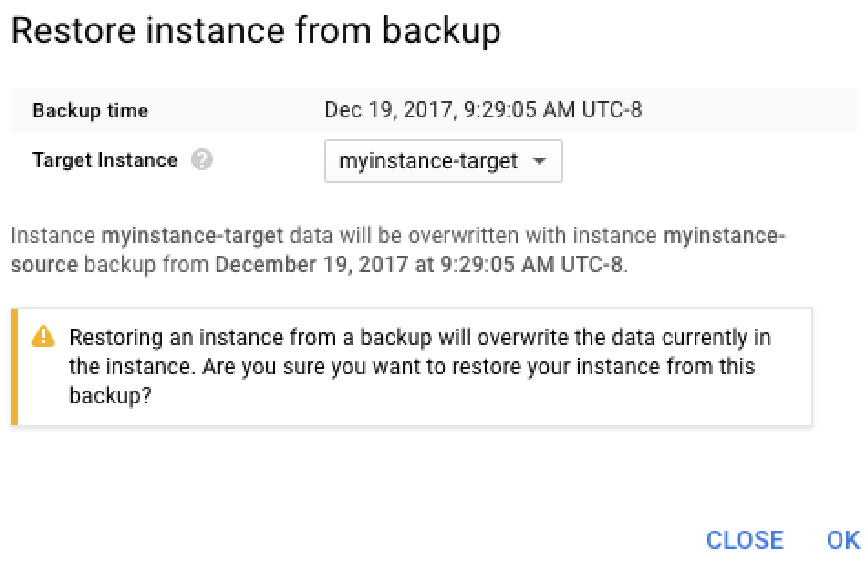
6. After the restore operation completes, recreate any replicas you deleted in the first step.
   You cannot reuse instance names for up to a week after an instance is deleted.

To restore your backup into another instance, follow the steps below:

**Warning:** The restore process overwrites all the current data on the instance.

1. Go to the Cloud SQL Instances page in the Google Cloud Platform Console.

2. If the target instance has any read replicas, use the More actions menu icon on the far-right side to delete them.
3. Click the source instance to open its Instance details page and select the Backups tab.
4. Find the backup you want to restore from and select Restore from the More actions menu icon.
5. In the Restore instance from backup dialog, select the Target instance and click OK.


Restore instance from backup

| | |
|---|---|
| Backup time | Dec 19, 2017, 9:29:05 AM UTC-8 |
| Target Instance | myinstance-target |

Instance myinstance-target data will be overwritten with instance myinstance-source backup from December 19, 2017 at 9:29:05 AM UTC-8.

Restoring an instance from a backup will overwrite the data currently in the instance. Are you sure you want to restore your instance from this backup?

6. You can check the status of the restore operation by going to the Operations tab of the target instance.
7. After the restore operation completes, recreate any replicas you deleted previously.
   You cannot reuse instance names for up to a week after an instance is deleted.

or use the following **gcloud** commands:

1. Describe the target instance to see whether it has any replicas, all listed under replicaNames:

```
gcloud sql instances describe [TARGET_INSTANCE_NAME]
```

2. Delete all replicas, repeat for all replicas:

```
gcloud sql instances delete [REPLICA_NAME]
```

3. List the backups for the source instance:

```
gcloud sql backups list --instance [SOURCE_INSTANCE_NAME]
```

4. Find the backup you want to use and record its ID value.
   Be sure to select a backup that is marked as Successful.

5. Restore from the specified backup to the target instance:

```
gcloud sql backups restore [BACKUP_ID] --restore-instance=[TARGET_INSTANCE_NAME] --backup-
        instance=[SOURCE_INSTANCE_NAME]
```

6. After the restore completes, recreate any replicas you deleted previously.
   You cannot reuse instance names for up to a week after an instance is deleted.

Currently, restore to point-in-time is not support in Cloud SQL for PostgreSQL (it is supported in Cloud SQL for MySQL)

## 8.5 RMAN operations comparison

| Description | Oracle | Cloud SQL for PostgreSQL |
|---|---|---|
| Scheduled Automatic backups | Create DBMS_SCHEDULER job that will execute your RMAN script on a scheduled basis. | gcloud sql instances patch [INSTANCE_NAME] --backup-start-time [HH:MM] |
| Manual full database backups | BACKUP DATABASE PLUS ARCHIVELOG; | gcloud sql backups create --async --instance [INSTANCE_NAME] |
| Restore database | RUN<br>{<br>SHUTDOWN IMMEDIATE;<br>STARTUP MOUNT;<br>RESTORE DATABASE;<br>RECOVER DATABASE;<br>ALTER DATABASE OPEN;<br>} | Find script 1 in the bottom of the topic |
| Incremental differential | BACKUP INCREMENTAL LEVEL 0 DATABASE;<br>BACKUP INCREMENTAL LEVEL 1 DATABASE; | All backups are incremental, no option to choose incremental type |
| Incremental cumulative | BACKUP INCREMENTAL LEVEL 0 CUMULATIVE DATABASE;<br>BACKUP INCREMENTAL LEVEL 1 CUMULATIVE DATABASE; | All backups are incremental, no option to choose incremental type |
| Restore database to a specific point-in-time | RUN { SHUTDOWN IMMEDIATE; STARTUP MOUNT; SET UNTIL TIME "TO_DATE('19-SEP-2017 23:45:00','DD-MON-YYYY HH24:MI:SS')"; RESTORE DATABASE; RECOVER DATABASE; ALTER DATABASE OPEN RESETLOGS; } | Currently, not available in Cloud SQL for PostgreSQL. |

| | | |
|---|---|---|
| Backup database Archive logs | BACKUP ARCHIVELOG ALL; | N/A |
| Delete old database Archive logs | CROSSCHECK BACKUP;<br>DELETE EXPIRED BACKUP; | Old backup records can be deleted:<br>gcloud sql backups delete [BACKUP_ID] --instance [INSTANCE_NAME] |
| Restore a single Pluggable database (12c) | RUN {<br> ALTER PLUGGABLE DATABASE pdb1, pdb2 CLOSE;<br>RESTORE PLUGGABLE DATABASE pdb1, pdb2;<br>RECOVER PLUGGABLE DATABASE pdb1, pdb2;<br>ALTER PLUGGABLE DATABASE pdb1, pdb2 OPEN;<br>} | Use script 1 to restore to another instance and then export the required database and import it to the current database, you can use the gcloud or the console to export and import:<br>gcloud sql instances export [INSTANCE_NAME] gs://bucketName/database_name.dmp --async --database=[DATABASE_NAME] |

## 8.6 Flashback database operations comparison

| Description | Oracle | Cloud SQL for PostgreSQL |
|---|---|---|
| Create a restore point | CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE; | This is equivalent for taking a backup:<br>gcloud sql backups create --async --instance [INSTANCE_NAME] |
| Configure flashback retention period | ALTER SYSTEM SET db_flashback_retention_target=2880; | Not needed in Cloud SQL |
| Flashback database to a previous restore point | shutdown immediate;<br>startup mount;<br>flashback database to restore point before_update; | Use script 1 to restore to the same instance |
| Flashback database to a previous point in time | shutdown immediate;<br>startup mount;<br>FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2017','MM/DD/YY')"; | Currently, not available in Cloud SQL for PostgreSQL. |

## 8.7 Flashback table operations comparison

| Description | Oracle | Cloud SQL for PostgreSQL |
|---|---|---|
| Create a restore point | CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE; | This is equivalent of taking a backup: gcloud sql backups create --async --instance [INSTANCE_NAME] |
| Configure flashback retention period | ALTER SYSTEM SET db_flashback_retention_target=2880; | Not needed in Cloud SQL |
| Flashback table to a previous restore point | shutdown immediate;<br>startup mount;<br>flashback database to restore point before_update; | Use script 1 to restore to another instance and then export the required table and import it to the current database, you can use the gcloud or the console to export and import:<br>gcloud sql instances export [INSTANCE_NAME] gs://bucketName/database_name.dmp --async --database=[DATABASE_NAME] --table=[TABLE_NAME]<br>gcloud sql instances import [INSTANCE_NAME] gs://bucketName/database_name.dmp --async |
| Flashback table to a previous point in time | shutdown immediate;<br>startup mount;<br>FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2017','MM/DD/YY')"; | Currently, not available in Cloud SQL for PostgreSQL. |

## 8.8 Additional scripts

Script 1:

```
echo '##################################################'
echo '####  WELCOME TO Cloud SQL RESTORE SCRIPT                 ###'
echo '##################################################'

echo 'All Cloud SQL instance (wait for instances list):'

gcloud sql instances list --format="value(name)" | sed 's/,/\n/g'

echo 'Choose the instance that own the backup: '

read selected_instance

echo 'Choose the target instance to restore to: '

read target_instance

echo 'All available backups for the selected instance  (wait for instances list):'

gcloud sql backups list --instance $selected_instance

echo 'Choose the backup ID to use for the restore: '

read backup_id

echo 'Running restore operation...'

for replica in  $(gcloud sql instances describe $target_instance --format="value(replicaNames.list())" | sed
's/,/\n/g')
do
    echo "Deleting replica:  $replica"
    for robot in $(gcloud sql instances delete $replica --quiet)
    do
        echo "    -> output $robot"
   done
done

echo 'Restoring to the target instance, it will be unavailable for a while'

gcloud sql backups restore $backup_id --restore-instance=$target_instance --backup-instance=$selected_instance
--quiet

echo 'Restore complete.'
```

## 8.9 Database Hints - Key Differences

In Oracle, a database hint lets developers and DBAs make decisions that are usually made by the optimizer. PostgreSQL does not support adding hints. You can change the optimizer/planner configuration to force the optimizer to choose a different plan.

## 8.10 Database Hints

Oracle uses hints for manual control of cost-based optimization. Oracle hints are enclosed within comments to the SQL commands Delete, Select, and Update. A hint is designated by two dashes and a plus sign and because the hint is inside a comment, it is important that the hint name is spelled correctly and that the hint is appropriate for the query. This format is identical for all three commands. A hint only affects the command block where it is located. If you must control the order of joins and index choices within a subquery, place the hint after the command keyword that starts the subquery. To affect the outer-query order of joins and index choices, place the hint immediately after the outer-query command. A large number of hints are supported by Oracle and each database hint can have zero or more arguments. There are also undocumented hints.
PostgreSQL does not implement hints. For testing or troubleshooting a query, you can change a planner configuration parameter and force the optimizer to choose a different plan.

Oracle Example: Force the Query Optimizer to use a specific index using the following command:

```
EXPLAIN PLAN FOR
SELECT /*+INDEX(LINE_NUMBERS_ACCOUNT_NUMBER_IX)*/
 INVOICE_ID, ACCOUNT_NUMBER
 FROM INVOICE_LINE_ITEMS
Execution Plan
-----------------------------------------------------------
Plan hash value: 1604387085
-----------------------------------------------------------------------------------------------
| Id  | Operation                | Name        | Cost (%CPU) |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                2   (0)|
|   1 |  VIEW| index$_join$_001      2   (0)|
|* 2 |   HASH JOIN         |       |
|   3|        INDEX FAST FULL SCAN|  LINE_ITEMS_ACCOUNT_NUMBER_IX |     1  (0)|
|   4|        INDEX FAST FULL FAST SCAN|  LINE_ITEMS_PK         |     1  (0)|
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - access(ROWID=ROWID))
```

## 8.11 PostgreSQL Planner Configuration Parameters

The planner uses configuration parameters to optimize a query. You can enable or disable certain parameters for troubleshooting.

PostgreSQL Example:

View the pg_settings parameters using the command below:

```
SELECT name FROM pg_settings WHERE name LIKE 'enable%';

Name
---------------
enable_bitmapscan
enable_hahagg
enable_hashjoin
enable_indexonlyscan
enable_material
enable_mergejoin
enable_nestloop
enable_seqscan
enable_sort
enable_tidscan
```

You could enable an index only to scan to make the optimizer use an index.

```
SET enable_indexonlyscan = true;
```

Another way to influence the optimizer/planner is to set the planner cost constants.

```
SELECT name from pg_settings WHERE name like '%cost%' and name NOT LIKE '%vacuum%';

name
--------------------
cpu_index_tuple_cost
cpu_operator_cost
cpu_tuple_cost
parallel_setup_cost
parallel_tuple_cost
random_page_cost
seq_page_cost
```

The cost variables are measured on an arbitrary scale. Only the relative values matter, so scaling them all up or down by the same factor will result in no change.

## 8.12 Database Links - Key Differences

Oracle supports connections to remote database tables and views with DML support using database links. When you create a named database link it is available until you drop the link. A remote database query looks similar to the SQL standard, except that the remote database connection is added to the table name in the FROM clause. You can also write heterogeneous queries in Oracle.

By creating a remote job or a remote procedure, you can run a DDL job on a remote procedure. PostgreSQL provides two methods to connect to remote servers, Database Link (dblink) and the Foreign Data Wrapper extension (FDW).

With DBlink, you must manually open the session. For each query, you must add parameters and data types. DBlink does not support heterogeneous queries.
FDWrapper supports creating a database link. The remote queries are similar to SQL standard. You cannot run a DDL command to a remote table.

For both PostgreSQL options, you should be aware of the following:

- Remote database username and password are stored as plain-text in the database. Using the DBlink function your code or procedures may store the password
- Any change to the user's password also requires an update in the FDW or DBlink specifications.
- Queries may fail, using the FDW extension, if the remote table column names have been modified or dropped. To resolve this issue, you must recreate the FDW tables.

## 8.13 Comparison Table

| Description | Oracle | PostgreSQL DBlink |
|---|---|---|
| Create a permanent named database link | CREATE DATABASE LINK remote_one CONNECT TO use1 IDENTIFIED BY password1 USING 'remote_one'; | Not supported.<br><br>You must manually open the connection to the remote database in your sessions or queries:<br><br>SELECT dblink_connect('thisconn', 'dbname=postgres17 port=5432 host=remote_host user=user1 password=password1'); |
| Query using a database link | SELECT * FROM invoices @remote_one; | SELECT * FROM dblink('thisconn','SELECT * FROM invoices') AS p(id int, invoice_date date); |
| DML using database link | INSERT INTO vendors@remote (id, company_name, company_email, contract_date, order_id) VALUES (44, 'Blue Plumbing', 'WJohns@test.com', SYSDATE, 430); | SELECT * FROM dblink('myconn',$$INSERT into vendors VALUES (45,'Paper Co','Calais, France')$$) AS t(id int, company_name text, company_address text); |
| Heterogeneous database link connections, such as Oracle to PostgreSQL or vice-versa | Supported. | Not supported. |
| Run DDL via a database link | Not supported directly, but, on the remote database, you can create a job or run procedures to execute the DDL commands.<br><br>dbms_job@remote.submit( l_job, 'execute immediate ''create table test_remote ( id int, message varchar2(50))''' ); commit; | SELECT * FROM dblink('myconn',$$CREATE table remote_ test (id int,notes text)$$) AS rt(a text); |
| Drop db link | drop database link thisconn; | Not supported.<br><br>Need to disconnect.<br><br>SELECT dblink_disconnect('thisconn'); |

## 8.14 Oracle Database links versus PostgreSQL DBLink and FDWrapper

You can use the Database Links, which are schema objects. Database links can interact with remote database objects. Database links are commonly used to select data from remote database tables.

**Note:** You must have Oracle Net Services installed to use database links. The services must be available on the local and the remote database servers.

In PostgreSQL, you have the following options to query data in a remote database:

- Database Link function (dblink).
- Foreign Data Wrapper extension (FDW).

The Foreign Data Wrapper (FDW) has similar functions as dblink. The advantages of FDW, the extension can improve performance and is closer to the SQL standard.
For more about the DBLink Link function, see Database Link Function.
For more about the FDW extension, see Foreign Data Wrapper.
Oracle Example:
Create a database link using the command below:

```
CREATE public DATABASE LINK my_link CONNECT TO remote_user1 IDENTIFIED BY remote_password1 USING
'homeserver:1521/mySID';

CREATE DATABASE LINK remotenoTNS CONNECT TOuser1 IDENTIFIED BY password1 USING
'(DESCRIPTION=(ADDRESS_LIST=(ADDRESS = (PROTOCOL = TCP)(HOST =192.168.10.10)(PORT =
1521)))(CONNECT_DATA =(SERVICE_NAME = orcl)))';
```

PostgreSQL Example:
Create the DBlink extension using the following query:

```
CREATE EXTENSION dblink;
```

After you have created the database link, add the database link name as a table suffix in a query.
Oracle Example:
Select to a remote table using the query below:

```
SELECT invoice_total, invoice_date FROM invoices @remote_db;
```

You can use the dblink_connect function to connect to a remote PostgreSQL database using the dblink_connect function. In the connection string, you must define the following:

- Connection name
- Database name
- Port

- User
- Password

PostgreSQL Example:
Connect to a remote database using the query below:

```
SELECT dblink_connect
('thisconn', 'dbname=postgres22 port=5432 host=myhost user=user1 password=password1');
```

You can use the connection to execute remote database queries. In the query, you must specify the connection name, the query, any parameters, and the data types for the selected columns.
PostgreSQL Example:
Select invoice_dates using the following query:

```
SELECT * from dblink
('thisconn', 'SELECT id, invoice_dates FROM INVOICES') AS i(id int,invoice_date date);
```

To preserve resources when you have completed your queries, close the connection.
PostgreSQL Example:
Disconnect from the remote server using the following query:

```
SELECT dblink_disconnect('thisconn');
```

You could also connect to the remote database and define a query in one connection string. To do this you must include the following information in the connection string:

- Database name
- Port
- Hostname
- Username
- Password
- Query, with any parameters and data types

Connecting to the remote server also executes the defined query.
PostgreSQL Example:
Connect and execute query using the query below:

```
SELECT * from dblink
('dbname=postgres22 port=5432 host=myhost user=user1e password=password1',
'SELECT id, invoice_date FROM EMPLOYEES') AS p(id int,invoice_date date);
```

In Oracle, you can also use database links to execute DML commands on remote tables or views.
Oracle Example:
Update and Delete a remote table using the following command:

```
INSERT INTO invoices @remote_db
(id, invoice_id, invoice_total, invoice_date) VALUES
(5453, 'HBM3232', 774.44, SYSDATE);
```

```
UPDATE jobs@remote_db SET invoice_total =775.33 WHEREid = 5453;

DELETE FROM invoices @remote_db WHERE id = 5453;
```

In PostgreSQL, you can perform DML commands on tables referenced through the dblink function.
PostgreSQL Example:
Insert and delete from a remote table using the query below:

```
SELECT * FROM dblink('thisconn',$$INSERT into cust_phone VALUES (34,565, '4825551212')$$) AS t(message text);

SELECT * FROM dblink('thisconn',$$DELETE FROM cust_phone WHERE id=34$$) AS t(message text);
```

You can query a remote table to create a local table.
PostgreSQL Example:
Create a local table using the following command:

```
SELECT cust_info.* INTO local_cust_info FROM dblink('thisconn','SELECT * FROM cust_info') AS emps(id int,
firstname varchar, lastname varchar);
```

You can create a join with a local table and remote table.
PostgreSQL Example:
Create a join with a remote table with command below:

```
SELECT local_cust_info.id , local_cust_info.name, o.order_date, o.total_items
FROM local_cust_info INNER JOIN
dblink('myconn','SELECT * FROM orders') AS o(id int, order_date date, total_items int)
ON local_cust_info.id = o.id;
```

You can run DDL statements in the remote database.
PostgreSQL Example:
Run a create table for a remote table with the following command:

```
SELECT * FROM dblink('thisconn',$$CREATE table cust_complaints (id int, complaint text)$$) AS cc(a text);
```

## 8.15 Working with the PostgreSQL Foreign Data Wrapper

To use the Foreign Data Wrapper, you must load the extension.
PostgreSQL Example:
Load an extension with the command below:

```
CREATE EXTENSION postgres_fdw;
```

You must create a connection to the remote database. To do this, you need to specify the following:

- Remote server (hostname)
- Database name
- Port

PostgreSQL Example:
Create a connection to a remote server using the following command:

```
CREATE SERVER remote_db17
 FOREIGN DATA WRAPPER postgres_fdw
 OPTIONS (host 'remote_server', dbname 'postgresql17', port '5432');
```

You create a user mapping with the following requirements:

- The local_user has user permissions on the current database
- Use the established connection
- The user name and password must have permissions in the remote database

PostgreSQL Example:
Create a user mapping using the following command:

```
CREATE USER MAPPING FOR user1
 SERVER remote_db17
 OPTIONS (user 'remote_user17', password 'password17');
```

The user mapping allows you to import some or all of the tables and views in the schema. You can also create a foreign table from the remote database that includes the schema name and table name properties.

PostgreSQL Example:
Create a remote table using the following command:

```
CREATE FOREIGN TABLE rem_cust_info (
        id int, firstname text, lastname text)
        SERVER remote_db17
        OPTIONS (schema_name 'customers', table_name 'cust_names');
```

A query on the local rem_cust_info table is selecting data from the remote customers.cust_names table.
PostgreSQL Example:
Execute a query on the table using the command below:

```
SELECT * FROM rem_cust_info;
```

If needed, you can import specific tables or a schema.
PostgreSQL Example:

Import table using the following command:

```
IMPORT FOREIGN SCHEMA customers LIMIT TO (cust_address)
   FROM SERVER remote_db17 INTO local_customers;
```

- The remote database username and password are stored as plain-text in several locations:
  - The pg_user_mapping view a view available to super users.
  - Your code or procedures may store the password in the database with the dblink function.
- User password changes must also be changed in the FDW/dblink specifications.
- When using FDW, queries will fail if remote table columns have been dropped or renamed. To resolve this issue, you must recreate the FDW tables.

## 8.16 PostgreSQL DBLink vs. FDW

| Description | PostgreSQL DBlink | PostgreSQL FDW |
|---|---|---|
| Use a database link object in the database | Not supported. | CREATE FOREIGN TABLE fdw_cust (id int, firstname text, lastname text, address text ) SERVER foreign_server OPTIONS (schema_name 'customer', table_name 'cust_info'); |
| Query remote data | SELECT * FROM dblink('thisconn','SELECT * FROM vendors') AS p(id int,company_name text); | SELECT * FROM fdw_cust; |
| DML on remote data | SELECT * FROM dblink('thisconn',$$INSERT into vendors VALUES (56,'Blue Moon Roofing')$$) AS v(id int, compnay_name text); | INSERT into fdw_cust VALUES (34,'Maria', Sotoyo,'Madrid, Spain'); |
| Run DDL on remote objects | SELECT * FROM dblink('myconn',$$CREATE table remote count_tbl (id int,cnt integer)$$) AS c(a text); | Not supported. |

## 8.17 Export / Import - Key Differences

There are not many equivalent PostgreSQL options to the Oracle Data Pump. Two different tools can achieve most of the major functionality for exporting and importing data to and from the PostgreSQL instance.

One of the main Cloud SQL advantages is that you can use the console to export and import. In addition, Cloud SQL will also use the Cloud Storage so there is no need to keep the files on another resource or transfer the files over the network

Oracle Example:
The following is an Oracle export example:

```
expdp system schemas=emp dumpfile=emp.dmp logfile=emp.log
```

PostgreSQL Example:
The following is an PostgreSQL export example:

```
pg_dump -h gcp-playbooks:us-central1:playbook -U emp_db_admin -d emp_db  -f dump_file.sql
```

Oracle Example:
The following is an Oracle import example:

```
impdp system schemas=emp dumpfile=emp.dmp logfile=emp.log REMAP_SCHEMA=emp:emp_copy
TRANSFORM=OID:N
```

PostgreSQL Example:
The following is a PostgreSQL import example:

```
pg_restore -h gcp-playbooks:us-central1:playbook -U hr -d hr_restore -p 5432 c:\Expor\hr.dmp
```

## 8.18 Oracle Data Pump and PostgreSQL utilities

Oracle Data Pump utility can be used for importing data into an Oracle database or exporting data from an Oracle database. The utility can copy a full database, complete schemas, or only selected schema objects. You could develop a backup strategy to use the Oracle Data Pump utility to restore selected database objects (such as tables, stored procedures, specific records, views, and so on). You could use this strategy instead of using snapshots or Oracle RMAN. For more about this utility, see Oracle Data Pump.

If you do not use the splfile parameter during export, the generated dumpfile is binary. You cannot open a binary file with a text editor, if needed.

The Oracle Data Pump supports the following functions:

- Importing data to an Oracle database with the IMPDP command.
- Exporting data from an Oracle database with the EXPDP command.

The Data Pump IMPDP command imports data and objects from a selected dump file created by Data Pump EXPDP command. You can filter the import from the IMPDP command and remap schema names and objects while importing.

The Data Pump EXPDP command creates a binary dump file of objects from the database by default. The objects can be exported with metadata only or data. You can perform an export for a selected SCN or timestamp to ensure consistency.

## 8.19. Notes:

- If you are using the Data Pump to import data into an existing table, the job is terminated if any row violates a constraint.
- If the main table is compressed, the Data Pump import job automatically compresses the data.
- Data Pump utility uses parallel execution.
- In interactive mode you can stop, start, change parameters, or kill a job while the job is running. If you stop the job, the master table is retained and you can restart the job. If you kill the job, the master table is dropped and the job cannot be restarted.
- You can estimate job time with the utility.
- A Data Pump dump file can be referred to as a logical backup.
- A default directory object, DATA_PUMP_DIR is created when the database is created or if the database dictionary is upgraded. This directory is available to only a few users. If you are not one of those users, a DBA or a user with Create Any Directory privilege must create a directory object for you. The user creating the directory must grant Read and Write permissions on the directory to other users. These permissions only mean that Oracle will read and write that file for you. You must have the appropriate operating system privileges to have direct access to the files outside of the Oracle database.

The EXPDP command contains the following parameters:

- Username and password to run the utility
- logical directory name
- Schema name to export
- Dump file name
- Log file name

Oracle Example:

Export the HR schema using the EXPDP command:

```
expdp system/**** directory=dpump_Finance schemas=Finance dumpfile=Finance.dmp logfile=Finance.log
```

The IMPDP command contains the following parameters:

- Username and password to run the utility
- logical directory name

- Schema name to import
- Dump file name
- Log file name
- REMAP_SCHEMA parameter

Oracle Example:
Import and rename a schema using the IMPDP command:

```
impdp system/**** directory=dpump_Finance schemas=Finance dumpfile=Finance.dmp logfile=Finance.log
REMAP_SCHEMA=Finance:Finance_copy
```

PostgreSQL has its own utilities for creating dump files (exporting data from the database). Some of the functionality is comparable to the Oracle EXPDP/IMPDP utilities. The pg_dump utility exports data from the database like EXPDPand pg_restore will import the data like IMPDP. These PostgreSQL tools can be used for backups or moving data, similar to the utilities in Oracle. The main difference is that the PostgreSQL export tool creates a plain-text file while EXPDP creates a binary file. Both PostgreSQL tools can be used against the Cloud SQL for PostgreSQL instance, but the tools must be run from another resource, such as a laptop, or in Cloud Compute with the PostgreSQL client installation.
If exporting more than one database, the pg_dumpall tool should be used.
The dump files created by pg_dump can be moved and kept in Cloud Storage in your GCP environment. These dump files can be used with restore operations or to copy data to a new instance.

**Note:**
The pg_dump tool creates consistent backups while application are using the database and it does not block readers or writers operations.

1. To create a data dump file with pg_dump, run the command from a laptop or Cloud Compute.

```
pg_dump -h gcp-playbooks:us-central1:playbook -U emp_db_admin -d emp_db  -f dump_file.sql
```

2. Create the dump file and upload it to GCP Storage using the gsutil utility:

```
pg_dump -h gcp-playbooks:us-central1:playbook -U emp_db_admin -d emp_db -f dump_file.sql | gsutil cp
gs://[DESTINATION_BUCKET_NAME]/pg_bck-$(date"+%Y-%m-%d-%H-%M-%S")
```

3. To restore a dump file with pg_restore, use the following command:

```
pg_restore -h gcp-playbooks:us-central1:playbook -U emp_db_admin -d emp_db_restored dump_file.sql
```

**Note:** The {-$(date "+%Y-%m-%d-%H-%M-%S")} format is valid on Linux operating system.

In order to download the dump file, use the command below:

```
gsutil cp gs://[DESTINATION_BUCKET_NAME]/pg_bck-$(date"+%Y-%m-%d-%H-%M-%S") pg_dump_to_restore.sql
```

In order to copy a database within the same instance use the following command:

```
CREATE DATABASE emp_copy TEMPLATE emp_db;
```

There is another way to use export and import by using the GCP console or gcloud command.
To export directly to GCP Storage, use the following example:

```
gcloud sql instances export [INSTANCE_NAME] gs://bucketName/database_name.dmp --async --
database=[DATABASE_NAME] --table=[TABLE_NAME]
```

For more about this command, see the Export Guide.

To import directly to GCP Storage, use the example below:

```
gcloud sql instances import  [INSTANCE_NAME] gs://bucketName/database_name.dmp --async
```

For more about this command, see the Import Guide.

## 8.20 Built-in Packages - Key Differences

The DBMS_OUTPUT package allows you to send messages from PL/SQL subprograms. The PostgreSQL RAISE function has similar functionality.

### Comparison Tables

| Feature | Oracle | PostgreSQL |
|---|---|---|
| Disables output of the message | DISABLE | Set the “client_min_message” or “log_min_message” configuration parameters |
| Enables output of the message | ENABLE | |
| From the buffer, returns one line | GET_LINE | Store a string in a temporary table or an array. Use the stored message in another function. |
| From the buffer, returns an array of lines | GET_LINES | |
| To the buffer, adds a partial line | PUT + NEW_LINE<br>BEGIN<br>DBMS_OUTPUT.PUT ('hello');<br>DBMS_OUTPUT.PUT('world,');<br>DBMS_OUTPUT.NEW_LINE();<br>END;<br>/ | Before RAISE, concatenate a string and assign the string to a variable.<br><br>do $$<br>DECLARE<br> str_msg text :='';<br>begin<br>  str_msg:= str_msg\|\| 'hello';<br>  str_msg:= str_msg\|\| 'world';<br>  RAISE NOTICE '%',<br>str_msg;<br>END$$; |
| To the buffer, adds one line | PUT_LINE<br><br>DBMS_OUTPUT.PUT_LINE ('The invoice | RAISE<br><br>RAISE NOTICE 'The invoice data is: % |

| | | |
|---|---|---|
| | data is:' \|\| d_invoice_date \|\| ' and the vendor is:' \|\| vendor_name); | and the vendor name is: %', d_inv_data, str_vendor_name |
| Retrieves the most recent exception number code | SQLCODE + SQLERRM | SQLSTATE + SQLERRM |
| Retrieves the error code and the error message | DECLARE<br>Name vendors.company_name%TYPE;<br>BEGIN<br> SELECT company_name INTO co_name FROM vendors<br>WHERE vendor_id = -1;<br>EXCEPTION<br> WHEN OTHERS then<br> DBMS_OUTPUT.PUT_LINE('Error code ' \|\| SQLCODE \|\| ': ' \|\| sqlerrm);<br>END;<br>/ | do $$<br>declare<br> Name vendors%ROWTYPE;<br>BEGIN<br> SELECT company_name INTO co_name FROM vendors WHERE vendor_id = -1;<br>EXCEPTION<br> WHEN OTHERS then<br> RAISE NOTICE 'Error code %: %', sqlstate, sqlerrm;<br>end$$; |

The Oracle DBMS_RANDOM package lets you generate random numbers and strings. The PostgreSQL RANDOM() function has some of the same abilities. You should not use either the DBMS_RANDOM package or the RANDOM() function for cryptography as those functions are not considered cryptographically safe.

The PostgreSQL RANDOM() function does not allow you to set the length of the string. You can write a user-defined function to match that capability in Oracle.

| Description | Oracle | PostgreSQL |
|---|---|---|
| Produce a random number | select dbms_random.value() from dual; | select random(); |
| Produce a random number from a range of numbers | select dbms_random.value(1,100) from dual; | select random() * 100; |
| Produce a string of random characters | select dbms_random.string('a',21) from dual; | select md5(random()::text); |
| Generate a lower case random string | select dbms_random.string('L',22) from dual; | select lower(md5(random()::text)); |

The DBMS_SQL package provides an interface to use dynamic SQL to parse any DML or DDL statement using PL/SQL. For example, you could create a procedure that uses dynamic SQL to allow users who may not have the correct permissions the ability to change their password.

PostgreSQL does not have all of the features of the DBMS_SQL package. The database does allow dynamic SQL and dynamic cursors.

The DBMS_SCHEDULER package is a more complex job scheduling engine. The job scheduling is the core of DBMS_SCHEDULER but you also have the following benefits:

- Logging of jobs
- Simple scheduling syntax
- Ability to run jobs outside of the database on the operating system
- Manage resources between different job classes
- Ability to use arguments
- Privilege-base security model
- Ability to name the jobs and add comments
- Ability to store and reuse schedules

For more about this package, see DBMS_SCHEDULER package.

## 8.21 DBMS_OUTPUT

The Oracle DBMS_OUTPUT package lets you send messages from stored procedures, triggers, and packages. The package is very useful when displaying PL/SQL debugging information. For more about this package, see Oracle DBMS_OUTPUT.
Typing SET SERVEROUTPUT ON has the effect of setting the DBMS_OUTPUT.ENABLE (buffer_size => NULL) parameter;
Oracle Example:
This is an example using SET SERVEROUTPUT ON:

```
SET SERVEROUTPUT ON
BEGIN
    dbms.output_put_line (user || ' has the following tables in the database: ');
    FOR tbl IN (SELECT table_name FROM user_tables)
    LOOP
          dbms_output.put_line(tbl.table_name);
    END LOOP;
END;
/
```

You can add a partial line or line to the buffer or retrieve a line or an array of lines from the buffer. The PUT procedure adds a partial line. The PUT_LINE procedure adds a line. Another PL/SQL procedure or package can read this buffer using the GET_LINE procedure, which reads one line and GET_LINES procedure, which retrieves an array of lines.

An option to the DBMS_OUTPUT is the PostgreSQL RAISE statement. You can report messages and raise errors. For more about this statement, see RAISE statement. With RAISE statement, you can add the level option. The level option sets the error severity. The following levels of severity are available in PostgreSQL:

- DEBUG
- LOG
- NOTICE
- INFO
- WARNING
- EXCEPTION

The EXCEPTION level is the default.
PostgreSQL Example:
This is an example using the RAISE statement:

```
DO $$
BEGIN
        RAISE DEBUG 'This is a debug message %', now();
        RAISE LOG ' This is a log message %', now();
        RAISE NOTICE ' This is a notice message %', now();
        RAISE INFO 'This is an information message %', now();
        RAISE  WARNING 'This is a warning message %', now();

END $$;

NOTICE: This is a notice message: 2018-12-26 15:56:30.50661+00
INFO: This is an information message 2018-12-26 15:56:32.50661+00
WARNING: This is a warning message  2018-12-26 15:56:34.50661+00
```

Observe that not every message level is returned. The message availability is controlled by the client_min_messages and the log_min_messages configuration parameters. The default are INFO, NOTICE, and WARNING messages.

## 8.22 DBMS_RANDOM

Oracle's DBMS_RANDOM package provides a built-in random number generator. Do not use DBMS_RANDOM for cryptography. The package is initialized with the current user name, current time, and current session. For more about this package, see DBMS_RANDOM package.
The following parameters produce these results:

DBMS_RANDOM.RANDOM generates integers in the [-2^^31, 2^^31] range.
DBMS_RANDOM.VALUE generates numbers in the [0,1] range with 38 digits of precision.
You can initialize DBMS_RANDOM explicitly. The package does not require initialization before using the random number generator. The package automatically initializes with date, userid, and process if not explicitly initialized.

If the package is seeded twice with the same seed, and then accessed using the same method, the package produces the same results.

The DBMS_RANDOM Package subprograms include the following options:

- SEED Regenerates the seed to a different value.
- NORMAL Generates random number in a normal distribution.

- VALUE Generates a number with a precision of 38. The number generated is greater than or equal to 0 and less than 1. The precision specifies the maximum number of decimal digits. To return integers in a given range, add the low_value and high_value arguments and truncate the decimals from the result. Note, the high value is not a possible result.
- STRING Generates random text strings. Add a code which indicates the type of string and the length.

Oracle Example:
This is an example using DBMS_RANDOM.VALUE:

```
select TRUNC(dbms_random.value(0, 50) from dual;

DBMS_RANDOM.VALUE(0, 50)
-------------------
42

 select TRUNC(dbms_random.value(0, 50) from dual;

DBMS_RANDOM.VALUE(0, 50)
-------------------
22
```

The function string generates a random string with two options. The first option defines the string type. The second option determines the length of the generated string.
The following options are possible:

- U or u: Uppercase alpha characters
- L or l: Lower case alpha characters
- A or a: Upper- and lower-case alpha characters
- X or x: Upper alpha and numeric characters
- P or p: Any printable character

Oracle Example:
This is an example using DBMS_RANDOM.STRING:

```
select dbms_random.string('U',25) from dual;
DBMS_RANDOM.STRING('U',25)
-----------------------------------------------------
NAXGOVAGRDMJNVBKXDKKXCJBI
```

```
select dbms_random.string('U',25) from dual;
DBMS_RANDOM.STRING('U',25)
------------------------------------------------------
DZIEGFHFJLHBQTXAAKYBKURRH
```

PostgreSQL does not offer a direct equivalent to the DBMS_RANDOM package.

PostgreSQL provides functions that may be used as a workaround in specific situations.
PostgreSQL provides the random() function to generate a random number between 0 and 1. The characteristics of the random() function depends on the system implementation. This function is not suitable for cryptographic applications. For more about this function, see Random() Function.

To generate a random string use the random() function with the MD5 () function. You can also create user-defined functions to generate strings of a specific length or range. For more about this function, see MD5 () function.

## 8.23 Examples

PostgreSQL Example:
This is an example using the random() function:

```
select random();
    random
-------------------
 0.59156496338546


--generate a random number between 1 and 15
select random() * 15 + 1 as RAND_15 ;
     RAND_15
-------------------
11.799183815252
```

PostgreSQL Example:
This is an example using the MD5() function:

```
select md5(random()::text);
            md5
----------------------------------
"bfd33f43e9318dade9ba6ec94349e63c"

 select md5(random()::text);
            md5
----------------------------------
"42fcfed745bb03e136555c21ea2e890e"
```

## 8.24 DBMS_SQL Package (11g & 12c)

Oracle allows you to write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are stored in character strings that are input to and built by the application at runtime. Dynamic SQL allows you to create statements that are more general-purpose.
DMBS_SQL provides an interface to use dynamic SQL to parse any DML or DDL statement from within PL/SQL.
The DBMS_SQL package is owned by the SYS schema. Any DBMS_SQL subprogram called from an anonymous PL/SQL block runs with the current user privileges.
Oracle Example:
This is an example using DMBS_SQL.OPEN:

```
DECLARE
cur1 integer := dbms_sql.open_cursor;
fdbk integer;
stmt varchar2(2000);
BEGIN
        FOR i in (SELECT null FROM user_objects WHERE object_name = 'test_seq') loop
         stmt := 'DROP SEQUENCE test_seq';
fdbk := dbms_sql.execute(cur1);
DBMS_SQL.CLOSE_CURSOR (cur1);
END loop;
END;
/
```

The declaration block defines the variables. The execution block assigns a valid DDL statement, binds the cursor to the statement, and runs the statement.
The execution flow is as follows:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE or BIND_ARRAY
- DEFINE_COLUMN, DEFINE_COLUMN_LONG, DEFINE_ARRAY
- EXECUTE
- FETCH_ROWS or EXECUTE_FETCH
- VARIABLE_VALUE, COLUMN_VALUE, or COLUMN_VALUE_LONG
- CLOSE_CURSOR

The following procedures are also available in the DBMS_SQL package:

- RETURN_RESULT
  (New in Oracle 12c) Rather than defining explicit ref cursor out parameters, the RETURN_RESULT procedure allows you to pass those parameters implicitly.

- TO_REFCURSOR
  This function converts a SQL cursor to a weakly-typed variable of PL/SQL data type REFCURSOR.

You can use this cursor in dynamic SQL statements. You must OPEN, PARSE, and EXECUTE before passing a SQL cursor number to DBMS_SQL.TO_REFCURSOR. After the SQL Cursor is converted to a REFCURSOR variable, DBMS_SQL operations can only access the REFCURSOR variable.

- TO_CURSOR_NUMBER
  This function converts a REFCURSOR variable to a SQL cursor number, which you can then pass to DBMS_SQL programs. The REFCURSOR variable can be either strongly typed or weakly typed. For more about this function, see TO_CURSOR_NUMBER.

PostgreSQL does not support all the features of DBMS_SQL; however, PostgreSQL does support dynamic cursors and dynamic SQL. For more about dynamic SQL, see dynamic SQL.
PostgreSQL Example:
This is an example using the REFCURSOR procedure:

```
CREATE FUNCTION reffunc(_ref)
RETURNS refcursor AS $$

BEGIN
        OPEN _ref FOR execute 'SELECT * FROM pg_tables';
        RETURN _ref;
END;
$$ LANGUAGE plpgsql;
```

## 8.25 DBMS_SCHEDULER

The DBMS_SCHEDULER package offers a collection of scheduling procedures and functions. These functions and procedures can be called from any PL/SQL program. The package ignores privileges granted on scheduler objects through roles. Object privileges must be granted directly to the user. For more about the DBMS_Scheduler package, see DBMS_SCHEDULER package.
There is no equivalent option in PostgreSQL. To get similar functionality, you can use a scheduled Cloud Function. To connect and execute the Cloud SQL for PostgreSQL instance from the Cloud Functions, see the example in the UTL_Packages topic.
To create a scheduled Cloud Function, see Using Pub/Sub to Trigger a Cloud Function.

## 8.26 EXECUTE IMMEDIATE - Key Differences

Oracle Example:
This is an example using the EXECUTE IMMEDIATE procedure:

```
DECLARE
    l_count varchar2(50);
BEGIN
EXECUTE IMMEDIATE 'SELECT count(1) from customers'
         into l_count;
 dbms_output.put_line(l_count);
END;
```

PostgreSQL Example:

This is an example using the EXECUTE procedure:

```
CREATE OR REPLACE FUNCTION Ad_count
 (a_date timestamptz) RETURNS BIGINT
AS $$
DECLARE
count BIGINT;
BEGIN
EXECUTE 'SELECT count(*) from Ad_Submissions WHERE
 ad_date >= $1' USING a_date INTO count;
 RETURN count;
 END;
$$ LANGUAGE plpgsql;
```

Execute INSERT with bind variables.

Oracle Example:

This is an example using the BEGIN EXECUTE IMMEDIATE procedure:

```
DECLARE
    l_deptname varchar2(50) := 'Quality Analysis';
    l_location varchar2(50) := 'Pune';
BEGIN EXECUTE IMMEDIATE 'insert into org.departments values (:1:2:3)'
         using 150, l_deptname, l_location;
    COMMIT;
END;
```

PostgreSQL Example:

This is an example using EXECUTE procedure with INSERT:

```
CREATE OR REPLACE FUNCTION insert_States (state_name text , short_name text)
RETURNS VOID AS
$$
BEGIN
EXECUTE format('INSERT INTO STATE(state_name, short_name)
      VALUES($1, $2;') using 'Washington', 'WA';
END
$$ LANGUAGE plpgsql;
```

Execute a DDL statement.

Oracle Example:

This is an example using the EXECUTE IMMEDIATE procedure to execute a DDL statement:

```
EXECUTE IMMEDIATE 'set role all';
```

PostgreSQL Example:

This is an example using the EXECUTE IMMEDIATE procedure to execute a DDL statement:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test_user(id SERIAL PRIMARY KEY, name TEXT);";
EXEC SQL END DECLARE SECTION

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

Execute an anonymous block.
Oracle Example:
Execute an anonymous block using the following command:

```
EXECUTE IMMEDIATE ‘BEGIN DBMS_OUTPUT.PUT_LINE(‘’ block’’); END;’;
```

PostgreSQL Example:
Use the following command to execute an anonymous block:

```
 DO $$DECLARE tn table_name
BEGIN
      For tn IN SELECT table_schema, table_name FROM information_schema.tables
       WHERE table_schema='public' AND table_type = 'BASE TABLE'
 Loop
EXECUTE 'GRANT ALL ON ' || quote_ident(tn.table_schema) || '.'||quote_ident(tn.table_name ||' to tempDBA';
END LOOP;
END$$;
```

## 8.27 EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE command dynamically prepares and executes a statement or runs a dynamic DDL statement.
The Oracle and PostgreSQL command can be used with bind variables.
The purpose of the EXECUTE IMMEDIATE command in Oracle is to parse and execute a dynamic SQL or an anonymous PL/SQL block. It also supports bind variables.
Oracle Example:
Run a dynamic SQL statement to update a table using the following command:

```
DECLARE
inv_id_var NUMBER;
terms_var NUMBER;
sql_stmt varchar2(400);

begin
   inv_id_var := &invoice_id;
   terms_var := &terms_id;
    sql_stmt := 'UPDATE INVOICES ' ||
          'SET terms_id = ' || terms_var || '' ||
        'WHERE invoice_id = ' || inv_id_var;

EXECUTE IMMEDIATE sql_stmt;
```

Oracle Example:  Run a DDL operation using the command below:

```
EXECUTE IMMEDIATE 'ALTER TABLE  CUST_ACCT ADD last_transaction_date DATE';
```

Oracle Example:
 Run an anonymous block with bind variables using the following command:

```
EXECUTE IMMEDIATE 'BEGIN raise_sal (:col_val, :col_name, :amount); END;' USING 134, 'Manager_ID', 10;
```

PostgreSQL Example:

Using the following example to execute a SQL SELECT query using bind variables:

```
DO $$
DECLARE
curr_tbl    varchar(35) := 'libraries_tbl';
id_to_find          integer := 76;
id_count            integer;
BEGIN
EXECUTE format('SELECT count(*) FROM %I WHERE branch_id = $1', curr_tbl)
INTO id_count USING id_to_find;
RAISE NOTICE 'Count is % int table %', id_count, curr_tbl;
END$$;
```

PostgreSQL Example:
 Execute a command with variables and without variables using the command below:

```
DO $$
 DECLARE
BEGIN
EXECUTE 'INSERT INTO number_counter VALUES (15')';
EXECUTE format('INSERT INTO number_counter VALUES (%s)', 16);
END$$;
;
```

**Note:**
Format specifiers, which start with the % character, indicate the location and method to translate the data to characters. The %s specifier formats the argument value null terminated string.

PostgreSQL Example:
Use the following example to execute a CREATE TABLE command:

```
DO $$
DECLARE
BEGIN
EXECUTE 'CREATE TABLE number_counter (counter integer)';
END
$$;
```

A statement created with the PREPARE keyword is a server-side object and can be used to optimize performance. When the PREPARE statement is executed, the defined statement is parsed, analyzed, and rewritten. The EXECUTE command plans and executes the statement. This division allows the execution plan to optimize on the parameter values supplied. Prepared statements can take parameters, these values are substituted when the statement is executed. Prepared statements only last for the duration of the current database session. A single prepared statement cannot be used by multiple database clients. Each client can create their own statement. You can clean up statements with the DEALLOCATE command. The best performance advantage of a prepared statement is when a single session is used to execute a large number of similar complex statements.

PostgreSQL Example:
The following example uses a PREPARE statement and EXECUTE command:

```
PREPARE emp (int, text) AS
INSERT INTO emp_info VALUES($1, $2);
EXECUTE emp(200, 'Anita Harris');
execute emp(301,'Maria Sotoyo');
execute emp(302,'Albert Jones');
execute emp(304,'Robert Travis');
```

## 8.28 Execution Plans - Key Differences

- Oracle and PostgreSQL both use cost-based optimizers.
- Oracle uses the EXPLAIN PLAN clause and saves the plan to a plan table.
- PostgreSQL uses EXPLAIN keyword and provides the execution plan in the result.

## 8.29 Execution Plans

In most database types, before the database executes a query, the optimizer creates execution plans. The query optimizer creates execution plans for all DMLs and queries in the database. The query optimizer performs the following steps:

- Generates a set of potential plans based on the access paths and hints.
- Estimates the cost of each plan based on statistics and storage characteristics.
- Compares the cost and chooses the lowest cost.

The query optimizer determines a set of instructions. The database system then executes the step-by-step plan. Users review execution plans to understand how the query is executed to tune the query performance.

Both Oracle and PostgreSQL use a cost-based optimizer (CBO) for queries. Cost-based optimization involves generating multiple execution plans and selecting the lowest cost execution plan to execute the query. For example, a review of the plan can determine if new indexes are needed. Execution plans

might have changed because of differences with one of these: data volumes, CPU, IO, memory, data statistics, and global or session parameters.
Oracle generates and displays execution plans by placing the plans into a table, which is normally called PLAN_TABLE. You can view the data with a one-line query.

While the display is commonly displayed in a tabular format, the plan is tree-shaped. The tabular representation is top-down, left-to-right traversal of the execution tree. To read the plan, you start from the bottom left and work across and then up.

The execution plan alone cannot differentiate between well-tuned queries and queries that, in reality, perform poorly. The query optimizer relies on statistics. If the table statistics have not been updated and the table is heavily used, the indexes are out-of-date. Sometimes the selected index is inefficient. In an execution plan, you should examine the following:

- Columns of the index being used.
- Filter Selectivity: The rows excluded.

In Oracle, the EXPLAIN PLAN statement displays execution plans chosen by the Oracle optimizer. The EXPLAIN PLAN results let you determine whether the optimizer selected a particular plan and understand the optimizer's decision. The EXPLAIN PLAN statement does not run the query.

PostgreSQL uses the EXPLAIN keyword. The EXPLAIN keyword displays the execution plan for a query. The PostgreSQL optimizer generates the estimated execution plan for SELECT, INSERT, UPDATE, and DELETE operations. The optimizer builds a B-Tree that represents the different actions. The EXPLAIN statement can also provide statistical information.
Running a query with the EXPLAIN command will not execute the query itself but will only create a cost estimation.

Oracle Example:
The following example uses EXPLAIN PLAN to generate an execution plan and query the PLAN_TABLE for the results:

```
EXPLAIN PLAN FOR
SELECT i.invoice_id, i.invoice_total, i.invoice_date, ili.line_item_amt FROM invoices i
INNER JOIN invoice_line_items ili
ON i.invoice_id = ili.invoice_id'
WHERE invoice_total > 100.00;

select plan_table_output from table(dbms_xplan.display('plan_table',null,'basic +cost'));

PLAN_TABLE_OUTPUT
----------------------------------------------------------
Plan hash value: 347543421
------------------------------------------------------------------------------------------
| Id  | Operation                  | Name       |  Cost (%CPU)|
-----------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT          |            |        6 (17)|
| 1 |       MERGE JOIN |            |  6   (17)|
| 2 |         TABLE ACCESS BY ROWID    | INVOICES |   |     2   (0)|
| 3|           INDEX FULL SCAN    |  INVOICE_PK   |   1 (0)|
| 4|            SORT JOIN       |        |    4 (25)|
| 5|              TABLE ACCESS FULL    |    INVOICE_LINE_ITEMS  |   3 (0)|
-----------------------------------------------------------------------------------------
```

The Invoice and Invoice_Line_Items tables contain indexes for INVOICE_ID columns.
Oracle Example:
The following is an example of a full table scan:

```
EXPLAIN PLAN FOR
SELECT invoice_id, invoice_total FROM invoices
WHERE invoice_total > 100.00;
Execution Plan
---------------------------------------------------------
Plan hash value: 2547190830
------------------------------------------------------------------------------
| Id  | Operation        | Name      |   Cost (%CPU) |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     3   (0)|
|  1 |       TABLE ACCESS FULL| INVOICES |        3   (0)|
--------------------------------------------------------------
```

PostgreSQL can perform different types of scans. If the table is almost empty, the optimizer performs a sequential scan. If the query is for only a small set of rows, the optimizer may use the index only scan because all columns are already in the index. The optimizer uses a bitmap scan when the query reads too many rows for an index scan and too few for a sequential scan, or the query is using multiple indexes to scan a single table.
PostgreSQL Example:

Displaying the execution plan of a SQL statement using the EXPLAIN command:

```
 EXPLAIN
                              SELECT id, FIRST_NAME, LAST_NAME, REGISTRATION FROM test_users
                              WHERE id IN (SELECT id FROM test_users WHERE id > 75 and id < 90);

                              ---------------------------------------------------------------------------------------
                              Hash Semi Join (cost=4.64..11.59 rows=15 width=27)
                              Hash Cond: (test_users.id = test_users_1.id)
                              ->  Seq Scan on test_users (cost=0.00..6.00 rows=300 width=27)
                              ->  Hash (cost=4.45..4.45 rows=15 width=4)
                            ->  Index Only Scan using test_users_pkey on test_users test_user_1 (cost=0.15..4.45
rows=15 width=4)
                                     Index Cond: ((id > 75) AND (id < 90))
```

The addition of the ANALYZE keyword executes the statement and provides additional details.
PostgreSQL Example:

Run the same statement after adding the ANALYZE keyword using the command below:

```
 EXPLAIN ANALYZE
                                          SELECT ID, FIRST_NAME, LAST_NAME, REGISTRATION FROM
test_users
                                          WHERE id IN (SELECT id FROM test_users WHERE id > 75 and id < 90);

                                          ----------------------------------------------------------------------------------------
                                          Hash Semi Join (cost=4.64..11.59 rows=15 width=27)
                                          Hash Cond: (test_users.id = test_users_1.id)
                                           ->  Seq Scan on test_users (cost=0.00..6.00 rows=300 width=27)
                                           ->  Hash (cost=4.45..4.45 rows=15 width=4)
                                                 Buckets: 1024 Batches: 1 Memory Usage: 9kB
                                           -> Index Only Scan using test_users_pkey on test_users test_users_1
(cost=0.15..4.45 rows=15 width=4) (actual time=0.004..0.005 rows=14 loops=1)
                                                                  Index Cond: ((id > 75) and (id < 90))
                                                                  Heap Fetches: 0
                                          Planning time: 0.339 ms
                                          Execution time: 0.081 ms
```

PostgreSQL Example:

Use the following example to view a PostgreSQL execution plan that shows a sequential scan:

```
EXPLAIN ANALYZE
    SELECT ID, FIRST_NAME, LAST_NAME, REGISTRATION FROM test_users
    WHERE ID > 100;

 ----------------------------------------------------------------------------------------
  Seq Scan on test_users  (cost=0.00..6.75 rows=200 width=27) (actual time=0.012..0.034 rows=200 loops=1)
   Filter: (id > 100)
   Rows Removed by Filter: 100
  Planning time: 0.067 ms
  Execution time: 0.047 ms
```

## 8.30 Indexes - Key Differences

Oracle database provides a wide variety of index types. The default index type is B-Tree.
The PostgreSQL CREATE INDEX syntax is similar to the Oracle syntax. PostgreSQL can re-index an existing index.
PostgreSQL does not have equivalent index types for the following Oracle indexes:

- Partial Indexes.
- Global Partitioned Indexes (Local Indexes can be considered equivalent to PostgreSQL Child table indexes).
- B-Tree Cluster indexes.
- Reverse Key indexes.

## 8.31 Comparison Table

| Oracle Indexes | PostgreSQL Availability Level | PostgreSQL Equivalent Level |
|---|---|---|
| CREATE INDEX… / DROP INDEX… | Available | Similar to Oracle |
| ALTER INDEX… (General Definitions) | Available | No equivalent |
| ALTER INDEX… REBUILD | Available | REINDEX |
| ALTER INDEX… REBUILD ONLINE | Limited Availability | CONCURRENTLY |
| Partial Indexes for Partitioned Tables (Oracle 12c) | Not available | No equivalent |
| Local and Global Indexes | Not available | No equivalent |
| Index Compression | No direct equivalent | No equivalent |
| Index Metadata | PG_INDEXES (Oracle USER_INDEXES) | No equivalent |
| Index-Organized Tables | Available | PostgreSQL CLUSTER |
| B-Tree Index | Available | B-Tree Index |
| B-tree cluster indexes | Not available | No equivalent |
| BITMAP Index / Bitmap Join Indexes | Not available | Could use BRIN index* |
| Reverse key indexes | Not available | No equivalent |
| Unique / non-unique Indexes | Available | Syntax is identical |
| Descending indexes | Available | ASC (default) / DESC |
| Index Tablespace Allocation | Available | SET TABLESPACE |
| Composite Indexes | Available | Multicolumn Indexes |
| Application Domain indexes | Not available | No equivalent |
| Function-Based Indexes | Available | PostgreSQL Expression Indexes |
| Index Parallel Operations | Not available | No equivalent |
| Invisible Indexes | Not available | Extension hypopg is not currently supported* |

## 8.32 Indexes - Key Differences

PostgreSQL supports multiple index types. Each type uses a different type of indexing algorithm. Each algorithm provides a benefit in performance for specific query types.
The PostgreSQL index types include:

- B-Tree
  Recommended to use this index type for queries with equality and range. When a row has a single key value use this type of index. These indexes can be applied against all data types and can retrieve NULL values. The default sort order is ascending.

- Hash
  Recommended for equality operators. The structure for a hash index is flat compared to a B-Tree index—the space difference can be noticeable. These index types are not transaction-safe and should not be used in replication. They must be manually rebuilt with REINDEX after a database crash if there were unwritten changes. You must rebuild hash indexes when you upgrade from any previous major PostgreSQL version.

- GIN (Generalized Inverted Indexes)
  This type of index is designed for indexing composite values, and the index must search for element values within the composite values. GIN indexes are recommended for indexing full-text search and working with array values.

- GiST (Generalized Search Tree)
  This index type is a balanced, tree-structured access method, and can act as a base template when you must implement different indexing schemes. GiST indexes allow you to build general B-Tree structures, R-Trees, and many other indexing schemes for complex operations. They are used to create indexes for geometric data types.GiSTindexes support full-text search.

- BRIN (Block Range Indexes)
  This index type reads the selected column's maximum and minimum values for each 8k page of stored data. PostgreSQL stores three pieces of information in the BRIN index: the page number, the minimum value, and the maximum value for the selected column. BRIN values are valid until any row of data stored on that page is updated. If the page has not been updated since the creation of the index or has been re-indexed since any row updates, PostgreSQL can verify if a value you are testing against is on that page. If not, the page is dropped from the result set. BRIN indexes are best used against data that you would consider immutable.

## 8.33 CREATE INDEX Synopsis

PostgreSQL Example:

Create index syntax with the following command:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table_name [ USING method ]
    ( { column | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

In the CREATE INDEX command, if you do not specify a type of query, the default index is B-Tree. For more about the CREATE INDEX command, see Create Index Syntax.
Oracle Example:
Create and drop an index using the command below:

```
CREATE UNIQUE INDEX Order_id_idx ON Orders (Order_ID DESC);
DROP INDEX Order_id_idx;
```

PostgreSQL Example:
Create and drop an index using the following example:

```
CREATE UNIQUE INDEX Order_id_idx ON Orders(order_ID DESC);
DROP INDEX Order_id_idx;
```

Oracle Example:
Use the following example to alter an index:

```
ALTER INDEX Order_id_idx RENAME TO Order_id_idx_OLD;
```

PostgreSQL Example:
Use the example below to alter an index:

```
ALTER INDEX Order_id_idx RENAME TO Order_id_idx_OLD;
```

For more about this command, see Alter an Index.
Oracle Example:
Alter an index to rebuild the tablespace with the following command:

```
ALTER INDEX Order_id_idx REBUILD TABLESPACE Test_IDX;
```

PostgreSQL Example:
Alter an index to rebuild the tablespace using the example below:

```
CREATE TABLESPACE SampleIDX LOCATION '/sample/indexes';
ALTER INDEX Order_id_idx SET TABLESPACE SampleIDX;
```

Oracle Example:
Use the following example to rebuild an index:

```
ALTER INDEX Order_id_idx REBUILD;
```

PostgreSQL Example:
Use the following command to re-index an index:

```
REINDEX INDEX Order_id_idx;
```

For more about the REINDEX INDEX command, see Reindex an Index.

Oracle Example:
Rebuild an index online using the command below:

```
ALTER INDEX Order_id_idx REBUILD ONLINE;
```

PostgreSQL Example:
Reindex online using the following command:

```
CREATE INDEX CONCURRENTLY Order_id_idx1 ON Order(Order_ID);
DROP INDEX CONCURRENTLY Order_id_idx;
```

## 8.34 Bitmap Indexes

A Bitmap index creates a separate bitmap, which is a sequence of 0 and 1, for each possible value of the column.
Characteristics of Bitmap Indexes:

- Best with columns with few unique values (low cardinality).
- Tables that have no, or little inserts or updates are good candidates.
- The bitmap index has a highly compressed structure, which makes them fast to read. The system can combine multiple indexes together for fast access to the underlying table.
- The overhead on maintaining bitmap indices can be large. A bitmap index modification requires more work by the system than a B-Tree index modification.

Oracle Example:
Create a bitmap index using the following example:

```
CREATE BITMAP INDEX IDX_BITMAP_Invoices_GEN ON Invoices(invoice_type);
```

A persistent bitmap index is not provided by PostgreSQL.
A BRIN index is a block range index. When you create a BRIN index PostgreSQL reads a selected column's maximum values and minimum values for each 8k page of stored data. PostgreSQL stores three pieces of information in the BRIN index: page number, minimum value and maximum value for the selected column.
After creation, the BRIN values are valid until the data is updated, which invalidates the BRIN index. You should use a BRIN index for immutable data. It is beneficial to cluster the column used for a BRIN index.
An Oracle BITMAP index and a PostgreSQL BRIN index do not have the same functionality. They are not equivalent index types.
PostgreSQL Example:
Use the example below to create a BRIN index:

```
CREATE INDEX IDX_BRIN_orders ON orders USING BRIN(total_amt);
```

## 8.35 B-Tree Indexes

The most common relational database index type are B-Tree indexes. This type is the default index. You walk the branches to find the node that contains the data you want to use. The idea is the amount of data on both sides of the tree are about the same and the number of levels to traverse to find a row is same approximate number. An Oracle B-Tree starts with only two nodes, one header and one leaf. The header contains a pointer to the leaf block and the values are stored in the leaf block. As the index grows, leaf blocks are added to the index. For more about B-Tree indexes, see B-Tree indexes.
To find a specific row, you check the header to find the range of values for each leaf and then traverse to that leaf node. Since a header only contains pointers, a single header can support a large number of leaf nodes. If the header block fills, a new header is established and the former header because a branch node.
They can enhance the performance of many types of queries. The B-Tree index is the better selection for equality and range queries.
This type of index is best when used for primary keys search or a column with a high cardinality and provide excellent performance with a many different types of query patterns. A B-Tree index can take a measurable amount of disk space.
Oracle Example:
Create a B-Tree index using the following command:

```
CREATE INDEX idx_invoice_date ON invoices (invoice_date);
```

In PostgreSQL, the default index is also B-Tree. PostgreSQL B-Tree indexes are similar to Oracle B-Tree indexes. With the certain operators in a query, the optimizer considers a B-Tree index. For more about B-Tree indexes, see Created Indexes. .
The operators include the following: =, <, >,<=, >=
Performance may be improved if the query uses the following clauses: BETWEEN, IS NULL, IS NOT NULL, or IN.
PostgreSQL Example:
Create an implicit B-Tree index using the example below:

```
CREATE INDEX idx_order_id ON orders(order_ID);
```

PostgreSQL Example:
Use the following example to create an explicit B-Tree index:

```
CREATE INDEX IDX_order_customer_id1 ON orders USING BTREE (order_customer_ID);
```

## 8.36 Invisible Indexes

In Oracle database, an Invisible Index is an index that is maintained by the database but ignored by the optimizer unless explicitly specified. The invisible index is an alternative to dropping an index or making an index unusable. This feature is also useful when certain modules of an application require a specific index without affecting the rest of the application.
Invisible indexes are used to:

- Test a dropped index and what effect this drop may have without dropping the index.
- Use a certain index for specific modules in an application or specific operations and to affect the complete application.
- Add an additional index to a collection of columns.

The Invisible Index feature is not supported in PostgreSQL.

For more about the Invisible Index feature, see Understand When to Create Multiple Indexes on the Same Set of Columns.

Notes:

- Rebuilding an invisible index will make it visible
- In a database hint, you can specify an invisible index.

Oracle Example:
Alter an existing index to invisible using the following example:

```
ALTER INDEX test_idx INVISIBLE;
```

Use the example below of an existing index to visible:

```
ALTER INDEX idx_test VISIBLE;
```

Create an invisible index using the following example:

```
CREATE INDEX idx_test ON orders(vendor_id) INVISIBLE;
```

Use this example to instruct the optimizer to include invisible indexes at the session level:

```
ALTER SESSION set optimizer_use_invisible_indexes = true;
```

## 8.37 Local and Global Partitioned Indexes

In Oracle, Partitioned Tables databases can use Local Indexes and Global Indexes. You must specify if the index is either local or global when creating a partitioned table index.

- Local Partitioned Index: Defined by the LOCAL keyword, this index type is a one-to-one mapping between an index partition and a table partition. Oracle automatically uses equal partitioning of the index based on the number of table partitions. Index maintenance operations can be performed independently because the index partition is separate. If needed, a single partition can be taken off-line and the index rebuilt. This operation does not affect the other table partitions. If a table partition is created or deleted Oracle automatically manages the index partitions.
- Global Partitioned Index: A global partitioned index is a one-to-many relationship, which allows one index partition to map to many table partitions. The default global index is non-partitioned, but you can partition the index if needed. The DBA can define as many partitions for the index as needed. There are specific index management and maintenance restrictions when creating a global partitioned index. For example, you must rebuild a global index if you drop a table partition because the entire index is affected. In the global index partition scheme, the index is harder to maintain since the index may span partitions in the base table.

For more about local and global partitioned indexes, see Partitioning Concepts.
Oracle Example:
Create a local index:

```
CREATE INDEX IDX_invoice_date_LOC ON invoices (invoice_DATE)
      LOCAL
        (PARTITION invoice_DATE_1,
         PARTITION invoice_DATE_2,
         PARTITION invoice_DATE_3);
```

Oracle Example:
Create a global index:

```
CREATE INDEX IDX_Invoice_Date_GLOB ON invoices(invoice_DATE)
      GLOBAL PARTITION BY RANGE (invoice_DATE) (
      PARTITION Invoice_DATE_1 VALUES LESS THAN (TO_DATE         ('03/01/2016','DD/MM/YYYY')),
      PARTITION Invoice_DATE_2 VALUES LESS THAN (TO_DATE('03/01/2017','DD/MM/YYYY')),
      PARTITION Invoice_DATE_3 VALUES LESS THAN (TO_DATE('03/01/2018','DD/MM/YYYY')),
      PARTITION Invoice_DATE_4 VALUES LESS THAN (MAXVALUE);
```

PostgreSQL has no direct equivalent to Local or Global indexes (there is Partition table in PostgreSQL 9.6).
PostgreSQL has a different method for partitioning tables while using table inheritance.
The Parent table is typically empty and an index on the Parent table would have no effect. The Child tables in the inheritance is used for table partition and indexes on the Child tables could be considered like the Local index. For more about partitioning tables, see Partitioning.
PostgreSQL Example:
Create the Parent Table using the following example:

```
CREATE TABLE daily_sales
(
      id   NUMERIC   PRIMARY KEY,
     sales_amt NUMERIC,
     sales_date    DATE   NOT NULL DEFAULT CURRENT_DATE
);
```

PostgreSQL Example: Use the following example to create Child Tables:

```
CREATE TABLE daily_sales_m1_to_m4
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '2018-01-01' AND sales_date < DATE '2018-05-01')
)
    INHERITS (daily_sales);

CREATE TABLE daily_sales_m5_to_m8
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '2018-05-01' AND sales_date < DATE '2018-09-01')
)
    INHERITS (daily_sales);

CREATE TABLE daily_sales_m9_to_m12
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '2018-09-01' AND sales_date < DATE '2019-01-01')
)
    INHERITS (daily_sales);
```

PostgreSQL Example:  Use the example below to create Child table indexes:

```
CREATE INDEX m1_to_m4_sales_date ON
daily_sales_m1_to_m4 (sales_date);

CREATE INDEX m5_to_m8_sales_date ON
daily_sales_m5_to_m8 (sales_date);

CREATE INDEX m9_to_m12_sales_date ON
daily_sales_m9_to_m12 (sales_date);
```

Local and Global indexes in Oracle do not have direct equivalents in PostgreSQL, but Child Table indexes are similar to Oracle Local Indexes.

## 8.38 Composite Indexes

When you create an index on multiple columns in a table. The index is called a composite or a concatenated index. You can create composite B-Tree indexes or bitmap indexes.

The optimizer uses the composite index when the WHERE clause in a query refers to all the columns in the index or even the first column. The order of the columns is important when creating a query. The most restrictive column should be first to improve query performance.

Oracle Example:
Use the example below to create a composite index:

```
CREATE INDEX IDX_cust_COMP ON
        cust(firstname, lastname, email);
```

Oracle Example:
Use the following example to drop a composite index:

```
DROP INDEX IDX_cust_COMP;
```

The PostgreSQL Multicolumn Index is equivalent to the Composite Index. You should create an index on every column that covers query conditions. The optimizer only optimizes the queries that reference the columns in the same order.

- Multicolumn Indexes are supported by GIN, B-Tree, BRIN, and GiST only.
- When creating a Multicolumn index, the maximum number of columns is 32.

PostgreSQL and Oracle use the same syntax.
For more information about the PostgreSQL Multicolumn Index, see Multicolumn Indexes.

PostgreSQL Example:
Use the example below to create a multicolumn index:

```
CREATE INDEX IDX_cust_MC
  ON cust( lastname, phone_num);
```

PostgreSQL Example:
Use the following example to drop the multiple-column index:

```
DROP INDEX IDX_cust_MC;
```

## 8.39 Index-Organized Table (IOT)

In an Index-organized table (IOT) the index determines the sort order for the table data and index level and is a special type of index/table hybrid. A common table is stored in a heap, which means the data is unsorted. An Index-organized table stores the table data in a B-Tree index structure sorted by the primary key. The structure of the index allows each leaf block to contain the all columns.

Based on the primary key, an IOT can provide an improvement in query performance due to the method of storing the table records.

The Oracle Index-Organized Table defined when the table is created, stays sorted by the index as long as the table is not dropped.

For more about the Oracle Index-Organized Table, see Indexes and Index-Organized Tables.
Oracle Example:
Create an index-organized table using the following example:

```
CREATE TABLE STATES_PROVINCES(
    ID NUMBER,
     NAME VARCHAR2(255) NOT NULL,
     SHORT_NAME VARCHAR2(255),
     COUNTRY VARCHAR(255),
     CONSTRAINT PK_ST_P_PK PRIMARY KEY(ID))
     ORGANIZATION INDEX;

INSERT INTO STATE_PROVINCES(id, name, short_name, country) SELECT ID, "NAME", "ABBREVIATION",
"COUNTRY" FROM state;

EXPLAIN PLAN FOR
SELECT * FROM state_provinces;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
----------------------------------------------------------
 Plant hash value: 4135328531


------------------------------------
    Id | Operation                    | Name           | Rows      | Bytes     | Cost  (%CPU)  |  |Time      |
   0  |  SELECT STATEMENT        |                      |   69        |  27600     |                2  (0)  |  00:00:1     |
|   1  |    INDEX FAST FULL SCAN|  PK_ST_P_PK |      69        |    27600 |                2 (0) |    00:00:01  |
```

PostgreSQL does not support IOTs, but has functionality that is somewhat similar with the CLUSTER feature. The CLUSTER statement requires an existing index and then defines the table sorting based on that index. The index determines how the table data is physically sorted.
You can recluster the table at any time with the CLUSTER statement. The table data is only in the cluster sort order when statement is run. If the table inserts, updates, or deletes the data, you should recluster the table to maintain the desired sort order.

PostgreSQL Example:
To create a table, insert data, and cluster the table:

```
CREATE TABLE cust_info (
      ID SERIAL,
       name VARCHAR(100) NOT NULL,
      Addr1 VARCHAR(200),
     City VARCHAR(100) NOT NULL,
          State VARCHAR(50) NOT NULL
   CONSTRAINT cust_info_pkey PRIMARY KEY (id));

INSERT INTO cust_info VALUES('Shaine Bennett','431-8211 Sit St.','Dublin','L');
INSERT INTO SYSTEM_EVENTS VALUES('Shaine Bennett','431-8211 Sit St.','Dublin','L');
INSERT INTO SYSTEM_EVENTS VALUES('Hamilton Monroe','P.O. Box 225, 3913 Magna Ave','Cork','M');

CLUSTER cust_info USING cust_info_pkey;
```

## 8.40 Function-Based Indexes

A Function-based index is an index created on the results of a function or expression. Typically, if you use a function on an indexed column in a WHERE clause, the index would not be used. Instead of indexing a column, you index the function on that column and store the result of the function. When a query could benefit from that index, the query is rewritten to allow the index to be used. For more information about function-based indexes, see Indexes and Index-Organized Tables.
Oracle Example:
Create a function-based index using the example below:

```
CREATE TABLE cust_name(
          CD_ID NUMERIC PRIMARY KEY,
           Name VARCHAR2(100) NOT NULL);


CREATE INDEX name_idx ON cust_name (UPPER(name));
```

The PostgreSQL Expression Indexes are similar to Oracle's Function-Based Indexes. Expression indexes are useful for queries that match on a function or modification of the data. PostgreSQL lets you index the result of the function, which allows searches to be as efficient as raw data values.

PostgreSQL Example:
Use the example below to create an Expression Index:

```
CREATE TABLE cust_name(
          CD_ID serial PRIMARY KEY,
           Name VARCHAR(100) NOT NULL);

CREATE INDEX upper_name_idx ON cust_name (UPPER(name));
```

Use EXPLAIN to verify the expression is being used.
PostgreSQL Example:
Use the following example to verify the expression is used:

```
 EXPLAIN
SELECT * FROM cust_name WHERE  upper(name) = 'OWENS';


                            QUERY PLAN
 ---------------------------------------------------------------------------------
 Index scan using upper_name_idx on cust_name (cost=0.14..9.16 rows =1 width=18)
   Index Cond:(upper((name)::text)='OWENS'::text)
```

**Partial Indexes**

PostgreSQL also has partial indexes. A partial index is a subset of the table's data. It is an index with a WHERE clause. The concept is to increase the index efficiency by reducing the size of the index. A smaller index takes less storage, is easier to maintain, and may be faster.
For more about partial indexes, see Create Index.

PostgreSQL Example:
Create a partial index using the following example:

```
CREATE INDEX IDX_access_log_client_ip_idx ON access_log(client_ip)
       WHERE (client_ip > inet '168.190.100.0' and client_ip < inet '168.190.100.255');

SELECT * FROM access_log WHERE client_ip = '168.190.100.56';
```

## 8.41 Information Views - Key Differences

Oracle database provides a Data Dictionary and the V$ views to monitor performance and activities in the database. The data dictionary stores all information that is used to manage the objects in the database.

Oracle database contains a set of underlying views maintained by the database server and accessible to the administrator. These views are called Dynamic performance views (V$ Views) because they are unceasingly updated while the database is open and in use.

PostgreSQL provides system catalogs, which may change with each version of PostgreSQL, statistic collector, and the SQL-92 standard information_schema views. Each system provides information on the current database state.

PostgreSQL database dynamic meta-data and static meta-data is stored in a collection of tables called the System Catalog Table. These tables are similar to the data dictionary.

PostgreSQL's statistics collector is a subsystem that supports the collection and reporting of information about server activity.

The information_schema views provide meta-data information about the tables, columns, and other database information.

For more about the System Catalog Table, see System Catalogs.

## 8.42 Comparison Table

| Information | Oracle | PostgreSQL |
|---|---|---|
| Current run-time parameters | V$PARAMETER | PG_SETTINGS |
| System statistics | V$SYSSTAT | PG_STAT_DATABASE |
| I/O operations | V$SEGSTAT | PG_STATIO_ALL_TABLES |
| Database properties | V$DATABASE | PG_DATABASE |
| Database tables | DBA_TABLES | PG_TABLES |
| Database sessions | V$SESSION | PG_STAT_ACTIVITY |
| Database locks | V$LOCKED_OBJECT | PG_LOCKS |
| Database roles | DBA_ROLES | PG_ROLES |
| Database users | DBA_USERS | PG_USER |
| Tables privileges | DBA_TAB_PRIVS | TABLE_PRIVILEGES |
| Table columns | DBA_TAB_COLS | PG_ATTRIBUTE |

## 8.43 Migrating Oracle V$View and the Data Dictionary to PostgreSQL System Catalog & The Statistics Collector

Oracle contains a set of performance views that are maintained by the database server and are accessible to the DBAs. These views provide continually updated operational statistics and state.

A collection of views and internal tables that supply database metadata is contained in the data dictionary. The data dictionary stores all information that is used to manage the objects in the database. The data dictionary contents are persistent. The data dictionary tables and views are not listed alphabetically but instead are grouped by function.

The data dictionary has the following primary uses:

- Finding information about users, schema objects, and storage structures
- The data dictionary is modified after each DDL statement
- Read-only reference for database information

With some exceptions, the names of the objects in the data dictionary begin with either USER, ALL, or DBA. The USER views typically show information about objects owned by the user account. The ALL views include the USER objects plus information about the objects on which privileges have been granted to PUBLIC or the user. DBA views encompass all database objects.

**Note:** The USER, ALL, and DBA views show the information about schema objects that are accessible to you at different levels of privilege.

The data dictionary includes the following information:

- Definition of all schema objects
- Space allocated for and currently used by schema objects
- Default values for all columns
- Integrity constraint information
- Oracle users
- Privileges and roles granted to each user
- Auditing information

The following is a small list of the views available:

- DBA_ALL_TABLES: Describes all object tables and relational tables in the database
- DBA_DATA_FILES: Describes the database files
- DBA_TABLES: Describes all relational tables in the database
- DBA_TABLESPACES: Describes all tablespaces in the database
- DBA_USERS: Describes all users of the database
- DBA_TAB_COLS: Describes the columns of all tables, views, and clusters in the database, includes hidden columns
- USER_TABLESPACES: Describes the tablespaces accessible to the current user
- USER_USER: Describes the current user

Oracle contains a set of underlying views maintained by the database server and accessible to the administrator. These views are called Dynamic performance views (V$ Views) because they are unceasingly updated while the database is open and in use.
These views provide data on internal disk structures and memory structures. They are read-only.
V$ views include the following:

- V$SESSION: lists session information for each current session
- V$LOCKED_OBJECT: lists all locks acquired by every transaction on the system
- V$INSTANCE: state of the current instance
- V$SESSION_LONGOPS: status of various operations that run longer than a set time period.
- V$MEMORY_TARGET_ADVICE: provides information about how the memory_target parameter should be sized based on the current sizing and satisfaction metrics

In PostgreSQL, to review information about the database state or current activities you can use three different sets of meta-data tables. These tables are similar the V$ performance views and Oracle data dictionary.

All users can create read-only queries for the system catalog, the statistics collector, and the information schema.

| Category | Description |
|---|---|
| System catalog | A schema with tables and views that contain meta-data about all database objects and other information. |
| Statistic collection views | A subsystem that supports collection and reporting about server activity. |
| Information schema views | A collection of views that contain information about current database objects. Some of these views are equivalent to the Oracle USER_* Data Dictionary tables. |

For more information about the data dictionary, see Catalog Views / Data Dictionary Views.

## 8.44 System Catalog Tables

PostgreSQL database dynamic meta-data and static meta-data is stored in a collection of tables called the System Catalog Table. These tables are similar to the data dictionary.
The pg_catalog schema is the standard PostgreSQL meta-data and core schema. General database information is stored in pg_database and statistics are stored in pg_stat_database.

The availability of certain system catalogs may change with every version of PostgreSQL.

For more information about the System Catalog Table, see System Catalogs.
PostgreSQL Example:
Use the following example to return the information in the pg_catalog schema:

```
select * from pg_tables where schemaname='pg_catalog';
```

The system catalog include the following:

| Table name | Purpose |
|---|---|
| pg_database | Contains a row for every database in the cluster. |
| pg_tables | Provides access to information about each table in the database |
| pg_index | Contains part of the information about indexes. The rest of the information is contained in pg_class. |
| pg_cursors | Lists the cursors that are currently available. |
| pg_stat_database | Each row in this table contains live data for each database. |

**Statistics Collector**
PostgreSQL's statistics collector is a subsystem that supports the collection and reporting of information about server activity.

The collector can access tables and indexes in both disk-block and individual-row terms. The collector also tracks the total number of rows in each table as well as the vacuum and analyze actions for each table.

PostgreSQL Example: Use the example below to return active current user activity:

```
SELECT * FROM pg_stat_activity WHERE STATE = 'active';
```

The statistics collector views include the following:

| Table name | Purpose |
|---|---|
| pg_stat_activity | One of two views that show current user activity. This view shows information about current connections to the database, and the type of locks they have acquired. |
| pg_stat_all_tables | Information for each table in the database, shows statistics about access to specific table. |
| pg_statio_all_tables | Information on performance and I/O output. |
| pg_stat_database | Information on database-wide statistics |
| pg_stat_bgwriter | Information on the processes that manage writing data to disk, either by checkpoint or background writer. |
| pg_stat_all_indexes | Information on each index in the current database. |

**Information_Schema**
The information schema automatically exists in all databases. The information_schema views provide metadata information about the tables, columns, and other database information.

- The information schema is defined in the SQL-92 standard and supported by PostgreSQL. The structure and information available in the schema is stable. Since they are defined by the standard, they may not have information about PostgreSQL-specific functions.
- The initial database user in the cluster owns the schema and has all privileges on the schema.

The information_schema views contain the following:

| Table name | Purpose |
|---|---|
| Schema | Lists the schemas owned by you |
| Tables | Lists all tables you have the right to access |

| | |
|---|---|
| Columns | Lists all columns in all tables you have the right to access |
| Views | Lists all views you have the right to access |
| Table_privileges | Lists the privileges you have been granted or own for each accessible object in the database |
| Check_constraints | Lists all CHECK constraints for objects you have the right to access |

PostgreSQL Example:
To return every table in a database, use the example below:

```
select * from information_schema.tables;
```

## 8.45 Index-Organized Tables - Key Differences

Oracle has an index-organized table that stores the data in a B-Tree index. PostgreSQL does not support index-organized tables, but you can use CLUSTER to organize the data by an index.

## 8.46 Index-Organized Table (IOT)

An index-organized table (IOT) is a table that stores data in a B-Tree index structure logically sorted in primary key order. This structure is different than normal relational tables which store rows in any order and are called heap-organized tables. If the primary key comprises most or all of the columns in a table, you could store the data in an IOT. An IOT is useful for frequently used lookup tables. The following are the properties and restrictions:

- Rows are accessed by logical rowid
- You must modify a IOT index property using ALTER TABLE. Using ALTER INDEX causes an error.
- You must have a primary key
- You cannot create an IOT in a cluster
- You cannot define an IOT with a column of LONG data type

Oracle Example:  An Oracle IOT with the Explain Plan:

```
CREATE TABLE STATES_PROVINCES(
    ID NUMBER,
     NAME VARCHAR2(255) NOT NULL,
     SHORT_NAME VARCHAR2(255),
     COUNTRY VARCHAR(255),
     CONSTRAINT PK_ST_P_PK PRIMARY KEY(ID))
     ORGANIZATION INDEX;

INSERT INTO STATE_PROVINCES(id, name, short_name, country) SELECT ID, "NAME", "ABBREVIATION",
"COUNTRY" FROM state;
EXPLAIN PLAN FOR
SELECT * FROM state_provinces;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------
 Plant hash value: 4135328531

-----------------------------------
    Id | Operation                    | Name            | Rows       | Bytes      | Cost  (%CPU)  |  |Time       |
   0  |  SELECT STATEMENT        |                      |    69          |  27600      |                 2  (0)  |  00:00:1     |
|   1  |     INDEX FAST FULL SCAN|  PK_ST_P_PK |       69         |    27600 |                  2 (0) |    00:00:01   |
```

PostgreSQL supports heap tables only. You can use the CLUSTER clause to align the contents of the heap table with an index. The clustering on an index forces the physical data to be in the same order as the chosen index. The CLUSTER command is used to create a complete copy of the table and the old table is dropped. The CLUSTER command requires virtually twice the disk space to hold the initial organized copy. You can have only one clustered index on a table. Clustering on an index in PostgreSQL does not maintain that order, you must reapply the CLUSTER process to maintain the order. The CLUSTER process acquires a table lock. You can schedule a cluster.
PostgreSQL Example:
Use the following example to create an index on a table, CLUSTER the table, ANALYZE the table, and a SELECT statement:

```
CREATE INDEX state_idx
on state
using btree (id)
with (FILLFACTOR=70);

ALTER TABLE state CLUSTER on state_idx;

ANALYZE state;

SELECT * FROM state;
```

## 8.47 JSON Support - Key Differences

JSON support was introduced in Oracle database 12c release 2 (12.1.0.2), including transactions, indexing, and views. Oracle works with JSON data and XML data in similar ways. Data is stored, indexed, and queried and can use relational database features including transactions, constraints, and views.

PostgreSQL supports native JSON and JSONB data types. PostgreSQL provides functions and operators for manipulating JSON data. PostgreSQL supports B-Tree, HASH, and GIN indexes on JSON tables.

## 8.48 Common Task Differences

**Specify a column to support JSON documents**

Oracle Example:

```
CREATE TABLE json_sample (
            ID RAW(16) NOT NULL,
            json_data CLOB NOT NULL
 );
```

PostgreSQL Example:

```
CREATE TABLE orders (id serial primary key, data jsonb);
```

**Select a JSON document**

Oracle Example:

```
SELECT json_data FROM json_sample;
```

PostgreSQL Example:

```
SELECT cust_info FROM customers;
```

**Select an element from a JSON document**

Oracle Example:

```
SELECT j.json_data.names FROM json_sample j;
```

PostgreSQL Example:

```
SELECT  cust_info->>'Region' from cust_info WHERE id = 1;
```

**Select a matching a requested pattern**

Oracle Example:

```
SELECT j.json_data FROM json j WHERE j.json_data like '%Sev%';
```

PostgreSQL Example:

Use jsonb_pretty to return JSON data and then convert jsonb to text:

```
SELECT * FROM (SELECT jsonb_pretty(data) AS raw_data FROM users) raw_users WHERE raw_data LIKE '%Mac%';
SELECT key, value FROM products, lateral jsonb_each_text(data) WHERE value LIKE '%M%';
```

**Select json elements that match a pattern in a specific field**

Oracle Example:

```
SELECT e.emp_data.name FROM employees e WHERE e.data.active = 'true';
```

PostgreSQL Example:

Use the following example to return only the results that match the lastname:

```
SELECT * FROM users WHERE data->>'LastName' = 'Kasper';
```

## 8.49 JSON Support

JSON (JavaScript Object Notation) is a language-independent open data format. JSON can be used for data interchange or to pass data between the database and a RESTful web service. JSON stores text in key-value pairs where the value can be a JSON object, JSON array, number, string, boolean, or null. A JSON object contains one-to-many key-value pairs separated by commas and placed in curly brackets. The key-value pairs can be different types. A JSON-array is a comma-separated list of objects inside square brackets. The objects can be different types. JSON data and XML data can be used in Oracle in similar ways. The latest versions of Oracle support JSON relational database features, including transactions, indexing, and views.

As a table, JSON data can be stored in the following ways;

- VARCHAR2
- CLOB
- BLOB

Oracle recommends, when you create the table, to add a check constraint. The check constraint ensures the validity of the column values.

A table with JSON columns can coexist with any other kind of data. A table can have multiple columns with JSON documents.
Oracle Example:
Use the example below to create a table, add constraints, and insert JSON data:

```
CREATE TABLE json_sample (id RAW(16) NOT NULL, json_data  CLOB,
CONSTRAINT json_sample_pk PRIMARY KEY (id),
CONSTRAINT json_sample_chk CHECK (data IS JSON));

INSERT INTO json_sample (id, json_data) VALUES (SYS_GUID(),
'{
    "Names"    : "Dominique Farmer",
      "org_id": "960302",
       "country":"Australia"}');
}');
```

PostgreSQL has native JSON support. PostgreSQL has two data types:
JSON: Basically, a blob that stores the JSON data in raw format. White space is preserved, along with the order of keys, even duplicate keys. The data type may be slow because a query must load and parse the JSON blob each time.

JSONB: Stores JSON data in a custom binary format optimized for querying and does not reparse the JSON blob each time.

The JSONB data type has the following considerations:

- JSONB is stricter and disallows characters unless the database encoding is UTF8 or higher.
- Rejects the NULL character (\u0000)
- Does not preserve white space
- Does not preserve the order of the object keys
- Does not keep duplicate object keys

If you know that you are not performing queries on the JSON blob, then use the JSON data type. The JSONB column type may be slightly slower to input, there is an added conversion overhead and it may take more storage space. JSONB is significantly faster to process, since there is no reparsing, and supports indexing.

Oracle, in the latest version, supplies other JSON tools, such as:

- JSON SQL
- JSON Conditionals
- JSON Functions

PostgreSQL Example:
To create a table and insert data, use the following example:

```
create table json_orders (id serial primary key, data jsonb);

INSERT INTO json_orders (DATA) VALUES (
'{
   "Number": "323",
      "userId":"214",
     "Items": [
           { "ProductID": "HB324",
           "Name":"Dutch Oven",
           "Price":"85.00" }
    ]
  }'
);
```

In Oracle, you can use dot notation to access JSON data in a table.
The following rules apply to use dot notation:

- Alias the table
- Column must be a JSON column.
- JSON object key must be case-sensitive and match the object key for the value you want to retrieve

Oracle Example:

Use dot notation to access JSON data:

```
SELECT j.data.Names,j.data.org_id,j.data.country
FROM json_sample j;

Names          org_id          country
-------             --------         ----------
Kasper, Gary    979443      New Zealand

1 row selected.
```

The following operators are used to query jsonb documents in PostgreSQL:

- -> returns the JSONB object field or array by key
- ->> returns the JSONB object field as text

PostgreSQL Example:
The following is an example of a simple select statement:

```
 SELECT
   u.id as id, u.data->>'FirstName' as first_name,
     u.data->>'LastName' as last_name,
     u.data->'Address'->>'Country' as country
 FROM json_users AS u;

 id        first_name        last_name       country
2032          Gary           Kasper          Macedonia
```

You can use the relational features in SQL and join two tables and use a GROUP BY to order the results.
PostgreSQL Example:
Join tables and use a GROUP BY:

```
 SELECT u.id as user_id,
    u.data->>'LastName' as last_name,
     COUNT(o.id) as orders_count
FROM orders AS o INNER JOIN users as u ON u.id = cast(o.data->>'UserId' as numeric)
 GROUP BY u.id, last_name;

id            last_name              orders_count
2032              Kasper               12
```

## 8.50 Indexing and Constraints with JSONB Columns

Without an index, each query is a table scan. To improve performance, you can add an index.
PostgreSQL Example:
To create a unique index, use the example below:

```
CREATE unique INDEX json_email_idx ON cust_info ((cust_info->>'email'));
```

PostgreSQL Example:
In the example below, the first insert fails because of the unique constraint:

```
INSERT INTO cust_info VALUES
('{ "name": "Thomas", "street_address": "438 Luctus St.", "Region": "M", "country":"Ireland", "email":
"consectetuer@Inmipede.ca"}');

INSERT INTO cust_info VALUES
('{ "name": "Sing", "street_address": "P.O. Box 953", "Region": "TR", "country":"France", "email":
"vSing@test.com"}');

ERROR: duplicate key value violates unique constraint "json_email_idx"  Detail: Key ((emp_data ->>
'email'::text))=(consectetuer@Inmipede.ca) already exists.
```

PostgreSQL supports B-Tree, HASH, and GIN for JSON documents.
A B-Tree index performs best when indexing searches for a single attribute or a few predetermined attributes. A HASH index performs well, but you should provide a WAL (write ahead log).

PostgreSQL supports the GIN (Generalized Inverted Index). The GIN is an inverted index. The key is stored in the index entry and the mapping information for the key is stored in the posting tree. GIN is useful when an index must map many values to a row, such as indexing an array or documents.
PostgreSQL Example:
To add a GIN index for a key and a GIN index for the JSON column, use the following example:

```
create index json_users_idx on users using gin((data->'FirstName'));
create index json_orders_idx on orders using gin((data));
```

## 8.51 Materialized View - Key Differences

The PostgreSQL materialized view has the following differences when compared to Oracle materialized views:

- PostgreSQL does not support automatic re-caching.
- PostgreSQL supports complete refresh only.
- In PostgreSQL, you cannot edit an existing materialized view.
- Inserting data into a materialized view isn't possible.
- The views cannot be user dependent or time dependent.

## 8.52 Common Task Differences

**Create materialized view**

Oracle Example:

```
CREATE MATERIALIZED VIEW mv1
 AS SELECT * FROM finance.invoices;
```

PostgreSQL Example:

```
CREATE MATERIALIZED VIEW mv_sample
 AS SELECT * FROM country;
```

**Refresh materialized view manually**

Oracle Example:

```
EXECUTE DBMS_MVIEW.REFRESH('mv1', 'F');
```

PostgreSQL Example:

```
REFRESH MATERIALIZED VIEW mv_sample;
```

**Fast refresh**

Oracle Example:

```
CREATE MATERIALIZED VIEW mv1 REFRESH FAST ON COMMIT AS SELECT * FROM finance.invoices;
```

PostgreSQL Example:

Create a trigger to initiate a refresh on the underlying tables. Automatic incremental refresh

Oracle Example:

```
CREATE MATERIALIZED VIEW LOG ON finance.invoices…
 INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW mv1 REFRESH FAST AS SELECT * FROM finance.invoices;
```

PostgreSQL Example:

Has no equivalent
Inserting data into certain types of materialized views is supported in Oracle but not in PostgreSQL.

A possible workaround would be to create regular tables and maintain them with triggers or scripts to “refresh” them.

## 8.53 Materialized Views

Materialized view are able to improve the processing speed of queries. Unlike a view, where only the defining SQL query is stored, a materialized view stores the result set as a container table. With time, the materialized view data can become stale and must be refreshed. The data in a materialized view is updated by either a complete, or (in Oracle), an incremental refresh.
Oracle supports multiple methods of refreshing a materialized view:

- Manually (ON DEMAND)
- Automatically (ON COMMIT, DBMS_JOB)

The data that is stored in the Materialized view will refresh in a defined interval relative to the query it was created with.

Oracle Materialized Views are handy with the following use cases:

- Mitigating remote databases, to improve performance and security
- Data warehouse and reporting
- Increasing query performance by storing complex query results
- Multi database data replication

A SELECT query defines the materialized view. The FROM clause of a materialized view definition query can reference tables, views, and other materialized views. The source objects are also called master tables (replication terminology) or detail tables (data warehouse terminology). The database that contains master tables are called the master database.

In Oracle, the BUILD IMMEDIATE option during the query creation instructs Oracle to populate the view immediately. This option is the default. A DEFERRED option instructs Oracle to update the view by the next REFRESH operation. This first refresh must always be a complete refresh. Until the materialized view is populated the view has a staleness value of UNUSABLE.
A PostgreSQL materialized view has the following limitations:

- Does not support automatic re-caching. The view may be refreshed either manually or a trigger to automate the refresh. The manual option could be a REFRESH MATERIALIZED VIEW job.
- Supports complete refresh only.
- Existing materialized views are not editable. You must drop and recreate the view and all indexes if you wish to change it.
- Inserting, updating, or deleting data are not supported.

- Definitions cannot be user-dependent or time-dependent. PostgreSQL does not support a materialized view defined with, for example, WHERE user=current_user( ).

## 8.54 Oracle Fast and Complete Refresh

You can specify REFRESH FAST for a single-table aggregate materialized view if you have created a materialized view log for the underlying table. If the log is not created or unavailable, the refresh command will fail.

A complete refresh occurs when the materialized view is initially defined. Additionally, you can request a complete refresh at any time. This refresh requires reading the master table to compute the results and can be a time-consuming process.

If you specify a REFRESH FAST, only the UPDATE, DELETE, and INSERT deltas are refreshed. A materialized view log must be present.

You can refresh automatically on the next COMMIT performed on the master table. The ON COMMIT refresh can be used on a single-table aggregate materialized view and materialized views containing only joins.

## 8.55 PostgreSQL REFRESH

In PostgreSQL, the command below completely replaces the contents of a materialized view:

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name [ WITH [ NO ] DATA ]
```

You must be the owner of the view.
The CONCURRENTLY option allows a populated view to be refreshed without locking out concurrent selects. The option may not be used if the materialized view is empty.

To use CONCURRENTLY, the view must have at least one UNIQUE index with these limitations:

- Defined with only column names
- Indexes all rows
- Cannot contain expressions
- Cannot include a WHERE clause.

Without CONCURRENTLY, a refresh of a large view will take less time and resources but could block other connections.

If the WITH DATA is defined, the query is executed to refresh the data and the materialized view is in a scannable state.

If the WITH NO DATA option is specified, no new data is generated, and the materialized view is left in an unscannable state, this state means that the materialized view can't be queried again until the next refresh.

## 8.56 Materialized View Refresh Strategy

PostgreSQL does not support automatic re-caching of materialized views. The view may be refreshed either by running a job with the REFRESH MATERIALIZED VIEW command or a trigger to automate a refresh.
Oracle Example:
Create a Materialized View:

```
CREATE MATERIALIZED VIEW mv_US_STATES
AS SELECT name as state_name, ABBREVIATION as short_name
FROM STATE
WHERE COUNTRY = 'USA';
```

Oracle Example:
Create a Materialized View log, a materialized view with REFRESH FAST ON COMMIT, index the materialized view, and gather statistics:

```
CREATE MATERIALIZED VIEW LOG ON time2
WITH ROWID, SEQUENCE (t_key, amt)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW mv_sample
REFRESH FAST ON COMMIT
AS SELECT time_key,
    sum(amt) as amount_sum,
      count(*) as row_count,
     count(amt) as amt_count
FROM time2
 GROUP BY time_key;

create index mv_time2_idx ON mv_sample(time_key);

execute dbms_stats.gather_table_stats(user,'mv_sample');
```

PostgreSQL Example:
Use statement triggers to refresh the materialized view:

```
DROP TABLE IF EXISTS "contact_info";

CREATE TABLE "contact_info" (
id SERIAL PRIMARY KEY,
 Names varchar(255) default NULL,
 Phone_Contact varchar(100) default NULL,
  email varchar(255) default NULL
);

DROP MATERIALIZED VIEW IF EXISTS mv_contacts;

CREATE MATERIALIZED VIEW mv_contacts AS
select id, names, phone_contact from contact_info order by id;

CREATE OR REPLACE FUNCTION tr_refresh_mv_contacts() RETURNS TRIGGER AS
$$
begin
        refresh materialized view mv_contacts;
         return null;
end;
$$ language plpgsql;

CREATE TRIGGER tr_01_refresh_mv_contacts AFTER TRUNCATE OR INSERT OR UPDATE OR DELETE
ON contact_info FOR EACH STATEMENT
execute procedure tr_refresh_mv_contacts();

INSERT INTO "contact_info" (Names,Phone_Contact,email) VALUES ('Denise Dillon','(944) 461-
2810','id.libero.Donec@ultriciesligulaNullam.ca'),('Hedda Gibbs','(213) 479-8938','amet@Nunc.com'),('Illana
Golden','(103) 564-1257','fermentum@dui.net'),('Logan Hernandez','(815) 887-9037','ut@magna.co.uk'),('Eaton
Rodriguez','(829) 584-0706','ac@acrisusMorbi.org'),('Armand Acosta','(355) 509-
2228','nunc.est.mollis@mi.com'),('Rhea Olson','(906) 879-5759','et.magnis@gravida.com'),('Neve Mcfadden','(608)
594-7514','Suspendisse.dui.Fusce@aliquetmolestie.org'),('Nissim English','(287) 337-
2664','ullamcorper.Duis.cursus@bibendum.com'),('Eagan Crosby','(910) 929-7513','tincidunt@semmolestie.org');

select count(*) from mv_contacts;

count
-----------
10
```

## 8.57 Partitioning - Key Differences

Oracle database provides a variety of partitioning strategies. Partitioning allows tables, indexes, and index-organized tables to be subdivided. This subdivision allows for access and management at a finer granular level.

PostgreSQL supports partitioning by table inheritance and checked conditions.

Using the INHERITS keyword, child tables are linked to parent tables, using the CHECK constraints ensures that the relevant data is inserted or updated into the specific child table, and, finally, triggers can redirect DML actions on the parent table to populate the child tables.

## 8.58 Table Partitioning

Partitioning enhances the performance, manageability, and availability of applications. Partitioning may also help with the total cost of ownership for large amounts of data. Partitioning allows tables, indexes and index-organized tables to be subdivided into smaller objects. This ability to support smaller objects helps in the management and accessibility of these objects.

Oracle provides a variety of partitioning strategies and extensions. A partition is designed to be transparent from the user perspective, so partitioning can be applied to application’s data without the need to update the application's architecture or code.

Partitioning allows a table, index, or index-organized table to be subdivided. Every partition has its own name and may have its own storage characteristics. Multiple partitions can be managed either collectively or individually. Each row in a partitioned table is assigned to a single partition. A partition key contains one or more columns that determine the partition that stores each row. Oracle automatically directs insert, update, and delete operations to the appropriate partition by using the partition key. Any table, except a table with a LONG or LONG RAW data type, can be partitioned.

**Partitioning Strategies**

Oracle supports the following fundamental data distribution methods:

- Hash
- List
- Range

With these methods, a table can be partitioned as a single-level or composite. Each partitioning strategy has different advantages and design considerations.

## 8.59 PostgreSQL Implementation

PostgreSQL supports partitioning by table inheritance and checked conditions. Child tables have the same structure as their parent table. A query run against the parent table also queries the child tables. The checked conditions provide a means to query specific children of the parent table.

The following caveats apply to PostgreSQL partitioned tables:

- No automatic way to verify if the check constraints are mutually exclusive.
- Designing the partitions in such a way that the partition key columns of a row do not change enough to move the row to a different partition. To handle this case, it is recommended to add update triggers to the partition tables, but this addition is more complicated.
- If you are using the VACUUM and ANALYZE commands manually, you must run them on each partition.

## 8.60 Single-Level Partitioning

**Hash Table Partitioning**
Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the identified partitioning key. A hashing algorithm evenly distributes rows among the same-size partitions. Hash partitions is the method for distributing data evenly across devices.

They are an alternative to range partitioning, especially if the data is not historical or has no obvious partition key.

**Note:** You cannot change the hashing algorithm used by partitioning.

Oracle Example:
Use the following example to create a hash partitioned table:

```
CREATE TABLE golf_sets
      (id    NUMBER NOT NULL,
       Manufacturer  VARCHAR2(60))
        PARTITION BY HASH (id)
              PARTITIONS 3
       STORE IN (golfset1, golfset2, golfset3);
```

## 8.61 List Table Partitioning

List partitioning enables you to explicitly control how rows map to partitions by specifying a list of values for the partitioning key in the description for each partition. This feature allows you to organize unordered or unrelated sets of data. A default partition enables you to avoid specifying all possible values for a list-partitioned table by using the default partition, if rows do not map to another partition, this will not cause an error.
Oracle Example:
Use the example below to create a list-partitioned table:

```
CREATE TABLE customer_locations
      (name    VARCHAR2(50) NOT NULL,
       email  VARCHAR2(100),
       state_code  VARCHAR2(2))
       PARTITION BY LIST (state_code)
      (PARTITION region_south VALUES ('GA', 'TN', 'KY'),
     PARTITION region_north VALUES ('MA', 'DE', 'CT') );
```

## 8.62 Range Table Partitioning

Range partitioning maps data to partitions based on a range of values of the partitioning key when you establish each partition. This is the most common type of partition and is often used with dates. Each

partition has a VALUES LESS THAN clause. The clause specifies the non-inclusive upper bound for the partitions. Any values of the partitioning key equal to or higher than this clause are added to the next higher partition.
Create a range-partitioned table using the example below:

```
CREATE TABLE SETUP1
(id    NUMBER NOT NULL,
 SETP_DATE  DATE   NOT NULL
)
 PARTITION BY RANGE (SETUP_DATE)
 (PARTITION SETUP_DATEp1 VALUES LESS THAN (TO_DATE('01/01/2016',
                                  'DD/MM/YYYY')),
  PARTITION SETUP_DATEp2 VALUES LESS THAN (TO_DATE('01/01/2017',
                                  'DD/MM/YYYY')) ,
  PARTITION SETUP_DATEp3 VALUES LESS THAN (MAXVALUE) );
```

PostgreSQL implements the following methods:

- Range Partitioning
- List Partitioning

## Steps to Create a PostgreSQL Range partition

PostgreSQL Example:
Use the example below to create a master table that contains data on sales stored on a daily basis. The records a user inserts into the master table are moved to the child table based on the sales_date:

```
CREATE TABLE daily_sales
(
       id   NUMERIC   PRIMARY KEY,
      sales_amt NUMERIC,
      sales_date    DATE   NOT NULL DEFAULT CURRENT_DATE
);
```

PostgreSQL Example:
Use the following example to create the child range partition tables:

```
CREATE TABLE daily_sales_m1_to_m4
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '1960-01-01' AND sales_date < DATE '1960-05-01')
)
    INHERITS (daily_sales);


CREATE TABLE daily_sales_m5_to_m8
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '1960-05-01' AND sales_date < DATE '1960-09-01')
)
```

```
    INHERITS (daily_sales);


CREATE TABLE daily_sales_m9_to_m12
(
    PRIMARY KEY (id, sales_date),
     CHECK (sales_date >= DATE '1690-09-01' AND sales_date < DATE '1961-01-01')
)
    INHERITS (daily_sales);
```

PostgreSQL Example:
You can verify that the tables are linked, and the partition is successfully created with the following query:

```
\d+ daily_sales
```

PostgreSQL Example:
To create indexes on the child tables, use the example below:

```
CREATE INDEX m1_to_m4_sales_date ON
daily_sales_m1_to_m4 (sales_date);

CREATE INDEX m5_to_m8_sales_date ON
daily_sales_m5_to_m8 (sales_date);

CREATE INDEX m9_to_m12_sales_date ON
daily_sales_m9_to_m12 (sales_date);
```

PostgreSQL Example:
Use the following example to create a trigger function:

```
CREATE OR REPLACE FUNCTION daily_sales_insert()
RETURNS trigger AS $$
BEGIN
    IF (NEW.sales_date >= DATE '1960-01-01' AND NEW.sales_date < DATE '1960-05-01') THEN
    INSERT INTO
     daily_sales_m1_to_m4 VALUES (NEW.*);
ELSEIF (NEW.sales_date >= DATE '1960-05-01' AND NEW.sales_date < DATE '1960-09-01') THEN
    INSERT INTO
     daily_sales_m5_to_m8 VALUES (NEW.*);
 ELSEIF (NEW.sales_date >= DATE '1960-09-01' AND NEW.sales_date < DATE '1961-01-01') THEN
    INSERT INTO
     daily_sales_m9_to_m12 VALUES (NEW.*);
    ELSE
      RAISE EXCEPTION 'Date is out of range. Check the sales_record_insert_trigger() function';
    END IF;
    RETURN NULL;
END;
```

```
$$
LANGUAGE plpgsql;
```

PostgreSQL Example:
Use the example below to create the trigger to execute the function:

```
CREATE TRIGGER daily_sales_trigger
    BEFORE INSERT ON daily_sales
    FOR EACH ROW
     EXECUTE PROCEDURE
daily_sales_insert();
```

PostgreSQL Example:
To set a constraint, use the example below:

```
SET constraint_exclusion = on;
```

PostgreSQL Example:
Insert records into the parent table using the following example:

```
INSERT INTO daily_sales (id, sales_amt, sales_date) VALUES (1, 330, TO_DATE('02/15/1960')), (2,770,
TO_DATE('03/01/1960')), (3, 50000, TO_DATE('06/06/1960')), (4, 12000, TO_DATE('08/14/1960')), (5, 7773,
TO_DATE('10-14-1960')), (6, 380238, TO_DATE('12/12/1960'));
```

PostgreSQL Example:
Use the example below to run a query on one of the child tables:

```
SELECT * FROM daily_sales_m5_to_m8;

id      sales_amt       sales_date
-------------------------------------
3        500000           1960/06/06
4        12000            1960/8/14
```

## 8.63 Steps to Create a PostgreSQL List Partition

The list partition is similar to the range partition.
PostgreSQL Example:
1. Create a master table:

```
CREATE TABLE daily_sales_listp
(
       id   NUMERIC   PRIMARY KEY,
      sales_amt NUMERIC,
     city    text
);
```

PostgreSQL Example:
2. Create the child tables:

```
CREATE TABLE daily_sales_p1
(
    PRIMARY KEY (id, sales_date),
     CHECK (city IN ('Minneapolis', 'Wellington')
)
    INHERITS (daily_sales_listp);

CREATE TABLE daily_sales_p2
(
    PRIMARY KEY (id, sales_date),
    CHECK (city IN ('Jakarta', 'Pune', 'Shanghai')
)
    INHERITS (daily_sales_listp);
```

PostgreSQL Example:
3. Create the indexes:

```
CREATE INDEX listp1_index ON
daily_sales_p1(city);

CREATE INDEX listp2 ON
daily_sales_p2(city);
```

PostgreSQL Example:
4. Create the trigger function:

```
CREATE OR REPLACE FUNCTION daily_sales_listp_insert()
RETURNS trigger AS $$
BEGIN
    IF (NEW.city IN ('Minneapolis', 'Wellington') ) THEN
    INSERT INTO
     daily_sales_p1 VALUES (NEW.*);
ELSEIF (NEW.city IN  ('Jakarta', 'Pune', 'Shanghai')(THEN
    INSERT INTO
    daily_sales_p1 VALUES (NEW.*);
    ELSE
      RAISE EXCEPTION 'city is not listed.';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

PostgreSQL Example:
5. Create the trigger to execute the trigger function:

```
CREATE TRIGGER daily_sales_listp_trigger
    BEFORE INSERT ON daily_sales_listp
    FOR EACH ROW
     EXECUTE PROCEDURE
daily_sales_listp_insert();
```

PostgreSQL Example:
6. Verify that the partition is linked:

```
\d+ daily_sales_listp
```

## 8.64 Stored Procedures - Key Differences

Unlike Oracle, PostgreSQL only supports creating functions there is no CREATE PROCEDURE command. In PostgreSQL, the purpose of a function is to process the input parameters and return a value. The function can be invoked from a SELECT statement, or another function. PostgreSQL allows function overloading. The same name can be used for different functions as long each function has distinct input argument types. A PostgreSQL function can be written in multiple languages, such as Python or R directly in the database.

An Oracle stored procedure can be converted to a PostgreSQL function. See detailed explanation further in this chapter.

PostgreSQL does not support packages. The database user should use schemas to group functions. There are no package-level variables. If needed, you can keep per-session state in temporary tables. Each PL/SQL object within the Package is converted to a PostgreSQL function.

For more information about creating functions, see Create Function.
For more about Oracle stored procedures, see Create Procedure Statement.

## 8.65 Procedures and Functions

PL/SQL is a procedural language developed by Oracle as an extension to SQL. The language allows sophisticated business rules and logic to be implemented at the database level without having to establish a separate connection. PL/SQL is used to create procedural objects, such as stored procedures, and functions. The procedural objects are stored in the database. The PL/SQL code can be tested and verified and then may be called from multiple applications. This feature insures the data is processed consistently.

PL/SQL is a block-structured language. A block is a unit of code of logically related declarations and statements. A block provides the execution and scoping boundaries for variable declarations and exception handling.  In Oracle, Procedures cannot be called directly from SELECT statements. Procedures are called with the EXEC keyword or from within another block.

The PostgreSQL variant of SQL is PL/pgSQL. PL/pgSQL is similar to PL/SQL in many ways. The PostgreSQL language is also imperative and block-structured. All variables must be declared. The assignments, conditionals, and loops are comparable. PostgreSQL only supports the creation of functions. A function in PostgreSQL can be created to replace an Oracle stored procedure.

The purpose of a stored procedure is to do the following:

- Can modify inputs and uses an IN, IN OUT, or OUT parameters.
- Perform an action without returning a result.
- Return one or more scalar values as OUT parameters.
- Return one or more result sets.

Conversely, the purpose of an Oracle Function is to perform a calculation and return a value using the RETURN keyword. A function does not modify its inputs and returns a single value. Functions can be invoked from a SELECT statement.

For more about PL/pgSQL, see PL/pgSQL - SQL Procedural Language.

## 8.66 Porting Oracle Stored Procedures and Functions to PostgreSQL Functions

This section describes the key differences between Oracle's implementation of a Stored Procedures and Functions, compared to PostgreSQL Functions.

## 8.67 Migrating an Oracle Stored Procedure to a PostgreSQL Function

Oracle Example:
A stored procedure:

```
CREATE OR REPLACE PROCEDURE DEACTIVATE_ACCT
(P_ACCOUNT_ID IN NUMBER )
IS
BEGIN
  UPDATE CUST_ACCT
  SET FLAG_DEACTIVATED=1
  WHERE ACCT_ID=P_ACCOUNT_ID;
  DBMS_OUTPUT.PUT_LINE('Account deactivated: '||P_ACCOUNT_ID||'.');

EXCEPTION
   WHEN OTHERS THEN
RAISE_APPLICATION_ERROR(-20001,'An error has occurred -   '||SQLCODE||' -ERROR- '||SQLERRM);
ROLLBACK;
COMMIT;
END;
/
-- Execute
SQL> EXEC DEACTIVATE_ACCT(3004);
```

PostgreSQL Example:
 The same procedure converted to a function:

```
CREATE OR REPLACE FUNCTION public.deactivate_acct(p_acct_id integer)
  RETURNS void AS
$$
       UPDATE "CUST_ACCT"
       SET "FLAG_DEACTIVATED"=1
       WHERE "ACCT_ID"=P_ACCT_ID;
$$
LANGUAGE SQL VOLITILE

psql> SELECT public.deactivate_acct(81);
```

The main differences are as follows:

- Not all Oracle data types are compatible with PostgreSQL, some data type substitution should occur. In the PostgreSQL code, you must replace the Oracle data type NUMBER with the PostgreSQL data type INTEGER.
- The RETURN keyword in the Oracle stored procedure prototype is changed to the RETURNS keyword in the PostgreSQL code.
- An Oracle stored procedure does not return a value. For a migrated stored procedure, use the PostgreSQL "RETURNS void" command in the header.
- The IS becomes AS in the PostgreSQL code.
- In PostgreSQL, the function body is considered to be a string literal.
- In PostgreSQL, you can't COMMIT inside a function, the function runs as a single transaction
- The use of the $$ (dollar-quoting). These symbols are used to replace single quotes almost anywhere in SQL scripts. Dollar-quoting symbols are a PL/pgSQL-specific substitutes for single quote marks to avoid issues within the function body. If single quote marks are used in the Function body, each mark must be preceded with an escape character. The dollar-quoting symbols are used after the AS keyword and after the END keyword.
- A LANGUAGE clause is added at the end because a function can be written in several programming languages, including PL/pgSQL.

## 8.68 Migrating an Oracle Function to a PostgreSQL Function

Oracle Example:
A function in Oracle:

```
CREATE OR REPLACE FUNCTION c_to_f(DEGREES NUMBER)
 RETURN NUMBER IS
F NUMBER;
BEGIN
        F:= (DEGREES * 9/5)+32;
   RETURN F;
END;
SQL> SELECT c_to_f(45) FROM DUAL ;
SQL> 59
```

PostgreSQL Example:
The same function converted:

```
CREATE OR REPLACE FUNCTION c_to_f(IN DEGS Numeric(10,2), OUT f Numeric(10,2))
AS $$
BEGIN
       f:=(DEGS * 9/5) + 32;
END; $$
LANGUAGE PLPGSQL;

SELECT public.c_to_f(15.0);

59.00
```

The main differences, along with the differences mentioned in the Migrating Procedures to Functions section are:

- The Oracle NUMBER data type has been changed to the PostgreSQL Numeric data type.
- In PostgreSQL, the FROM clause is not mandatory in a SELECT statement. The DUAL table can be created in PostgreSQL as a view to eliminate migration issues.

## 8.69 Dropping Procedures and Functions

To drop a procedure, use the DROP PROCEDURE command.  The database user must either own the procedure or have the DROP ANY PROCEDURE system privilege.

```
drop procedure deactivate_account;
```

To drop a function, use the DROP FUNCTION command. The database user must own the function or have the DROP ANY PROCEDURE system privilege.

```
drop function overdue_status;
```

In PostgreSQL, the DROP FUNCTION removes the existing function. The database user must be the owner of the function.

```
drop function addcustacct(varchar(25), varchar(60), integer)
```

The main difference is, for the PostgreSQL command, the argument types to the function must be listed. Different functions can have the same name as long as each function has a different argument list.

## 8.70 Handling Oracle Package and Package bodies

PostgreSQL does not have an equivalent to Packages. A package is a container or a namespace that organizes and groups related stored procedures and user-defined functions. Packages allow multiple procedures to use the same variables, constants, and cursors.

The following are the parts of a package:

- CREATE PACKAGE: Creates the specification for the package. A package specification lists the names and parameters for the procedures and functions contained in the package.
- CREATE PACKAGE BODY: Contains the code for its procedures and functions. The names, parameters, and return types must match the names, parameters, and return types listed in the specification.

Oracle Example:
A package specification:

```
CREATE OR REPLACE PACKAGE PCK_ACCT_MGMT
AS
PROCEDURE addCustAcct(P_DATE_JOINED CUST_ACCT.DATEJOINED%TYPE,
P_City CUST_ACCT.CITY%TYPE
P_PIN CUST_ACCT.PIN%TYPE);
PROCEDURE delCUSTACCT(P_ID CUST_ACCT.ACCT_ID%TYPE);
END PCK_ACCT_MGMT;
```

Oracle Example:
A package body:

```
CREATE OR REPLACE PACKAGE BODY PCK_ACCT_MGMT
AS
PROCEDURE addCustAcct (
(P_DATE_JOINED CUST_ACCT.DATE_JOINED%TYPE,
P_City  CUST_ACCT.CITY%TYPE,
P_PIN CUST_ACCT.PIN%TYPE)
IS
BEGIN
    INSERT INTO CUST_ACCT (
DATE_JOINED, CITY, PIN)
VALUES (P_DATEJOINED, P_CITY, P_PIN);
END;

PROCEDURE delCUST_ACCT (P_id  CUST_ACCT.ACCT_ID%TYPE)
IS
BEGIN
  DELETE FROM CUST_ACCT
  WHERE ACCT_ID = P_ID
END;
END PCK_ACCT_MGMT;

DECLARE
code CUST_ACCT.ACCT_ID%TYPE := 1050;
BEGIN
   pck_acct_mgmt.addCustAcct('07/21/2018', 'Belgrade','4434');
   pck_acct_mgmt.delCustAcct(code);
END;
```

PostgreSQL Example:
The following example shows the procedure converted to a function:

```
CREATE OR REPLACE FUNCTION ADDCUSTOMERACCT(P_DATE_JOINED varchar(255), P_CITY varchar(255),
P_PIN INTEGER)
RETURNS VOID
AS
$$
      INSERT INTO "CUST_ACCT" (
        "DATE_JOINED"
        ,"CITY"
        ,"PIN"
      )VALUES (P_DATE_JOINED, P_CITY, P_PIN);
 $$
Language SQL;
```

PostgreSQL Example:
The example below displays the procedure converted to a function:

```
CREATE OR REPLACE FUNCTION DELCUSTACCT(P_ID INTEGER)
RETURNS VOID
AS
$$
      DELETE FROM "CUST_ACCT"
        WHERE "ACCT_ID" = P_ID;
```

```
 $$
Language SQL;
```

The key difference is that PostgreSQL does not support Packages. The database user should use schemas to group functions. There are no package-level variables. Each PL/SQL object within the Package is converted to a PostgreSQL function.

## 8.71 Privileges Required

The privileges needed for procedures and functions cannot be granted via roles. The privileges must be granted directly to the owner of the procedure or function. The necessary privileges are as follows:

- In your schema, to create a procedural object you must have CREATE PROCEDURE.
- In another user's schema, you must have CREATE ANY PROCEDURE.
- You must GRANT another user the EXECUTE privilege on that object.

In PostgreSQL, the database user who creates a function is the owner of the function. The database user must have USAGE privileges on the argument types, the return type, and the language. The database user must own the function to use ALTER FUNCTION and have CREATE privilege on the schema.

## 8.72 Comparison table

| Description | Oracle | PostgreSQL |
|---|---|---|
| Stored Procedures | Available | Not supported. Must be converted to PostgreSQL function with RETURNS void |
| Functions | Available | Available. Can be written in multiple languages. |
| Packages | Available | Not supported. Use schemas to group functions. Each object within the package is converted to a PostgreSQL function. |
| DROP | Available | The argument types must be listed because function names can be overloaded. |

## 8.73 Users and Roles - Key Differences

Oracle allows user accounts and has wider security options, comparing to PostgreSQL.
PostgreSQL represents user accounts as roles. A login role has database access permissions. A group role is a collection of privileges and typically does not have login access. A schema is created as a separate object in the database.

## Common DBA tasks differences:

**Display all roles**

Oracle Example:

```
SELECT * FROM dba_roles;
```

PostgreSQL Example:

```
SELECT * FROM pg_roles;
```

**List all users**

Oracle Example:

```
SELECT * FROM dba_users;
```

PostgreSQL Example:

```
SELECT * FROM pg_user;
```

**Create a role**

Oracle Example:

```
CREATE ROLE c##myrole1;

Or

CREATE ROLE local_myrole1;
```

PostgreSQL Example:

```
CREATE ROLE qa_role1;
```

**Create a user or a login role**

Oracle Example:

```
CREATE USER c##user1 IDENTIFIED BY user1;
```

PostgreSQL Example:

```
CREATE ROLE qa_role1 WITH LOGIN PASSWORD 'welcome1';
```

**Change a user password**

Oracle Example:

```
ALTER USER c##user1 IDENTIFIED BY welcome1;
```

PostgreSQL Example:

```
ALTER ROLE qa_role1 WITH LOGIN PASSWORD 'welcome';
```

**Lock an account or a set NOLOGIN for a role**

Oracle Example:

```
ALTER USER c##user1 ACCOUNT LOCK;
```

PostgreSQL Example:

```
ALTER ROLE qa_group1 WITH NOLOGIN;
```

**Unlock an account or add login to a role**

Oracle Example:

```
ALTER USER c##user1 ACCOUNT UNLOCK;
```

PostgreSQL Example:

```
ALTER ROLE qa_role2 WITH LOGIN;
```

**Set tablespace quotas**

Oracle Example:

```
Alter User c##myrole1 QUOTA UNLIMITED ON TABLESPACE users;
```

Tablespace quotas are supported in PostgreSQL.

**Limit user sessions or connections**

Oracle Example:

```
CREATE PROFILE app_myusers LIMIT SESSIONS_PER_USER 15;
 ALTER USER c##user1 PROFILE app_myusers;
```

PostgreSQL Example:

```
ALTER ROLE qa_user1 WITH CONNECTION LIMIT 5;
```

**Assign a user to a role**

Oracle Example:

```
GRANT myrole1 TO c##user1;
```

PostgreSQL Example:

```
GRANT qa_group1 TO qa_role1;
```

**Assign a role's privileges to another role**

Oracle Example:

```
GRANT local_myrole1 TO local_myrole2;
```

PostgreSQL Example:

```
GRANT qa_role1 to qa_role2;
```

**Assign system privileges**

Oracle Example:

```
GRANT CREATE TABLE TO local_role;
```

PostgreSQL Example:

```
GRANT CREATE ON DATABASE postgresdb to qa_role1;
```

**Assign object privileges**

Oracle Example:

```
GRANT INSERT on orders to myrole1;
```

PostgreSQL Example:

```
GRANT INSERT, DELETE ON Samples.warehouse to qa_role1;
```

**Assign EXECUTE privileges**

Oracle Example:

```
GRANT EXECUTE ON Finance.check_results to c##user1;

GRANT EXECUTE on function "newdate"(arguments) to qa_user1;
```

**Create a schema**

Oracle Example:

```
CREATE USER myuser_schema IDENTIFIED BY welcome1;
```

PostgreSQL Example:

```
CREATE SCHEMA Samples;
```

**External authentication**

Oracle Example:

```
CREATE USER OPS$$User1 IDENTIFIED EXTERNALLY;
```

PostgreSQL supports integration with LDAP authentication and Kerberos authentication, see Authentication Methods.

## 8.74 Users & Roles

Oracle allows user accounts, which can have privileges and many more security options.
Privileges that are grouped together and granted to database users are Oracle roles. Individual system and object permissions as well as other roles can be granted to a role. In one operation, you can grant multiple privileges.

Oracle 12c supports the creation of both common roles and local roles in a multi-tenant database architecture:

- Common Roles: Created in the root container (CDB) and exist in all containers. The role may have a different collection of privileges in each container. The role can be granted to either other roles or common or local users.
- Local Roles: A role created in a specific pluggable database (PDB) and exists only in that PDB. The roles can be granted only locally to either other roles or common users or local users.
- User-created Common role names must begin with c## or C## prefix. In Oracle 12.1.0.2 and afterward, the SYSDBA can alter the prefix using the COMMON_USER_PREFIX command.
- The CREATE ROLE statement has a CONTAINER clause. The common role is created with "container=all" and the local role is created with a specific container.

PostgreSQL represents user accounts as roles. Recent versions of PostgreSQL avoid using the terms users and groups. The terms are now mapped to login roles and group roles. For compatibility reasons, CREATE USER and CREATE GROUP will still work, but the documentation recommends using CREATE ROLE. For more information, see Create Role.
PostgreSQL manages database access permissions using roles. A database-level role has database-wide scope similar to an Oracle common role. A role can be considered either a database user or a group of users. A login role can access the database and could be considered a user. A group role does not have login privileges. Roles defined at the database cluster level are valid in each database.

Schemas in PostgreSQL are separate from users. A schema is similar to a namespace. The schema organizes and identifies information into logical groups. Schema names must be unique in the database and serve to avoid name collision.

**Note:** The PostgreSQL CREATE USER command is an alias for the CREATE ROLE command. If you create a role with CREATE USER, the role has the default ability to log into the database. The CREATE ROLE command has a NOLOGIN default value, but you can change the value to enable log in.

Important differences:

- Common user (12c) and the database user (11g) are equal to a PostgreSQLdatabase role with login ability.
- Local user (12c) does not have an equivalent role in PostgreSQL.

- Database role is equal to the PostgreSQLdatabase role without a login ability.
- Database user name is identical to the schema name in Oracle. In PostgreSQL, the schema is created as a separate object.

For more information about security options, see Configuring Privilege and Role Authorization.

Oracle Example:

Create a common role:

```
CONN / AS SYSDBA

CREATE ROLE c##role1;
GRANT c##role1 to c##user1 container=all;
```

Oracle Example:

Create a local role, assign a user to a local role, grant and revoke privileges to a role:

```
CONN / AS SYSDBA

ALTER SESSION SET CONTAINER = pdbAlpha;
CREATE ROLE local_test1;

Grant local_test1 to c##user2;
GRANT CREATE SESSION to local_test1;
REVOKE CREATE SESSION from local_test1;
```

PostgreSQL Example:

Create a role and grant the ability to create other roles:

```
CREATE ROLE test_role;
 ALTER ROLE test_role CREATEROLE;
```

PostgreSQL Example:

Create a login role and assign a password and specifies when the role should expire and lose any granted privileges:

```
CREATE ROLE whse_user  LOGIN PASSWORD 'Welcome1' CREATEDB VALID UNTIL '2022-11-1';
```

PostgreSQL Example:

Create a schema Samples and create a table:

```
CREATE SCHEMA Samples AUTHORIZATION postgres;

 CREATE TABLE Samples.warehouse (warehouse_id integer NOT NULL, warehouse_name text NOT NULL,
CONSTRAINT "PRIM_KEY" PRIMARY KEY(warehouse_id));
```

## 8.75 Large Objects - Key Differences

Oracle supports four LOB data types. You can store LOBs in SecureFIles when you create a table. SecureFiles also provide encryption, deduplication, and compression options.

In PostgreSQL, all large objects are stored in a single system table, named pb_largeobject. A large object can be created, modified, and deleted using a read/write API. The API operations are similar to standard read/write operations.

PostgreSQL also supports a storage system, called "TOAST". This system automatically stores values larger than a single database page into a secondary storage area for each table. The limitations to TOAST are that the TOAST fields can be, at most, 1 GB and read or write operations treat the large object as one unit.

PostgreSQL does not support the SecureFiles options.

## 8.76 Large Objects (LOBs)

Oracle supports four LOB types. The character data types are the following:

- CLOB: Character Large Object
- NCLOB: National Character Large Object

These data types are commonly used to store text or XML files. The CLOB type uses one byte per character and uses the ASCII character set. The NCLOB uses two or three bytes per character to store characters in the Unicode character set.

The following data types are also LOB types:

- BLOB: Binary Large Object
- BFILE: Binary File

These data types store data in a binary format. The BLOB type can store binary files, such as a PDF file, along with image, sound, and video files. The BFILE type stores a pointer to a binary file that is outside of the database. The binary file must be accessible through the host computer file system.

Oracle has SecureFiles, a LOB storage architecture. You can create a SecureFiles LOB with the SECUREFILE keyword in the CREATE TABLE statement. BasicFiles LOBs is the default storage and would be used if SECUREFILE is not defined.

Oracle Example:
Create a table and store the table as a Securefile:

```
CREATE TABLE sample_table (id NUMBER, Large_object CLOB) LOB(Large_Object) STORE AS SECUREFILE;
```

Oracle recommends you should enable compression, deduplication or encryption through the CREATE TABLE statement.

SecureFIles provides the following additional features:

- Compression: Seamlessly analyzes LOB data and compresses to save space
- Deduplication: Enables Oracle to detect automatically duplicate LOB data within a LOB column or partition. This process conserves space by storing only one copy.
- Encryption: Data is encrypted using Transparent Data Encryption (TDE). This encryption allows the data to be securely stored and allows for read and write access.

If you decide to enable these features later, through an ALTER TABLE, all SecureFiles LOB data is read, modified, and written. These events cause the database to lock the table and could be potentially a lengthy process.

Oracle Example:

Create a table, store the table as a Securefile, and compress the table:

```
CREATE TABLE sample_table (id NUMBER,COL2_CLOB CLOB) LOB(Large_Object) STORE AS SECUREFILE
COMPRESS_LOB(COMPRESS HIGH);
```

In PostgreSQL, all large objects are stored in a system table named pg_largeobject. Each large object also has an entry in the pg_largeobject_metadata system table. PostgreSQL uses a read/write API to create, modify or delete large objects.

PostgreSQL also supports a TOAST (The Oversized-Attribute Storage Technique) storage system. This system automatically stores values larger than a single database page into a secondary storage area for each table. TOAST has the following limitations:

- Fields are limited to 1 GB.
- Read and write operations use the whole value, which can be slower.

The large object facility allows up to 4 TB in size and can read and write to portions of a large object.

In PostgreSQL, there are two ways to store a Binary Large Object.

The BYTEA data type supports two formats for input and output: "hex" and an older "escape" format. Either of these formats are accepted on input. The output format depends on the bytea_output configuration parameter. The default is hex.

The BYTEA data type is appropriate for storing raw data. The differences between BYTEA and character strings are as follows:

- Binary string is similar to character strings with the following characteristics:
- Allows storing octets of value zero and non-printable octets.
- Any operations process the actual bytes.

- Can be used to store a URL reference

The TEXT type stores strings of any length.

- Data type for storing strings with unlimited length.
- Longer strings have four bytes overhead and are compressed by the system automatically.
- Very long strings are stored in background tables. This storage avoids interference with access to shorter column values.
- The longest possible character string is about 1 GB.
- If you do not specify a length to the character or varchar definition the type accepts a string of any size.

## 8.77 SQL*Loader - Key Differences

The Oracle SQL*Loader utility provides a way to load data from another data source into Oracle. The utility only reads flat files. If you are loading data from another database, you must export the data to a flat file before loading the files. The flat files can be delimited or fixed-length.

SQL*Loader has dozens of options and should be used for high performance data loads. The utility reads a data file and a description of the data. This description is contained in a control file. If needed, and if the data meets certain criteria, you can use SQL*Loader express mode, which does not require a control file.

During processing, SQL*Loader writes messages to a log file, any bad rows to a bad file, and if a row fails a specified criteria, the row is written to a discard file.

There is no direct equivalent for SQL*Loader in PostgreSQL. To load data to your database you should import data, and there are two main options to do that:

1. Use the pg_restore utility
2. Use the gcloud utility or the Google Cloud Console to load data from Google Cloud Storage. You can also load CSV files with this option.

For additional details, review the Export/Import topic.

## 8.78 Table Statistics - Key Differences

Oracle table statistics are collected by the DBMS_STATS package. In PostgreSQL, the statistics are stored in pg_class and pg_statistics.

PostgreSQL has an autovacuum feature that combines the VACUUM and ANALYZE functions.

## 8.79 Table Statistics

In Oracle, table statistics can affect the optimization of a query and the execution plan. The query optimizer chooses an execution plan based on the table statistics. Common sources of poor execution plans include: missing indexes, missing statistics on tables, columns, and indexes involved in the query. In Oracle, the DBMS_STATS package manages and controls the table statistics. Table statistics can be collected manually or automatically. If the execution plan is poor, the first task is to regenerate statistics for every table and index relevant to the query.

The following statistics can be collected:

- Number of table rows
- Number of table blocks
- Average row length
- Sample size
- Last analyzed
- Last analyzed since

In PostgreSQL, table statistics are stored in two places: pg_class and pg_statistic. The pg_class system catalog contains one row for each table and information about views, indexes, disc blocks, and sequences. The pg_statistic system catalog stores statistics about each column, such as the percentage of null values, most common values, and histogram bounds.

When a table is created the pg_class estimates the size of the table using default values. As you insert and delete, the pg_class estimates are not updated. You must use the following three commands to update the pg_class estimates:

- Vacuum
- Analyze
- Create Index

PostgreSQL database require periodic maintenance known as vacuuming. PostgreSQL has an optional but recommended feature called autovacuum. The purpose of the autovacuum is to automate the execution of the VACUUM and the ANALYZE commands. When this feature is enabled, the autovacuum checks for tables with a large number of inserts, updates, and deletes. The checks use the statistic collection facility. Autovacuum cannot be used unless track_counts is set to true.

The ANALYZE command collects statistics for the following:

- Database
- Table
- Column

The command stores the results in the pg_statistics system catalog. You must make sure to run ANALYZE frequently, preferably using autovacuum. For large tables, ANALYZE takes a random table sample. The sample allows large tables to be analyzed in less time.

The default_statistics_target variable, in postgresql.conf, is 100. Increasing this variable may provide a more accurate planner estimate. The price is ANALYZE may take more time to complete and consume more space. Oracle, using the database scheduler and automated maintenance tasks, collects table and index statistics during predefined maintenance windows. Oracle's data modification monitoring feature tracks the approximate number of INSERTs, UPDATEs, and DELETEs to determine which table statistics should be collected.

For information about table statistics, see Managing Optimizer Statistics: Basic Topics.

## 8.80 Manual Optimizer Statistics Collection

When the automatic statistics collection is not suitable for a particular use case, the optimizer statistics collection can be performed manually at several levels. To generate statistics, use the dbms_stats package. The package contains the following procedures:

- GATHER_DATABASE_STATS procedures:  Collects statistics for all database objects.
- GATHER_DICTIONARY_STATS procedures: Collects statistics for the Fixed (in-memory) and Real (normal) dictionary tables.
- GATHER_FIXED_OBJECTS_STATS procedures: Collects statistics for all fixed objects (dynamic performance tables).
- GATHER_INDEX_STATS procedures: Collects index statistics.
- GATHER_SCHEMA_STATS procedures: Collects statistics for all objects in a schema.
- GATHER_SYSTEM_STATS procedures: Collects system statistics.
- GATHER_TABLE_STATS procedures: Collects table and column (and index) statistics.

Oracle Example:
Table-level statistics collection can be performed using the example below:

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('Finance','Invoices');
END;
/

PL/SQL procedure successfully completed.
```

Oracle Example:
Column-level statistics collection can be performed using the following example:

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('Finance','Invoices',
        METHOD_OPT=>'FOR COLUMNS Invoice_total');
END;
/

PL/SQL procedure successfully completed.
```

PostgreSQL Example:
Collect statistics for a database.

```
ANALYZE;
```

PostgreSQL Example:
To collect statistics for a table with VERBOSE, use the following example:

```
ANALYZE VERBOSE Warehouse;
```

PostgreSQL Example:
To collect statistics for a column, use the example below:

```
ANALYZE Invoices(Invoice_creation_date);
```

PostgreSQL Example:
To define  a default_statistics_target parameter for a column and reset the parameter, use the following example:

```
ALTER TABLE Invoices ALTER COLUMN Total_Invoices SET STATISTICS 150;

 ALTER TABLE Invoices ALTER COLUMN Total_Invoices SET STATISTICS -1;
```

PostgreSQL Example:
To examine  the default_statistics_target parameter, modify the parameter to 150 and analyze the Invoices table like the example below:

```
SHOW default_statistics_target ;
 SET default_statistics_target to 150;
 ANALYZE Invoices;
```

PostgreSQL Example:
To run  a command to examine the last time statistics were collected for a table, use the following example:

```
SELECT relname, last_analyze FROM pg_stat_all_tables;
```

## 8.81 Table Constraints - Key Differences

Constraints allow a DBA to enforce data integrity across the database.
Oracle and PostgreSQL use the same syntax for most of the constraint creation. However, there are some differences:

- Oracle has the REF constraint which is not supported in PostgreSQL.
- Oracle views support primary key, foreign key, and unique constraints, while PostgreSQL not.
- When you delete a foreign key in Oracle, the ON DELETE clause allows:

- ON UPDATE CASCADE
- ON UPDATE RESTRICT

- PostgreSQL has the EXCLUSION constraint which is not supported in Oracle.
- PostgreSQL views do not support constraints.
- In PostgreSQL, the ON DELETE clause and the ON UPDATE clause have the same options:
    - ON DELETE | ON UPDATE CASCADE
    - ON DELETE | ON UPDATE RESTRICT
    - ON DELETE | UPDATE NO ACTION

## 8.82 Comparison table

| Oracle Constraint options | Cloud SQL for PostgreSQL Constraint options |
|---|---|
| PRIMARY KEY | PRIMARY KEY |
| FOREIGN KEY | FOREIGN KEY |
| UNIQUE | UNIQUE |
| CHECK | CHECK |
| NOT NULL | NOT NULL |
| REF | Not Supported |
| DEFERRABLE | DEFERRABLE |
| NOT DEFERRABLE | NOT DEFERRABLE |
| SET CONSTRAINTS | SET CONSTRAINTS |
| INITIALLY IMMEDIATE | INITIALLY IMMEDIATE |
| INITIALLY DEFERRED | INITIALLY DEFERRED |
| ENABLE | Default, not supported as keyword |
| DISBALE | Not supported as keyword, NOT VALID can use instead |
| ENABLE VALIDATE | Default, not supported as keyword |
| ENABLE NOVALIDATE | NOT VALID |
| DISABLE VALIDATE | Not supported. |
| DISABLE NOVALIDATE | Not supported. |
| USING_INDEX_CLAUSE | table_constraint_using_index |
| View Constraints | Not supported. |
| Metadata: DBA_CONSTRAINTS | Metadata: PG_CONSTRAINT |

## 8.83 Table Constraints

Oracle Table Constraints is a feature that lets the DBA enforce data integrity across the database, using CREATE TABLE or ALTER TABLE you can add many types of constraint to check the data or to link a specific value to be verified with parent table.

This topic will compare the Oracle Table Constraints with PostgreSQL Table Constraints. PostgreSQL supports the following types of table constraints: primary key, foreign key, check unique, not null, exclusion.

**Note:** There is no Oracle REF constraint equivalent in PostgreSQL

**Note: EXCLUSION** constraints - ensures if any two rows are compared on the specified columns or expressions using the specified operators that at least one comparison return false or NULL.

PostgreSQL allows constraints to be created in-line or out-of-line. You can define a constraint with either CREATE / ALTER TABLE.

For more information about Oracle Table Constraints, see Constraint.

## 8.84 Oracle Integrity Constraint Types:

- Primary Key: All values are UNIQUE and NOT NULL.
- Foreign Key: All values exist in the referenced table.
- Unique: No duplicate values are allowed.
- Check: Run condition check must return TRUE.
- Not Null: No NULL values are allowed.
- REF: References between objects.

## 8.85 Constraint Creation

Both database systems support creating constraints in two ways:

- Inline or column-level: Defined when the column is defined. A NOT NULL constraint is only available at the column-level.

```
CREATE TABLE EMPLOYEES (EMP_ID NUMBER PRIMARY KEY,…);
```

- Out-Of-Line or table-level: Defined at the table level

```
CREATE TABLE EMPLOYEES (EMP_ID NUMBER,…,CONSTRAINT PK_EMP_ID PRIMARY KEY(EMP_ID));
```

Constraints can be specified with the following syntax:

- CREATE / ALTER TABLE
- CREATE / ALTER VIEW

**Note:** An Oracle view support primary key, foreign key, and unique constraints. PostgreSQL does not support view constraints.

## 8.86 PRIMARY KEY Constraints

PostgreSQL uses the same ANSI SQL syntax as Oracle. The main difference is the data types defined for the columns are different. The primary key constraint uniquely identifies each row in a database table. You can have only one primary key. The primary key cannot contain NULL values. A table can have many columns with unique constraints. At the column-level, you declare the PRIMARY KEY keyword only. When you create the constraint at the table-level, you can specify one column or a combination of columns. If you do not assign a name to a primary key constraint, PostgreSQL assigns a default name, such as [table-name]_pkey.

When you add a primary key, PostgreSQL creates a unique B-Tree index on the selected column or columns. If you decide to drop a primary key constraint, the index generated by the system is dropped but a user-defined index is not dropped.

- In Oracle, primary keys cannot be created on columns defined with the following data types: LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, BFILE, REF, TIMESTAMP WITH TIME ZONE.
- Primary keys can enforce one or more columns. A multiple column primary key is also called a composite key and are limited to 32 columns.
- A column can't use both PRIMARY KEY and UNIQUE constraints
- Defining the same column as both a primary key and as a unique constraint is not allowed.
- Uniquely identifies each record and cannot contain a NULL value.
- PostgreSQL generates constraint names automatically, or you can specify a name during creation.
- In PostgreSQL the names of primary key and unique key constraints must be unique within the schema.

Oracle Example:
Create a column-level primary key. The primary key will have a system-generated primary key name, like the example below:

```
CREATE TABLE BOOKSTORE (
        BOOKSTORE_ID NUMBER PRIMARY KEY,
       TITLE   VARCHAR2(100),
        AUTHOR   VARCHAR2(200),
        EMAIL       VARCHAR2(100));
```

Oracle Example:
To create an inline primary key using a user-specified primary key constraint name, use the following example:

```
CREATE TABLE BOOKSTORE(
       BOOKSTORE_ID NUMBER CONSTRAINT PK_BKSTRE_ID PRIMARY KEY,
        TITLE VARCHAR2(100),
        AUTHOR VARCHAR2(200),
        EMAIL VARCHAR2(100));
```

Oracle Example:
To create an out-of-line primary key, use the example below:

```
CREATE TABLE BOOKSTORE(
      BOOKSTORE_ID NUMBER,
      TITLE  VARCHAR2(100),
      AUTHOR   VARCHAR2(200),
      EMAIL       VARCHAR2(100));
      CONSTRAINT PK_BKSTRE_ID PRIMARY KEY (BOOKSTORE_ID));
```

Oracle Example:
It is rare to add a primary key to an existing table. You must use the ALTER TABLE statement, such as the example below:

```
ALTER TABLE WAREHOUSE_EVENTS
    ADD CONSTRAINT PK_WAREHOUSE_EVENTS_ID PRIMARY KEY (EVENT_CODE, WAREHOUSE_ID);
```

PostgreSQL uses the same ANSI SQL syntax to create primary keys. The main difference is the data types defined for the columns.
PostgreSQL Example:
To create an inline primary key constraint with a system-generated constraint name, use the following example:

```
CREATE TABLE CONTACT_INFO(
      ID SERIAL PRIMARY KEY,
      FIRST_NAME  VARCHAR(50),
      LAST_NAME   VARCHAR(50),
      EMAIL       VARCHAR(100));
```

PostgreSQL Example:
To define a name when creating an inline primary key constraint, use the example below:

```
CREATE TABLE CONTACT_INFO (
      ID SERIAL CONSTRAINT PK_CNTINFO_ID PRIMARY KEY,
      FIRST_NAME  VARCHAR(50),
      LAST_NAME   VARCHAR(50),
      EMAIL       VARCHAR(100));
```

PostgreSQL Example:
At the table-level, create a primary key constraint, using the following example:

```
CREATE TABLE CONTACT_INFO(
      ID SERIAL NOT NULL,
      FIRST_NAME  VARCHAR(50),
      LAST_NAME   VARCHAR(50),
      EMAIL       VARCHAR(100));
      CONSTRAINT PK_CNTINFO_pkey PRIMARY KEY (ID));
```

PostgreSQL Example:
For an existing table, add a primary key constraint, use the example below:

```
ALTER TABLE SYSTEM_EVENTS
            ADD COLUMN PK_EMP_ID PRIMARY KEY (EVENT_CODE, EVENT_TIME);

--Add an automatically named primary key
ALTER TABLE vendors ADD PRIMARY KEY (vendor_id);

--Recreate a primary key constraint without blocking updates while the index is built
CREATE UNIQUE INDEX CONCURRENTLY vendors_id_pk_idx ON VENDORS(vendors_id);
ALTER TABLE vendors DROP CONSTRAINT vendors_pkey,
    ADD CONSTRAINT vendors_pkey PRIMARY KEY USING INDEX vendors_id_idx;
```

PostgreSQL Example:
Use the following example to drop a primary key:

```
ALTER TABLE products DROP CONSTRAINT products_pkey;
```

## 8.87 FOREIGN KEY Constraint

A foreign key is a column or group of columns that uniquely identifies a row in another table, which is, typically, the table's primary key. The table containing the foreign key is called the referencing table or child table. The table the foreign key references is called the referenced or parent table. A table can have multiple foreign keys.

In PostgreSQL, you define a foreign key through the foreign key constraint. A foreign key maintains the referential integrity between the tables.

- In PostgreSQL, creating a FOREIGN KEY constraint is done with the standard ANSI SQL syntax.
- Foreign keys can be created at the column level or at the table-level when creating the table.
- The REFERENCES clause specifies the table referenced by the constraint.
- When defining the REFERENCES, if there is no column list in the referenced table, the primary key is used.
- Key names are either generated by the database or defined during constraint creation.
- You can add a foreign key to an existing table with an ALTER TABLE statement.
- In cases of parent record deletions, use the ON DELETE clause.

## 8.88 Limitations

- The primary key or unique constraints on a parent table must be created before the foreign key can reference the columns.
- A CREATE TABLE statement with a subquery clause cannot have Foreign key constraints.
- Foreign keys cannot be created on columns defined with the following data types: LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, BFILE, REF, TIMESTAMP WITH TIME ZONE. In PostgreSQL, it is also good practice not to use a BOOLEAN column.
- Composite Foreign key constraints, a foreign key comprised of multiple columns, cannot have more than 32 columns in the definition.

## 8.89 ON DELETE Clause

The effect of deleting values from a parent table on the referenced records of a child table is handled in the ON DELETE clause. In Oracle, if the ON DELETE clause is not specified, you are unable to delete referenced values in a parent table that have dependent rows in the child table.

- ON DELETE CASCADE: The referenced values from the parent table and the dependent foreign key values in a child table are removed.
- ON DELETE RESTRICT: PostgreSQL option - Delete the child table rows before the parent table.
- ON DELETE NO ACTION: In PostgreSQL, this is the default action. If any referencing rows still exist when the constraint is checked, an error is raised.
- ON DELETE NULL: The dependent foreign key values in a child table are updated to null.

Only in PostgreSQL, there is also ON UPDATE clause option and it has the same options as the ON DELETE clause:

- ON UPDATE CASCADE
- ON UPDATE RESTRICT
- ON UPDATE NO ACTION

Oracle and PostgreSQL use the SQL ANSI standard code to define a foreign key.
Oracle Example:
To create a column-level foreign key, use the following example:

```
CREATE TABLE BOOKSTORES(
      BOOKSTORE_ID   NUMBER PRIMARY KEY,
      TITLE    VARCHAR2(100),
      AUTHOR     VARCHAR2(200),
      EMAIL        VARCHAR2(100) ,
     LOCATION_ID REFERENCES LOCATIONS(LOCATION_ID));
```

Oracle Example:
To create a table-level foreign key, use the example below:

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID   NUMBER PRIMARY KEY,
    FIRST_NAME    VARCHAR2(50),
    LAST_NAME     VARCHAR2(50),
    EMAIL         VARCHAR2(100),
    BOOKSTORE_ID NUMBER,
    CONSTRAINT FK_BOOKSTORE_EMP_ID
    FOREIGN KEY(BOOKSTORE_ID) REFERENCES BOOKSTORES(BOOKSTORE_ID));
```

Oracle Example:
To create a foreign key with ON DELETE CASCADE, use the example below:

```
CREATE TABLE EMPLOYEES (
     EMPLOYEE_ID   NUMBER PRIMARY KEY,
     FIRST_NAME    VARCHAR2(500),
     LAST_NAME     VARCHAR2(50),
     EMAIL         VARCHAR2(100),
     BOOKSTORE_ID NUMBER,
    CONSTRAINT FK_BOOKSTORE_ID
    FOREIGN KEY(BOOKSTORE_ID) REFERENCES BOOKSTORES(BOOKSTORE_ID)
    ON DELETE CASCADE);
```

Oracle Example:
Use the example below to add a foreign key to an existing table:

```
ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_LOCATION_ID
     FOREIGN KEY(LOCATION_ID) REFERENCES LOCATIONS(LOCATION_ID);
```

PostgreSQL Example: Foreign key columns must have a specified data type while Oracle does not. In PostgreSQL, create a column-level foreign key.

Notice the REFERENCES clause to define a foreign key constraint to the Invoices table:

```
CREATE TABLE INVOICE_ITEMS (
        ITEM_ID   INTEGER PRIMARY KEY,
        INVOICE_ID    INTEGER REFERENCES INVOICES(INVOICE_ID),
        PRODUCT_ID     INTEGER,
        QTY       INTEGER,
        NET_PRICE NUMERIC);
```

PostgreSQL Example:
Another way to define a foreign key constraint is to add the constraint at the table-level:

```
CREATE TABLE INVOICE_ITEMS(
        ITEM_ID   INTEGER,
        INVOICE_ID   INTEGER,
        PRODUCT_ID     INTEGER,
        QTY INTEGER,
        NET_PRICE NUMERIC,
        PRIMARY KEY (item_id, invoice_id),
        FOREIGN KEY (invoice_id) REFERENCES INVOICES(invoice_id));
```

PostgreSQL Example:
Each line item of the invoice must belong to a specific invoice. If a row in the Invoices table is deleted, what should happen to the rows in invoice_items that reference the row? PostgreSQL allows
DELETE RESTRICT, DELETE CASCADE and NO ACTION:

```
CREATE TABLE INVOICE_ITEMS (
        ITEM_ID INTEGER NOT NULL,
        INVOICE_ID INTEGER REFERENCES INVOICES(invoice_id) ON DELETE RESTRICT,
        PRODUCT_ID INTEGER REFERENCES PRODUCTS(product_id) ON DELETE CASCADE,
        QTY INTEGER,
        NET_PRICE NUMERIC,
        PRIMARY KEY (item_id, invoice_id));
```

PostgreSQL Example:
To add a foreign key to an existing table, use the following example:

```
ALTER TABLE INVOICE_ITEMS  ADD CONSTRAINT FK_EMP_ID
          FOREIGN KEY(EMPLOYEE_ID) REFERENCES EMPLOYEES(EMPLOYEE_ID);
```

## 8.90 UNIQUE Constraints

A unique constraint specifies that the values in a column, or multiple columns, must be unique. You can create unique constraints either in the CREATE TABLE or ALTER TABLE statements. The unique constraint automatically creates a B-Tree index to manage the constraint. A unique constraint is considered a table-level constraint.
A unique constraint can contain null values.
The Oracle and PostgreSQL syntax to create a unique constraint is identical. The following is true of unique constraints:

- Guarantees a value in a column, or multiple columns, is unique.
- UNIQUE constraint fails if duplicate values exist in the column(s) and returns an error message.
- In PostgreSQL, UNIQUE constraints allow NULL values.
- Name can be system-generated or explicitly defined.

## 8.91 Limitations

- Cannot be created with the following data types: LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, BFILE, REF, TIMESTAMP WITH TIME ZONE.
- Are limited to 32 columns.

- Since a primary key and a unique constraint are essentially the same, you cannot create both on the same column or columns.

Oracle Example:
To create a unique constraint for a group of columns, use the following example:

```
CREATE TABLE BOOKSTORES(
      BOOKSTORE_ID   NUMBER PRIMARY KEY,
      TITLE    VARCHAR2(100),
      AUTHOR     VARCHAR2(200),
      EMAIL        VARCHAR2(100),
      CONSTRAINT u_TITLE_AUTHORS UNIQUE(TITLE, AUTHOR));
```

PostgreSQL Example:
To create a unique constraint for one column, use the example below:

```
CREATE TABLE PRODUCTS(
       PRODUCT_ID   INTEGER PRIMARY KEY,
       PRODUCT_NAME    TEXT UNIQUE,
       DESCRIPTION     TEXT,
       isAVAILABLE BOOLEAN);
```

## 8.92 CHECK Constraint

Check constraints are Boolean expressions. The check constraint validate that specific column values meet certain criteria or conditions. If the values do not satisfy the Boolean expressions the values are not added to the table. Check constraints are column-level constraints. A check constraint can help transfer the application-level logical integrity validation to the database. You can create boundary expressions and check if a value is greater than, less than or equal to another value. If someone tries to update a value that is does not meet the criteria the update fails with an error message. PostgreSQL allows inherited tables. A table designated as a child table may have its own columns and all columns of the parent tables. Any structural changes made to a parent table are also propagated to the child automatically. Check constraints are inherited.

- CHECK constraints guarantee column values satisfy a defined requirement.
- Are only defined with a Boolean data type.
- Names can be system-generated or explicitly defined.

### In-Line vs. Out-Of-Line

Creating a check constraint at the column-level (in-line) defines the constraint for that column. Creating the constraint at the table-level (out-of-line) allows you to define the constraint for multiple columns.

## Limitations

The following limitations apply to all CHECK constraints:

- Cannot perform validation on other tables or columns in another table.
- Can be defined only with functions that are deterministic.
- Cannot be defined with user-defined functions.
- Cannot be defined with pseudo columns such as: CURRVAL or ROWNUM.

Oracle Example:

To create a table with a column-level check constraint, use the example below:

```
CREATE TABLE INVOICES(
      INVOICE_ID   NUMBER PRIMARY KEY,
      INVOICE_TOTAL   NUMBER(9,2) NOT NULL CHECK (invoice_total >= 0),
      INVOICE_DATE DATE);
```

PostgreSQL Example:

To create a table-level check constraint, use the following example:

```
CREATE TABLE PRODUCTS(
      PRODUCT_ID INTEGER PRIMARY KEY,
       PRODUCT_NAME TEXT,
       CURRENT_PRICE CONSTRAINT PRODUCTS_PRICE CHECK (price > 0));
```

# 8.93 Not Null Constraints

A Not Null constraint applies to a single column only. The constraint restricts the column to make an entry mandatory and prevent null values. The database user cannot insert a row without a value for that column. The NOT NULL keyword must be specified at the column-level. There is no option to add a NOT NULL constraint at the table-level. If the column is defined without NOT NULL, the column allows null values. You can use ALTER TABLE in case every row in that column has data, to add a NOT NULL constraint to a column. You cannot remove a NOT NULL constraint using the ALTER TABLE statement.

PostgreSQL and Oracle use the same ANSI SQL syntax. The following is true of NOT NULL constraints:

- Enforce that column cannot accept NULL values.
- Can only be defined inline at table creation.
- Define a name for the NOT NULL constraint when the constraint is used with a CHECK constraint.

PostgreSQL Example:

Use the following example to define not null constraint on the PRODUCT_NAME column:

```
CREATE TABLE PRODUCTS (
       PRODUCT_ID INTEGER PRIMARY KEY,
       PRODUCT_NAME  VARCHAR(50) NOT NULL);
```

## 8.94 REF Constraints

REF constraints define a relationship between a column of type REF and the object it references. You can create the REF constraint as either in line or out of line, but for an out-of-line specification you must specify the REF column or attribute. Both specifications allow you to define the following: a scope constraint, a rowid constraint, or a referential integrity constraint on a REF column.

The REF constraint use the ref_constraint syntax:

- For ref_column, specify the REF column of an object or table.
- For ref_attribute, specify an embedded ref attribute with an object column of a table.

The following restrictions apply f:

- Cannot add a scope constraint to an existing column unless the table is empty.
- Cannot specify a scope constraint for the ref elements of a VARRAY column.
- Must specify this clause if you specify as a subquery and the subquery returns user-defined refs.
- Cannot drop a scope constraint from a REF column.
- Cannot define a rowid constraint for a ref element of a VARRAY column.
- Cannot subsequently drop a rowid constraint from a ref column.
- Clause is ignored if the ref column or ref attribute is scoped.

Oracle Example:
To create a new Oracle type object, use the example below:

```
CREATE TYPE DEP_TYPE AS OBJECT (
      DEP_NAME    VARCHAR2(60),
      DEP_ADDRESS VARCHAR2(300));
```

Oracle Example:
To create  a table based on the previously created type object, use the following example:

```
CREATE TABLE DEPARTMENTS_OBJ_T OF DEP_TYPE;
```

Oracle Example:
To create the EMPLOYEES table with a reference to the previously created DEPARTMENTS table that is based on the DEP_TYPE object, use the example below:

```
CREATE TABLE SUPERVISOR(
     SUPERVISR_NAME   VARCHAR2(60),
     SUPERVISOR_EMAIL  VARCHAR2(60),
     SUPERVISOR_DEPT   REF DEPARTMENT_TYP REFERENCES DEPARTMENTS_OBJ_T);
```

## 8.95 Special Constraint States

You can specify, in Oracle, that a constraint is enabled or disabled. An enabled constraint checks the data as it is entered or updated in the database, if the data does not match the constraint, the data is prevented from being entered. If the constraint is disabled, the data is allowed to enter the database whether that data conforms or not.

Additionally, you can specify that existing data conforms to the constraint with a VALIDATE. A setting of NOVALIDATE means that the existing data is not checked against the constraint.
An integrity constraint can be in one of the following states:

- ENABLE, VALIDATE
- ENABLE, NONVALIDATE
- DISABLE, VALIDATE
- DISABLE, NOVALIDATE

Oracle Example:
Use the following example to modify the state of the constraint to ENABLE NOVALIDATE:

```
ALTER TABLE EMPLOYEES ADD
                    CONSTRAINT CHK_EMP_NAME CHECK(FIRST_NAME LIKE 'a%')  ENABLE NOVALIDATE;
```

These options are not supported in PostgreSQL

## 8.96 Exclusion Constraints

PostgreSQL has exclusion constraints to allow users to create constraints which are hard to do at the application-level. The exclusion constraint ensures that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of those operators returns false or null. One example for this type of constraint is to avoid bookings that overlap. A car rental agency does not want a car rented out to multiple people at the same time or that a driver does not have two rental cars at the same time.

PostgreSQL Example:
To create an exclusion constraint, use the following example:

```
CREATE TABLE car_reservation (
car_reservation_id SERIAL NUMERIC,
during tszrange,
EXCLUDE USING GIST (car WITH =, during WITH &&));
```

This constraint is not available in Oracle databases

## 8.97 Using Existing Indexes to Enforce Constraint Integrity (using_index_clause)

All enabled unique and primary key constraints require an index. You can create the indexes yourself rather than let the database create the index. Note the following:

- Constraints use existing indexes where possible
- Unique and primary keys can use non-unique as well as unique indexes.
- At most one unique or primary key can use each non-unique index.
- The index column order and the constraint do not need to match

You should also always create an index from foreign keys.

You can create a stand-alone index in Oracle and PostgreSQL. The PostgreSQL documentation recommends creating a constraint, which automatically creates an index, instead of using an index to enforce a constraint.

### PostgreSQL SET CONSTRAINTS Synopsis

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are checked when the transaction commits. Each constraint has an IMMEDIATE or DEFERRED mode. Every constraint has one of these characteristics: DEFERRABLE, INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE or NOT DEFERRABLE. The IMMEDIATE characteristic is not affected by the SET CONSTRAINTS command. If the constraint has one of the other characteristics, their behavior can change during a transaction with SET CONSTRAINTS. NOT NULL constraints and CHECK constraints are always IMMEDIATE.

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

- The VALIDATE CONSTRAINT option in ALTER TABLE - Validates foreign key or check constraints that were previously created as NOT VALID. The table is scanned to ensure no rows which do not match the constraint.
- With foreign key or check constraints, the NOT VALID option can used, it will enforce that no new values will be validated, only when issues the VALIDATE CONSTRAINT command is executed

PostgreSQL Example:
Using the ALTER TABLE command in PostgreSQL, it can also be used with CREATE TABLE command, like the following example:

```
ALTER TABLE PRODUCTS ADD CONSTRAINT FK_DEPT FOREIGN KEY (department_id)
      REFERENCES DEPARTMENTS (department_id) NOT VALID;

 ALTER TABLE PRODUCTS VALIDATE CONSTRAINT FK_DEPT;
```

For more information about the Alter Table command, see Alter Table.
For more information about the Create Table command, see Constraints.

Temporary Tables - Key Differences

- Although the syntax is similar, the Oracle temporary table is defined only once and its structure (DDL definition) exists within every session connected to the database. PostgreSQL requires each session to create its own temporary table.
- PostgreSQL allows the GLOBAL and LOCAL keywords in the temporary table definition, but the keywords have no effect on the definition. As PostgreSQL may implement the GLOBAL and LOCAL keywords in the future, using these keywords in current table definitions may cause unwanted results after an upgrade.
- The Oracle syntax for creating temporary tables includes specifying if the temporary table is or is not GLOBAL and if the database should or should not preserve the rows on commit:
CREATE [GLOBAL TEMPORARY] TABLE name ON COMMIT PRESERVE ROWS | DELETE ROWS
- The PostgreSQL syntax is as follows: CREATE [GLOBAL|LOCAL] TEMPORARY|TEMP TABLE [IF NOT EXISTS] name
ON COMMIT PRESERVE ROWS | DELETE ROWS | DROP

## 8.98 Temporary Tables

In Oracle, for a temporary table, the table structure is not temporary and is visible for all sessions. Global Temporary Tables store data in the Temporary Tablespace. The data is local to that session or transaction. After a commit or a session disconnects, the data is deleted. The temporary table creation is similar to the definition of a non-temporary table with the addition of the GLOBAL TEMPORARY keywords and the ON COMMIT clause.

Temporary tables have many of the features of the non-temporary tables but cannot have foreign keys related to other temporary tables or non-temporary tables. Temporary table can be used with ALTER TABLE, DROP TABLE, and CREATE INDEX and support triggers. A temporary table can generate an UNDO and REDO for the UNDO.

The following limitations for temporary tables are:

- Cannot partition or cluster a temporary table.
- Cannot create a temporary table as an index-organized table.
- Does not support parallel UPDATE, DELETE, or MERGE executions.
- Cannot have the same name as an existing table object.
- Drops fail is the table contains data even if the data is from another session.

The primary use-cases for temporary tables include the following:

- Staging tables store intermediate results when batch processing rows.
- Storing session data when the session disconnects, the session data is cleared.

PostgreSQL requires each session to use its own temporary table. Different sessions can use the same temporary table name for different purposes and the table structure may not be the same.

In review, Oracle stores the temporary table structure but does not store the data. The temporary table structure is available after a database restart. PostgreSQL creates the table structure for that session only. When that session disconnects, the temporary table is dropped.

## 8.99 ON COMMIT

Both Oracle and PostgreSQL use the ON COMMIT clause, but the clause is enforced differently. The options are as follows:

- PRESERVE ROWS - The table is session based. The data truncated either by issuing a TRUNCATE or the session terminates.
- DELETE ROWS - The table is transaction based. The database truncates the data after each commit.
- DROP - The table is dropped on COMMIT.

Oracle only allows the PRESERVE ROWS | DELETE ROWS options. If the ON COMMIT clause is omitted, the default behavior is ON COMMIT DELETE ROWS.

PostgreSQL has the same ON COMMIT clause, but there are differences. If the ON COMMIT clause is omitted, the default behavior is ON COMMIT PRESERVE ROWS. The ON COMMIT DROP option does not exist in Oracle.

## 8.100 Global Temporary Table Statistics

Oracle 12c introduces temporary table session-private statistics for global temporary tables. Session-private statistics are controlled by using the GLOBAL_TEMP_TABLE_STATS global preference in the DBMS_STATS package. the setting can be set to SHARED or SESSION.

## 8.101 Global Temporary Table Undo

Oracle 12c provides an option to store the temporary undo data in the temporary tablespace with "alter session set temp_undo_enables=true". This ability reduced the usage of the database primary UNDO tablespace for operations done on temporary tables.

Oracle Example:
Use the example below to create an Oracle Global Temporary Table with ON COMMIT DELETE ROWS behavior:

```
CREATE GLOBAL TEMPORARY TABLE my_sample_table(
      id                    NUMBER NOT NULL,
      first_name        VARCHAR2(100) NOT NULL,
      last_name         VARCHAR2(100) NOT NULL
)
ON COMMIT DELETE ROWS;

--an insert before a commit
INSERT INTO my_sample_table VALUES(1,'Margaret','Hanson'),
(2,'Jon','Bostrom'),(3,'Anita','Hanus');

--first query
SELECT COUNT(*) from my_sample_table;

COMMIT;

--second query
SELECT COUNT(*) FROM my_sample_table;

-- first query result
Count(*)
----------
      3

--second query result
 COUNT(*)
---------
     0
```

PostgreSQL Example:
To create and use a temporary table with ON COMMIT DELETE ROWS behavior, use the following example:

```
CREATE GLOBAL TEMPORARY TABLE SAMPLE_TEMP (
ID                          NUMERIC PRIMARY KEY
,FIRST_NAME       VARCHAR(60) NOT NULL
,LAST_NAME        VARCHAR(60) NOT NULL
)
ON COMMIT DELETE ROWS;


BEGIN;

INSERT INTO SAMPLE_TEMP
VALUES(1,'Bernard','Sheehan'),(2,'Sue','Peck'),(3,'Roxanne','McNeil')(4,'Donald','Max');

--first query
SELECT FIRST_NAME as FIRST_QUERY FROM SAMPLE_TEMP WHERE id = 3;

COMMIT;
```

```
--second query
SELECT FIRST_NAME as SECOND_QUERY FROM SAMPLE_TEMP WHERE id = 4;

--first query result

FIRST_QUERY
------------
Roxanne

--second query
SECOND_QUERY
---------
Donald

 DROP TABLE SAMPLE_TEMP;
```

PostgreSQL Example:
To create and use a Temporary Table with ON COMMIT PRESERVE ROWS behavior, use the following example:

```
CREATE TEMPORARY TABLE SAMPLE_TEMP (
          EMP_ID       NUMERIC PRIMARY KEY,
          FIRST_NAME   VARCHAR(60) NOT NULL,
          LAST_NAME    VARCHAR(60) NOT NULL
         )
 ON COMMIT PRESERVE ROWS;

 INSERT INTO EMP_TEMP VALUES(1, 'Bernard',
'Sheehan'),(2,'Ronald','White'),(3,'Teri','McHale'),(4,'Kavitha','Gowda');

 SELECT LAST_NAME AS FIRST_QUERY FROM SAMPLE_TEMP WHERE id = 4;

 COMMIT;

 SELECT LAST_NAME as SECOND_QUERY FROM SAMPLE_TEMP WHERE id = 1;

 FIRST_QUERY
 ------------
 Gowda

 SECOND_QUERY
 ------------
 Sheehan

 DROP TABLE SAMPLE_TEMP;
```

## 8.102. Transaction Model and Locking - Key Differences

Both Oracle and PostgreSQL use Multiversion Concurrency Control (MVCC) model for data consistency. In MVCC, each SQL statement sees a snapshot of the data. This model prevents statements from viewing inconsistent or dirty data produced by transactions performing updates on the same data rows and provides transaction isolation. MVCC minimizes lock contentions to allow for better performance in multi-user environments. One advantage is the locks acquired for reading data do not conflict with locks acquired for writing data. Table-level and row-level locking facilities are available; however, a design that uses MVCC properly will generally provide better performance than setting locks.

## 8.103 Concurrency Issues

Concurrency is the ability of a system to support two or more transactions interacting with the same data at the same time. Oracle can automatically prevent some concurrency issues by using locks. A lock stops the execution of a transaction if the transaction conflicts with a transaction already running. Concurrency is an issue when data is modified.
The following is a list of the common concurrency issues:

| Issue | Description |
|---|---|
| Lost Updates | When two transactions try to update the same row based on the values originally selected. The last update overwrites the first update. |
| Dirty Reads | When a transaction selects data that has not been committed by another transaction. |
| Non-repeatable reads | When two SELECT statements return different values because another transaction has updated the data in the time between the SELECT statements. |
| Phantom Reads | When one transaction performs an update or delete on a set of rows when another transaction is performing an insert or delete on the same set of rows. |

## 8.104 ANSI Isolation Levels

The ANSI/ISO SQL standard defines four levels of transaction isolation. The same task performed in the same way with the same inputs may result in different answers, depending on the selected isolation level. The transaction levels are defined in terms of how they handle concurrency issues.

The four isolation levels define what may not happen, but not what must happen, allowing each DBMS to interpret the standard with individual differences.

| Isolation level | Dirty read | Non-repeatable read | Phantom read |
|---|---|---|---|
| Read Uncommitted | Possible | Possible | Possible |
| Read Committed | Not permitted | Possible | Possible |
| Repeatable Read | Not permitted | Not permitted | Possible |
| Serializable | Not permitted | Not permitted | Not permitted |

## 8.105 Isolation Levels Supported by Oracle

Oracle explicitly supports the Read Committed and Serializable isolation levels as these levels are defined in the standard. It also provides a Read-Only isolation level.

- Read Committed (default): A transaction may read only data that has been committed in the database. There are no dirty reads, no non-repeatable reads, or phantom reads.
- Serializable: Every transaction is isolated from any other transaction.
- Read-Only: Every transaction is isolated from any other transaction, but a transaction cannot modify the database.

## 8.106 Setting Isolation Levels in Oracle

Isolation levels can be changed at the transaction and session levels.

Oracle Example:

To change the isolation level at the transaction-level, use the following example:

```
--At the transaction level
 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READONLY;

--At the session level
ALTER SESSION SET ISOLATION_LEVEL READ COMMITTED;
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;
ALTER SESSION SET ISOLATION_LEVEL READONLY;
```

## 8.107 Isolation Levels Supported by PostgreSQL

In PostgreSQL, you can request any of the four standard transaction isolation levels. Internally, there are only three levels, which are the Read Committed, Repeatable Read, and Serializable. The Read Uncommitted is equivalent to the Read Committed, the default isolation level. The three isolation levels map to the PostgreSQL MVCC architecture. PostgreSQL complies with the same ANSI/ISO SQL (SQL92) isolation levels with differences from Oracle's implementation:

| Isolation Level | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|---|---|---|---|
| Read Uncommitted | Allowed but not implemented in PostgreSQL | Possible | Possible |
| Read Committed | Not possible | Possible | Possible |
| Repeatable Read | Not Possible | Not Possible | Possible but not implemented in PostgreSQL |
| Serializable | Not Possible | Not Possible | Not Possible |

## 8.108 Setting Isolation Levels in PostgreSQL

Isolation levels can be configured at several levels:

- Session level.
- Transaction level.

PostgreSQL Example:
Configure the isolation level for a specific transaction or session using the example below:

```
--At transaction level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

--at session level
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

PostgreSQL Example:
To view the current isolation level, use the following example:

```
SELECT CURRENT_SETTING('TRANSACTION_ISOLATION'); -- Session
SHOW DEFAULT_TRANSACTION_ISOLATION;              -- Instance
```

## 8.109 PostgreSQL SET TRANSACTION

The SET TRANSACTION command sets the characteristics of the current transaction. The command has no effect on subsequent transactions. to set the characteristics at a session-level, use SET SESSION CHARACTERISTICS. The session characteristics can be overridden at the transaction-level by SET TRANSACTION. If the SET TRANSACTION is executed without a prior START TRANSACTION or BEGIN, there is a warning and the command has no effect.

The SET TRANSACTION SNAPSHOT command allows a new transaction to run in the same snapshot as an existing transaction. The snapshot identifier must be written as a string.

The transaction must be set for SERIALIZABLE mode or REPEATABLE READ mode. The READ COMMITTED mode automatically takes a new snapshot for each command.
For more information about the Set Transaction command, see Set Transaction.

PostgreSQL Example:
The following options are available:

```
SET TRANSACTION transaction_mode [,...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [...]

where transaction_mode is one of:

ISOLATION LEVEL {
SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
}
READ WRITE | READ ONLY [ NOT ] DEFERRABLE
```

The following table displays the common database features of Oracle and PostgreSQL and how they may differ:

| Database Feature | Oracle | PostgreSQL | Comments |
|---|---|---|---|
| AutoCommit | Off | On | Can be set to OFF with SET AUTOCOMMIT |
| MVCC | Yes | Yes | |
| Default Isolation Level | Read Committed | Read Committed | |
| Other Supported Isolation Levels | Serializable Read-only | Repeatable Reads Serializable | |
| Configure Session Isolation Levels | Yes | Yes | |
| Configure Transaction Isolation Levels | Yes | Yes | |
| Nested Transaction Support | Yes | No | Consider using SAVEPOINT |
| Support for Transaction SAVEPOINTs | Yes | Yes | |

## 8.110 Lock Modes

Locks are database mechanisms to prevent destructive interaction between transactions accessing the same resource. Holding locks on a database object can cause other concurrent sessions to wait. Block can occur when concurrent sessions are forced to wait. Oracle and PostgreSQL provide various lock modes to control concurrent access to database objects. These modes are used for controlled locking when the MVCC does not give the desired behavior.

In PostgreSQL, to examine a list of outstanding locks in a database server, use pg_locks system view.

Oracle uses the following modes of locking in a multiuser database:

- Exclusive lock mode prevents the resource from being shared. The first transaction to acquire the lock on a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
- Share lock mode allows a resource to be shared, depending on the operations. Holding a share lock prevents an exclusive lock until the lock is released. Several transactions can acquire share locks on the same resource.

Both database systems automatically acquire the appropriate locks to ensure that the tables are not dropped or modified in an incompatible way. For example, a TRUNCATE command obtains an exclusive lock on the table because this command cannot safely operate with other concurrent operations on the same table. The MVCC architecture prevents viewing inconsistent data produced by concurrent transactions performing updates on the same rows and provides strong transaction isolation for each database session and minimizes lock-contention in multiuser environments. All locks acquired by statements within a transaction are held for the duration of the transaction. Changes made by the transaction become visible only after the first transaction is committed.

PostgreSQL provides multiple lock modes. The MVCC provides transaction isolation for each session. In a multiuser environment the MVCC minimizes the lock-contention.

- MVCC locks acquired for reading data do not conflict with locks acquired for writing data.
- A table is not locked for writes when a threshold is exceeded for row locks.

Lock types can be divided into four categories:

- DML locks - DML locks protect data.
- DDL locks - DDL locks protect the structure of schema objects.
- Internal locks and latches - protect internal database structures, such as datafiles.
- Explicit (Manual) data locking.

Internal locks and latches are entirely automatic.
The following sections describe each category:

| Statement | Row Locks | Table Lock Mode | RS | RX | S | SRX | X |
|---|---|---|---|---|---|---|---|
| SELECT ... FROM table... | — | none | Y | Y | Y | Y | Y |
| INSERT INTO table... | Yes | SX | Y | Y | N | N | N |
| UPDATE table ... | Yes | SX | Y | Y | N | N | N |
| MERGE INTO table ... | Yes | SX | Y | Y | N | N | N |
| DELETE FROM table... | Yes | SX | Y | Y | N | N | N |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SELECT ... FROM table FOR UPDATE OF... | Yes | SX | Y | Y | N | N | N |
| LOCK TABLE table IN... | — | | | | | | |
| ROW SHARE MODE | | SS | Y | Y | Y | Y | N |
| ROW EXCLUSIVE MODE | | SX | Y | Y | N | N | N |
| SHARE MODE | | S | Y | N | Y | N | N |
| SHARE ROW EXCLUSIVE MODE | | SSX | Y | N | N | N | N |
| EXCLUSIVE MODE | | X | N | N | N | N | N |

## 8.111 DML Locks

A DML lock is a lock acquired on a table undergoing an INSERT, UPDATE, or DELETE operation. The lock guarantees the integrity of the data being accessed concurrently. DML statements automatically acquire locks at the table-level and row-level.

- Row Locks (TX): A lock on a single row of a table. A transaction acquires a row lock for each row modified by one of the following operations: INSERT, DELETE, MERGE, and SELECT ... FOR UPDATE. The row lock exists until the transaction commits or rolls back.
- Table Locks (TM): A transaction automatically acquires a table lock with the following operations: INSERT, UPDATE, DELETE, and SELECT ... FOR UPDATE. These operations require a table lock to prevent operations that would conflict with the transaction.

## 8.112 Table Locks

The following list describes the table locks from the least restrictive to the most restrictive:

- Row Share Table Locks (RS) - indicates that the transaction holding the lock on the table has locked rows and intends to update the rows.
- Row Exclusive Table Locks (RX) - indicates the transaction holding the lock has mode one or more updates to the rows in the table.
- Share Table Locks (S) - indicates the table held by the transaction allows other transactions to query the table. No updates are allowed by other transactions.
- Share Rows Exclusive Table Locks (SRX) - only one transaction can acquire a SRX lock on a table. Other transactions may query but not update the table.
- Exclusive Table Locks (X) - Allows the transaction that holds the lock exclusive write access to the table.

PostgreSQL acquires locks automatically to control concurrent access. The following locks types are available:

**PostgreSQL Table-level Locks:**

| Requested Lock Mode VS current | ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
|---|---|---|---|---|---|---|---|---|
| ACCESS SHARE | | | | | | | | X |
| ROW SHARE | | | | | | | X | X |
| ROW EXCLUSIVE | | | | | X | X | X | X |
| SHARE UPDATE EXCLUSIVE | | | | X | X | X | X | X |
| SHARE | | | X | X | X | X | X | X |
| SHARE ROW EXCLUSIVE | | | X | X | X | X | X | X |
| EXCLUSIVE | | X | X | X | X | X | X | X |
| ACCESS EXCLUSIVE | X | X | X | X | X | X | X | X |

PostgreSQL and Oracle lock table syntax are almost the same except for the following:

- ACCESS SHARE,
- ACCESS EXCLUSIVE
- SHARE UPDATE EXCLUSIVE

PostgreSQL Row-level Locks:

| Requested Lock Mode VS current | FOR KEY SHARE | FOR SHARE | FOR NO KEY UPDATE | FOR UPDATE |
|---|---|---|---|---|
| FOR KEY SHARE | | | | X |
| FOR SHARE | | | X | X |
| FOR NO KEY UPDATE | | X | X | X |
| FOR UPDATE | X | X | X | X |

3. Page-level Locks: Can be either shared or exclusive. In the shared buffer pool, these locks control read or write access to table pages. After a row is fetched or updated, the locks are released.

4. Deadlocks: A situation when multiple transactions wait for each other to release a lock. PostgreSQL resolves deadlocks by aborting a transaction and the other transactions continue.

## 8.113 DDL Locks

A data dictionary lock (DDL) protects the definition of a schema object while that object is acted upon or referred to by an ongoing DDL operation. For example, for a stored procedure, Oracle automatically acquires DDL locks for all schema objects referenced by the stored procedure definition to prevent those objects from being altered or dropped before the procedure completes.

## 8.114 Internal locks and Latches

Latches and internal locks protect database and memory structures. They are inaccessible to database users.

## 8.115 External (Manual) Data Locks

You can override the Oracle default locking mechanism at the transaction level or session level.
At the transaction level, the following statements override Oracle's default locking:

- SET TRANSACTION ISOLATION LEVEL
- LOCK TABLE
- SELECT ... FOR UPDATE

Locks acquired by these statements are released after the transaction commits or rolls back.
At the session level, the session can set the required transaction isolation level with the ALTER SESSION statement.

Oracle Example:
In two sessions, the first session locks the table. The second session cannot finish until the first session issues a COMMIT.

```
-- Session 1
INSERT INTO test VALUES ('day','welcome');

BEGIN;
SELECT * FROM items WHERE value1 = 'day' FOR UPDATE;

-- Session 2
UPDATE test SET value = 'bonjour' WHERE value1 = 'day';

-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

## 8.116 Advisory Locks

PostgreSQL provides a way to create locks with application-defined meanings. These locks are called advisory locks. The system does not enforce these locks, the application does. Advisory locks can be useful for locking strategies that do not fit the MVCC architecture.

## 8.117 Comparison table

| Description | Oracle | PostgreSQL |
|---|---|---|
| "Dictionary" tables to obtain information about locks | v$lock;<br>v$locked_object;<br>v$session_blockers; | pg_locks<br>pg_stat_activity |
| Lock a table | BEGIN;<br>LOCK TABLE employees IN SHARE ROW EXCLUSIVE MODE; | LOCK TABLE employees IN SHARE ROW EXCLUSIVE MODE; |
| Explicit Locking | SELECT * FROM Invoices WHERE Invoice_id=1002 FOR UPDATE; | BEGIN;<br>SELECT * FROM Invoices WHERE Invoice_id=1002 FOR UPDATE; |
| Explicit Locking, options | SELECT…FOR UPDATE | SELECT ... FOR SHARE<br>SELECT ... FOR UPDATE<br>SELECT ... FOR NO KEY UPDATE (When an UPDATE does not update columns related to keys)<br>SELECT ... FOR KEY SHARE (to reduce lock contention for foreign key triggers) |

## 8.118 Triggers - Key Differences

An Oracle trigger is defined in much the same way as a package. The trigger has a declaration and a body. The declaration section describes the trigger, and when the trigger is fired. The body section is an anonymous block of PL/SQL code. The trigger does not return a value. The trigger is planned to fire on a specific event and timing. The trigger can be designed to perform related actions or to centralize global actions.

For detailed information about Oracle triggers, see Create Trigger Statement.
A PostgreSQL trigger is a function invoked by any of the following events:  Insert, Update, Delete, or Truncate. A trigger is a user-defined function bound to a table. A database user creates the function and then binds this function to a table. PostgreSQL has several types of triggers, on row and statement levels. A difference between the types is when the trigger is invoked and the number of times it is invoked. For example, an Update trigger for 20 rows would invoke the row level trigger for each row, but the statement level trigger is invoked once.

For more information about PostgreSQL triggers, see Trigger Procedures.

## 8.119 Triggers

As a database grows, so does its complexity. The database administrator and database user can utilize triggers to monitor, troubleshoot, and enforce business rules. A trigger is a user-defined procedure or function that is bound to a table, automatically invoked when a selected event occurs.. The database user cannot call a trigger directly, as the trigger is automatically invoked when an event occurs. An excessive use of triggers may result in complex inter-dependencies.

A trigger can be set to execute before an operation is attempted on a row, after the operation has completed, or instead of the operation.

You can do the following with triggers:

- Control the behavior of DDL statements.
- Control the behavior of DML statements.
- Control the sequence of and synchronize calls to triggers.
- Enforce referential integrity, complex business rules, and security policies.
- Control and redirect DML statements when changing data in a view.
- Audit information of system access and behavior by populating logs.

## 8.120 Privileges Required to Use Triggers

In Oracle, the database user must have the CREATE TRIGGER system privilege. If you are required to interact with a trigger that is owned by another user, you must be granted the ALTER TRIGGER privilege on the trigger.

In PostgreSQL, the user must have the TRIGGER privilege on the table and the EXECUTE privilege on the trigger.

The following is an example of an Oracle "After Update trigger" executed after a row is inserted into the table.

```
CREATE OR REPLACE TRIGGER widgets_au_trigger
    AFTER UPDATE OF price ON widgets
    FOR EACH ROW
DECLARE v_user VARCHAR2(100);
    BEGIN
      SELECT user INTO v_user FROM dual;
      INSERT INTO widgets_audit_log(id, widgets_id, description, created_by, old_price, new_price, log_date)
       VALUES (widgets_audit_log_seq.NEXTVAL, :new.id, :new.description, v_user, :old.price,:new.price, sysdate);
END;
/

Trigger created.
```

```
Update widgets SET price = 4.99 WHERE id = 1;

SELECT created_by, old_price, new_price FROM widgets_audit_log;

created_by        old_price        new_price
----------------------  ---------------------- ------------------------
user name         3.99             4.99
```

The following example demonstrates an Oracle schema trigger on a table that fires if a TRUNCATE command is executed for an object in the schema. The table is not truncated, the user information is logged, and raises an application error.

```
CREATE OR REPLACE TRIGGER NO_TRUNCATE_TRIGGER
    BEFORE TRUCATE ON SCHEMA
DECLARE PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
INSERT INTO log_table(id, user_computer, user_ip, user_name)
SELECT log_table_id_seq.nextval,
SYS_CONTEXT('USERENV','TERMINAL'),
SYS_CONTEXT('USERENV',IP_ADDRESS'),
SYS_CONTEXT('USERENV','OS_USER') FROM dual;

CONMMIT;
       RAISE_APPLICATION_ERROR (num => -20000,
       msg => 'You are not allowed to truncate a table. Contact your DBA');
END;
/

Trigger created.

TRUNCATE TABLE REGIONS;

ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: You are not allowed to truncate a table. Contact your DBA.
ORA-06512: at line 13

SELECT * FROM log_table;

ID                USER_COMPUTER         USER_IP              USER_NAME
-----------------   ------------------            -------------------          -----------------
6                 unknown                     228.209.72.249      John Smith
```

The following is an example of an AFTER INSERT trigger for PostgreSQL:

```
CREATE TRIGGER log_new AFTER INSERT ON warehouse
 FOR EACH ROW EXECUTE PROCEDURE logfunc();
 CREATE OR REPLACE FUNCTION logfunc()
 RETURNS TRIGGER AS
 $$
 BEGIN
```

```
                              INSERT INTO log(log_time, description) VALUES (current_timestamp, 'new data inserted
into warehouse');
 RETURN NEW;
 END;
 $$ LANGUAGE plpgsql;

INSERT INTO public.warehouse(
                    warehouse_id, warehouse_name, year_created, street_address, city, state, zip
                    ) VALUES (6, 'Regency', 2018, '801 Main', 'North Adams', 'MA', '10101');

 SELECT * FROM log;

 log_id     log_time       description
 ----------  --------------   --------------
 6        15:08:06       new data inserted into warehouse
```

## 8.121 Drop Triggers

The DROP command deletes a trigger in both databases, like in the following example:

```
 DROP TRIGGER name;
```

## 8.122 User-Defined Functions - Key Differences

One way to extend the database is through user-defined functions. This functionality allows commonly required code to be written and tested once, and then used by any application that requires the code.

- PostgreSQL supports the creation of User-Defined Functions using the CREATE FUNCTION statement. The PostgreSQL extended SQL language, PL/pgSQL, is the primary language to use while migrating from Oracle’s PL/SQL User-Defined Functions.
- Oracle database uses the DUAL table for a SELECT statement where a FROM clause is not needed, because a FROM clause is mandatory. In PostgreSQL, the FROM clause can be omitted.
- Oracle and PostgreSQL data types are different. For example, in Oracle a function may use the NUMBER data type. The appropriate data type for that function in PostgreSQL are either DECIMAL or NUMERIC data types.
- Oracle uses the RETURN clause in a function statement specifies the return value data type. In an Oracle function, there must be at least one execution path that leads to a return clause. PostgreSQL can use either the INOUT parameter, OUT parameter or RETURNS to specify the return data type.
- PostgreSQL has a LANGUAGE clause. The function can be implemented in SQL, C, INTERNAL, or a procedural language.

## 8.123 User-Defined Functions (UDFs)

An Oracle User-Defined Function (UDF) can be written in PL/SQL, Java, or the C language. A UDF helps the DBA to achieve a functionality that does not available in SQL or with SQL built-in functions but needed by the application. Although PL/SQL is a powerful language, there are tasks a different programming language can achieve more easily. You can use an external C program to interface with embedded systems, solve engineering problems or analyze data. The Java language has mature, reusable libraries providing common design patterns. Functions are also one of the keys to modular, reusable PL/SQL code. Functions are stored in a compact, compiled form. When invoked, they are loaded and processed immediately. As a subprogram, the function takes advantage of shared memory. One copy of the UDF is loaded into memory for execution by multiple users.A UDF can be found in SQL statements where built-in SQL functions can be used in any of the following areas:

- While performing DML operations.
- Return a single value from a SELECT statement (scalar function).
- With WHERE, CONNECT BY, ORDER BY, HAVING, GROUP BY, and START WITH clauses.

PostgreSQL provides the following kinds of functions:

- Query language: usually written in SQL.
- Procedural language functions: usually written in a procedural language (PL) such as PL/pgSQL, PL/Tcl, PL/Perl, or PL/Python.
- Internal functions: usually functions written in C and statically linked into the PostgreSQL server.
- C-language functions.

PostgreSQL functions allow any collection of commands to be packaged together and defined. The function can include, besides SELECT queries, INSERT, UPDATE, and DELETE queries. You cannot include transaction control commands, such as COMMIT, or SAVEPOINT or a variety of utility commands, such as VACUUM.

Every PostgreSQL function has a volatility classification. You should label your functions with the strictest volatility classification that is valid for the function. The classifications are the following:

- Volatile: can do anything, including modifying the database. The function can return different results on successive calls. You cannot use a volatile function in an index scan condition.
- Stable: Cannot modify the database and returns the same results given the same arguments. You can use the function in an index scan condition.
- Immutable: Cannot modify the database and is guaranteed to return the same results given the same arguments forever.

For more information about the Oracle User-Defined Function, see Create Function Statement.
Oracle Example:
Use UDF to convert Celsius to Fahrenheit:

```
CREATE OR REPLACE FUNCTION c_to_f(C_DEGREES NUMBER)
RETURN NUMBER
IS
F_DEGREES NUMBER;
BEGIN
 F_DEGS:=(C_DEGREES * 9/5) + 32;
RETURN F_DEGREES;
END;
```

```
/
-- Verifying
SELECT c_to_f(14) FROM DUAL;
57.2
…
```

PostgreSQL Example:
The same syntax for create a Celsius to Fahrenheit conversion function in PostgreSQL UDF.

```
CREATE OR REPLACE FUNCTION c_to_f(IN C_DEGREES NUMERIC(10,2), OUT F_DEGREES NUMERIC(10,2))
AS
$$
BEGIN
      F_DEGREES:=(C_DEGS * 9/5)+32;
END;
$$
LANGUAGE plpgsql;

SELECT public.c_to_f(14);
57.20
```

For more information about PostgreSQL User-defined Functions, see User-defined Functions.

## 8.124 Required System Privileges

In Oracle, for a database user to create a function in their own schema, the user must have the CREATE PROCEDURE system privilege. To create a function in another user's schema, the database user must have the CREATE ANY PROCEDURE system privilege. To replace a function in another user's schema, the database user must have the ALTER ANY PROCEDURE system privilege.

In PostgreSQL, the database user must have the USAGE privilege on the language, the argument types, and the return type.

## 8.125 Common UTL Packages - Key Differences

Oracle's database UTL packages add to database some more complex capabilities which aren't usually related to regular database types, the packages that will be covered in this topic allow an authorized user to write and read a file, send an email from the database, using either UTL_MAIL or UTL_SMTP. PostgreSQL does not support sending email directly from the database, but you can use other services on Google Cloud Platform to achieve the same result.

## 8.126 Common UTL Packages

WIth Oracle’s UTL_FILE package, PL/SQL programs can read and write operating system text files. The package is available in client-side and server-side PL/SQL.

UTL_FILE has the following rules and limits:

- Cannot use operating system-specific parameters in the file location or file name parameters.
- I/O capabilities are similar to the standard operating system stream file capabilities, with some limitations.
- In a PL/SQL file I/O, errors are returned using PL/SQL exceptions.

The following subprograms are typically used:

- FCLOSE procedures - closes a file
- FOPEN function - opens a file for input or output
- GET_LINE procedure - reads text from an open file
- PUT procedure - writes a string to a file

Oracle Example:

To create and write text to a file, use the following example:

```
DECLARE
fhandle utl_file.file_type;
BEGIN
  fhandle := utl_file.fopen (
            'UTL_DIR'    -- location of file
             ,'sample_file.txt' --name of file
             , 'w' --w = write
                 );

utl_file.put(fhandle, 'Writing to file - hello world'
                || CHR(12)):
utl_file.put(fhandle,'Thank you for reading this.');

 utl_file.fclose(fhandle);
exception
   when others then
       dbms_output.put_line('ERROR: ' || SQLCODE || ' - ' || SQLERRM);
        raise;
 END;
/
```

The UTL_MAIL package lets an authorized PL/SQL user sends an email from Oracle.

UTL_MAIL interfaces with the smtp_out_server parameter to specify the outbound destination for email. You can specify multiple service by separating the server names by a comma. If the first server is unavailable, the package tries each of the other servers listed in turn. The package uses utl_tcp and utl_smtp internally and simplifies the mail process.

Oracle Example:

Run the following script to install the packages:

```
@{ORACLE_HOME}/rdbms/admin/utlmail.sql
@{ORACLE_HOME}/rdbms/admin/prvtmail.plb
GRANT execute on utl_mail to sample_user;
```

Before the packages can be used, the SMTP gateway must be defined with the SMTP_OUT_SERVER parameter. The instance may require a restart before an email can be sent.
Oracle Example:
Configure the SMTP_OUT_SERVER location using the following example:

```
ALTER SYSTEM SET smtp_out_server = 'smtp.my_domain.com' SCOPE=BOTH;
shutdown immediate
startup
```

You can send an email using the send procedure.
Oracle Example:
Use the following example to send an email:

```
begin
   utl_mail.send(sender     =>'user1@test.com',
                 recipient => 'user2@test.com',
                 subject =>'Test UTL_MAIL',
                 message => 'Good morning. If you are reading this, the UTL_MAIL works.');
end;
/
```

The UTL_SMTP utility allows the database to send message by email to any valid email address. Emails can be sent to production support personnel when certain events occur. Any event that can be monitored can be sent by email. The UTL_SMTP package provides PL/SQL a way to send messages over the Simple Mail Transfer Protocol (SMTP). Depending on your Oracle version, the package may require the JServer option, this option can be loaded by the Database Configuration Assistant (DBCA) or by running a script.

Oracle Example:

Run the script to install UTL_SMTP, based on Oracle version:

```
In oracle 12c:
@{ORACLE_HOME}/rdbms/admin/utlsmtp.sql

In oracle 11g:
@{ORACLE_HOME}/javavm/install/initjvm.sql
@{ORACLE_HOME}/rdbms/admin/initplsj.sql
```

The following events, procedures, and functions are called:

- Open an SMTP connection.

- HELO function and procedure performs the initial handshake with the SMTP server.
- MAIL function and procedure initiates an email transaction with the server, the destination is a valid email address.
- RCPT function defines the recipient.
- DATA function and procedure sends the email body.
- QUIT function and procedure terminates the SMTP session and disconnects from the server.

Oracle Example:
To send an email, use the following example:

```
DECLARE
smtpconn utl_smtp.connection;
BEGIN
smtpconn := UTL_SMTP.OPEN_CONNECTION('smtp.test.com', 25);
UTL_SMTP.HELO(smtpconn, 'smtp.test.com');
UTL_SMTP.MAIL(smtpconn, 'user1@test.com');
UTL_SMTP.RCPT(smtpconn, 'user2@test.com');
UTL_SMTP.DATA(smtpconn,'This is a test of the UTL_SMTP package. If you read this, it works.');
UTL_SMTP.QUIT(smtpconn);
END;
/
```

In PostgreSQL, there is no option to send email directly from the database; however, you can utilize other products in Google Cloud Platform such as AppEngine, GKE and Cloud Functions to get the same functionality.

For example, Google Cloud Platform has a serverless service called Cloud Functions, this service allows you to invoke a function that will query a table in the database, this table will store all the required details to send an email:

1. Sender
2. Recipient
3. Subject
4. Body
5. Sending status

After querying the table, the function can send the email to the email service (e.g SendGrid)and  mark the email as sent.

here you can find a tutorial on how to send emails using SendGrid, but other services  are available as well.

Writing a mail-sending function with Google Cloud Functions

Before you start, prepare your environment, than run the command below to add the required libraries to your Node.js client so it can connect to the Cloud SQL for PostgreSQL instance:

```
npm install pg
```

The command above will put the files in the current directory that your CLI is pointing on.

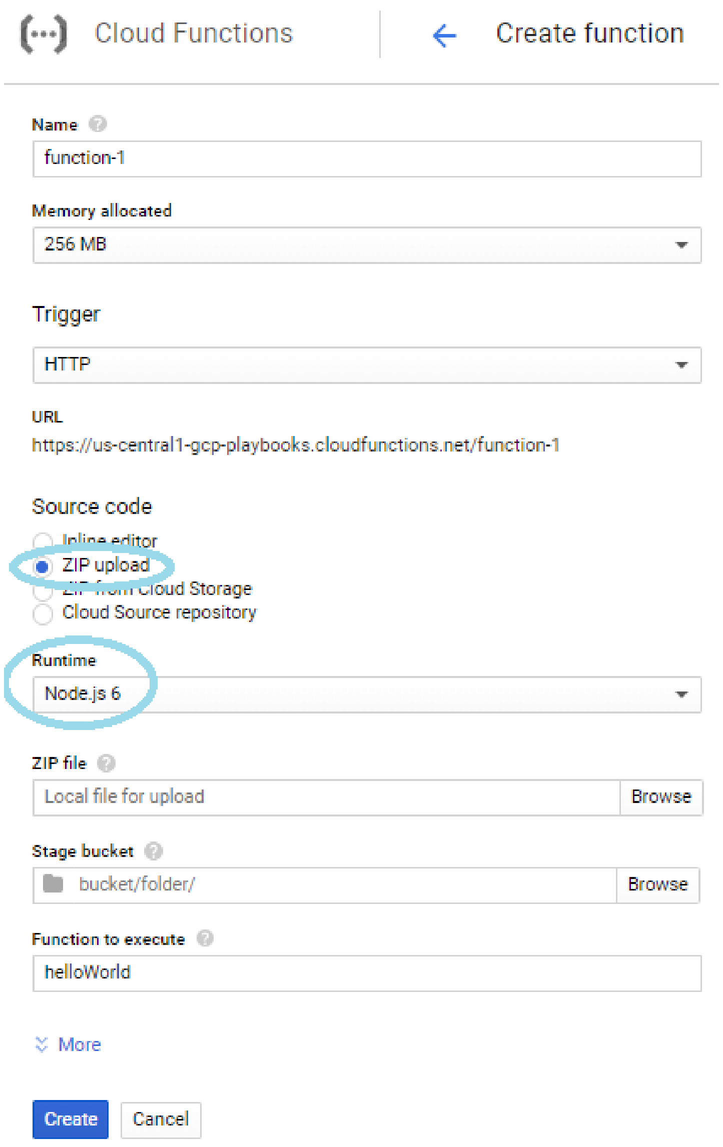
The npm command will create a file and a folder named package-lock.json and node_modules.

Next, create a file named index.js in the folder that you run the npm command, and put the code snippet below as its content, explanation about the code can be found here:

```
const pg = require('pg');

 const connectionName =
   process.env.INSTANCE_CONNECTION_NAME || 'gcp-playbooks:us-central1:test-playbook';
 const dbUser = process.env.SQL_USER || 'postgres';
 const dbPassword = process.env.SQL_PASSWORD || 'Aa123456';
 const dbName = process.env.SQL_NAME || 'postgres';

 const pgConfig = {
   max: 1,
   user: dbUser,
   password: dbPassword,
   database: dbName,
 };

 if (process.env.NODE_ENV === 'production') {
   pgConfig.host = `/cloudsql/${connectionName}`;
 }

 let pgPool;

 exports.postgresDemo = (req, res) => {
   if (!pgPool) {
     pgPool = new pg.Pool(pgConfig);
   }

   pgPool.query('SELECT * FROM new_emails_tbl', (err, results) => {
     if (err) {
       console.error(err);
       res.status(500).send(err);
     } else {
       res.send(JSON.stringify(results));
     }
   });

/*GO OVER THE RESULTS AND CALL THE EMAIL COMMAND:*/
curl -X POST "https://YOUR_REGION-
YOUR_PROJECT_ID.cloudfunctions.net/sendgridEmail?sg_key=YOUR_SENDGRID_KEY" --data
'{"to":"YOUR_RECIPIENT_ADDR",
  "from":"YOUR_SENDER_ADDR",
   "subject":"YOUR_SUBJECT",
   "body":"THE MESSAGE BODY"}' --header "Content-Type: application/json"
  pgPool.query('UPDATE new_emails_tbl SET sent=1', (err, results) => {
     if (err) {
       console.error(err);
       res.status(500).send(err);
     } else {
       res.send(JSON.stringify(results));
     }
   });

 };
```

Now you have 2 files and one directory in the project folder:  index.js, package-lock.json and node_modules directory, take those 3 items and zip them into a single zip file.

When creating a new function, make sure you are using "ZIP upload" as the source code and "Node.js 6" as the runtime, fill the other details and click Create.



Cloud Functions ← Create function

Name
function-1

Memory allocated
256 MB

Trigger
HTTP

URL
https://us-central1-gcp-playbooks.cloudfunctions.net/function-1

Source code
- Inline editor
- [✓ ZIP upload]
- ZIP from Cloud Storage
- Cloud Source repository

Runtime
Node.js 6

ZIP file
Local file for upload
Browse

Stage bucket
bucket/folder/
Browse

Function to execute
helloWorld

To make this function run in specified time intervals use this guide.

## 8.127 Views - Key Differences

Both Oracle and PostgreSQL use standard ANSI SQL to create views.

In PostgreSQL, to replace a view, the defining query must define the same names, same data types, and list the columns in the same order. If you must reorder or make changes to the view, it is recommended to remove the view and create a new view. Access to the view is determined by the view owner.

## 8.128 Views

A database view is a searchable object in the database. A view is defined by a query. A view does not store data. You can query a view as if it was a table. A view can be based on the data from one or more base tables. The syntax and purpose of the Oracle and PostgreSQL views are similar. A view can provide simple, granular security, by using the view to limit the data a user can see. The benefit of views is the following:

- Enforce business rules
- Consistency
- Security
- Simplify the database structure
- Conserve space

To change the defining query of a PostgreSQL view, use the CREATE OR REPLACE VIEW statement. PostgreSQL does not support removing an existing column in a view when replacing the view. The defining query must define the same names, same data types, and have the columns in the same order. PostgreSQL allows additional columns at the end of the column list. If you must make changes, it is recommended to remove the view and create a new one.

## 8.129 View Parameters

| Oracle View Parameter | Description |
|---|---|
| CREATE OR REPLACE | Create a view or replace an existing view |
| FORCE | Create the view even if the defining query cannot be executed. |
| VISIBLE \| INVISIBLE | By default, view columns are visible regardless of their visibility on the base tables unless you specify invisible |
| WITH READ ONLY | Indicates that the table or view cannot be updated |
| WITH CHECK OPTION | Indicates that Oracle prohibits any changes to the table or view that would produce rows not included in the subquery |

There are two types of views: simple view and complex view. Simple views can only contain one base table. Complex views can be constructed from one or more base tables. Complex views can contain joins, group by clauses, and order by clause. DML operations can be executed on a simple view but cannot always be performed on a complex view.

Oracle Example:
To create and update a simple view, use the following example:

```
CREATE OR REPLACE VIEW vw_EMP(employee_id, fName, salary, hire_date, job_id, dept_id)
AS
(SELECT EMPLOYEE_ID, CONCAT(CONCAT(first_name,' '),last_name) as fName, SALARY, HIRE_DATE, JOB_ID,
DEPARTMENT_ID
FROM EMPLOYEES);

UPDATE VW_EMP
SET dept_id=110
WHERE employee_id = 205;
```

Oracle Example:
The following example creates and updates a complex view, then tries to update the view, which is not allowed:

```
CREATE OR REPLACE VIEW vw_dpt_emp(d_id, cn_emp)
AS
SELECT department_id, COUNT(employee_id) AS cn_emp
FROM EMPLOYEES
GROUP BY department_id;

UPDATE VW_dpt_emp
SET cn_emp = cn_emp +1
WHERE d_id=50;

ORA-01732: data manipulation operation not legal on this view
```

- CREATE [OR REPLACE] VIEW: In PostgreSQL, the CREATE OR REPLACE VIEW command is similar to Oracle. Simple views are automatically updatable. To create an updatable view, the following must be true:
  - Only one table or another updatable view - a view with a join is not updatable
  - No WITH, GROUP BY, HAVING, DISTINCT, LIMIT, or OFFSET clauses allowed in the view definition
  - The view columns cannot contain set operations such as UNION, INTERSECT or EXCEPT
  - The SELECT list cannot contain any aggregates, window functions, or set-returning functions
  - The Check Option is only applied to an updatable view

- WITH [ CASCADED | LOCAL ] CHECK OPTION:This option controls the behavior of automatically updatable views. If you have a WHERE clause in the defining query, you can still update or delete rows that are not visible in the view. The CHECK OPTION avoids this case.

CHECK OPTION:

- LOCAL: New rows are checked against the conditions defined in the view itself.
- CASCADED: New rows are checked against the conditions of the view and all underlying base views.

PostgreSQL Example:
To create a view, use the following example:

```
CREATE OR REPLACE VIEW public.dup_users
        AS SELECT test_users.user_id,
         concat(test_users.first_name, ' ', test_users.last_name) AS full_name,
         replace(test_users.enrollment::text, '-'::text, ''::text) AS enrollment
          FROM test_users;
```

## 8.130 Drop Views

The DROP VIEW command is similar in both Oracle and PostgreSQL. In PostgreSQL, you must be the owner of the view. In PostgreSQL, the IF EXISTS, CASCADE | RESTRICT parameters are optional.

- IF EXISTS: do not throw an error if the view does not exist.
- CASCADE: automatically drop objects that depend on the view. RESTRICT - do not drop the view if there are dependencies.

Oracle Example:

```
DROP VIEW view_name;
```

PostgreSQL Example:

```
DROP VIEW [IF EXISTS ] view_name [CASCADE | RESTRICT];
```

## 8.131 Privileges

To create a view, you must meet the following requirements:

- Granted CREATE VIEW, to create a view in your schema, or CREATE ANY VIEW, to create a view in another user's schema, either explicitly or through a role.
- Granted explicitly the SELECT, INSERT, UPDATE, or DELETE object privileges on all base objects underlying the view or the SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, or DELETE ANY TABLE system privileges. You cannot obtain these privileges through a role.

- In order to grant other users access to your view, you must have received object privileges to the base objects with GRANT OPTION.
- EXECUTE privilege on the function.

In PostgreSQL, access to tables referenced in a view is determined by the view owner. Functions called in a view are treated as if the function had been called from a query using the view. The user of a view must have permissions to call all functions used in a view. A role must be granted SELECT, CREATE and DML privileges on the base tables.

## 8.132 Virtual Columns - Key Differences

Oracle database can define a virtual column that can be derived from the other columns in the table.

PostgreSQL does not support a virtual column. The functionality can be emulated and the alternatives are the following:

- View: With a function populating a column from other columns.
- Function: Populates a column
- Trigger: Populates the column on INSERT or UPDATE using a function.

## 8.133 Virtual Columns

Oracle Virtual Column can be derived from the other columns of the table, be the result of a function, or be a constant expression. No physical data can be stored in the column and no DML operations are permitted against the column. A virtual column can simplify queries and you can apply indexes to the column.

The following provides notes and restrictions on virtual columns:

- Indexes defined against virtual columns are equivalent to function-based indexes.
- Can be referenced in a WHERE clause of updates and deletes.
- Cannot be modified by DML.
- Not supported for index-organized, external, objects, cluster, or temporary tables.
- The output of the expression used in the virtual column definition must be a scalar value.

### Generating a Virtual Column

In the syntax for generating a virtual column, the GENERATED ALWAYS keywords and VIRTUAL keyword are optional.
Oracle Example:
To create a table with a virtual column, use the following example:  The creation of a table with a virtual column:

```
CREATE TABLE bkstore_location (
      bkstore_location_ID NUMBER(8) NOT NULL,
      bk_location   VARCHAR2(50),
      area_desc    VARCHAR2(50) GENERATED ALWAYS AS
        (CASE WHEN substr(bk_location, 1,1,) = 'S'
          THEN 'Suburb'
          ELSE 'City' END) VIRTUAL
);
```

Oracle Example:
Insert a new record into the table using the following example. You do not insert values for the virtual column.

```
INSERT INTO bkstore_location (bkstore_location_id, bk_location) values (20, 'M211');
```

Oracle Example:
To run a SELECT statement for the virtual column, use the example below:

```
SELECT area_desc FROM bkstore_location;

AREA_DESC
--------------------
 City
```

PostgreSQL does not have the concept of a virtual column. This functionality can be emulated. The PostgreSQL alternatives are the following:

- View: Using a function to populate a column.
- Function: Populates a column based on the values in another column or columns.
- Trigger: Populate the column on INSERT or UPDATE using a function.

In the following example, data was imported from a legacy system. Some users were created with duplicate enrollment numbers. The main difference is the placement of a dash in the enrollment number.

PostgreSQL Example:
To create  a table, use the following example:

```
CREATE TABLE test_users (
       user_ID serial PRIMARY KEY,
       FIRST_NAME   VARCHAR,
       LAST_NAME    VARCHAR,
       enrollment    VARCHAR);
```

Use the following example to create  a PL/pgSQL function which strips out the dash and finds the duplicate users:

```
CREATE OR REPLACE FUNCTION duplicate_enrollment (test_users)
      RETURNS text AS $$
      SELECT replace($1.enrollment,'-','')
      $$ STABLE LANGUAGE SQL;
```

Insert data to the table, using INSERT INTO:

```
 INSERT INTO "test_users"(FIRST_NAME,LAST_NAME,ENROLLMENT) VALUES ('Denise','Gilliam','48249077-8');
```

Use the duplicate_enrollment function as part of a SELECT statement:

```
SELECT test_users.duplicate_enrollment, string_agg(last_name,', ') as last_names
group by test_users.duplicate_enrollment
having count(*) > 1;
      FROM test_users ;

 duplicate_enrollment      last_names
 ------------------------     -----------------------
 133392491                   Witt, Lowe
 334607422                   Jefferson, Daugherty
```

Use the example below to create a view that mimics the function:

```
CREATE VIEW dup_enrollments as
 SELECT user_ID, CONCAT(first_name, ' ' , last_name) as full_name, replace(enrollment,'-','') as enrollment
      FROM test_users;


 SELECT enrollment, string_agg(full_name,',' ) from dup_enrollments group by enrollment having count(*) > 1;
```

The function is created as STABLE in the volatility category, you can create an index:

```
CREATE INDEX dup_enrollments_idx ON test_users(duplicate_enrollment(test_users));
```

Use the EXPLAIN keyword to display the execution plan. The table sample is small, and the sequence scan would probably perform better. You use a command to turn off sequence scan to see the index:

```
SET enable_seqscan = OFF;

 EXPLAIN
  SELECT test_users.duplicate_enrollment, string_agg(last_name,', ') as last_names
                              from test_users
                              group by test_users.duplicate_enrollments
                              having count(*) > 1;

                                 QUERY PLAN
```

```
---------------------------------------------------------------------------
-> Index Scan using duplicate_enrollment_idx on test_users (cost=0.14..13.02 rows=50 width=64
```

## DML Support

You can also create a trigger to populate the column values automatically. For this approach, create the following objects:

- Function: Contains to populate the column.
- Trigger: Invokes the function.

The area_desc column is populated using data from the bk_location column.
For more information about Triggers, see Create Trigger.
PostgreSQL Example:
Use the following example to create the table:

```
CREATE TABLE bk_location(
        ID serial PRIMARY KEY,
        bk_location   VARCHAR,
        area_desc    VARCHAR
 );
```

Use the example below to a function to populate the area_desc column:

```
CREATE OR REPLACE FUNCTION set_area_desc ()
       RETURNS trigger as
                      $$
               BEGIN
                  if substr(new.bk_location,1,1)='S'
                         then new.area_desc = 'Suburb';
                         else new.area_desc = 'City';
                         end if;
              RETURN NEW;
               END;
              $$
 LANGUAGE plpgsql;
```

Use the following example to create a trigger that invokes the function:

```
CREATE TRIGGER TRG_area_desc BEFORE INSERT OR UPDATE
        ON bk_location FOR EACH ROW
        EXECUTE PROCEDURE set_area_desc();
```

To run a SELECT statement to view the results, use the example below:

```
INSERT INTO bk_location(bk_location) values ('SM-311'), ('TM-334');
```

```
SELECT  * FROM bk_location;

id | bk_location | area_desc
-------------+------------+-----------+-------------
 1 | SM-311      |Suburb
 2 | TM-335      |City
```

Use the following example to create an Index based on the virtual area_desc column:

```
CREATE INDEX bk_area_desc_idx ON bk_location(area_desc);
```

Use the example below to check if  the execution plan uses the index with EXPLAIN:

```
SET enable_seqscan = OFF;

 EXPLAIN
  SELECT * FROM bk_location
  WHERE area_desc = 'Suburb';

                              QUERY PLAN
---------------------------------------------------------------------------
  Index Scan using bk_area_desc_idx on bk_location (cost=0.13..8.15 rows=1 width=68)
    Index Cond: ((area_desc)::text = 'Suburb'::text)
```

## 8.134 Synonyms

Oracle sSynonyms feature is not supported in PostgreSQL but, depending on the scenario, you may find an alternative feature in PostgreSQL that can achieve the same behaviour.

In Oracle databases, Synonyms are the feature that gives the ability to give another name for an object, the reference can also point to an object that belongs to another schema.

There is also a property that can make a Synonym object accessible to all users.

PostgreSQL doesn't have the concept of a synonym in this type of feature, however, for most use cases, there are workarounds. The alternative in PostgreSQL depends on the object type that is being referred to in the source database.

For example: iIf you just want to refer to an object in a different schema without specifying the fully qualified name, you can just set the SEARCH_PATHsearch_path parameter in PostgreSQL— as described later in this chapter will be described later.

Consult the table below to decide which object in PostgreSQL you should use:

| Object type being referenced in Oracle | Object type that can be used in PostgreSQL |
|---|---|
| Table | View |
| View | View |
| Function | Function that wrap another function |
| Procedure | Function that wrap another function |

Oracle Example:
The statement below, creates a sSynonym called “emps” that references to a table named “employees”.

```
CREATE SYNONYM emps FOR employees;
```

PostgreSQL Example:
The statement below will achieve the same functionality by creating a view named “emps” that will query the “employees” table

```
 CREATE VIEW emps AS SELECT * FROM employees;
```

Oracle Example:
The statement below will create a synonym named “emps_func” that refers to a function called “get_emps "

```
CREATE SYNONYM emps_func FOR get_emps;
```

PostgreSQL Example:
The code below will create a function named “emps_func” that calls the “get_emps” function.

```
CREATE OR REPLACE FUNCTION emps_func() RETURNS DECIMAL AS

$BODY$

BEGIN

        return get_emps();

END

$BODY$

LANGUAGE plpgsql;
```

**Note:** Make sure that your referencing refers function returns the same data type as the being referenced function being referenced.

If you don't want to create a new object and maintain them, another easy and useful solution for this will be to set the "search_path" parameter in PostgreSQL.

The search_path parameter will determine what are the schemas that should be checked when looking for an object.

For example, if the value of search_path is "user1,user2,public" then when I run any statement the PostgreSQL engine will check first if it can find the relevant object under user1, if the object can't be found it will search in user2, if the object can't be found in user2 as well and will check in public and just after all users were checked and the object wasn't found the PostgreSQL engine will return an error saying that the object does not exists.

This can be useful when designing your schemas to create some object in the public schema or create specific schemas to specific objects and add the schemas names to the search_path parameter.

In order to check the variable value, use the command below:

```
show search_path;
```

To change it, just use the command below:

```
SET search_path TO "Samples", user1, public;
```

**Note:** PostgreSQL converts object names to lowercase letters, if one of your schemas to check is not fully lowercase, mark it with apostrophes.

# 9. Copying Data from Oracle Source Databases to Cloud SQL PostgreSQL Targets

This section names several tools that can help with the data movement aspect of migrating Oracle source databases to Cloud SQL for PostgreSQL. Please note that the list provided below is only partial and additional tools which can be used for data movement exists.

## 9.1 Data movement concepts across heterogeneous databases

**Batch data load**
Users can offload data from the source Oracle database and copy it to PostgreSQL as a one-off process. The batch data load process involves copying data from the source database that is accurate to a specific point in time. Consider suspending write activities on the source database until the data transfer completes, as new changes applied to the source database will not be copied to the target.

This approach is mostly useful for smaller databases or database systems where write downtime can be scheduled for the duration of the data copy process.

**Advantages:**

- Considerably simpler process.
- There are plenty of tools which can facilitate batch data loads across heterogeneous database systems, including most ETL tools.
- **In addition to using 3rd party tools, users can facilitate the one-off copy process using scripts.**

**Disadvantages:**

- Only suitable for non-write intensive databases and when down-time is acceptable by the application.
- The throughput to the target database can be a major factor if the source database is more than few TB of data.
- Downtime might be required from the start of the copy process and until the end of the process, as new “writes” applied on the source database will not be copied to the target.

**Available tools:**

- Ora2pg: Free, open source tool. Most frequently used for the schema conversion aspects of heterogeneous database migrations, Ora2pg also supports data copy between Oracle and PostgreSQL databases.

- Open source/free to use ETL tools: These include CloverETL, Talend, and Pentaho Kettle, as well as others.
- Real-time data replication: Also known as CDC (Changed Data Capture) data transfer. Unlike a one-off batch copy of source Oracle data to your target PostgreSQL databases, you can also employ a solution that captures changes applied on the source database and streams them to the target database. These types of solutions are suited for large databases and database where only minimal downtime can be tolerated as part of the data replication process. Products that enable real-time data replication usually work by mining (or extracting) changes that were applied on the source database by constantly reading the source database transaction logs ("redo logs" in the Oracle database), converting these changes to ANSI compliant SQL statements and applying them on the target database. As such, these types of solutions can be used to replicate data across heterogeneous database environments.

**Advantages**:

- Enables real-time replication of data from source databases to target databases essnetially minimizing any downtime that is required during the data copy process as the source and target databases are always kept in sync.
- Suitable for very large databases.

**Disadvantages:**

- Real-time data replication solutions are often more difficult to setup and configure.
- Most of the products that exist in this category are commercial and require purchasing a license.
- Certain prerequisites are required on the source and target databases such as ensuring all tables have primary keys, ensuring all data types are supported, disabling triggers on the target database during replication, etc.

**Some of the available tools include:**

- Oracle GoldenGate: Oracle's own heterogeneous real time data replication solution. Oracle Golden Gate is a product that enables real-time data replication across heterogeneous database targets. Oracle Golden Gate is relatively expensive and requires extensive DBA expertise to setup and configure. Existing Oracle customers may already have Oracle GoldenGate licenses which could be leveraged as part of the migration process. To view the documentation for PostgreSQL GoldenGate configuration, see Fusion Middleware Installing and Configuring Oracle GoldenGate for PostgreSQL.
- Attunity Replicate: Used for real-time data replication across heterogeneous targets. Considered easier to used compared to Oracle GoldenGate. Commercial product.
- HVR: Real-time data replication solution. Commercial product.

**Note:** most real-time data replication software connect to the target database using either ODBC or JDBC connections. Therefore, any data type that cannot be handled by ODBC can't be migrated by this tool.

## 9.2 Schema prerequisites for enabling data replication

There are certain considerations and prerequisites you should be aware of before initiating data copy from the source Oracle database to the target PostgreSQL database. These considerations include:

- Data replication products do not usually create the target schema. These products focus on copying the data from the source database to the target database and usually require that the target database schema have been already created. You will need to use a schema conversion tool (such as ora2pg or migVisor) to convert your Oracle schema to PostgreSQL dialect before initiating the data copy.
- Ensure source tables contain valid primary keys. Most, if not all, CDC data replication tools rely on identifying rows using a logical identifier. Most often this logical identifier is the primary key of the row. Tables without primary keys will require specific attention.
- Ensure that all your source Oracle data types are compatible with the target database. Pay special attention to legacy Oracle data-types such as RAW or LONG.
- When using CDC real-time data replication tools, ensure that your target database contains the same primary key indexes as your source Oracle database, otherwise you will experience performance issues during replication.
- When using CDC real-time data replication tools, be sure to disable triggers and cascading foreign keys on the target database while the replication is enabled to prevent potential data duplication issues.
- Disable autovacuum on PostgreSQL target tables before initiating the data copy. Enable autovacuum once the load is complete. This can improve performance as the autovacuum can interrupt the copy process and incur performance penalties during replication.
-

**Useful PostgreSQL-specific tools and extensions**
pgpool: Load balancer option to distribute queries between more than one PostgreSQL instances.

Basically, it provides a load balancer option to distribute requests between more than one PostgreSQL instances.

DBeaver: Free to use IDE for developing and running SQL code on top of a variety of database engines, including both Oracle and PostgreSQL. You can use DBeaver to query your databases (source or target), run scripts, extract DDL from schema objects, etc.
migVisor: Database migration assessment product that can scan an existing fleet of Oracle or SQL Server source databases and provide a recommended migration path to Google Cloud SQL database targets. migVisor ranks your source databases based on migration complexity, identify potential migration challenges and assist in the schema conversion process.

**Additional post-migration tips**
Run the statement below to ensure that the query planner will work properly with partitions and recognize them

```
SET constraint_exclusion = on;
```

When possible, review your pg/sql code and replicate Oracle's behavior by adding COMMIT before/after DDL statements.

If you are using packages and you want to keep the same logic, you can create a schema that will act as a package, containing all the functions that were related to each package. This strategy will allow you to use the same reference that was used in Oracle database, instead of having packages, you'll use schemas. You can call each function with schema_name(same as the package).

For more about autocommit in PostgreSQL, see Set Autocommit.

In PostgreSQL, collations can be defined at the server and database levels, if you have several Oracle databases with different encoding, think about consolidating these databases to the same Cloud SQL instance to reduce costs.

When loading data to PostgreSQL, make sure that there are no indexes, no constraints, and no enabled triggers.

# 10. Testing and Validating Data Migrations

This topic covers all of the actions that need to be executed in order to ensure that the data migration is consistent and complete.

**Steps for checking data migration**
The most common initial check would be counting the rows for each table, based on your use-case, determine if you need the exact number of rows or an estimation.

**Caution:** These queries can consume a lot of resources, so run them responsibly or not on a production database

To generate the Oracle database query that will determine the number of rows for each table use the query template below:

```
SELECT 'select count(*), ''' || TABLE_NAME || ''' FROM ' || TABLE_NAME || ';'  FROM ALL_TABLES WHERE OWNER
= 'SCHEMA_NAME';
```

**Note:** Remember to change the SCHEMA_NAME to the correct schema name.

The  output from this query will be similar to the examples  below:

```
select count(*), 'EMPS' FROM EMPS;
select count(*), 'DAPS' FROM DAPS;
```

The query above will run every command separately. To order all commands as one query, use the example below:

```
SELECT 'select count(*), ''' || TABLE_NAME || ''' FROM ' || TABLE_NAME || ' UNION '  FROM ALL_TABLES WHERE
owner = 'SCHEMA_NAME';
```

The output from this query will be similar to the examples below

```
select count(*), 'EMPS' FROM EMPS UNION


select count(*), 'DAPS' FROM DAPS UNION
```

Replace the last "UNION" with ";". This generates the result for all tables in the schema at once. The fixed query will be formatted like the following example:

```
select count(*), 'EMPS' FROM EMPS UNION

select count(*), 'DAPS' FROM DAPS;
```

After migrating the data, compare the above results with the number of rows in the PostgreSQL instance.  It will be similar to the method you used in Oracle.

To generate the PostgreSQL query that will determine the number of rows for each table, use the query template below:

```
SELECT 'select count(*), ''' || TABLE_NAME || ''' FROM ' || TABLE_NAME || ';'

from information_schema.tables where table_schema = 'SCHEMA_NAME';
```

**Note:** Remember to change the SCHEMA_NAME to the correct  schema name.

To run all queries as one query, use the query below:

```
SELECT 'select count(*), ''' || TABLE_NAME || ''' FROM ' || TABLE_NAME || ' UNION '

from information_schema.tables where table_schema = SCHEMA_NAME';
```

Additional recommended post-migration checks are:

- Checking data samples to verify that the data is correct—there are no missing values or invalid characters like '?' and others.

- Run several queries that are using Views or Functions and check that the results are consistent for both source and target.
- Check that all objects in the destinations are valid (in PostgreSQL it's mostly relevant for indexes)

**Use the following query to check that all objects in the destinations are valid:**

```
SELECT pg_class.relname FROM pg_class, pg_index

WHERE pg_index.indisvalid = false AND pg_index.indexrelid = pg_class.oid;
```

**Application testing**

Connect your application/s to the CloudSQL for PostgreSQL database, run a full functionality tests and performance tests. Be sure to test all your application's features and flows, including those that are kept for backward compatibility.

If your application is supposed to interact with both Oracle Database and CloudSQL for PostgreSQL database after the migration, check that both of those paths are working and providing consistent results.

When running your tests, don't just test the data retrieving functionality, but also the data modification options in your application, resulting in DML statements on PostgreSQL.

Make sure to run manually your scheduled tasks and scripts to make sure that they still run against your new CloudSQL for PostgreSQL target.

**Commercial and open-source tools**
There aren't many commercial and open-source tools available that allow you to write your own testing scripts.

For commercial options, you can use:
- DTM Data Comparer by sqledit
- DataDiff CrossDB by DataNamic

---

To get started, sign up here for a free 1-hour, in-person guided tour of Google Cloud Platform.

Contact us here for any further questions:
www.Troposphere.tech/contact
support@troposphere.tech
(877) 256-8349

# 11. Glossary

| | |
|---|---|
| **ACID** | Atomicity, Consistency, Isolation, Durability |
| **AES** | Advanced Encryption Standard |
| **ANSI** | American National Standards Institute |
| **API** | Application Programming Interface |
| **BLOB** | Binary Large Object |
| **CLI** | Command Line Interface |
| **CLOB** | Character Large Object |
| **CSV** | Comma Separated Values |
| **CTE** | Common Table Expression |
| **DB** | Database |
| **DDL** | Data Definition Language |
| **DML** | Data Manipulation Language |
| **GCP** | Google Cloud Platform |
| **IP** | Internet Protocol |
| **ISO** | International Organization for Standardization |
| **JSON** | JavaScript Object Notation |
| **PDF** | Portable Document Format |
| **QA** | Quality Assurance |
| **SQL** | Structured Query Language |
| **SSD** | Solid State Disk |
| **TDE** | Transparent Data Encryption |
| **UDF** | User Defined Function |
| **UDT** | User Defined Type |
| **UTC** | Universal Time Coordinated |
| **XML** | Extensible Markup Language |