Oracle Database 11g/12c
To Amazon Aurora with PostgreSQL Compatibility (9.6.x)

Migration Playbook

Version: 1.1, January 2018
 Written by: David Yahalom, Nimrod Keinan
Notices

This document is provided for informational purposes only. It represents AWS’s current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS’s products or services, each of which is provided “as is” without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.
Introduction

To migrate an Oracle database to Amazon Aurora with PostgreSQL Compatibility, you usually need to perform both automated and manual tasks. The automated tasks involve data migration and schema conversion using the AWS Database Migration Service (AWS DMS) and AWS Schema Conversion Tool (AWS SCT). The manual tasks involve post-migration “touch-ups” for certain database objects that can’t be migrated automatically.

This whitepaper primarily focuses on the manual aspects of database migration and includes step-by-step instructions that you can adapt for your own requirements. In this document, we focus on how to manually migrate specific Oracle database objects and features to Amazon Aurora with PostgreSQL Compatibility equivalents. We also include a brief overview that explains how to use the AWS Schema Conversion Tool (AWS SCT) for automatic migrations of schema objects. You can use this document as a supplementary guide for your database migrations – both as a guide to expand your PostgreSQL competency if you come from an Oracle database background and as a reference to help build Oracle-comparable functionality in Amazon Aurora with PostgreSQL compatibility.

This document does not yet cover all Oracle Database features and capabilities from a migration perspective. For the first release, we focus on some of the most important features and will continue to expand the Playbook document over time. Not all Oracle features have direct and fully compatible equivalents in PostgreSQL. In these cases, we present our recommendations for the best-possible equivalent features in Amazon Aurora with PostgreSQL compatibility.

We also plan to expand this document in the future and add new chapters specifically dedicated to advanced topics such as Oracle security, High Availability and Disaster Recovery, Performance Tuning, and more.

Disclaimer

The various code snippets, commands, guides, best practices, and scripts included in this document should be used for reference only and are provided as-is without warranty. Please be sure to test all of the code, commands, best practices, and scripts outlined in this document in a non-production environment first. Amazon and its affiliates are not responsible for any direct or indirect damage that can occur from the information contained in this document.
## Automatic Schema Migration

### Help Section

- [Link] Using the AWS Schema Conversion Tool for automatic schema conversion

### SQL & PL/SQL (Manual)

<table>
<thead>
<tr>
<th>Oracle Feature</th>
<th>Aurora PostgreSQL Feature</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Link] Anonymous Block</td>
<td>Do</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Execute Immediate</td>
<td>Execute &amp; Prepare</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] DBMS_RANDOM</td>
<td>random()</td>
<td>Yes*</td>
</tr>
<tr>
<td>[Link] DBMS_OUTPUT</td>
<td>RAISE</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Procedures &amp; Functions</td>
<td>Functions</td>
<td>Yes*</td>
</tr>
<tr>
<td>[Link] User Defined Functions (UDFs)</td>
<td>Functions</td>
<td>Yes*</td>
</tr>
<tr>
<td>[Link] UTL_FILE</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td>[Link] JSON Document Support</td>
<td>JSON Document Support</td>
<td>Yes*</td>
</tr>
<tr>
<td>[Link] OLAP Functions</td>
<td>Window Functions</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] PL/SQL Cursors</td>
<td>Cursors</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Single Row &amp; Aggregate Functions</td>
<td>Single Row &amp; Aggregate Functions</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Merge</td>
<td>SQL Merge</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Create Table As Select (CTAS)</td>
<td>Create Table As Select (CTAS)</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Common Table Expression (CTE)</td>
<td>Common Table Expression (CTE)</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Insert From Select</td>
<td>Insert From Select</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] Inline Views</td>
<td>Inline Views</td>
<td>Yes</td>
</tr>
<tr>
<td>[Link] DB Hints</td>
<td>Query Planning</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
## Tables & Indexes (Manual)

<table>
<thead>
<tr>
<th>Oracle Feature</th>
<th>Aurora PostgreSQL Feature</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Index Organized Tables (IOTs)</td>
<td>PostgreSQL “Cluster” Tables</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Common Data Types</td>
<td>Common Data Types</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Table Constraints</td>
<td>Table Constraints</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Table Partitioning including: RANGE, LIST, HASH, COMPOSITE, Automatic LIST</td>
<td>Table Partitioning including: RANGE, LIST</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Exchange and Split Partitions</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td>Link Temporary Tables</td>
<td>Temporary Tables</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Unused Columns</td>
<td>ALTER TABLE DROP COLUMN</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Virtual Columns</td>
<td>Views and/or Function as a Column</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link User Defined Types (UDTs)</td>
<td>User Defined Types (UDTs)</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Read Only Tables and Table Partitions</td>
<td>Read Only Roles and/or Triggers</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Index Types</td>
<td>Index Types</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link B-Tree Indexes</td>
<td>B-Tree Indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Composite Indexes</td>
<td>Multi-Column Indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Link BITMAP Indexes</td>
<td>BRIN Indexes</td>
<td>Minimal</td>
</tr>
<tr>
<td>Link Function-Based Indexes</td>
<td>Expression Indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Local and Global Partitioned Indexes</td>
<td>Partitioned Indexes</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Identity Columns</td>
<td>Serial Data Type</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link MVCC (Table and Row Locks)</td>
<td>MVCC (Table and Row Locks)</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Character Sets</td>
<td>Encoding</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Transaction Model</td>
<td>Transactional Model</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link LOBs and SecureFile LOBs</td>
<td>LOBs</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

## Database Objects (Manual)

<table>
<thead>
<tr>
<th>Oracle Feature</th>
<th>Aurora PostgreSQL Feature</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Materialized Views</td>
<td>Materialized Views</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Common Data Types</td>
<td>Common Data Types</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Oracle Triggers</td>
<td>PostgreSQL Trigger Procedure</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Views</td>
<td>Views</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Sequences</td>
<td>Sequences</td>
<td>Yes</td>
</tr>
<tr>
<td>Link</td>
<td>Database Links</td>
<td>PostgreSQL DBLink and FDWrapper</td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>---------------------------------</td>
</tr>
</tbody>
</table>
# Database Administration (Manual)

<table>
<thead>
<tr>
<th>Oracle Feature</th>
<th>Aurora PostgreSQL Feature</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Recovery Manager (RMAN)</td>
<td>Amazon Aurora Snapshots</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Flashback Database</td>
<td>Amazon Aurora Snapshots</td>
<td>Yes</td>
</tr>
<tr>
<td>Link 12c Multi-Tenant Architecture: PDBs and CDB</td>
<td>Databases</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Tablespaces and DataFiles</td>
<td>Tablespaces</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Data Pump</td>
<td>pg_dump and pg_restore</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Resource Manager</td>
<td>Separate Amazon Aurora Clusters</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Database Users</td>
<td>Database Roles</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Database Roles</td>
<td>Database Roles</td>
<td>Yes</td>
</tr>
<tr>
<td>Link SGA &amp; PGA Memory</td>
<td>Memory Buffers</td>
<td>Yes</td>
</tr>
<tr>
<td>Link V$ Views &amp; the Data Dictionary</td>
<td>System Catalog Tables, Statistics Collector, Amazon Aurora Performance Insights</td>
<td>Yes*</td>
</tr>
<tr>
<td>Link Log Miner</td>
<td>Logging Options</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Instance &amp; Database Parameters (SPFILE)</td>
<td>Amazon Aurora Parameter Groups</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Session Parameters</td>
<td>Session Parameters</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Alert.log (error log)</td>
<td>Error Log via AWS Management Console</td>
<td>Yes</td>
</tr>
<tr>
<td>Link Viewing Execution Plans</td>
<td>Viewing Execution Plans</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Automatic Migration of Oracle Schema Objects
Using the AWS Schema Conversion Tool
Automatic Schema Migration

This section provides a step-by-step process for using the AWS Schema Conversion Tool (AWS SCT) to migrate an Oracle database to an Aurora with PostgreSQL compatibility database cluster. Amazon SCT can automatically migrate most of the database objects.

While this document primarily covers the best practices, feature-parity aspects of manual database migrations, and Oracle to Amazon Aurora with PostgreSQL compatibility migration best practices, we recommend using AWS SCT as the first step of the process.

AWS SCT is a downloadable Java utility that runs locally on your computer. It connects to the source and target databases, scans the source database schema objects (tables, views, indexes, procedures, etc.), and converts them to the target database objects.

For more information, see
Download AWS SCT and Install JDBC Drivers
JDBC drivers are required for database connectivity to both the source and target databases.

1. Download SCT:  

2. Download the Oracle JDBC Driver (ojdbc7.jar):  

3. Download the PostgreSQL JDBC Driver (postgresql-9.4-1204.jdbc42.jar):  
   https://jdbc.postgresql.org/download.html

Configure SCT for Database Migration

1. Launch SCT.

2. Choose the JAR files path under SCTs Global Settings

3. Click Global Settings > Drivers

4. Add the file path to the Oracle and the PostgreSQL JDBC drivers

5. Use the following filenames:

   Oracle JDBC JAR - ojdbc7.jar
   PostgreSQL JDBC JAR - postgresql-9.4-1204.jdbc42.jar

6. Click OK.
AWS SCT – Database Migration Project Configuration
Create a new SCT project which will guide you, step-by-step, through the schema conversion process.

1. Click **SCT > File > New Project Wizard**

2. Select a source database for migration (the Oracle Database to migrate to Aurora with PostgreSQL Compatibility).

3. Enter a project name, specify the location of the SCT project files, choose the source database workload characteristics (OLTP or OLAP), and select the source database engine (Oracle).
4. Configure the source database connection properties:
   - Server hostname
   - Oracle Net Listener port number
   - Oracle Database SID
   - Privileged username and password. For example, the Oracle system user.

5. Click **Next**.

6. Select the source Oracle schema for migration.
7. SCT analyzes the source database schema objects and produces the Database Migration Assessment report. Review the report.

8. Specify the target database configuration:
   - **Target Database Engine:** Amazon Aurora (PostgreSQL compatible).
   - **Server hostname:** Aurora *Cluster* Endpoint.
   - **Server port number:** 5432 (default PostgreSQL network port).
   - **Database:** The name of the target database that will store the migrated schema objects.
   - **The privileged target database username and password.** Deployment of the converted schema in the target database will use these credentials.
9. Click **Finish** when you are done. Note that at this stage in the process the migrated target schema **has not** yet been deployed to your target database.

10. Explore the AWS SCT Project Main Page and the other information pages. Select Oracle schema objects from the left Oracle pane to view the Oracle syntax.
AWS SCT – Database Migration Assessment

1. Explore the objects in your source database and be sure to note any database objects that SCT cannot automatically migrate to your target database syntax. SCT flags objects with potential migration issues with a RED exclamation mark. These objects require manual intervention for successful migration.

2. Right click the Oracle schema for migration and select **Create Report** to view the complete Database Migration Assessment report.

3. Click the **Action Items** tab. This section of the report provides information about potential migration issues.
4. Click the migration issues highlighted by SCT to view a detailed overview of the exact source syntax that failed the automatic migration process.
AWS SCT – Convert Source to Target Database Syntax
This step converts the source database schema objects to your target database using target database syntax.

1. Right click the Oracle Schema and select **Convert Schema**.

2. The new schema and objects appear in the right-side pane under the target database. Compare the source database objects (left-side pane) to the converted target database objects (right-side pane). Note that the converted schema has **not yet been deployed** to the target PostgreSQL database.

3. Examine any gaps in objects that AWS SCT could not automatically convert.
AWS SCT – Deploy the Converted Schema to the Target Database

1. In the right-side **Target Database** pane, right-click the PostgreSQL schema corresponding to the source database schema name.

2. Select **Apply to database**.

3. Click **Yes** to continue. **This step creates the new schema in the target database.**
Manual Migration and Best Practices of Oracle Schema Objects and Database Features
Overview
Oracle’s PL/SQL is a procedural extension of SQL. The PL/SQL program structure divides the code into blocks that can be distinguished by the following keywords: DECLARE, BEGIN, EXCEPTION, and END.

An unnamed PL/SQL code block (code not stored in the database as a procedure, function, or package) is known as an anonymous block. An anonymous block serves as the basic unit of Oracle PL/SQL and contains the following code sections:

- **The Declarative Section** (Optional)
  Contains variables (names, data types, and initial values).

- **The Executable Section** (Mandatory)
  Contains executable statements (each block structure must contain at least one executable PL/SQL statement).

- **The Exception-Handling Section** (Optional)
  Contains elements for handling exceptions or errors in the code.

Examples
Simple structure of an Oracle Anonymous Block:

```sql
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
   DBMS_OUTPUT.PUT_LINE('hello world');
END;
/
hello world
```

PL/SQL procedure successfully completed.
Oracle PL/SQL Anonymous blocks can contain advanced code elements such as functions, cursors, dynamic SQL, and conditional logic. The following anonymous block uses a cursor, conditional logic, and exception-handling:

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2  v_sal_chk        NUMBER;
  3  v_emp_work_years NUMBER;
  4  v_sql_cmd        VARCHAR2(2000);
  5  BEGIN
  6      FOR v IN (SELECT EMPLOYEE_ID, FIRST_NAME||' '||LAST_NAME AS EMP_NAME, HIRE_DATE, SALARY FROM EMPLOYEES)
  7          LOOP
  8              v_emp_work_years:=EXTRACT(YEAR FROM SYSDATE) - EXTRACT (YEAR FROM v.hire_date);
  9              IF v_emp_work_years>=10 and v.salary <= 6000 then
 10                 DBMS_OUTPUT.PUT_LINE('Consider a Bonus for: '||v.emp_name);
 11              END IF;
 12          END LOOP;
 13  EXCEPTION WHEN OTHERS THEN
 14      DBMS_OUTPUT.PUT_LINE('CODE ERR: '||sqlerrm);
 15  END;
 16
```

The above example calculates the years each employee has worked based on the `HIRE_DATE` column of the `EMPLOYEES` table. If the employee has worked for ten or more years and has a salary of $6000 or less, the system prints the message “Consider a Bonus for: <employee name>”.

For additional details:
https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/controlstructures.htm#CJAEDEIH
Migration to: PostgreSQL DO

Overview
PostgreSQL version 9.6 supports capabilities similar to Oracle’s anonymous blocks. In PostgreSQL, you can execute PL/pgSQL code that is not stored in the database as an independent code segment using a PL/pgSQL DO statement.

PL/pgSQL is a PostgreSQL extension to the ANSI SQL and has many similar elements to Oracle PL/SQL. PostgreSQL DO uses a similar code structure to an Oracle anonymous block:

- **Declarative Section** (Optional)
- **Executable Section** (Mandatory)
- **Exception-Handling Section** (Optional)

Examples
PostgreSQL DO simple structure:

```sql
psql=> SET CLIENT_MIN_MESSAGES = 'debug';
-- Equivalent To Oracle SET SERVEROUTPUT ON

psql=> DO $$
BEGIN
  RAISE DEBUG USING MESSAGE := 'hello world';
END $$;

DEBUG: hello world
```

The PostgreSQL PL/pgSQL DO statement supports the use of advanced code elements such as functions, cursors, dynamic SQL, and conditional logic.
The following example is a more complex PL/pgSQL DO code structure converted from Oracle’s “employee bonus” PL/SQL anonymous block example presented in the previous section:

```
psql=> DO $$
DECLARE
    v_sal_chk DOUBLE PRECISION;
    v_emp_work_years DOUBLE PRECISION;
    v_sql_cmd CHARACTER VARYING(2000);
    v RECORD;
BEGIN
    FOR v IN
    SELECT employee_id, CONCAT_WS('', first_name, ' ', last_name) AS emp_name, hire_date, salary
    FROM employees
    LOOP
        v_emp_work_years := EXTRACT (YEAR FROM now()) - EXTRACT (YEAR FROM v.hire_date);
        IF v_emp_work_years >= 10 AND v.salary <= 6000 THEN
            RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'Consider a Salary Raise for: ', v.emp_name);
        END IF;
    END LOOP;
EXCEPTION
WHEN others THEN
    RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'CODE ERR: ', SQLERRM);
END $$;
```

For additional information on PostgreSQL DO:
https://www.postgresql.org/docs/current/static/sql-do.html
Migrating from: Oracle EXECUTE IMMEDIATE

Overview
Oracle’s EXECUTE IMMEDIATE statement can be used to parse and execute a dynamic SQL statement or an anonymous PL/SQL block. It also supports bind variables.

Example
Run a dynamic SQL statement from within a PL/SQL procedure:

1. Create a PL/SQL procedure named `raise_sal`.

2. Define a SQL Statement with a dynamic value for the column name included in the where statement.

3. Use the EXECUTE IMMEDIATE command supplying the two bind variables to be used as part of the SELECT statement:
   - `amount`
   - `col_val`

   ```plsql
   CREATE OR REPLACE PROCEDURE raise_sal (col_val NUMBER, emp_col VARCHAR2, amount NUMBER) IS
     col_name VARCHAR2(30);
     sql_stmt VARCHAR2(350);
   BEGIN
     -- determine if a valid column name has been given as input
     SELECT COLUMN_NAME INTO col_name FROM USER_TAB_COLS
     WHERE TABLE_NAME = 'EMPLOYEES' AND COLUMN_NAME = emp_col;
     -- define the SQL statement (with bind variables)
     sql_stmt := 'UPDATE employees SET salary = salary + :1 WHERE ' || col_name || ' = :2';
     -- Execute the command
     EXECUTE IMMEDIATE sql_stmt USING amount, col_val;
   END raise_sal;
   /
   ```

4. Run the DDL operation from within an EXECUTE IMMEDIATE command:

   ```plsql
   EXECUTE IMMEDIATE 'CREATE TABLE link_emp (idemp1 NUMBER, idemp2 NUMBER)';
   EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
   ```

5. Run an anonymous block with bind variables using EXECUTE IMMEDIATE:

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

For additional details:
https://docs.oracle.com/database/121/LNPLS/dynamic.htm#LNPLS01115
Overview
The PostgreSQL EXECUTE command prepares and executes commands dynamically. The EXECUTE command can also run DDL statements and retrieve data using SQL commands. Similar to Oracle, the PostgreSQL EXECUTE command can be used with bind variables.

Example
1. Execute a SQL SELECT query with the table name as a dynamic variable using bind variables. This query returns the number of employees under a manager with a specific ID.

```sql
DO $$DECLARE
    Tabname  varchar(30) := 'employees';
    num      integer := 1;
    cnt      integer;
BEGIN
    EXECUTE format('SELECT count(*) FROM %I WHERE manager = $1', tabname)
        INTO cnt
    USING num;
    RAISE NOTICE 'Count is % int table %', cnt, tabname;
END$$;
```

2. Execute a DML command – first with no variables and then with variables:

```sql
DO $$DECLARE
BEGIN
    EXECUTE 'INSERT INTO numbers (a) VALUES (1)';
    EXECUTE format('INSERT INTO numbers (a) VALUES (%s)', 42);
END$$;
```

Notes
- `%s` formats the argument value as a simple string. A null value is treated as an empty string.
- `%I` treats the argument value as an SQL identifier and double-quotings it if necessary. It is an error for the value to be null.

3. Execute a DDL command:

```sql
DO $$DECLARE
BEGIN
    EXECUTE 'CREATE TABLE numbers (num integer)';
END$$;
```

For additional details:
https://www.postgresql.org/docs/9.3/static/functions-string.html
**PostgreSQL Prepare**

Using a `PREPARE` statement can improve performance for reusable SQL statements. The `PREPARE` command can receive a `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `VALUES` statement and parse it with a user-specified qualifying name so the `EXECUTE` command can be used later without the need to re-parse the SQL statement on each execution.

- When using `PREPARE` to create a prepared statement, it will be viable for the scope of the current session.
- If a DDL command is executed on a database object referenced by the prepared SQL statement, the next `EXECUTE` command requires a hard parse of the SQL statement.

**Example**

Use `PREPARE` and `EXECUTE` commands in tandem:

The SQL command is prepared with a user-specified qualifying name.

The SQL command is executed several times, without the need for re-parsing.

```sql
PREPARE numplan (int, text, bool) AS
    INSERT INTO numbers VALUES($1, $2, $3);
EXECUTE numplan(100, 'New number 100', 't');
EXECUTE numplan(101, 'New number 101', 't');
EXECUTE numplan(102, 'New number 102', 'f');
EXECUTE numplan(103, 'New number 103', 't');
```

**PL/pgSQL EXECUTE vs. Oracle implicit cursor**

<table>
<thead>
<tr>
<th>Functionality</th>
<th>PostgreSQL - EXECUTE</th>
<th>Oracle – EXECUTE IMMEDIATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute SQL with results and bind variables</td>
<td>EXECUTE format('select salary from employees WHERE %I = $1', col_name) INTO amount USING col_val;</td>
<td>EXECUTE IMMEDIATE 'select salary from employees WHERE '</td>
</tr>
<tr>
<td>Execute DML with variables and bind variables</td>
<td>EXECUTE format('UPDATE employees SET salary = salary + $1 WHERE %I = $2', col_name) USING amount, col_val;</td>
<td>EXECUTE IMMEDIATE 'UPDATE employees SET salary = salary + :1 WHERE '</td>
</tr>
<tr>
<td>Execute DDL</td>
<td>EXECUTE 'CREATE TABLE link_emp (idemp1 integer, idemp2 integer)';</td>
<td>EXECUTE IMMEDIATE 'CREATE TABLE link_emp (idemp1 NUMBER, idemp2 NUMBER)';</td>
</tr>
<tr>
<td>Execute Anonymous block</td>
<td>DO $$DECLARE BEGIN ... END$$;</td>
<td>EXECUTE IMMEDIATE 'BEGIN DBMS_OUTPUT.PUT_LINE(‘Anonymous Block’); END;';</td>
</tr>
</tbody>
</table>

For additional details:
[https://www.postgresql.org/docs/current/static/plpgsql-statements.html](https://www.postgresql.org/docs/current/static/plpgsql-statements.html)
Overview
Oracle’s DBMS_RANDOM package enables you to generate a random number or string as part of a SQL statement or PL/SQL procedure.

**DBMS_RANDOM Package Stored Procedures include:**
1. **NORMAL** – returns random numbers in a standard normal distribution.
2. **SEED** – resets the seed that generates random numbers or strings.
3. **STRING** – returns a random string.
4. **VALUE** – returns a number that is greater than or equal to 0 and less than 1 with 38 digits to the right of the decimal. Alternatively, you could get a random Oracle number that is greater than or equal to a low parameter and less than a high parameter.

**Notes:**
- DBMS_RANDOM.RANDOM produces integers in \([-2^{31}, 2^{31})\).
- DBMS_RANDOM.VALUE produces numbers in \([0,1]\) with 38 digits of precision.

**Example**
1. Generate a random number:

```sql
SQL> select dbms_random.value() from dual;

DBMS_RANDOM.VALUE()
-------------------
.859251508

SQL> select dbms_random.value() from dual;

DBMS_RANDOM.VALUE()
-------------------
.364792387
```
2. Generate a random string. The **first character** determines the returned string type and the **number** specifies the length:

```
SQL> select dbms_random.string('p',10) from dual;
```

```
DBMS_RANDOM.STRING('P',10)
------------------------------------------
la'?z[Q&/2
```

```
SQL> select dbms_random.string('p',10) from dual;
```

```
DBMS_RANDOM.STRING('P',10)
------------------------------------------
t?!Gf2M60q
```

*For additional details:*
https://docs.oracle.com/database/121/ARPLS/d_random.htm
Migration To: PostgreSQL random()

Overview
PostgreSQL does not provide a dedicated package equivalent to Oracle DBMS_RANDOM – a 1:1 migration is not possible. However, other PostgreSQL functions can be used as workarounds under certain conditions. For example, generating random numbers can be performed using the random() function. For generating random strings, you can use the value returned from the random() function coupled with an md5() function.

Example
1. Generate a random number:

```
mydb=> select random();
random
-------------------
0.866594325285405
(1 row)
mydb=> select random();
random
-------------------
0.524613124784082
(1 row)
```

2. Generate a random string:

```
mydb=> select md5(random()::text);
md5
----------------------------------
f83e73114eccfed571b43777b99e0795
(1 row)
mydb=> select md5(random()::text);
md5
----------------------------------
d46de3ce24a99d5761bb34bfb6579848
(1 row)
```
### Oracle dbms_random vs. PostgreSQL random()

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate a random number</td>
<td><code>select dbms_random.value()</code></td>
<td><code>select random()</code></td>
</tr>
<tr>
<td></td>
<td>from dual;</td>
<td></td>
</tr>
<tr>
<td>Generate a random number between 1 to 100</td>
<td><code>select dbms_random.value(1,100)</code></td>
<td><code>select random()*100;</code></td>
</tr>
<tr>
<td></td>
<td>from dual;</td>
<td></td>
</tr>
<tr>
<td>Generate a random string</td>
<td><code>select dbms_random.string('p',10)</code></td>
<td><code>select md5(random()::text);</code></td>
</tr>
<tr>
<td></td>
<td>from dual;</td>
<td></td>
</tr>
<tr>
<td>Generate a random string in upper case</td>
<td><code>select dbms_random.string('U',10)</code></td>
<td><code>select upper(md5(random()::text));</code></td>
</tr>
<tr>
<td></td>
<td>from dual;</td>
<td></td>
</tr>
</tbody>
</table>

For additional details:
- [https://www.postgresql.org/docs/current/static/functions-math.html](https://www.postgresql.org/docs/current/static/functions-math.html)
- [https://www.postgresql.org/docs/current/static/functions-string.html](https://www.postgresql.org/docs/current/static/functions-string.html)
Overview

Oracle’s DBMS_OUTPUT package is typically used for debugging or for displaying output messages from PL/SQL procedures.

Example

In the following example, DBMS_OUTPUT with PUT_LINE is used with a combination of bind variables to dynamically construct a string and print a notification to the screen from within an Oracle PL/SQL procedure.

In order to display notifications on to the screen, you must configure the session with `SET SERVEROUTPUT ON`.

```sql
SET SERVEROUTPUT ON
DECLARE
    CURSOR c1 IS
        SELECT last_name, job_id FROM employees
        WHERE REGEXP_LIKE (job_id, 'S\[HT\]_CLERK')
        ORDER BY last_name;
    v_lastname employees.last_name%TYPE;  -- variable to store last_name
    v_jobid     employees.job_id%TYPE;     -- variable to store job_id
BEGIN
    OPEN c1;
    LOOP  -- Fetches 2 columns into variables
        FETCH c1 INTO v_lastname, v_jobid;
        DBMS_OUTPUT.PUT_LINE ('The employee id is:' || v_jobid || ' and his last name is:' || v_lastname);
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
    CLOSE c1;
END;
```

In addition to the output of information on the screen, the PUT and PUT_LINE procedures in the DBMS_OUTPUT package enable you to place information in a buffer that can be read later by another PL/SQL procedure or package. You can display the previously buffered information using the GET_LINE and GET_LINES procedures.

For additional details:
https://docs.oracle.com/database/121/ARPLS/d_output.htm#ARPLS036
**Overview**

The PostgreSQL RAISE statement can be used as an alternative to `DBMS_OUTPUT`. You can combine RAISE with several levels of severity including:

<table>
<thead>
<tr>
<th>Severity</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG1..DEBUG5</td>
<td>Provides successively-more-detailed information for use by developers.</td>
</tr>
<tr>
<td>INFO</td>
<td>Provides information implicitly requested by the user</td>
</tr>
<tr>
<td>NOTICE</td>
<td>Provides information that might be helpful to users</td>
</tr>
<tr>
<td>WARNING</td>
<td>Provides warnings of likely problems</td>
</tr>
<tr>
<td>ERROR</td>
<td>Reports an error that caused the current command to abort.</td>
</tr>
<tr>
<td>LOG</td>
<td>Reports information of interest to administrators, e.g., checkpoint activity.</td>
</tr>
<tr>
<td>FATAL</td>
<td>Reports an error that caused the current session to abort.</td>
</tr>
<tr>
<td>PANIC</td>
<td>Reports an error that caused all database sessions to abort.</td>
</tr>
</tbody>
</table>

**Examples**

1. **Use RAISE DEBUG** (where DEBUG is the configurable severity level) for similar functionality as Oracle’s `DBMS_OUTPUT.PUT_LINE` feature.

   ```
   psql=> SET CLIENT_MIN_MESSAGES = 'debug';
   -- Equivalent To Oracle SET SERVEROUTPUT ON
   psql=> DO $$
   BEGIN
   RAISE DEBUG USING MESSAGE := 'hello world';
   END $$;
   DEBUG:  hello world
   DO
   ```

2. **Use the client_min_messages parameter** to control the level of message sent to the client. The default is NOTICE. **Use the log_min_messages parameter** to control which message levels are written to the server log. The default is WARNING.

   ```
   SET CLIENT_MIN_MESSAGES = 'debug';
   ```

For additional details:

- [https://www.postgresql.org/docs/current/static/plpgsql-errors-and-messages.html](https://www.postgresql.org/docs/current/static/plpgsql-errors-and-messages.html)
- [https://www.postgresql.org/docs/current/static/runtime-config-logging.html#GUC-LOG-MIN-MESSAGES](https://www.postgresql.org/docs/current/static/runtime-config-logging.html#GUC-LOG-MIN-MESSAGES)
## Oracle DBMS_OUTPUT.PUT_LINE vs. PostgreSQL RAISE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disables message output</td>
<td>DISABLE</td>
<td>Configure &quot;client_min_message&quot; or &quot;log_min_message&quot; for the desired results</td>
</tr>
<tr>
<td>Enables message output</td>
<td>ENABLE</td>
<td>Consider storing messages in an array or temporary table so that you can retrieve them from another procedure or package</td>
</tr>
<tr>
<td>Retrieves one line from buffer</td>
<td>GET_LINE</td>
<td>Store and concatenate the message string in a varchar variable before raising</td>
</tr>
<tr>
<td>Retrieves an array of lines from buffer</td>
<td>GET_LINES</td>
<td></td>
</tr>
<tr>
<td>Terminates a line created with PUT</td>
<td>PUT + NEW_LINE</td>
<td></td>
</tr>
<tr>
<td>Places a partial line in the buffer</td>
<td>PUT_LINE</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------</td>
<td></td>
</tr>
<tr>
<td>Places line in buffer</td>
<td>DBMS_OUTPUT.PUT_LINE('1,');</td>
<td>RAISE NOTICE 'The employee id is: % and his last name is: %', v_jobid, v_lastname;</td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.PUT('2,');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.PUT('3,');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.PUT('4');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBMS_OUTPUT.NEW_LINE();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Returns the number code of the most recent exception</td>
<td>SQLCODE + SQLERRM</td>
<td>SQLSTATE + SQLERRM</td>
</tr>
<tr>
<td>Returns the error message associated with its error-number argument.</td>
<td>DECLARE Name employees.last_name%TYPE; BEGIN SELECT last_name INTO name FROM employees WHERE employee_id = -1; EXCEPTION WHEN OTHERS then DBMS_OUTPUT.PUT_LINE('Error code '</td>
<td></td>
</tr>
</tbody>
</table>

For additional details:  
Overview
Oracle PL/SQL is Oracle’s built-in database programming language providing several methods to store and execute reusable business logic from within the database. Procedures and functions are reusable snippets of code created using the `CREATE PROCEDURE` and the `CREATE FUNCTION` statements.

Stored Procedures and Stored Functions are PL/SQL units of code consisting of SQL and PL/SQL statements that solve specific problems or perform a set of related tasks.

- Procedure – used to perform database actions with PL/SQL.
- Function – used to perform a calculation and return a result.

Privileges for Creating Procedures and Functions

- To create procedures and functions in their own schema, Oracle database users must have the `CREATE PROCEDURE` system privilege.
- To create procedures or functions in other schemas, the database user must have the `CREATE ANY PROCEDURE` privilege.
- To execute a procedure or function, the database user must have the `EXECUTE` privilege.

Package and Package Body
In addition to stored procedures and functions, Oracle also provides “Packages” that encapsulate related procedures, functions, and other program objects.

- Package: declares and describes all the related PL/SQL elements.
- Package body: contains the executable code.

To execute a stored procedure or function created inside a package, the `package name` and the `stored procedure or function` name must be specified.

```sql
SQL> EXEC PKG_EMP.CALCULTE_SAL('100');
```
Examples

1. Create an Oracle stored procedure using the `CREATE OR REPLACE PROCEDURE` statement. The optional `OR REPLACE` clause overwrites an existing stored procedure with the same name, if exists.

```
SQL> CREATE OR REPLACE PROCEDURE EMP_SAL_RAISE
           (P_EMP_ID IN NUMBER, SAL_RAISE IN NUMBER)
           AS
              V_EMP_CURRENT_SAL NUMBER;
           BEGIN
               SELECT SALARY INTO V_EMP_CURRENT_SAL FROM EMPLOYEES WHERE
               EMPLOYEE_ID=P_EMP_ID;

               UPDATE EMPLOYEES
               SET SALARY=V_EMP_CURRENT_SAL+SAL_RAISE
               WHERE EMPLOYEE_ID=P_EMP_ID;

               DBMS_OUTPUT.PUT_LINE('New Salary For Employee ID: '||P_EMP_ID||' Is
               '||(V_EMP_CURRENT_SAL+SAL_RAISE));

               EXCEPTION WHEN OTHERS THEN
                   RAISE_APPLICATION_ERROR(-20001,'An error was encountered -
                   '||SQLCODE||' -ERROR- '||SQLERRM);
                   ROLLBACK;
               COMMIT;
           END;
/
-- Execute
SQL> EXEC EMP_SAL_RAISE(200, 1000);
```
2. **Create a function using the `CREATE OR REPLACE FUNCTION` statement:**

```sql
SQL> CREATE OR REPLACE FUNCTION EMP_PERIOD_OF_SERVICE_YEAR
  (P_EMP_ID NUMBER)
  RETURN NUMBER
AS
  V_PERIOD_OF_SERVICE_YEARS NUMBER;
BEGIN
  SELECT EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM TO_DATE(HIRE_DATE))
  INTO V_PERIOD_OF_SERVICE_YEARS
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID=P_EMP_ID;
  RETURN V_PERIOD_OF_SERVICE_YEARS;
END;
/
```

```sql
SQL> SELECT EMPLOYEE_ID,
       FIRST_NAME,
       EMP_PERIOD_OF_SERVICE_YEAR(EMPLOYEE_ID) AS PERIOD_OF_SERVICE_YEAR
FROM EMPLOYEES;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>PERIOD_OF_SERVICE_YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>174</td>
<td>Ellen</td>
<td>13</td>
</tr>
<tr>
<td>166</td>
<td>Sundar</td>
<td>9</td>
</tr>
<tr>
<td>130</td>
<td>Mozhe</td>
<td>12</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>12</td>
</tr>
<tr>
<td>204</td>
<td>Hermann</td>
<td>15</td>
</tr>
<tr>
<td>116</td>
<td>Shelli</td>
<td>12</td>
</tr>
<tr>
<td>167</td>
<td>Amit</td>
<td>9</td>
</tr>
<tr>
<td>172</td>
<td>Elizabeth</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. **Create a Package using the `CREATE OR REPLACE PACKAGE` statement:**

```sql
SQL> CREATE OR REPLACE PACKAGE PCK_CHINOOK_REPORTS
AS
  PROCEDURE GET_ARTIST_BY_ALBUM(P_ARTIST_ID ALBUM.TITLE%TYPE);
  PROCEDURE CUST_INVOICE_BY_YEAR_ANALYZE;
END;
```
4. Create a new Package using the `CREATE OR REPLACE PACKAGE BODY` statement:

```sql
CREATE OR REPLACE PACKAGE BODY PCK_CHINOOK_REPORTS
AS
  PROCEDURE GET_ARTIST_BY_ALBUM(P_ARTIST_ID ALBUM.TITLE%TYPE) IS
    V_ARTIST_NAME ARTIST.NAME%TYPE;
    BEGIN
      SELECT ART.NAME INTO V_ARTIST_NAME
      FROM ALBUM ALB JOIN ARTIST ART USING(ARTISTID)
      WHERE ALB.TITLE=P_ARTIST_ID;
      DBMS_OUTPUT.PUT_LINE('ArtistName: '||V_ARTIST_NAME);
    END;

  PROCEDURE CUST_INVOICE_BY_YEAR_ANALYZE AS
    V_CUST_GENRES VARCHAR2(200);
    BEGIN
      FOR V IN(SELECT CUSTOMERID, CUSTNAME, LOW_YEAR, HIGH_YEAR, CUST_AVG FROM TMP_CUST_INVOICE_ANALYSE)
      LOOP
        IF SUBSTR(V.LOW_YEAR, -4) > SUBSTR(V.HIGH_YEAR , -4) THEN
          SELECT LISTAGG(GENRE, ',') WITHIN GROUP (ORDER BY GENRE) INTO V_CUST_GENRES
          FROM (SELECT DISTINCT
              FUNC_GENRE_BY_ID(TRC.GENREID) AS GENRE
              FROM TMP_CUST_INVOICE_ANALYSE TMPTBL JOIN INVOICE INV
              USING(CUSTOMERID)
              JOIN INVOICELINE INVLIN
              ON INV.INVOICEID = INVLIN.INVOICEID
              JOIN TRACK TRC
              ON TRC.TRACKID = INVLIN.TRACKID
              WHERE CUSTOMERID=V.CUSTOMERID);

          DBMS_OUTPUT.PUT_LINE('Customer: '||UPPER(V.CUSTNAME)||' - Offer a Discount According To Preferred Genres: '||UPPER(V_CUST_GENRES));
          END IF;
        END LOOP;
      END;
    END;
END;
```

The above examples demonstrate basic Oracle PL/SQL procedure and function capabilities. Oracle PL/SQL provides a vast number of features and capabilities that are not within the scope of this document.

For additional details:
- [https://docs.oracle.com/cd/E18283_01/appdev.112/e17126/create_procedure.htm](https://docs.oracle.com/cd/E18283_01/appdev.112/e17126/create_procedure.htm)
- [https://docs.oracle.com/database/121/LNPLS/create_procedure.htm](https://docs.oracle.com/database/121/LNPLS/create_procedure.htm)
- [https://docs.oracle.com/cd/E18283_01/appdev.112/e17126/create_function.htm](https://docs.oracle.com/cd/E18283_01/appdev.112/e17126/create_function.htm)
- [https://docs.oracle.com/database/121/LNPLS/create_function.htm#LNPLS01370](https://docs.oracle.com/database/121/LNPLS/create_function.htm#LNPLS01370)
- [https://docs.oracle.com/database/121/LNPLS/create_package.htm#LNPLS01371](https://docs.oracle.com/database/121/LNPLS/create_package.htm#LNPLS01371)
Migration to: PostgreSQL Functions

Overview
PostgreSQL version 9.6 provides support for both stored procedures and stored functions using the `CREATE FUNCTION` statement. To emphasize, the procedural statements used by PostgreSQL version 9.6 support the `CREATE FUNCTION` statement only. The `CREATE PROCEDURE` statement is not compatible with this PostgreSQL version.

**PL/pgSQL** is the main database programming language used for migrating from Oracle’s PL/SQL code. PostgreSQL support additional programming languages, also available in Amazon Aurora PostgreSQL:

- PL/pgSQL
- PL/Tcl
- PL/Perl

Use the `psql=> show.rds.extensions` command to view all available extensions for Amazon Aurora.

Interchangeability Between Oracle PL/SQL and PostgreSQL PL/pgSQL
PostgreSQL’s PL/pgSQL language is often considered the ideal candidate to migrate from Oracle’s PL/SQL code because many of the Oracle PL/SQL syntax elements are supported by PostgreSQL PL/pgSQL code.

For example, Oracle’s `CREATE OR REPLACE PROCEDURE` statement is supported by PostgreSQL PL/pgSQL. Many other PL/SQL syntax elements are also supported making PostgreSQL and PL/pgSQL natural alternatives when migrating from Oracle.

PostgreSQL create function privileges
To create a function, a user must have USAGE privilege on the language. When creating a function, a language parameter can be specified as shown in the examples.

Examples
Converting Oracle Stored Procedures and Functions to PostgreSQL PL/pgSQL:

1. Use the PostgreSQL `CREATE FUNCTION` command to create a new function named `FUNC_ALG`:

   ```sql
   psql=> CREATE OR REPLACE FUNCTION FUNC_ALG(P_NUM NUMERIC)
   RETURNS NUMERIC
   AS $$
   BEGIN
   RETURN P_NUM * 2;
   END; $$
   LANGUAGE PLPGSQL;
   ```

   - Using a `CREATE OR REPLACE` statement creates a new function, or replaces an existing function, with these limitations:
     - You cannot change the function name or argument types.
     - The statement does not allow changing the existing function return type.
     - The user must own the function to replace it.
• **INPUT parameter** *(P_NUM)* is implemented similarly to Oracle’s PL/SQL INPUT parameter.
• The $$ signs are used to prevent the need to use single-quoted string escape elements. With the $$ sign, there is no need to use escape characters in the code when using single quotation marks (’). The $$ sign appears after the keyword AS and after the function keyword END.
• Use the **LANGUAGE PLPGSQL** parameter to specify the language for the created function.

2. Convert the Oracle **EMP_SAL_RAISE** PL/SQL function to PostgreSQL PL/pgSQL:

```sql
psql=> CREATE OR REPLACE FUNCTION EMP_SAL_RAISE
(IN P_EMP_ID DOUBLE PRECISION, IN SAL_RAISE DOUBLE PRECISION)
RETURNS VOID
AS $$
DECLARE
  V_EMP_CURRENT_SAL DOUBLE PRECISION;
BEGIN
  SELECT SALARY INTO STRICT V_EMP_CURRENT_SAL
  FROM EMPLOYEES WHERE EMPLOYEE_ID = P_EMP_ID;
  UPDATE EMPLOYEES
  SET SALARY = V_EMP_CURRENT_SAL + SAL_RAISE
  WHERE EMPLOYEE_ID = P_EMP_ID;
  RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'NEW SALARY FOR EMPLOYEE ID: ', P_EMP_ID, ' IS '
  , (V_EMP_CURRENT_SAL + SAL_RAISE));
EXCEPTION
  WHEN OTHERS THEN
    RAISE USING ERRCODE := '20001', MESSAGE :=
    CONCAT_WS('','AN ERROR WAS ENCOUNTERED - ', SQLSTATE, ' -ERROR- '
    , SQLERRM);
END; $$
LANGUAGE PLPGSQL;
```

```sql
psql=> select emp_sal_raise(200, 1000);
```

3. Convert the Oracle **EMP_PERIOD_OF_SERVICE_YEAR** PL/SQL function to PostgreSQL PL/pgSQL:

```sql
psql=> CREATE OR REPLACE FUNCTION EMP_PERIOD_OF_SERVICE_YEAR
(IN P_EMP_ID DOUBLE PRECISION)
RETURNS DOUBLE PRECISION
AS $$
DECLARE
  V_PERIOD_OF_SERVICE_YEARS DOUBLE PRECISION;
BEGIN
  SELECT
    EXTRACT (YEAR FROM NOW()) - EXTRACT (YEAR FROM (HIRE_DATE))
  INTO STRICT V_PERIOD_OF_SERVICE_YEARS
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = P_EMP_ID;
  RETURN V_PERIOD_OF_SERVICE_YEARS;
END; $$
LANGUAGE PLPGSQL;
```

```sql
psql=> SELECT EMPLOYEE_ID,
     FIRST_NAME,
     EMP_PERIOD_OF_SERVICE_YEAR(EMPLOYEE_ID) AS
      PERIOD_OF_SERVICE_YEAR
FROM EMPLOYEES;
```
Oracle Packages and Package Bodies

PostgreSQL version 9.6 does not support Oracle Packages and Package Bodies. All PL/SQL objects must be converted to PostgreSQL functions. The following examples describe how the Amazon Schema Conversion Tool (SCT) handles Oracle Packages and Package Body names:

Oracle:
- Package Name: PCK_CHINOOK_REPORTS
- Package Body: GET_ARTIST_BY_ALBUM

Oracle:
```
SQL> EXEC PCK_CHINOOK_REPORTS.GET_ARTIST_BY_ALBUM('');
```

PostgreSQL (converted using Amazon SCT):
- The $ sign separates the package and the package name.

PostgreSQL (converted using Amazon SCT):
```
psql=> SELECT PCK_CHINOOK_REPORTS$GET_ARTIST_BY_ALBUM('');
```

Examples

Convert an Oracle Package and Package Body to PostgreSQL PL/pgSQL:

1. Oracle Package - PCK_CHINOOK_REPORT, Oracle Package Body - GET_ARTIST_BY_ALBUM:

   ```
   psql=> CREATE OR REPLACE FUNCTION chinook."PCK_CHINOOK_REPORTS$GET_ARTIST_BY_ALBUM"
   (p_artist_id text)
   RETURNS void
   LANGUAGE plpgsql
   AS $function$
   DECLARE
       V_ARTIST_NAME CHINOOK.ARTIST.NAME%TYPE;
   BEGIN
       SELECT
           art.name
       INTO STRICT V_ARTIST_NAME
       FROM chinook.album AS alb
       JOIN chinook.artist AS art
       USING (artistid)
       WHERE alb.title = p_artist_id;
       RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'ArtistName: ', V_ARTIST_NAME);
   END;
$function$;
   -- Procedures (Packages) Verification
   psql=> set client_min_messages = 'debug';
   -- Equivalent to Oracle SET SERVEROUTPUT ON
   psql=> select chinook.pck_chinook_reports$get_artist_by_album('Fireball');
   ```
2. Oracle Package - PCK_CHINOOK_REPORTS, Oracle Package Body - CUST_INVOICE_BY_YEAR_ANALYZE:

```sql
psql=> CREATE OR REPLACE FUNCTION
    chinook."pck_chinook_reports$cust_invoice_by_year_analyze"()
RETURNS void
LANGUAGE plpgsql
AS $function$
DECLARE
  v_cust_genres CHARACTER VARYING(200);
  v RECORD;
BEGIN
  FOR v IN
    SELECT
      customerid, custname, low_year, high_year, cust_avg
    FROM chinook.tmp_cust_invoice_analyse
    LOOP
    IF SUBSTR(v.low_year, - 4) > SUBSTR(v.high_year, - 4) THEN
      -- Altering Oracle LISTAGG Function With PostgreSQL STRING_AGG
      Function
      select string_agg(genre, ',') into v_cust_genres
      from (select distinct
                chinook.func_genre_by_id(trc.genreid)
               as genre
               from chinook.tmp_cust_invoice_analyse tmptbl
               join chinook.INVOICE inv using(customerid)
               join chinook.INVOICELINE invlin
               on inv.invoiceid = invlin.invoiceid
               join chinook.TRACK trc
               on trc.trackid = invlin.trackid
               where customerid=v.CUSTOMERID) a;
      -- PostgreSQL Equivalent To Oracle DBMS_OUTPUT.PUT_LINE()
      RAISE DEBUG USING MESSAGE := CONCAT_WS('', 'Customer: ',
      UPPER(v.custname), ' - Offer a Discount According To Preferred
      Genres: ', UPPER(v_cust_genres));
    END IF;
  END LOOP;
$function$;
-- Executing
psql=> SELECT chinook.pck_chinook_reports$cust_invoice_by_year_analyze();
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createfunction.html
https://www.postgresql.org/docs/9.6/static/plpgsql.html
https://www.postgresql.org/docs/9.6/static/xplang.html
https://www.postgresql.org/docs/9.6/static/xfunc-sql.html
Migrating from: Oracle UDFs

Overview
You can create an Oracle User-Defined Function (UDF) using PL/SQL, Java, or C. UDFs are useful for providing functionality not available in SQL or SQL built-in functions. They can appear in your SQL statements wherever built-in SQL functions appear.

UDFs Usage:
- Can be used to return a single value from a SELECT statement (scalar function).
- Can be used while performing DML operations.
- Can be used in WHERE, GROUP BY, ORDER BY, HAVING, CONNECT BY, and START WITH clauses.

Example
Create a simple Oracle UDF that receives each employee’s HIRE_DATE and SALARY values as INPUT parameters and calculates the overall salary over the employee’s years of service for the company.

```
SQL> CREATE OR REPLACE FUNCTION TOTAL_EMP_SAL_BY_YEARS
  2  (p_hire_date DATE, p_current_sal NUMBER)
  3  RETURN NUMBER
  4  AS
  5  v_years_of_service NUMBER;
  6  v_total_sal_by_years NUMBER;
  7  BEGIN
  8      SELECT EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM to_date(p_hire_date))
  9          INTO v_years_of_service FROM dual;
 10      v_total_sal_by_years:=p_current_sal*v_years_of_service;
 11      RETURN v_total_sal_by_years;
 12  END;
 13  /

-- Verifying
SQL> SELECT EMPLOYEE_ID,
  2       FIRST_NAME,
  3       TOTAL_EMP_SAL_BY_YEARS(HIRE_DATE, SALARY)AS TOTAL_SALARY
  4  FROM EMPLOYEES;
EMPLOYEE_ID FIRST_NAME TOTAL_SALARY
----------- -------------------- ------------
100 Steven                     364000
101 Neena                      204000
102 Lex                        272000
103 Alexander                   99000
104 Bruce                       60000
105 David                       57600
...
```

For additional details:
https://docs.oracle.com/cd/E24693_01/server.11203/e17118/functions256.htm
Migration to: PostgreSQL User-Defined Functions

Overview
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.

PostgreSQL Create Function Privileges
To create a function, a user must have USAGE privilege on the language.

Example
Convert the Oracle User-Defined Function from the previous Oracle section to a PostgreSQL PL/pgSQL function:

```
psql=> CREATE OR REPLACE FUNCTION total_emp_sal_by_years
       (P_HIRE_DATE DATE, P_CURRENT_SAL NUMERIC)
       RETURNS NUMERIC
       AS
       $BODY$
       DECLARE
       V_YEARS_OF_SERVICE NUMERIC;
       V_TOTAL_SAL_BY_YEARS NUMERIC;
       BEGIN
       SELECT EXTRACT(YEAR FROM NOW()) - EXTRACT(YEAR FROM
       (P_HIRE_DATE)) INTO V_YEARS_OF_SERVICE;

       V_TOTAL_SAL_BY_YEARS:=P_CURRENT_SAL*V_YEARS_OF_SERVICE;
       RETURN V_TOTAL_SAL_BY_YEARS;
       END;
       $BODY$
       LANGUAGE  PLPGSQL;

psql=> SELECT EMPLOYEE_ID,
       FIRST_NAME,
       TOTAL_EMP_SAL_BY_YEARS(HIRE_DATE, SALARY)AS TOTAL_SALARY
       FROM EMPLOYEES;

<table>
<thead>
<tr>
<th>employee_id</th>
<th>first_name</th>
<th>total_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>364000.00</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>204000.00</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>272000.00</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>99000.00</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>60000.00</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>57600.00</td>
</tr>
<tr>
<td>106</td>
<td>Valli</td>
<td>52800.00</td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>42000.00</td>
</tr>
</tbody>
</table>
```

For additional details:
https://www.postgresql.org/docs/current/static/xfunc.html
https://www.postgresql.org/docs/9.6/static/sql-createfunction.html
Overview
Oracle’s UTL_FILE PL/SQL package enables you to read and write files stored outside of the database server, such as files stored on the O/S, your database server, or a connected storage volume. The UTL_FILE.FOPEN, UTL_FILE.GET_LINE, and UTL_FILE.PUT_LINE are procedures within the UTL_FILE package used to open, read, and write files.

Example
Run an anonymous PL/SQL block that reads a single line from file1 and writes it to file2.

- Use UTL_FILE.FILE_TYPE to create a handle for the file.
- Use UTL_FILE.FOPEN to open streamable access to the file and specify:
  - The logical Oracle directory object that was created pointing to the O/S folder where the file resides.
  - The file name.
  - The file access mode:
    - A: append mode.
    - W: write mode.
- Use UTL_FILE.GET_LINE to read a line from the input file into a variable.
- Use UTL_FILE.PUT_LINE to write a single line to the output file.

```
DECLARE
  strString1 VARCHAR2(32767);
  fileFile1 UTL_FILE.FILE_TYPE;
BEGIN
  fileFile1 := UTL_FILE.FOPEN('FILES_DIR','File1.tmp','R');
  UTL_FILE.GET_LINE(fileFile1,strString1);
  UTL_FILE.FCLOSE(fileFile1);
  fileFile1 := UTL_FILE.FOPEN('FILES_DIR','File2.tmp','A');
  utl_file.PUT_LINE(fileFile1,strString1);
  utl_file.fclose(fileFile1);
END;
/
```

For additional details:
https://docs.oracle.com/database/121/ARPLS/u_file.htm
Migration to:

Amazon Aurora PostgreSQL does not support a direct comparable alternative for Oracle UTL_FILE.
Overview
JSON documents are based on JavaScript syntax and allow serialization of objects. Oracle support for JSON document storage and retrieval enables you to extend the database capabilities beyond purely relational use-cases and allows the Oracle database to support semi-structured data. Oracle JSON support also includes full-text search and several other functions dedicated to querying JSON documents.

Additional details:

Examples
Create a table to store a JSON document in a data column and insert a JSON document into the table:

```sql
CREATE TABLE json_docs (  
id    RAW(16) NOT NULL,  
data  CLOB,  
CONSTRAINT json_docs_pk PRIMARY KEY (id),  
CONSTRAINT json_docs_json_chk CHECK (data IS JSON)
);

INSERT INTO json_docs (id, data)  
VALUES (SYS_GUID(),  
'{
    "FName"    : "John",
    "LName"    : "Doe",
    "Address"  : {
        "Street" : "101 Street",
        "City"   : "City Name",
        "Country": "US",
        "Pcode"  : "90210"
    }
  }');
```

Unlike XML data, which is stored using the SQL data type XMLType, JSON data is stored in an Oracle Database using the SQL data types VARCHAR2, CLOB, and BLOB. Oracle recommends that you always use an is_json check constraint to ensure the column values are valid JSON instances. Or, add a constraint at the table-level (CONSTRAINT json_docs_json_chk CHECK (data IS JSON)).
You can query a JSON document directly from a SQL query without the use of special functions. Querying without functions is called *Dot Notation*.

```sql
SELECT a.data.FName,
       a.data.LName,
       a.data.Address.Pcode AS Postcode
FROM   json_docs a;
```

<table>
<thead>
<tr>
<th>FNAME</th>
<th>LNAME</th>
<th>POSTCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Doe</td>
<td>90210</td>
</tr>
</tbody>
</table>

1 row selected.

In addition, Oracle provides multiple SQL functions that integrate with the SQL language and enable querying JSON documents (such as `IS JSON`, `JSON_VALUE`, `JSON_EXISTS`, `JSON_QUERY`, and `JSON_TABLE`).

For additional details:

[http://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246](http://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246)
Migration to: PostgreSQL JSON Support

Overview
PostgreSQL provides native JSON Document support using the JSON data types JSON and JSONB.

**JSON:** Stores an exact copy of the input text, which processing functions must reparse on each execution. It also preserves semantically-insignificant white space between tokens and the order of keys within JSON objects.

**JSONB:** Stores data in a decomposed binary format causing slightly slower input performance due to added conversion to binary overhead. But, it is significantly faster to process since no reparsing is needed on reads.
- Does not preserve white space.
- Does not preserve the order of object keys.
- Does not keep duplicate object keys. If duplicate keys are specified in the input, only the last value is retained.

Most applications store JSON data as JSONB unless there are specialized needs.

For additional information about the differences between JSON and JSONB datatypes: https://www.postgresql.org/docs/9.6/static/datatype-json.html

In order to adhere to the full JSON specification, database encoding must be set to UTF8. If the database codepage is set to non-UTF8, characters that can be represented in the database encoding, but not in UTF8, are allowed. This condition is not desirable.

Creating JSON Tables in PostgreSQL and Inserting Data:

1. Create a PostgreSQL table named `json_docs` with a single JSON column:
   
   ```sql
   CREATE TABLE json_docs (data jsonb);
   ```

2. Create a PostgreSQL table named `employees` with two scalar datatype columns and a single JSON column:
   
   ```sql
   CREATE TABLE employees (emp_id int, emp_name varchar(100), emp_data jsonb);
   ```

3. Insert JSON data into the table:
   
   ```sql
   INSERT INTO employees VALUES (1, 'First Employee', '{ "address": "1234 First Street, Capital City", "phone numbers": { "home": "123456789", "mobile": "987654311" } }');
   ```

Oracle uses VARCHAR/BLOB/CLOB data types to store JSON data, but PostgreSQL uses the special JSON and JSONB data types. Validations of proper JSON formats are performed during insert. You cannot store invalid JSON in a JSON/JSONB data type.
Querying JSON/JSONB data with operators

Querying JSON data in PostgreSQL uses different query syntax from Oracle – you must change application queries. Examples of PostgreSQL-native JSON query syntax are provided below:

1. Return the JSON document stored in the `emp_data` column associated with `emp_id=1`:
   ```sql
   SELECT emp_data FROM employees WHERE emp_id = 1;
   ```

2. Return all JSON documents stored in the `emp_data` column having a key named `address`:
   ```sql
   SELECT emp_data FROM employees WHERE emp_data ? 'address';
   ```

3. Return all JSON items that have an `address` key or a `hobbies` key:
   ```sql
   SELECT * FROM employees WHERE emp_data ?| array['address', 'hobbies'];
   ```

4. Return all JSON items that have both an `address` key and a `hobbies` key:
   ```sql
   SELECT * FROM employees WHERE emp_data ?? array['a', 'b'];
   ```

5. Return the value of `home` key in the `phone numbers` array:
   ```sql
   SELECT emp_data ->'phone numbers'->'home' FROM employees;
   ```

6. Return all JSON documents where the `address` key is equal to a specified value and return all JSON documents where `address` key contains a specific string (using `like`):
   ```sql
   SELECT * FROM employees WHERE emp_data->>'address' = '1234 First Street, Capital City';
   SELECT * FROM employees WHERE emp_data->>'address' like '%Capital City%';
   ```

For additional details:
[https://www.postgresql.org/docs/9.6/static/functions-json.html](https://www.postgresql.org/docs/9.6/static/functions-json.html)
# Oracle vs. PostgreSQL JSON Support

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle (Dot-Notation)</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return the full JSON document / all JSON documents</td>
<td>emp_data is a column that stores json documents: SELECT emp_data FROM employees;</td>
<td>emp_data is a column that stores JSON documents: SELECT emp_data FROM employees;</td>
</tr>
<tr>
<td>Return a specific element from a JSON document</td>
<td>Return only the address property: SELECT e.emp_data.address FROM employees e;</td>
<td>Return only the address property, for emp_id 1 from the emp_data JSON column in the employees table: select emp_data-&gt;&gt;'address' from employees where emp_id = 1;</td>
</tr>
<tr>
<td>Return JSON documents matching a pattern in any field</td>
<td>Return the JSON based on a search of on all JSON properties. Could be returned even if element is equal to the pattern SELECT e.emp_data FROM employees e WHERE e.emp_data like '%patten%';</td>
<td>Either use jsonb_pretty to flatten the JSON and search or, preferably, convert it to text and make like search on value select * from (select jsonb_pretty(emp_data) as raw_data from employees) raw_jason where raw_data like '%1234%';</td>
</tr>
<tr>
<td>Return JSON documents matching a pattern in specific fields (root level)</td>
<td>SELECT e.emp_data.name FROM employees e WHERE e.data.active = 'true';</td>
<td>Only return results where the “finished” property in the JSON document is true: SELECT * FROM employees WHERE emp_data-&gt;&gt;'active' = 'true';</td>
</tr>
<tr>
<td>Define a column in a table that supports JSONB documents</td>
<td>1. Create a table with a CLOB column. 2. Define an “IS JSON” constraint on the column. CREATE TABLE json_docs (id RAW(16) NOT NULL, data CLOB, CONSTRAINT json_docs_pk PRIMARY KEY (id), CONSTRAINT json_docs_json_chk CHECK (data IS JSON) );</td>
<td>1. Create a table with a column defined as JSON: CREATE TABLE json_docs (id integer NOT NULL, data jsonb );</td>
</tr>
</tbody>
</table>
Indexing and Constraints with JSONB Columns

You can use the `CREATE UNIQUE INDEX` statement to enforce constraints on values inside JSON documents stored in PostgreSQL. For example, you can create a unique index that forces values of the `address` key to be unique.

```sql
CREATE UNIQUE INDEX employee_address_uq ON employees( (emp_data->>'address') ) ;
```

This index allows the first SQL insert statement to work and causes the second to fail:

```sql
INSERT INTO employees VALUES (2, 'Second Employee', '{ "address": "1234 Second Street, Capital City"}');

INSERT INTO employees VALUES (3, 'Third Employee', '{ "address": "1234 Second Street, Capital City"}');

ERROR: duplicate key value violates unique constraint "employee_address_uq" SQL state: 23505 Detail: Key ((emp_data ->> 'address'::text))=(1234 Second Street, Capital City) already exists.
```

For JSON data, PostgreSQL Supports B-Tree, HASH, and GIN indexes (Generalized Inverted Index). A GIN index is a special inverted index structure that is useful when an index must map many values to a row (such as indexing JSON documents).

When using GIN indexes, you can efficiently and quickly query data using only the following JSON operators: `@>`, `?`, `?&`, `?|`

Without indexes, PostgreSQL is forced to perform a full table scan when filtering data. This condition applies to JSON data and will most likely have a negative impact on performance since Postgres has to step into each JSON document.

1. Create an index on the address key of `emp_data`:
   ```sql
   CREATE INDEX idx1_employees ON employees ( (emp_data->>'address') ) ;
   ```

2. Create a GIN index on a specific key or the entire `emp_data` column:
   ```sql
   CREATE INDEX idx2_employees ON cards USING gin ( (emp_data->>'tags') ) ;
   CREATE INDEX idx3_employees ON employees USING gin ( emp_data ) ;
   ```

For additional details:
- [https://www.postgresql.org/docs/9.6/static/datatype-json.html](https://www.postgresql.org/docs/9.6/static/datatype-json.html)
- [https://www.postgresql.org/docs/9.6/static/functions-json.html](https://www.postgresql.org/docs/9.6/static/functions-json.html)
**Overview**
Oracle OLAP functions extend the functionality of standard SQL analytic functions by providing capabilities to compute aggregate values based on a group of rows. You can apply the OLAP functions to logically “partitioned” sets of results within the scope of a single query expression. OLAP functions are usually used in combination with Business Intelligence reports and analytics. They can help boost query performance – an alternative to achieving the same result using more complex non-OLAP SQL code.

**Common Oracle OLAP Functions:**

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td><code>average_rank, avg, count, dense_rank, max, min, rank, sum</code></td>
</tr>
<tr>
<td>Analytic</td>
<td><code>average_rank, avg, count, dense_rank, lag, lag_variance, lead_variance_percent, max, min, rank, row_number, sum, percent_rank, cume_dist, ntile, first_value, last_value</code></td>
</tr>
<tr>
<td>Hierarchical</td>
<td><code>hier_ancestor, hier_child_count, hier_depth, hier_level, hier_order, hier_parent, hier_top</code></td>
</tr>
<tr>
<td>Lag</td>
<td><code>lag, lag_variance, lag_variance_percent, lead, lead_variance, lead_variance_percent</code></td>
</tr>
<tr>
<td>OLAP DML</td>
<td><code>olap_dml_expression</code></td>
</tr>
<tr>
<td>Rank</td>
<td><code>average_rank, dense_rank, rank, row_number</code></td>
</tr>
</tbody>
</table>

For additional details:
https://docs.oracle.com/cd/E11882_01/olap.112/e23381/olap_functions.htm#OLAXS169
https://docs.oracle.com/database/121/OLAXS/olap_functions.htm#OLAXS174
Migration to: PostgreSQL Window Functions

Overview
PostgreSQL refers to ANSI SQL analytical functions as “Window Functions”. They provide the same core functionality as SQL Analytical Functions and Oracle extended OLAP functions. Window functions in PostgreSQL operate on a logical “partition” or “window” of the result set and return a value for rows in that “window”.

From a database migration perspective, you should examine PostgreSQL Window Functions by type and compare them with the equivalent Oracle’s OLAP functions to verify compatibility of syntax and output.

Note: Even if a PostgreSQL window function provides the same functionality of a specific Oracle OLAP function, the returned data type may be different and require application changes.

PostgreSQL provides support for two main types of Window Functions:
- Aggregation functions.
- Ranking functions.

PostgreSQL Window Functions by Type:

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>avg, count, max, min, sum, string_agg</td>
</tr>
<tr>
<td>Ranking</td>
<td>row_number, rank, dense_rank, percent_rank, cume_dist, ntile, lag, lead, first_value, last_value, nth_value</td>
</tr>
</tbody>
</table>

Oracle’s OLAP Functions vs. PostgreSQL Window Functions:

<table>
<thead>
<tr>
<th>Oracle OLAP Function</th>
<th>Returned Data Type</th>
<th>PostgreSQL Window Function</th>
<th>Returned Data Type</th>
<th>Compatible Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>number</td>
<td>count</td>
<td>bigint</td>
<td>Yes</td>
</tr>
<tr>
<td>max</td>
<td>number</td>
<td>max</td>
<td>numeric, string, date/time, network or enum type</td>
<td>Yes</td>
</tr>
<tr>
<td>min</td>
<td>number</td>
<td>min</td>
<td>numeric, string, date/time, network or enum type</td>
<td>Yes</td>
</tr>
<tr>
<td>avg</td>
<td>number</td>
<td>avg</td>
<td>numeric, double, otherwise same datatype as the argument</td>
<td>Yes</td>
</tr>
<tr>
<td>sum</td>
<td>number</td>
<td>sum</td>
<td>bigint, otherwise same datatype as the argument</td>
<td>Yes</td>
</tr>
<tr>
<td>rank()</td>
<td>number</td>
<td>rank()</td>
<td>bigint</td>
<td>Yes</td>
</tr>
<tr>
<td>row_number()</td>
<td>number</td>
<td>row_number()</td>
<td>bigint</td>
<td>Yes</td>
</tr>
<tr>
<td>dense_rank()</td>
<td>number</td>
<td>dense_rank()</td>
<td>bigint</td>
<td>Yes</td>
</tr>
<tr>
<td>percent_rank()</td>
<td>number</td>
<td>percent_rank()</td>
<td>double</td>
<td>Yes</td>
</tr>
<tr>
<td>cume_dist()</td>
<td>number</td>
<td>cume_dist()</td>
<td>double</td>
<td>Yes</td>
</tr>
<tr>
<td>ntile()</td>
<td>number</td>
<td>ntile()</td>
<td>integer</td>
<td>Yes</td>
</tr>
<tr>
<td>lag()</td>
<td>same type as value</td>
<td>lag()</td>
<td>same type as value</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle OLAP Function</td>
<td>Returned Data Type</td>
<td>PostgreSQL Window Function</td>
<td>Returned Data Type</td>
<td>Compatible Syntax</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------</td>
<td>-----------------------------</td>
<td>--------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>lead()</td>
<td>same type as value</td>
<td>lead()</td>
<td>same type as value</td>
<td>Yes</td>
</tr>
<tr>
<td>first_value()</td>
<td>same type as value</td>
<td>first_value()</td>
<td>same type as value</td>
<td>Yes</td>
</tr>
<tr>
<td>last_value()</td>
<td>same type as value</td>
<td>last_value()</td>
<td>same type as value</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Example
The Oracle `rank()` function & PostgreSQL `rank()` function providing the same results

Oracle:

```sql
SQL> SELECT department_id, last_name, salary, commission_pct,
         RANK() OVER (PARTITION BY department_id
ORDER BY salary DESC, commission_pct) "Rank"
FROM employees WHERE department_id = 80;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>Russell</td>
<td>14000</td>
<td>.4</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>Partners</td>
<td>13500</td>
<td>.3</td>
<td>2</td>
</tr>
<tr>
<td>80</td>
<td>Errazuriz</td>
<td>12000</td>
<td>.3</td>
<td>3</td>
</tr>
</tbody>
</table>

PostgreSQL:

```
hr=# SELECT department_id, last_name, salary, commission_pct,
       RANK() OVER (PARTITION BY department_id
ORDER BY salary DESC, commission_pct) "Rank"
FROM employees WHERE department_id = 80;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>Russell</td>
<td>14000.00</td>
<td>0.40</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>Partners</td>
<td>13500.00</td>
<td>0.30</td>
<td>2</td>
</tr>
<tr>
<td>80</td>
<td>Errazuriz</td>
<td>12000.00</td>
<td>0.30</td>
<td>3...</td>
</tr>
</tbody>
</table>

Note: The returned formatting for certain numeric data types is different.

**Oracle CONNECT BY Equivalent in PostgreSQL:**
PostgreSQL provides two workarounds as alternatives to Oracle’s hierarchical statements such as the CONNECT BY function:

- Use PostgreSQL `generate_series` function.
- Use PostgreSQL recursive views.

For more information: [https://www.postgresql.org/docs/9.6/static/sql-createview.html](https://www.postgresql.org/docs/9.6/static/sql-createview.html)
Example

PostgreSQL `generate_series` function:

demo=> SELECT "DATE"
    FROM generate_series(timestamp '2010-01-01',
                  timestamp '2017-01-01',
                  interval '1 day') s("DATE")

                    DATE
            ---------------------
2010-01-01 00:00:00  
2010-01-02 00:00:00  
2010-01-03 00:00:00  
2010-01-04 00:00:00  
2010-01-05 00:00:00  
...

For additional details:
https://www.postgresql.org/docs/9.6/static/functions-window.html
https://www.postgresql.org/docs/9.6/static/functions-aggregate.html

Extended Support for Analytic Queries and OLAP

For advanced analytic purposes and use cases, consider using Amazon Redshift as a purpose-built data warehouse cloud solution. You can run complex analytic queries against petabytes of structured data using sophisticated query optimization, columnar storage on high-performance local disks, and massive parallel query execution. Most results are returned in seconds.

Amazon Redshift is specifically designed for online analytic processing (OLAP) and business intelligence (BI) applications, which require complex queries against large datasets. Because it addresses very different requirements, the specialized data storage schema and query execution engine that Amazon Redshift uses is completely different from the PostgreSQL implementation. For example, Amazon Redshift stores data in columns, also known as a columnar-store database.
Amazon Redshift Window functions by type:

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Related Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>AVG, COUNT, CUME_DIST, FIRST_VALUE, LAG, LAST_VALUE, LEAD, MAX, MEDIAN, MIN, NTH_VALUE, PERCENTILE_CONT, PERCENTILE_DISC, RATIO_TO_REPORT, STDDEV_POP, STDDEV_SAMP (synonym for STDDEV), SUM, VAR_POP, VAR_SAMP (synonym for VARIANCE)</td>
</tr>
<tr>
<td>Ranking</td>
<td>DENSE_RANK, NTILE, PERCENT_RANK, RANK, ROW_NUMBER</td>
</tr>
</tbody>
</table>

For additional details:
http://docs.aws.amazon.com/redshift/latest/dg/c_Window_functions.html
http://docs.aws.amazon.com/redshift/latest/dg/r_Window_function_examples.html
Overview

PL/SQL cursors are pointers to data sets on which application logic can iterate. A PL/SQL cursor holds the rows returned by a SQL statement. Using cursors, PL/SQL code can iterate over the rows and execute business logic one row at a time. You can refer to the active data set in named cursors from inside a program.

There are two types of cursors in PL/SQL:

1. **Implicit Cursors** – Session cursors constructed and managed by PL/SQL automatically without being created or defined by the user. PL/SQL opens an implicit cursor each time you run a SELECT or DML statement. Implicit cursors are also called “SQL Cursors”.

2. **Explicit Cursors** – Session cursors created, constructed, and managed by a user. You declare and define an explicit cursor giving it a name and associating it with a query. Unlike an implicit cursor, you can reference an explicit cursor using its name. An explicit cursor or cursor variable is called a “named cursor”.

When migrating Oracle PL/SQL Cursors to PostgreSQL, most of the focus is on application-controlled (or programmatically-controlled) cursors, which are Explicit Cursors.

Examples

1. Define an **explicit** PL/SQL cursor named `c1`.
2. The cursor executes an SQL statement to return rows from the database.
3. The PL/SQL Loop reads data from the cursor, row by row, and stores the values into two variables:
   - `v_lastname`
   - `v_jobid`
4. The loop terminates when the last row is read from the database using the `%NOTFOUND` attribute.

```sql
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;
  v_lastname  employees.last_name%TYPE;  -- variable to store last_name
  v_jobid     employees.job_id%TYPE;     -- variable to store job_id
BEGIN
  OPEN c1;
  LOOP  -- Fetches 2 columns into variables
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
END;
```
1. Define an **implicit** PL/SQL Cursor using a **FOR Loop**.
2. The cursor executes a query and stores values returned into a record.
3. A loop iterates over the Cursor data set and prints the result.

```plsql
BEGIN
    FOR item IN (SELECT last_name, job_id
                  FROM employees
                  WHERE job_id LIKE '%MANAGER%' AND manager_id > 400
                  ORDER BY last_name)
    LOOP
        DBMS_OUTPUT.PUT_LINE ('Name = ' || item.last_name || ', Job = ' || item.job_id);
    END LOOP;
END;
/
```

*For additional details:*
https://docs.oracle.com/database/121/LNPLS/explicit_cursor.htm#LNPLS01313
Migration to: PostgreSQL PL/pgSQL Cursors

Overview
Similar to Oracle PL/SQL Cursors, PostgreSQL has PL/pgSQL cursors that enable you to iterate business logic on rows read from the database. They can encapsulate the query and read the query results a few rows at a time. All access to cursors in PL/pgSQL is performed through cursor variables, which are always of the refcursor data type.

Create a PL/pgSQL cursor by declaring it as a variable of type refcursor.

Example: Declare a Cursor

1. Declare a Cursor in PL/pgSQL to be used with any query:

```sql
DECLARE
  c1 refcursor;
```

The variable `c1` is unbound since it is not bound to any particular query.

2. Declare a Cursor in PL/pgSQL with a bound query:

```sql
DECLARE
  c2 CURSOR FOR SELECT * FROM employees;
```

`FOR` can be replaced by `IS` for Oracle compatibility:

```sql
DECLARE
  c2 CURSOR IS SELECT * FROM employees;
```

3. Declare a Cursor in PL/pgSQL with a parameterized bound query:

```sql
DECLARE
  c3 CURSOR (var1 integer) FOR SELECT * FROM employees where id = var1;
```

- The `id` variable is replaced by an integer parameter value when the cursor is opened.
- When declaring a Cursor with `SCROLL` specified, the Cursor can scroll backwards.
- If `NO SCROLL` is specified, backward fetches are rejected.

4. Declare a backward-scrolling compatible Cursor using the `SCROLL` option:

```sql
DECLARE
  c3 SCROLL CURSOR FOR SELECT id, name FROM employees;
```

Notes:
- `SCROLL` specifies that rows can be retrieved backwards. `NO SCROLL` specifies that rows cannot be retrieved backwards.
- Depending upon the complexity of the execution plan for the query, `SCROLL` might create performance issues.
- Backward fetches are not allowed when the query includes `FOR UPDATE` or `FOR SHARE`. 
Example: Open a Cursor

You must open a cursor before you can use it to retrieve rows.

1. **Open a Cursor variable that was declared as Unbound and specify the query to execute:**

   ```sql
   BEGIN
   OPEN c1 FOR SELECT * FROM employees WHERE id = emp_id;
   ```

2. Open a Cursor variable that was declared as Unbound and specify the query to execute as a string expression. This approach provides greater flexibility.

   ```sql
   BEGIN
   OPEN c1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1', tabname) USING keyvalue;
   ```

   Parameter values can be inserted into the dynamic command using `format()` and `USING`. For example, the table name is inserted into the query using `format()`. The comparison value for `col1` is inserted using a `USING` parameter.

3. Open a Cursor that was **bound to a query** when the Cursor was declared and that was declared to take arguments.

   ```sql
   DO $$
   DECLARE
   c3 CURSOR (var1 integer) FOR SELECT * FROM employees where id = var1;
   BEGIN
   OPEN c3(var1 := 42);
   END$$;
   ```

   For the `c3` Cursor, supply the argument value expressions.

   If the Cursor was not declared to take arguments, the arguments can be specified outside the Cursor:

   ```sql
   DO $$
   DECLARE
   var1 integer;
   c3 CURSOR FOR SELECT * FROM employees where id = var1;
   BEGIN
   var1 := 1;
   OPEN c3;
   END$$;
   ```
Example: Fetch a Cursor
The PL/pgSQL FETCH command retrieves the next row from the Cursor into a variable.

1. Fetch the values returned from the c3 Cursor into a row variable:

```sql
DO $$
DECLARE
  c3 CURSOR FOR SELECT * FROM employees;
  rowvar employees%ROWTYPE;
BEGIN
  OPEN c3;
  FETCH c3 INTO rowvar;
END$$;
```

2. Fetch the values returned from the c3 Cursor into two scalar datatypes:

```sql
DO $$
DECLARE
  c3 CURSOR FOR SELECT id, name FROM employees;
  emp_id integer;
  emp_name varchar;
BEGIN
  OPEN c3;
  FETCH c3 INTO emp_id, emp_name;
END$$;
```

3. PL/pgSQL supports a special direction clause when fetching data from a Cursor using the NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD arguments. Omitting direction is equivalent to as specifying NEXT. For example, fetch the last row from the Cursor into the declared variables:

```sql
DO $$
DECLARE
  c3 CURSOR FOR SELECT id, name FROM employees;
  emp_id integer;
  emp_name varchar;
BEGIN
  OPEN c3;
  FETCH c3 INTO emp_id, emp_name;
END$$;
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-fetch.html
Example: Close a Cursor
Close a PL/pgSQL cursor using the **close** command:

```sql
DO $$
DECLARE
  c3 CURSOR FOR SELECT id, name FROM employees;
  emp_id integer;
  emp_name varchar;
BEGIN
  OPEN c3;
  FETCH LAST FROM c3 INTO emp_id, emp_name;
  close c3;
END$$;
```

Example: Iterating Through a Cursor
PL/pgSQL supports detecting when a cursor has no more data to return and can be combined with loops to iterate over all rows of a Cursor reference.

The following PL/pgSQL code uses a loop to fetch all rows from the Cursor and then exit after the last record is fetched (using **EXIT WHEN NOT FOUND**):

```sql
DO $$
DECLARE
  c3 CURSOR FOR SELECT * FROM employees;
  rowvar employees%ROWTYPE;
BEGIN
  OPEN c3;
  LOOP
    FETCH FROM c3 INTO rowvar;
    EXIT WHEN NOT FOUND;
  END LOOP;
  CLOSE c3;
END$$;
```

Example: Move Cursor Without Fetching Data
**MOVE** repositions a cursor without retrieving any data and works exactly like the **FETCH** command, except it only repositions the cursor in the dataset and does not return the row to which the cursor is moved. The special variable **FOUND** can be checked to determine if there is a next row.

1. Move to the last row (null or no data found) for cursor **c3**:
   ```sql
   MOVE LAST FROM c3;
   ```
2. Move the Cursor two records back:
   ```sql
   MOVE RELATIVE -2 FROM c3;
   ```
3. Move the **c3** Cursor two records forward.
   ```sql
   MOVE FORWARD 2 FROM c3;
   ```
Example: Update/Delete Current
When a cursor is positioned on a table row, that row can be updated or deleted. There are restrictions on what the cursor’s query can select for this type of DML to succeed.

For example, the current row to which the C3 Cursor is pointed to is updated:

```
UPDATE employee SET salary = salary*1.2 WHERE CURRENT OF c3;
```

Example: Use an Implicit Cursor (FOR Loop Over Queries)

```
DO $$
DECLARE
    item RECORD;
BEGIN
    FOR item IN (    SELECT last_name, job_id
        FROM employees
        WHERE job_id LIKE 'MANAGER'
        AND manager_id > 400
        ORDER BY last_name )
    LOOP
        RAISE NOTICE 'Name = %, Job=%', item.last_name, item.job_id;
    END LOOP;
END $$;
```

Comparing Oracle PL/SQL and PostgreSQL PL/pgSQL syntax:

<table>
<thead>
<tr>
<th>Action</th>
<th>PostgreSQL PL/pgSQL</th>
<th>Oracle PL/SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare a bound explicit cursor</td>
<td>c2 CURSOR FOR SELECT * FROM employees;</td>
<td>CURSOR c1 IS SELECT * FROM employees;</td>
</tr>
<tr>
<td>Open a cursor</td>
<td>OPEN c2;</td>
<td>OPEN c1;</td>
</tr>
<tr>
<td>Move Cursor to next row and fetch into a record variable</td>
<td>FETCH c2 INTO rowvar;</td>
<td>FETCH c1 INTO rowvar;</td>
</tr>
<tr>
<td>(rowvar was declared in the DECLARE section)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Cursor to next row and fetch into multiple scalar data types</td>
<td>FETCH c2 INTO emp_id, emp_name, salary;</td>
<td>FETCH c1 INTO emp_id, emp_name, salary;</td>
</tr>
<tr>
<td>(emp_id, emp_name, salary was declared in the DECLARE section)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Iterate through an Implicit Cursor via a Loop | FOR item IN (    SELECT last_name, job_id
    FROM employees)                                                             | FOR item IN (    SELECT last_name, job_id
    FROM employees)                       |
<table>
<thead>
<tr>
<th>Action</th>
<th>PostgreSQL PL/pgSQL</th>
<th>Oracle PL/SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE job_id LIKE '%CLERK%' AND manager_id &gt; 120 ORDER BY last_name) LOOP &lt;&lt; do something &gt;&gt; END LOOP;</td>
<td>WHERE job_id LIKE '%CLERK%' AND manager_id &gt; 120 ORDER BY last_name) LOOP &lt;&lt; do something &gt;&gt; END LOOP;</td>
<td></td>
</tr>
<tr>
<td>Declare a cursor with variables</td>
<td>C2 CURSOR (key integer) FOR SELECT * FROM employees WHERE id = key;</td>
<td>CURSOR c1 (key NUMBER) IS SELECT * FROM employees WHERE id = key;</td>
</tr>
<tr>
<td>Open a cursor with variables</td>
<td>OPEN c2(2); or OPEN c2(key := 2); EXIT WHEN NOT FOUND;</td>
<td>OPEN c1(2);</td>
</tr>
<tr>
<td>Exit a loop after no data found</td>
<td>FOUND</td>
<td>%FOUND</td>
</tr>
<tr>
<td>Detect if a Cursor has rows remaining in its dataset</td>
<td>Not Supported but you can run with every DML GET DIAGNOSTICS integer_var = ROW_COUNT; and save the results in an array</td>
<td>%BULK_ROWCOUNT</td>
</tr>
<tr>
<td>Determine how many rows were affected from any DML statement</td>
<td>-</td>
<td>%BULK_EXCEPTIONS</td>
</tr>
<tr>
<td>Determine which DML execution failed with the relevant error code</td>
<td>-</td>
<td>%ISOPEN</td>
</tr>
<tr>
<td>Detect if the Cursor is open</td>
<td>NOT FOUND</td>
<td>%NOTFOUND</td>
</tr>
<tr>
<td>Detect if a Cursor has no rows remaining in its dataset</td>
<td>GET DIAGNOSTICS integer_var = ROW_COUNT;</td>
<td>%ROWCOUNT</td>
</tr>
<tr>
<td>Returns the number of rows affected by a Cursor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For additional information on PostgreSQL PL/pgSQL:
https://www.postgresql.org/docs/current/static/plpgsql-cursors.html
https://www.postgresql.org/docs/current/static/plpgsql-statements.html
Overview
Oracle provides two main categories of built-in SQL functions based on the amount of rows used as input and generated as output.

- **Single-row functions** (also known as Scalar Functions) return a single result for each row of the queried table or view. The implementation of single-row functions can be performed with a `SELECT` statement in the `WHERE` clause, the `START WITH` clause, the `CONNECT BY` clause, and the `HAVING` clause.

  The single-row functions are divided into groups according to the datatypes, such as:
  
  - NUMERIC functions.
  - CHAR functions.
  - DATETIME functions.

- **Aggregative functions** (also known as Group functions) are used to summarize a group of values into a single result. Examples include: `AVG`, `MIN`, `MAX`, `SUM`, `COUNT`, `LISTAGG`, `FIRST`, and `LAST`.

See the next section for a comparison of Oracle and PostgreSQL single-row functions.

For additional details:
https://docs.oracle.com/database/121/SQLRF/functions002.htm#SQLRF20031
https://docs.oracle.com/database/121/SQLRF/functions003.htm#SQLRF20035
Overview
PostgreSQL provides an extensive list of single-row and aggregative functions. Some functions are similar to their Oracle counterparts (by name and functionality, or under a different name but with similar functionality). Other functions can have identical names to their Oracle counterparts but offer different functionality. The “Equivalent” column in the table below indicates functional equivalency.

<table>
<thead>
<tr>
<th>Oracle Function</th>
<th>Function Definition</th>
<th>PostgreSQL Function</th>
<th>Function Definition</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NUMERIC FUNCTIONS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>Absolute value of n: abs (-11.3) --&gt; 11.3</td>
<td>ABS(n)</td>
<td>Absolute value of n: abs (-11.3) --&gt; 11.3</td>
<td>Yes</td>
</tr>
<tr>
<td>CEIL</td>
<td>Returns the smallest integer that is greater than or equal to n: ceil (-24.9) --&gt; -24</td>
<td>CEIL / CEILING</td>
<td>Returns the nearest integer greater than or equal to argument: ceil (-24.9) --&gt; -24</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOOR</td>
<td>Returns the largest integer equal to or less than n: floor (-43.7) --&gt; -44</td>
<td>FLOOR</td>
<td>Returns nearest integer less than or equal to argument: floor (-43.7) --&gt; -44</td>
<td>Yes</td>
</tr>
<tr>
<td>MOD</td>
<td>Remainder of n2 divided by n1: mod(10,3) --&gt; 1</td>
<td>MOD</td>
<td>Remainder of y/x: mod (10,3) --&gt; 1</td>
<td>Yes</td>
</tr>
<tr>
<td>ROUND</td>
<td>Returns n rounded to integer places to the right of the decimal point: round (3.49, 1) --&gt; 3.5</td>
<td>ROUND</td>
<td>Round to nearest integer: round (3.49, 1) --&gt; 3.5</td>
<td>Yes</td>
</tr>
<tr>
<td>TRUNC (Number)</td>
<td>Returns n1 truncated to n2 decimal places: trunc(13.5) --&gt; 13</td>
<td>TRUNC (Number)</td>
<td>Truncate to s decimal places: trunc (13.5) --&gt; 13</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>CHARACTER FUNCTIONS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONCAT</td>
<td>Returns char1 concatenated with char2: concat('a', 1) --&gt; a1</td>
<td>CONCAT</td>
<td>Concatenate the text representations of all the arguments: concat('a', 1) --&gt; a1</td>
<td>Partially</td>
</tr>
<tr>
<td>LOWER / UPPER</td>
<td>Returns char, with all letters lowercase or uppercase: lower ('MR. Smith') --&gt; mr. smith</td>
<td>LOWER / UPPER</td>
<td>Convert string to lower or upper case: lower ('MR. Smith') --&gt; mr. smith</td>
<td>Yes</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td><strong>LPAD / RPAD</strong></td>
<td>Returns expr1, left or right padded to length n characters with the sequence of characters in expr2:</td>
<td>LPAD('Log-1',10,'*')</td>
<td>-&gt; *****Log-1</td>
<td></td>
</tr>
<tr>
<td><strong>REGEXP_REPLACE</strong></td>
<td>Search a string for a regular expression pattern:</td>
<td>regexp_replace('John', '[hn].', '\1')</td>
<td>-&gt; Jo1</td>
<td></td>
</tr>
<tr>
<td><strong>REGEXP_SUBSTR</strong></td>
<td>Extends the functionality of the SUBSTR function by searching a string for a regular expression pattern:</td>
<td>REGEXP_SUBSTR('<a href="http://www.aws.com/products','http://(%5B%5B:alnum:%5D%5D+%5C.?)%7B3,4%7D/">http://www.aws.com/products','http://([[:alnum:]]+\.?){3,4}/</a>?')</td>
<td>-&gt; <a href="http://www.aws.com/">http://www.aws.com/</a></td>
<td></td>
</tr>
<tr>
<td><strong>REPLACE</strong></td>
<td>Returns char with every occurrence of search string replaced with a replacement string:</td>
<td>replace ('abcdef', 'abc', '123')</td>
<td>-&gt; 123def</td>
<td></td>
</tr>
<tr>
<td><strong>LTRIM / RTRIM</strong></td>
<td>Removes from the left or right end of char all of the characters that appear in set:</td>
<td>ltrim('zyzyaws', 'xyz')</td>
<td>-&gt; aws</td>
<td></td>
</tr>
<tr>
<td><strong>SUBSTR</strong></td>
<td>Return a portion of char, beginning at character position, substring length characters long:</td>
<td>substring ('John Smith', 6,1)</td>
<td>-&gt; S</td>
<td></td>
</tr>
<tr>
<td><strong>LPAD</strong></td>
<td>Fill up the string to length by prepending the characters fill left or right:</td>
<td>LPAD('Log-1',10,'*')</td>
<td>-&gt; *****Log-1</td>
<td></td>
</tr>
<tr>
<td><strong>REGEXP_REPLACE</strong></td>
<td>Replace substring(s) matching a POSIX regular expression:</td>
<td>regexp_replace('John', '[hn].', '\1')</td>
<td>-&gt; Jo1</td>
<td></td>
</tr>
<tr>
<td><strong>REPLACE</strong></td>
<td>Replace all occurrences in string of substring from with substring to:</td>
<td>replace ('abcdef', 'abc', '123')</td>
<td>-&gt; 123def</td>
<td></td>
</tr>
<tr>
<td><strong>LTRIM / RTRIM</strong></td>
<td>Remove the longest string containing only characters from characters (a space by default) from the start of string:</td>
<td>ltrim('zyzyaws', 'xyz')</td>
<td>-&gt; aws</td>
<td></td>
</tr>
<tr>
<td><strong>SUBSTRING</strong></td>
<td>Extract substring:</td>
<td>substring ( 'John Smith', 6,1)</td>
<td>-&gt; S</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>TRIM</td>
<td>ASCII</td>
<td>INSTR</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>substr('John Smith', 6,1)</code> --&gt; S</td>
<td>Trim leading or trailing characters (or both) from a character string: <code>trim (both 'x' FROM 'xJohnxx')</code> --&gt; John</td>
<td>-</td>
<td>Returns the decimal representation in the database character set of the first character of char: <code>ascii('a')</code> --&gt; a</td>
<td>N/A</td>
</tr>
<tr>
<td><code>trim (both 'x' FROM 'xJohnxx')</code> --&gt; John</td>
<td>Remove the longest string containing only characters from characters (a space by default) from the start, end, or both ends: <code>trim (both from 'yxJohnxx', 'xyz')</code> --&gt; John</td>
<td>-</td>
<td>ASCII code of the first character of the argument: <code>ascii('a')</code> --&gt; a</td>
<td>N/A</td>
</tr>
<tr>
<td><code>TRIM</code></td>
<td>-</td>
<td>Partly</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>ASCII</code></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>INSTR</code></td>
<td>-</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>LENGTH</code></td>
<td>-</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>REGEXP_COUNT</code></td>
<td>-</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>REGEXP_INST</code></td>
<td>-</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**DATETIME FUNCTIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>PostgreSQL</th>
<th>CURRENT_DATE</th>
<th>Partly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ADD_MONTHS</code></td>
<td>Returns the date plus integer months: <code>add_months(sysdate, 1)</code></td>
<td>No</td>
<td>PostgreSQL</td>
<td>Partly</td>
</tr>
<tr>
<td><code>CURRENT_DATE</code></td>
<td>Returns the current date in the session time zone: <code>select current_date from dual</code> --&gt; 2017-01-01 13:01:01</td>
<td>-</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Available for Redshift</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
<td>-------------------------</td>
<td></td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Returns the current date and time in the session time zone:</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>select current_timestamp from dual; --&gt; 2017-01-01 13:01:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Current date and time:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>select current_timestamp; --&gt; 2017-01-01 13:01:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTRACT (date part)</td>
<td>Returns the value of a specified datetime field from a datetime or interval expression:</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXTRACT (YEAR FROM DATE '2017-03-07') --&gt; 2017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTRACT (date part)</td>
<td>Retrieves subfields such as year or hour from date/time values:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXTRACT (YEAR FROM DATE '2017-03-07') --&gt; 2017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST_DAY</td>
<td>Returns the date of the last day of the month that contains date:</td>
<td>N/A</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The LAST_DAY function can be used with Amazon Redshift if necessary or can be created using PostgreSQL built-in functions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MONTHS_BETWEEN</td>
<td>Returns number of months between dates date1 and date2:</td>
<td>N/A</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MONTHS_BETWEEN (sysdate, sysdate-100) --&gt; 3.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>As an alternative solution create a function from PostgreSQL built-in functions to achieve the same functionality. Example for a possible solution without decimal values:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE_PART ('month', now()) - DATE_PART('month', now()- interval '100 days') --&gt; 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSDATE</td>
<td>Returns the current date and time set for the operating system on which the database server resides:</td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>select sysdate from dual --&gt; 2017-01-01 13:01:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSTIMESTAMP</td>
<td>Returns the system date, including fractional seconds and time zone:</td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Select systimestamp from dual --&gt; 2017-01-01 13:01:01.123456+00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCALTIMESTAMP</td>
<td>Returns the current date and time in the session time zone in a value of data type TIMESTAMP:</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>select localtimestamp --&gt; 2017-01-01 10:01:10.123456</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>SQL statement to select the localtimestamp from dual</td>
<td><code>select localtimestamp from dual</code> -&gt; <code>01-JAN-17 10.01.10.123456 PM</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TO_CHAR</td>
<td>Converts a datetime or timestamp to a value of VARCHAR2 data type</td>
<td><code>to_char(sysdate, 'DD-MON-YYYY HH24:MI:SS')</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>to a value of VARCHAR2 data type in the format specified by the date</td>
<td><code>---&gt; 01-JAN-2017 01:01:01</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>format:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUNC</td>
<td>Returns date with the time portion of the day</td>
<td><code>trunc(systimestamp)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>truncated to the unit specified by the format model:</td>
<td><code>---&gt; 2017-01-01 00:00:00</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATE_TRUNC</td>
<td>Truncate to specified precision:</td>
<td><code>date_trunc('day', now())</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>---&gt; 2017-01-01 00:00:00</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCODING AND DECODING FUNCTIONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECODE</td>
<td>Compares expr to each search value one by one using the functionality of</td>
<td><code>DECODE</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>an IF-THEN-ELSE statement</td>
<td>PostgreSQL Decode function acts differently from Oracle’s,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PostgreSQL decode binary data from textual representation in string</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and does not have the functionality of an IF-THEN-ELSE statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUMP</td>
<td>Returns a VARCHAR2 value containing the data type code, length in bytes,</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>and internal representation of expr.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORA_HASH</td>
<td>Computes a hash value for a given expression.</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL FUNCTIONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASE</td>
<td>The CASE statement chooses from a sequence of conditions and runs a</td>
<td>The PostgreSQL CASE expression is a generic conditional expression,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>corresponding statement:</td>
<td>similar to if/else statements in other programming languages:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CASE WHEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>condition THEN result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### CASE WHEN

<table>
<thead>
<tr>
<th>CASE WHEN condition THEN result</th>
<th>[WHEN ...]</th>
<th>[ELSE result]</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### COALESCE

<table>
<thead>
<tr>
<th>COALESCE</th>
<th>Returns the first non-null expr in the expression list:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>coalesce (null, 'a', 'b')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COALESCE</th>
<th>Returns the first of its arguments that is not null:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>coalesce (null, 'a', 'b')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

### NULLIF

<table>
<thead>
<tr>
<th>NULLIF</th>
<th>Compares expr1 and expr2. If they are equal, then the function returns null. If they are not equal, then the function returns expr1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NULLIF('a', 'b')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NULLIF</th>
<th>Returns a null value if value1 equals value2 otherwise it returns value1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NULLIF ('a', 'b')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

### NVL

<table>
<thead>
<tr>
<th>NVL</th>
<th>Replace null (returned as a blank) with a string in the results of a query:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NVL (null, 'a')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVL2</th>
<th>Determine the value returned by a query based on whether a specified expression is null or not.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COALESCE</th>
<th>Returns the first of its arguments that is not null:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>coalesce (null, 'a')</td>
</tr>
<tr>
<td></td>
<td>--&gt; a</td>
</tr>
</tbody>
</table>

### ENVIRONMENT AND IDENTIFIER FUNCTIONS

<table>
<thead>
<tr>
<th>SYS_GUID</th>
<th>Generates and returns a globally unique identifier (RAW value) made up of 16 bytes:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>select sys_guid() from dual</td>
</tr>
<tr>
<td></td>
<td>--&gt; 5A280ABA8C76201EE0530100007FF691</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UUID_GENERATE_V1()</th>
<th>Generates a version 1 UUID:</th>
</tr>
</thead>
<tbody>
<tr>
<td>select uuid_generate_v1( ) --&gt; 90791a6-a359-11e7-a61c-12803bf1597a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UID</th>
<th>Returns an integer that uniquely identifies the session user (the user who logged on):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>select uid from dual --&gt; 84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USER</th>
<th>Returns the name of the session user:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>select user from dual</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USER</th>
<th>SESSION_USER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CURRENT_USER</td>
</tr>
<tr>
<td></td>
<td>CURRENT_SCHEMA()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USER</th>
<th>User name or schema of current execution context:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Select user; or select current_schema();</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>USERENV</td>
<td>Returns information about the current session using parameters:</td>
</tr>
<tr>
<td>CAST</td>
<td>Converts one built-in data type or collection-typed value into another built-in data type or collection-typed value:</td>
</tr>
<tr>
<td>CONVERT</td>
<td>Converts a character string from a one-character set to another:</td>
</tr>
<tr>
<td>TO_CHAR (string / numeric)</td>
<td>Converts NCHAR, NVARCHAR2, CLOB, or NCLOB data to the database character set:</td>
</tr>
<tr>
<td>TO_DATE</td>
<td>Converts char of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of DATE data type:</td>
</tr>
<tr>
<td>TO_NUMBER</td>
<td>Converts expr to a value of NUMBER data type:</td>
</tr>
<tr>
<td>AVG</td>
<td>AVG returns average value of expr:</td>
</tr>
<tr>
<td>AVG</td>
<td>Average (arithmetic mean) of all input values:</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>COUNT</strong></td>
<td>Returns the number of rows returned by the query:</td>
</tr>
<tr>
<td><strong>LISTAGG</strong></td>
<td>Orders data within each group specified in the ORDER BY clause and then concatenates the values of the measure column:</td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>Returns maximum value of expr:</td>
</tr>
<tr>
<td><strong>MIN</strong></td>
<td>Returns minimum value of expr:</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td>Returns the sum of values of expr:</td>
</tr>
<tr>
<td><strong>FETCH</strong></td>
<td>Retrieves rows of data from the result set of a multi-row query:</td>
</tr>
<tr>
<td><strong>COUNT</strong></td>
<td>The number of input rows:</td>
</tr>
<tr>
<td><strong>STRING_AGG</strong></td>
<td>Input values concatenated into a string, separated by delimiter:</td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>Maximum value of expression across all input values:</td>
</tr>
<tr>
<td><strong>MIN</strong></td>
<td>Minimum value of expression across all input values:</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td>Sum of expression across all input values:</td>
</tr>
<tr>
<td><strong>FETCH</strong></td>
<td>Retrieve just a portion of the rows that are generated by the rest of the query:</td>
</tr>
</tbody>
</table>

For additional details:
https://www.postgresql.org/docs/current/static/functions.html
https://www.postgresql.org/docs/current/static/functions-math.html
https://www.postgresql.org/docs/current/static/functions-string.html
https://www.postgresql.org/docs/current/static/uuid-ossp.html
Overview
The **MERGE** statement provides a way to specify single SQL statements that can conditionally perform **INSERT**, **UPDATE**, or **DELETE** operations on the target table – a task that would otherwise require multiple logical statements. The **MERGE** statement selects record(s) from the source table and then, by specifying a logical structure, automatically performs multiple DML operations on the target table. Its main advantage is to help avoid the use of multiple inserts, updates or deletes. It is important to note that **MERGE** is a deterministic statement. That is, once a row has been processed by the **MERGE** statement, it cannot be processed again using the same **MERGE** statement. **MERGE** is also sometimes known as **UPSET**.

Example
Using Oracle **MERGE** to insert or update employees who are entitled to a bonus (by year):

```sql
SQL> CREATE TABLE EMP_BONUS
    2 (EMPLOYEE_ID NUMERIC,
       BONUS_YEAR VARCHAR2(4),
       SALARY NUMERIC,
       BONUS NUMERIC,
       PRIMARY KEY (EMPLOYEE_ID, BONUS_YEAR)
    3 );

SQL> MERGE INTO EMP_BONUS E1
USING (SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
       FROM EMPLOYEES) E2
ON (E1.EMPLOYEE_ID = E2.EMPLOYEE_ID)
WHEN MATCHED THEN
    UPDATE SET E1.BONUS = E2.SALARY * 0.5
    DELETE WHERE (E1.SALARY >= 10000)
WHEN NOT MATCHED THEN
    INSERT (E1.EMPLOYEE_ID, E1.BONUS_YEAR, E1.SALARY, E1.BONUS)
    VALUES (E2.EMPLOYEE_ID, EXTRACT(YEAR FROM SYSDATE), E2.SALARY,
            E2.SALARY * 0.5)
    WHERE (E2.SALARY < 10000);

SQL> SELECT * FROM EMP_BONUS;

EMPLOYEE_ID  BONUS_YEAR  SALARY  BONUS
-----------  ----------  ------  -----
    103       2017       9000   4500
    104       2017       6000   3000
    105       2017       4800   2400
    106       2017       4800   2400
    107       2017       4200   2100
    109       2017       9000   4500
    110       2017       8200   4100
    111       2017       7700   3850
    112       2017       7800   3900
    113       2017       6900   3450
    115       2017       3100   1550
```

For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_9016.htm#SQLRF01606
Migration to: PostgreSQL Merge SQL Syntax

Overview
Currently, PostgreSQL version 9.6 does not support the use of the MERGE SQL command. As an alternative, consider using the INSERT... ON CONFLICT clause, which can handle cases where insert clauses might cause a conflict, and then redirect the operation as an update.

Example
Using the ON CONFLICT clause to handle a similar scenario as shown for the Oracle MERGE command:

```sql
demo=> CREATE TABLE EMP_BONUS (
    EMPLOYEE_ID NUMERIC,
    BONUS_YEAR VARCHAR(4),
    SALARY NUMERIC,
    BONUS NUMERIC,
    PRIMARY KEY (EMPLOYEE_ID, BONUS_YEAR));

demo=> INSERT INTO EMP_BONUS (EMPLOYEE_ID, BONUS_YEAR, SALARY)
    SELECT EMPLOYEE_ID,
    EXTRACT(YEAR FROM NOW()),
    SALARY
    FROM EMPLOYEES
    WHERE SALARY < 10000
    ON CONFLICT (EMPLOYEE_ID, BONUS_YEAR)
    DO UPDATE SET BONUS = EMP_BONUS.SALARY * 0.5;

demo=> SELECT * FROM EMP_BONUS;

<table>
<thead>
<tr>
<th>employee_id</th>
<th>bonus_year</th>
<th>salary</th>
<th>bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>2017</td>
<td>9000.00</td>
<td>4500.00</td>
</tr>
<tr>
<td>104</td>
<td>2017</td>
<td>6000.00</td>
<td>3000.00</td>
</tr>
<tr>
<td>105</td>
<td>2017</td>
<td>4800.00</td>
<td>2400.00</td>
</tr>
<tr>
<td>106</td>
<td>2017</td>
<td>4800.00</td>
<td>2400.00</td>
</tr>
<tr>
<td>107</td>
<td>2017</td>
<td>4200.00</td>
<td>2100.00</td>
</tr>
<tr>
<td>109</td>
<td>2017</td>
<td>9000.00</td>
<td>4500.00</td>
</tr>
<tr>
<td>110</td>
<td>2017</td>
<td>8200.00</td>
<td>4100.00</td>
</tr>
<tr>
<td>111</td>
<td>2017</td>
<td>7700.00</td>
<td>3850.00</td>
</tr>
<tr>
<td>112</td>
<td>2017</td>
<td>7800.00</td>
<td>3900.00</td>
</tr>
<tr>
<td>113</td>
<td>2017</td>
<td>6900.00</td>
<td>3450.00</td>
</tr>
<tr>
<td>115</td>
<td>2017</td>
<td>3100.00</td>
<td>1550.00</td>
</tr>
<tr>
<td>116</td>
<td>2017</td>
<td>2900.00</td>
<td>1450.00</td>
</tr>
<tr>
<td>117</td>
<td>2017</td>
<td>2800.00</td>
<td>1400.00</td>
</tr>
<tr>
<td>118</td>
<td>2017</td>
<td>2600.00</td>
<td>1300.00</td>
</tr>
</tbody>
</table>

Running the same operation multiple times using the ON CONFLICT clause does not generate an error because the existing records are redirected to the update clause.
For additional details:
https://www.postgresql.org/docs/9.6/static/sql-insert.html
Overview
To create a new table based on an existing table, use the Create Table As Select (CTAS) statement. The CTAS statement copies the table DDL definitions (column names and column datatypes) and the data to a new table. The new table is populated from the columns specified in the SELECT statement, or all columns if you use SELECT * FROM. You can filter specific data using the WHERE and AND statements. Additionally, you can create a new table having a different structure using joins, GROUP BY, and ORDER BY.

Example
Oracle Create Table As Select (CTAS):

```sql
SQL> CREATE TABLE EMPS
AS
SELECT * FROM EMPLOYEES;
```

```sql
SQL> CREATE TABLE EMPS
AS
SELECT EMPLOYEE_ID, FIRST_NAME, SALARY FROM EMPLOYEES
ORDER BY 3 DESC;
```
Overview

PostgreSQL conforms to the ANSI/SQL standard for CTAS functionality and is compatible with an Oracle CTAS statement. For PostgreSQL, the following CTAS standard elements are optional:

- The standard requires parentheses around the SELECT statement; PostgreSQL does not.
- The standard requires the WITH [ NO ] DATA clause; PostgreSQL does not.

PostgreSQL CTAS Synopsis

```sql
CREATE [ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ] TABLE [ IF NOT EXISTS ]
  table_name
  [ (column_name [, ...] ) ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ WITH OIDS | WITHOUT OIDS ]
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
  [ TABLESPACE tablespace_name ]
  AS query
  [ WITH [ NO ] DATA ]
```

Examples

1. PostgreSQL CTAS:

   ```
   pg_demo=> CREATE TABLE EMPS
             AS
             SELECT * FROM EMPLOYEES;
   ```

   ```
   pg_demo=> CREATE TABLE EMPS
             AS
             SELECT EMPLOYEE_ID, FIRST_NAME, SALARY FROM EMPLOYEES
             ORDER BY 3 DESC;
   ```

2. PostgreSQL CTAS with no data:

   ```
   pg_demo=> CREATE TABLE EMPS
             AS
             SELECT * FROM EMPLOYEES
             WITH NO DATA; -- optionally
   ```

For additional details:

https://www.postgresql.org/docs/9.6/static/sql-createtableas.html
Overview
Common Table Expressions provide a way to implement the logic of sequential code or to reuse code. You can define a named subquery and then use it multiple times in different parts of a query statement. CTE is implemented using a `WITH` clause, which is part of the ANSI SQL-99 standard and has existed in Oracle since version 9.2. CTE usage is similar to an inline view or a temporary table. Its main purpose is to reduce query statement repetition and make complex queries simpler to read and understand.

CTE General Syntax

```sql
WITH <subquery name> AS (
    <subquery code>)
[...]
SELECT <Select list> FROM <subquery name>;
```

Example
Create a subquery of the employee count for each department and then use the result set of the CTE in a query:

```sql
SQL> WITH DEPT_COUNT
        (DEPARTMENT_ID, DEPT_COUNT) AS
          (SELECT DEPARTMENT_ID, COUNT(*)
           FROM   EMPLOYEES
           GROUP BY DEPARTMENT_ID)

     SELECT E.FIRST_NAME || ' ' || E.LAST_NAME AS EMP_NAME,
            D.DEPT_COUNT AS EMP_DEPT_COUNT
     FROM   EMPLOYEES E JOIN DEPT_COUNT D
            USING  (DEPARTMENT_ID)
     ORDER BY 2;
```
Overview
PostgreSQL confirms to the ANSI SQL-99 standard. Implementing CTEs in PostgreSQL is done in a similar way to Oracle as long as you are not using native Oracle elements (for example, `CONNECT BY`).

Example
A PostgreSQL CTE:

```sql
SQL> WITH DEPT_COUNT
       (DEPARTMENT_ID, DEPT_COUNT) AS
          (SELECT DEPARTMENT_ID, COUNT(*)
           FROM employees
           GROUP BY DEPARTMENT_ID)
       SELECT E.FIRST_NAME || ' ' || E.LAST_NAME AS EMP_NAME,
              D.DEPT_COUNT AS EMP_DEPT_COUNT
       FROM employees E JOIN DEPT_COUNT D
       USING (DEPARTMENT_ID)
       ORDER BY 2;
```

PostgreSQL provides an additional feature when using CTE as a recursive modifier. The following example uses a recursive `WITH` clause to access its own result set:

```sql
demo=> WITH RECURSIVE t(n) AS
          (VALUES (0)
              UNION ALL
              SELECT n+1 FROM t WHERE n < 5
          )
          SELECT * FROM t;

WITH RECURSIVE t(n) AS
          (VALUES (0)
              UNION ALL
              SELECT n+1 FROM t WHERE n < 5
          )
          SELECT * FROM t;

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
```

For additional details:
https://www.postgresql.org/docs/9.6/static/queries-with.html
Overview
You can insert multiple records into a table from another table using the \texttt{INSERT FROM SELECT} statement. The \texttt{INSERT FROM SELECT} statement is a derivative of the basic \texttt{INSERT} statement. The column ordering and data types must match between the target and the source tables.

Example
Simple \texttt{INSERT FROM SELECT} (Explicit):

\begin{verbatim}
SQL> INSERT INTO EMPS (EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID)
       SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
       FROM EMPLOYEES
       WHERE SALARY > 10000;
\end{verbatim}

Simple \texttt{INSERT FROM SELECT} (Implicit):

\begin{verbatim}
SQL> INSERT INTO EMPS
       SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
       FROM EMPLOYEES
       WHERE SALARY > 10000;
\end{verbatim}

The following example produces the same effect as the preceding example using a subquery in the \texttt{DML_table_expression_clause}:

\begin{verbatim}
SQL> INSERT INTO
       (SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID FROM EMPS)
VALUES (120, 'Kenny', 10000, 90);
\end{verbatim}
Logging Errors Using Oracle *error_logging_clause:*

```sql
SQL> ALTER TABLE EMPS ADD CONSTRAINT PK_EMP_ID PRIMARY KEY(employee_id);

SQL> EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('EMPS', 'ERRLOG');

SQL> INSERT INTO EMPS
       SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
       FROM EMPLOYEES
       WHERE SALARY > 10000
       LOG ERRORS INTO errlog ('Cannot Perform Insert') REJECT LIMIT 100;

0 rows inserted
```

When inserting an existing EMPLOYEE ID into the EMPS table, the insert does not fail because the invalid records are redirected to the ERRLOG table.

*For additional details:*
[https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_9014.htm#SQLRF01604](https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_9014.htm#SQLRF01604)
Migration to: PostgreSQL Insert From Select

Overview

PostgreSQL INSERT FROM SELECT syntax is mostly compatible with the Oracle syntax, except for a few Oracle-only features such as the conditional_insert_clause (ALL|FIRST|ELSE). Also, PostgreSQL does not support the Oracle error_logging_clause. As an alternative, PostgreSQL provides the ON CONFLICT clause to capture errors, perform corrective measures, or log errors.

PostgreSQL Insert Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
{ DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
[ ON CONFLICT [ conflict_target ] conflict_action ]
[ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

where conflict_target can be one of:

```
( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] ) [ WHERE index_predicate ]
ON CONSTRAINT constraint_name
```

and conflict_action is one of:

```
DO NOTHING
DO UPDATE SET { column_name = ( expression | DEFAULT ) | ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) | ( column_name [, ...] ) = ( sub-SELECT ) } [, ...] [ WHERE condition ]
```

Example

1. Simple INSERT FROM SELECT (Explicit):

```sql
demo=> INSERT INTO EMPS (EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID)
    SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
    FROM EMPLOYEES
    WHERE SALARY > 10000;
```
2. Simple Insert from Select (Implicit):

```sql
demo=> INSERT INTO EMPS
    SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
    FROM EMPLOYEES
    WHERE SALARY > 10000;
```

3. The following example is not compatible with the supported syntax PostgreSQL:

```sql
demo=> INSERT INTO
    (SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID FROM EMPS)
    VALUES (120, 'Keñny', 10000, 90);
```

4. Using the PostgreSQL ON CONFLICT clause:

```sql
demo=> ALTER TABLE EMPS ADD CONSTRAINT PK_EMP_ID PRIMARY KEY(employee_id);
demo=> INSERT INTO EMPS
    SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
    FROM EMPLOYEES
    WHERE SALARY > 10000
    ON CONFLICT on constraint PK_EMP_ID DO NOTHING;
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-insert.html
Overview
Oracle’s Index-Organized Table (IOT) is a special type of index/table hybrid that \textit{physically} controls how data is stored at the table and index level. A common table, or heap-organized table, stores the data unsorted (as a heap). In an Index-Organized Table, the actual table data is stored in a B-tree index structure sorted by the row’s primary key. Each leaf block in the index structure stores both the primary key and non-key columns.

Index-Organized Table benefits include:

- The table records are sorted (clustered) using the primary key, which provides performance benefits. Accessing data using the primary key is faster because the key and data are located physically in the same structure.
- The total size of storage is reduced because primary key duplication is prevented.

Example
Create an Oracle Index-Organized Table storing “ordered” table data based on the PK\_EVENT\_ID primary key:

\begin{verbatim}
SQL> CREATE TABLE SYSTEM_EVENTS (
    EVENT_ID NUMBER,
    EVENT_CODE VARCHAR2(10) NOT NULL,
    EVENT_DESCRIPTION VARCHAR2(200),
    EVENT_TIME DATE NOT NULL,
    CONSTRAINT PK_EVENT_ID PRIMARY KEY(EVENT_ID)) ORGANIZATION INDEX;
SQL> INSERT INTO SYSTEM_EVENTS VALUES(9, 'EVNT-A1-10', 'Critical', '01-JAN-2017');
SQL> INSERT INTO SYSTEM_EVENTS VALUES(1, 'EVNT-C1-09', 'Warning', '01-JAN-2017');
SQL> INSERT INTO SYSTEM_EVENTS VALUES(7, 'EVNT-E1-14', 'Critical', '01-JAN-2017');
SQL> SELECT * FROM SYSTEM_EVENTS;

<table>
<thead>
<tr>
<th>EVENT_ID</th>
<th>EVENT_CODE</th>
<th>EVENT_DESCRIPTION</th>
<th>EVENT_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EVNT-C1-09</td>
<td>Warning</td>
<td>01-JAN-17</td>
</tr>
<tr>
<td>7</td>
<td>EVNT-E1-14</td>
<td>Critical</td>
<td>01-JAN-17</td>
</tr>
<tr>
<td>9</td>
<td>EVNT-A1-10</td>
<td>Critical</td>
<td>01-JAN-17</td>
</tr>
</tbody>
</table>
\end{verbatim}

Note: The EVENT\_ID records are sorted in the reverse order from which they were inserted.

For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28310/tables012.htm#ADMIN11684
https://docs.oracle.com/database/121/CNCPT/indexiot.htm#CNCPT721
Overview
PostgreSQL does not support IOTs directly, but offers partially similar functionality using the `CLUSTER` feature. The PostgreSQL `CLUSTER` statement specifies table sorting based on an index already associated with the table. When using the PostgreSQL `CLUSTER` command, the data in the table is physically sorted based on the index, possibly using a primary key column.

Note: Unlike an Oracle Index-Organized Table which is defined during table creation and persists data sorting (the IOT will always remain sorted), the PostgreSQL `CLUSTER` does not provide persistent sorting; it is a one-time operation. When the table is subsequently updated, the changes are not clustered/sorted.

The `CLUSTER` statement can be used as needed to re-cluster the table.
Example
Using the PostgreSQL CLUSTER command:

```
demo=> CREATE TABLE SYSTEM_EVENTS (  
    EVENT_ID NUMERIC,  
    EVENT_CODE VARCHAR(10) NOT NULL,  
    EVENT_DESCRIPTION VARCHAR(200),  
    EVENT_TIME DATE NOT NULL,  
    CONSTRAINT PK_EVENT_ID PRIMARY KEY(EVENT_ID));

demo=> INSERT INTO SYSTEM_EVENTS VALUES(9, 'EV-A1-10', 'Critical', '01-JAN-2017');
demo=> INSERT INTO SYSTEM_EVENTS VALUES(1, 'EV-C1-09', 'Warning', '01-JAN-2017');
demo=> INSERT INTO SYSTEM_EVENTS VALUES(7, 'EV-E1-14', 'Critical', '01-JAN-2017');

demo=> CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID;
demo=> SELECT * FROM SYSTEM_EVENTS;
```

<table>
<thead>
<tr>
<th>event_id</th>
<th>event_code</th>
<th>event_description</th>
<th>event_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EVNT-C1-09</td>
<td>Warning</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>7</td>
<td>EVNT-E1-14</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>9</td>
<td>EVNT-A1-10</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
</tbody>
</table>

demo=> INSERT INTO SYSTEM_EVENTS VALUES(2, 'EV-E2-02', 'Warning', '01-JAN-2017');
demo=> SELECT * FROM SYSTEM_EVENTS;
```

<table>
<thead>
<tr>
<th>event_id</th>
<th>event_code</th>
<th>event_description</th>
<th>event_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EVNT-C1-09</td>
<td>Warning</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>7</td>
<td>EVNT-E1-14</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>9</td>
<td>EVNT-A1-10</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>2</td>
<td>EVNT-E2-02</td>
<td>Warning</td>
<td>2017-01-01</td>
</tr>
</tbody>
</table>

demo=> CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID; -- Run CLUSTER again to re-cluster
demo=> SELECT * FROM SYSTEM_EVENTS;
```

<table>
<thead>
<tr>
<th>event_id</th>
<th>event_code</th>
<th>event_description</th>
<th>event_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EVNT-C1-09</td>
<td>Warning</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>2</td>
<td>EVNT-E2-02</td>
<td>Warning</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>7</td>
<td>EVNT-E1-14</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
<tr>
<td>9</td>
<td>EVNT-A1-10</td>
<td>Critical</td>
<td>2017-01-01</td>
</tr>
</tbody>
</table>
```

For additional details:
https://www.postgresql.org/docs/current/static/sql-cluster.html
https://www.postgresql.org/docs/9.6/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY
# Migrating from: Oracle Common Data Types

**Overview**

Oracle provides a set of primitive data types that can be used for table columns or PL/SQL code variables. The assigned data types for table column or PL/SQL code (such as stored procedures and triggers) define valid values that each column or argument can store.

## Oracle Data Types vs. PostgreSQL Data Types

<table>
<thead>
<tr>
<th>Oracle Data Type Family</th>
<th>Oracle Data Type</th>
<th>Oracle Data Type Characteristic</th>
<th>PostgreSQL Identical Compatibility</th>
<th>PostgreSQL Corresponding Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Types</td>
<td>CHAR(n)</td>
<td>Maximum size of 2000 bytes</td>
<td>✓</td>
<td>CHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHARACTER(n)</td>
<td>Maximum size of 2000 bytes</td>
<td>✓</td>
<td>CHARACTER(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NCHAR(n)</td>
<td>Maximum size of 2000 bytes</td>
<td>✗</td>
<td>CHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR(n)</td>
<td>Maximum size of 2000 bytes</td>
<td>✓</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NCHAR</td>
<td>Varying-length UTF-8 string</td>
<td>✗</td>
<td>CHARACTER VARYING(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHARING(n)</td>
<td>Maximum size of 4000 bytes</td>
<td>✗</td>
<td>CHARACTER VARYING(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR2(n)</td>
<td>Maximum size of 4000 bytes</td>
<td>✗</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR2(n) 11g</td>
<td>Maximum size of 4000 bytes</td>
<td>✗</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR2(n) 12g</td>
<td>Maximum size of 32767 bytes</td>
<td>✗</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NVARCHAR2(n)</td>
<td>Maximum size of 4000 bytes</td>
<td>✗</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONG</td>
<td>Maximum size of 2GB</td>
<td>✗</td>
<td>TEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAW(n)</td>
<td>Maximum size of 2000 bytes</td>
<td>✗</td>
<td>BYTEA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONG RAW</td>
<td>Maximum size of 2GB</td>
<td>✗</td>
<td>BYTEA</td>
<td></td>
</tr>
<tr>
<td>Numeric Types</td>
<td>NUMBER</td>
<td>Floating-point number</td>
<td>✗</td>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMBER(*)</td>
<td>Floating-point number</td>
<td>✗</td>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMBER(p,s)</td>
<td>Precision can range from 1 to 38</td>
<td>✗</td>
<td>DECIMAL(p,s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC(p,s)</td>
<td>Precision can range from -84 to 127</td>
<td>✓</td>
<td>NUMERIC(p,s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT(p)</td>
<td>Floating-point number</td>
<td>✓</td>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEC(p,s)</td>
<td>Fixed-point number</td>
<td>✓</td>
<td>DEC(p,s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p,s)</td>
<td>Fixed-point number</td>
<td>✓</td>
<td>DECIMAL(p,s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>38 digits integer</td>
<td>✓</td>
<td>INTEGER / NUMERIC(38,0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>38 digits integer</td>
<td>✓</td>
<td>INTEGER / NUMERIC(38,0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>38 digits integer</td>
<td>✓</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>Floating-point number</td>
<td>✗</td>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
<td>Floating-point number</td>
<td>✓</td>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td>Date &amp;Time Types</td>
<td>DATE</td>
<td>DATE data type stores date and time data (year, month, day, hour, minute and second)</td>
<td>✓</td>
<td>TIMESTAMP(0)</td>
<td></td>
</tr>
<tr>
<td>Oracle Data Type Family</td>
<td>Oracle Data Type</td>
<td>Oracle Data Type Characteristic</td>
<td>PostgreSQL Identical Compatibility</td>
<td>PostgreSQL Corresponding Data Type</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
<td>---------------------------------</td>
<td>------------------------------------</td>
<td>-------------------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(p)</td>
<td>Date and time with fraction</td>
<td>√</td>
<td>TIMESTAMP(p)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(p)</td>
<td>Date and time with fraction and time zone</td>
<td>√</td>
<td>TIMESTAMP(p) WITH TIME ZONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTERVAL YEAR(p) TO MONTH</td>
<td>Date interval</td>
<td>√</td>
<td>INTERVAL YEAR TO MONTH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTERVAL DAY(p) TO SECOND(s)</td>
<td>Day and time interval</td>
<td>√</td>
<td>INTERVAL DAY TO SECOND(s)</td>
<td></td>
</tr>
<tr>
<td>LOB Data Types</td>
<td>BFILE</td>
<td>Pointer to binary file</td>
<td>☒</td>
<td>VARCHAR (255) / CHARACTER VARYING (255) BYTEA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLOB</td>
<td>Binary large object</td>
<td>☒</td>
<td>BYTETA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLOB</td>
<td>Character large object</td>
<td>☒</td>
<td>TEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NCLOB</td>
<td>Variable-length Unicode string</td>
<td>☒</td>
<td>TEXT</td>
<td></td>
</tr>
<tr>
<td>ROWID Data Types</td>
<td>ROWID</td>
<td>Physical row address</td>
<td>☒</td>
<td>CHARACTER (255)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UROWID(n)</td>
<td>Universal row id</td>
<td>☒</td>
<td>CHARACTER VARYING</td>
<td></td>
</tr>
<tr>
<td>XML Data Type</td>
<td>XMLTYPE</td>
<td>XML data</td>
<td>☒</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td>Logical Data Type</td>
<td>BOOLEAN</td>
<td>Values TRUE / FALSE and NULL</td>
<td>√</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be assign to a database table column</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Types</td>
<td>SDO_GEOMETRY</td>
<td>The geometric description of a spatial object</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SDO_TOPO_GEO METRY</td>
<td>Describes a topology geometry</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SDO_GEORASTER</td>
<td>A raster grid or image object is stored in a single row</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Media Types</td>
<td>ORDDicom</td>
<td>Supports the storage and management of audio data</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORDDicom</td>
<td>Supports the storage and management of Digital Imaging and Communications in Medicine (DICOM),</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORDDoc</td>
<td>Supports storage and management of any type of media data</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORDImage</td>
<td>Supports the storage and management of image data</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ORDVideo</td>
<td>Supports the storage and management of video data</td>
<td>☒</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The “PostgreSQL Identical Compatibility” column indicates if you can use the exact Oracle data type syntax when migrating to Amazon Aurora PostgreSQL.
Oracle Character Column Semantics
Oracle supports both \texttt{BYTE} and \texttt{CHAR} semantics for column size, which determines the amount of storage allocated for \texttt{CHAR} and \texttt{VARCHAR} columns.

- If you define a field as \texttt{VARCHAR2(10\ BYTE)}, Oracle can use up to 10 bytes for storage. However, based on your database codepage and NLS settings, you may not be able to store 10 \texttt{characters} in that field because the physical size of some non-English characters exceeds one byte.
- If you define a field as \texttt{VARCHAR2(10\ CHAR)}, Oracle can store 10 characters no matter how many bytes are required to store each non-English character.

```
CREATE TABLE table1 (col1 VARCHAR2(10\ \texttt{CHAR}),
col2 VARCHAR2(10\ \texttt{BYTE}));
```

By default, Oracle will use \texttt{BYTE} semantics. When using a multi-byte character set such as UTF8, you must do one of the following:

- Use the \texttt{CHAR} modifier in the \texttt{VARCHAR2/CHAR} column definition
- Modify the session or system parameter \texttt{NLS\_LENGTH\_SEMANTICS} to change the default from \texttt{BYTE} to \texttt{CHAR}:

```
ALTER system SET nls_length_semantics=char scope=both;
ALTER system SET nls_length_semantics=byte scope=both;

ALTER session SET nls_length_semantics=char;
ALTER session SET nls_length_semantics=byte;
```

For additional details:
https://docs.oracle.com/cd/E11882_01/server.112/e41084/sql_elements001.htm#SQLRF0021
https://docs.oracle.com/database/121/SQLRF/sql_elements001.htm#SQLRF30020
Overview
PostgreSQL provides multiple data types which are equivalent to certain Oracle data types. The following table provides the full list of PostgreSQL datatypes:

<table>
<thead>
<tr>
<th>PostgreSQL Data Type Family</th>
<th>PostgreSQL Data Type</th>
<th>PostgreSQL Data Type Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Data Types</td>
<td>CHAR</td>
<td>Stores a single character</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
<td>Stores a single character</td>
</tr>
<tr>
<td></td>
<td>CHAR(n)</td>
<td>Stores exactly (n) characters</td>
</tr>
<tr>
<td></td>
<td>VARCHAR(N)</td>
<td>Stores a variable number of characters, up to a maximum of n characters</td>
</tr>
<tr>
<td></td>
<td>TEXT</td>
<td>Specific variant of varchar, which does not require you to specify an upper limit on the number of characters</td>
</tr>
<tr>
<td>Numeric Data Types</td>
<td>NUMERIC (P, S)</td>
<td>Exact numeric of selectable precision</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>Single precision floating-point number (4 bytes)</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
<td>Double precision floating-point number (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>A signed 4-byte integer that can store −2147483648 to +2147483647</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>A signed 4-byte integer that can store −2147483648 to +2147483647</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>A signed 2-byte integer that can store −32768 to +32767</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>A signed 8-byte integer, giving approximately 18 digits of precision</td>
</tr>
<tr>
<td></td>
<td>BIT</td>
<td>Stores a single bit, 0 or 1</td>
</tr>
<tr>
<td></td>
<td>BIT VARYING</td>
<td>Stores a string of bits</td>
</tr>
<tr>
<td></td>
<td>MONEY</td>
<td>Equivalent to NUMERIC (9,2), storing 4 bytes of data. Its use is discouraged</td>
</tr>
<tr>
<td>Date &amp; Time Data Types</td>
<td>TIMESTAMP</td>
<td>Stores dates and times from 4713 BC to 1465001 AD, with a resolution of 1 microsecond - 8 bytes</td>
</tr>
<tr>
<td></td>
<td>INTERVAL</td>
<td>Stores an interval of approximately +/- 178,000,000 years, with a resolution of 1 microsecond - 16 bytes</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>Stores dates from 4713 BC to 32767 AD, with a resolution of 1 day - 4 bytes</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>Stores a time of day, from 0 to 23:59:59.99, with a resolution of 1 microsecond - 8 bytes with no timezone, 12 bytes with timezone</td>
</tr>
<tr>
<td>Logical Data Type</td>
<td>BOOLEAN</td>
<td>Holds a truth value. Will accept values such as TRUE, 't','true', 'y', 'yes', and '1' as true. Uses 1 byte of storage, and can store NULL. This type can be used upon table creation</td>
</tr>
<tr>
<td>XML Data Type</td>
<td>XML</td>
<td>XML data</td>
</tr>
<tr>
<td>Geometric Data Types</td>
<td>POINT</td>
<td>An x,y value</td>
</tr>
<tr>
<td></td>
<td>LINE</td>
<td>A line (pt1, pt2)</td>
</tr>
<tr>
<td></td>
<td>LSEG</td>
<td>A line segment (pt1, pt2)</td>
</tr>
<tr>
<td>PostgreSQL Data Type Family</td>
<td>PostgreSQL Data Type</td>
<td>PostgreSQL Data Type Characteristic</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>BOX</td>
<td>A box specified by a pair of points</td>
<td></td>
</tr>
<tr>
<td>PATH</td>
<td>A sequence of points, which may be closed or open</td>
<td></td>
</tr>
<tr>
<td>POLYGON</td>
<td>A sequence of points, effectively a closed path</td>
<td></td>
</tr>
<tr>
<td>CIRCLE</td>
<td>A point and a length, which specify a circle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PostgreSQL Data Types</th>
<th>SERIAL</th>
<th>A numeric column in a table that increases each time a row is added</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
<td>An object identifier. Internally, PostgreSQL adds, by default, a hidden oid to each row, and stores a 4-byte integer</td>
<td></td>
</tr>
<tr>
<td>CIDR</td>
<td>Stores a network address of the form x.x.x.x/y where y is the netmask</td>
<td></td>
</tr>
<tr>
<td>INET</td>
<td>Similar to cidr, except the host part can be 0</td>
<td></td>
</tr>
<tr>
<td>MACADDR</td>
<td>MAC (Media Access Control) address</td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>Textual JSON data</td>
<td></td>
</tr>
<tr>
<td>JSONB</td>
<td>Binary JSON data, decomposed</td>
<td></td>
</tr>
<tr>
<td>PG_LSN</td>
<td>PostgreSQL Log Sequence Number</td>
<td></td>
</tr>
<tr>
<td>BYTEA</td>
<td>Binary data (&quot;byte array&quot;)</td>
<td></td>
</tr>
<tr>
<td>TSQUERY</td>
<td>Text search query</td>
<td></td>
</tr>
<tr>
<td>TSVECTOR</td>
<td>Text search document</td>
<td></td>
</tr>
<tr>
<td>TXID_SNAPSHOT</td>
<td>User-level transaction ID snapshot</td>
<td></td>
</tr>
<tr>
<td>UUID</td>
<td>Universally unique identifier</td>
<td></td>
</tr>
</tbody>
</table>

**PostgreSQL Character Column Semantics**

PostgreSQL only supports **CHAR** for column size semantics. If you define a field as `VARCHAR (10)`, PostgreSQL can store 10 characters regardless of how many bytes it takes to store each non-English character. `VARCHAR (n)` stores strings up to n characters (not bytes) in length.

**Migration of Oracle Datatypes to PostgreSQL datatypes**

Automatic migration and conversion of Oracle Tables and Data Types can be performed using Amazon's Schema Conversion Tool (Amazon SCT).
Examples

To demonstrate SCT’s capability for migrating Oracle tables to their PostgreSQL equivalents, a table containing columns representing the majority of Oracle data types was created and converted using Amazon SCT.

Source Oracle compatible DDL for creating the `DATATYPES` table:

```sql
CREATE TABLE "DATATYPES" (  "BFILE" BFILE,  "BINARY_FLOAT" BINARY_FLOAT,  "BINARY_DOUBLE" BINARY_DOUBLE,  "BLOB" BLOB,  "CHAR" CHAR(10 BYTE),  "CHARACTER" CHAR(10 BYTE),  "CLOB" CLOB,  "NCLOB" NCLOB,  "DATE" DATE,  "DECIMAL" NUMBER(3,2),  "DEC" NUMBER(3,2),  "DOUBLE_PRECISION" FLOAT(126),  "FLOAT" FLOAT(3),  "INTEGER" NUMBER(*,0),  "INT" NUMBER(*,0),  "INTERVAL_YEAR" INTERVAL YEAR(4) TO MONTH,  "INTERVAL_DAY" INTERVAL DAY(4) TO SECOND(4),  "LONG" LONG,  "NCHAR" NCHAR(10),  "NCHAR_VARYING" NVARCHAR2(10),  "NUMBER" NUMBER(9,9),  "NUMBER1" NUMBER(9,0),  "NUMBER(*)" NUMBER,  "NUMERIC" NUMBER(9,9),  "NVARCHAR2" NVARCHAR2(10),  "RAW" RAW(10),  "REAL" FLOAT(63),  "ROW_ID" ROWID,  "SMALLINT" NUMBER(*,0),  "TIMESTAMP" TIMESTAMP(5),  "TIMESTAMP_WITH_TIME_ZONE" TIMESTAMP(5) WITH TIME ZONE,  "UROWID" UROWID(10),  "VARCHAR" VARCHAR2(10 BYTE),  "VARCHAR2" VARCHAR2(10 BYTE),  "XMLTYPE" XMLTYPE );
```
Target PostgreSQL compatible DDL for creating the `DATATYPES` table migrated from Oracle with Amazon SCT.

```sql
CREATE TABLE IF NOT EXISTS datatypes(
    bfile character varying(255) DEFAULT NULL,
    binary_float real DEFAULT NULL,
    binary_double double precision DEFAULT NULL,
    blob bytea DEFAULT NULL,
    char character(10) DEFAULT NULL,
    character character(10) DEFAULT NULL,
    clob text DEFAULT NULL,
    ncllob text DEFAULT NULL,
    date TIMESTAMP(0) without time zone DEFAULT NULL,
    decimal numeric(3,2) DEFAULT NULL,
    dec numeric(3,2) DEFAULT NULL,
    double_precision double precision DEFAULT NULL,
    float double precision DEFAULT NULL,
    integer numeric(38,0) DEFAULT NULL,
    int numeric(38,0) DEFAULT NULL,
    interval_year interval year to month(6) DEFAULT NULL,
    interval_day interval day to second(4) DEFAULT NULL,
    long text DEFAULT NULL,
    nchar character(10) DEFAULT NULL,
    nchar_varying character varying(10) DEFAULT NULL,
    number numeric(9,9) DEFAULT NULL,
    number1 numeric(9,0) DEFAULT NULL,
    "number(*)" double precision DEFAULT NULL,
    numeric numeric(9,9) DEFAULT NULL,
    nvarchar2 character varying(10) DEFAULT NULL,
    raw bytea DEFAULT NULL,
    real double precision DEFAULT NULL,
    row_id character(255) DEFAULT NULL,
    smallint numeric(38,0) DEFAULT NULL,
    timestamp TIMESTAMP(5) without time zone DEFAULT NULL,
    timestamp_with_time_zone TIMESTAMP(5) with time zone DEFAULT NULL,
    urowid character varying DEFAULT NULL,
    varchar character varying(10) DEFAULT NULL,
    varchar2 character varying(10) DEFAULT NULL,
    xmltype xml DEFAULT NULL
) WITH (OIDS=FALSE);
```

**Note:** While most of the datatypes were converted successfully, a few exceptions were raised for datatypes that Amazon SCT is unable to automatically convert and where SCT recommended manual actions:

- **PostgreSQL does not have a BFILE data type**
  BFILEs are pointers to binary files.

  **Recommended actions:** either store a named file with the data and create a routine that retrieves the file from the file system or store the data inside a blob datatype in your table.
• **PostgreSQL doesn’t have a ROWID data type**
  ROWIDs are physical row addresses inside Oracle’s storage subsystems. The **ROWID** datatype is primarily used for values returned by the **ROWID** pseudocolumn.

  **Recommended actions:** while PostgreSQL contains a **ctid** column that is the physical location of the row version within its table, it does not have a comparable data type. However, you can use **CHAR** as a partial datatype equivalent. Note: If you are using ROWID datatypes in your code, modifications may be necessary.

• **PostgreSQL does not have a UROWID data type**
  Universal rowid, or **UROWID**, is a single Oracle datatype that supports both logical and physical rowids of foreign table rowids such as non-Oracle tables accessed through a gateway.

  **Recommended actions:** PostgreSQL does not have a comparable data type. You can use **VARCHAR(n)** as a partial datatype equivalent. However, if you are using **UROWID** datatypes in your code, modifications may be necessary.

*For additional details:*
https://www.postgresql.org/docs/current/static/ddl-system-columns.html
https://www.postgresql.org/docs/current/static/datatype.html
https://aws.amazon.com/documentation/SchemaConversionTool
Overview
The Oracle database provides six types of constraints to enforce data integrity on table columns. Constraints ensure that data inserted into tables is controlled and satisfies logical requirements.

Oracle integrity constraint types:
- Primary key: enforces that row values in a specific column are unique and not null.
- Foreign key: enforces that values in the current table exist in the referenced table.
- Unique: prevents data duplication on a column, or combination of columns, and allows one null value.
- Check: enforces that values comply with a specific condition.
- Not null: enforces that null values cannot be inserted into a specific column.
- REF: references an object in another object type or in a relational table.

Constraint Creation
Oracle allows to create new constraints in two ways:

- Inline: Defines a constraint as part of a table column declaration:
  ```sql
  SQL> CREATE TABLE EMPLOYEES (
    EMP_ID NUMBER PRIMARY KEY,
  ...);
  ```

- Out-Of-Line: Defines a constraint as part of the table DDL during table creation:
  ```sql
  SQL> CREATE TABLE EMPLOYEES (
    EMP_ID NUMBER,
    ...
    CONSTRAINT PK_EMP_ID PRIMARY KEY(EMP_ID)) ;
  ```

Note: NOT NULL constraints must be declared using the inline method.

Oracle constraints can be specified with the following syntax:
- `CREATE / ALTER TABLE`
- `CREATE / ALTER VIEW`

Note: Views have only a primary key, foreign key, and unique constraints.
Major Constraint Types

PRIMARY KEY Constraint
A unique identifier for each record in a database table can appear only once and cannot contain NULL values. A table can only have one primary key.

When creating a primary key constraint inline, you can specify only the PRIMARY KEY keyword. When you create the constraint out-of-line, you must specify one column or combination of columns.

Creating a new primary key constraint will also implicitly create a unique index on the primary key column if no such index already exists. When dropping a primary key constraint, the system generated index is also dropped. If a user defined Index was used, the index is not dropped.

Limitations
- 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. Note: The data type TIMESTAMP WITH LOCAL TIME ZONE is allowed as primary key.
- Primary keys can be created from multiple columns (composite PK), limited to a total of 32 columns.
- Defining the same column as both a primary key and as a unique constraint is not allowed.

Examples
1. Create an Inline primary key using a system-generated primary key constraint name:

   ```sql
   SQL> CREATE TABLE EMPLOYEES (
   2   EMPLOYEE_ID NUMBER PRIMARY KEY,
   3   FIRST_NAME  VARCHAR2(20),
   4   LAST_NAME   VARCHAR2(25),
   5   EMAIL       VARCHAR2(25));
   ```

2. Create an inline primary key using a user-specified primary key constraint name:

   ```sql
   SQL> CREATE TABLE EMPLOYEES (
   2   EMPLOYEE_ID NUMBER CONSTRAINT PK_EMP_ID PRIMARY KEY,
   3   FIRST_NAME  VARCHAR2(20),
   4   LAST_NAME   VARCHAR2(25),
   5   EMAIL       VARCHAR2(25));
   ```

3. Create an out-of-line primary key:

   ```sql
   SQL> CREATE TABLE EMPLOYEES ( 
   2   EMPLOYEE_ID NUMBER,
   3   FIRST_NAME  VARCHAR2(20),
   4   LAST_NAME   VARCHAR2(25),
   5   EMAIL       VARCHAR2(25));
   6   CONSTRAINT PK_EMP_ID PRIMARY KEY (EMPLOYEE_ID));
   ```
4. Add a primary key to an existing table:

   SQL> ALTER TABLE SYSTEM_EVENTS
       ADD CONSTRAINT PK_EMP_ID PRIMARY KEY (EVENT_CODE, EVENT_TIME);

FOREIGN KEY Constraint
Foreign key constraints identify the relationship between column records defined with a foreign key constraint and a referenced primary key or a unique column. The main purpose of a foreign key is to enforce that the values in table A also exist in table B, as referenced by the foreign key.

A referenced table is known as a parent table while the table on which the foreign key was created is known as a child table. Foreign keys created in child tables generally reference a primary key constraint in a parent table.

Limitations
- 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.
- Composite Foreign key constraints, comprised from multiple columns, cannot have more than 32 columns.
- Foreign key constraints cannot be created in a CREATE TABLE statement with a subquery clause.
- A referenced primary key or unique constraint on a parent table must be created before the foreign key creation command.

ON DELETE Clause
The ON DELETE clause specifies the effect of deleting values from a parent table on the referenced records of a child table. If the ON DELETE clause is not specified, Oracle does not allow deletion of referenced key values in a parent table that has dependent rows in the child table.

- ON DELETE CASCADE: Any dependent foreign key values in a child table are removed along with the referenced values from the parent table.
- ON DELETE NULL: Any dependent foreign key values in a child table are updated to NULL.

Examples

1. Create an inline foreign key with a user-defined constraint name:

   SQL> CREATE TABLE EMPLOYEES (
       EMPLOYEE_ID NUMBER PRIMARY KEY,
       FIRST_NAME VARCHAR2(20),
       LAST_NAME VARCHAR2(25),
       EMAIL VARCHAR2(25),
       DEPARTMENT_ID REFERENCES DEPARTMENTS(DEPARTMENT_ID));
2. Create an Out-Of-Line foreign key with a system-generated constraint name:

```sql
SQL> CREATE TABLE EMPLOYEES (  
    EMPLOYEE_ID   NUMBER PRIMARY KEY,
    FIRST_NAME    VARCHAR2(20),
    LAST_NAME     VARCHAR2(25),
    EMAIL         VARCHAR2(25),
    DEPARTMENT_ID NUMBER,
    CONSTRAINT FK_FEP_ID
        FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID)) ;
```

3. Create a foreign key using the `ON DELETE CASCADE` clause:

```sql
SQL> CREATE TABLE EMPLOYEES (  
    EMPLOYEE_ID   NUMBER PRIMARY KEY,
    FIRST_NAME    VARCHAR2(20),
    LAST_NAME     VARCHAR2(25),
    EMAIL         VARCHAR2(25),
    DEPARTMENT_ID NUMBER,
    CONSTRAINT FK_FEP_ID
        FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID)
        ON DELETE CASCADE) ;
```

4. Add a foreign key to an existing table:

```sql
SQL> ALTER TABLE EMPLOYEES  
ADD CONSTRAINT FK_FEP_ID  
FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID) ;
```

**UNIQUE Constraint**

A unique constraint is similar to a primary key constraint. A unique constraint specifies that the values in a single column, or combination of columns, must be unique and cannot repeat in multiple rows.

The main difference from primary key constraint is that the unique constraint can contain NULL values. NULL values in multiple rows are also supported provided the combination of values is unique.

**Limitations**

- A unique constraint cannot be created on columns defined with the following data types: LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, BFILE, REF, TIMESTAMP WITH TIME ZONE
- A unique constraint comprised from multiple columns cannot have more than 32 columns.
- Primary key and unique constraints cannot be created on the same column or columns.
Example
Create an inline unique Constraint:

```
SQL> CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID   NUMBER PRIMARY KEY,  
  FIRST_NAME    VARCHAR2(20),  
  LAST_NAME     VARCHAR2(25),  
  EMAIL         VARCHAR2(25) CONSTRAINT UNIQ_EMP_EMAIL UNIQUE,  
  DEPARTMENT_ID NUMBER);  
```

Check Constraint
Check constraints are used to validate that values in specific columns meet specific criteria or conditions. For example, a check constraint on an `EMPLOYEE_EMAIL` column can be used to validate that each record has an “@aws.com” suffix, if a record fails the “check” validation, an error is raised and the record is not inserted.

Using a check constraint can help transfer some of the logical integrity validation from the application to the database.

In-Line vs. Out-Of-Line
When creating a check constraint as inline, it can only be defined on a specific column. When using the out-of-line method, the check constraint can be defined on multiple columns.

Limitations
- Check constraints cannot perform validation on columns of other tables.
- Check constraints cannot work with functions that not deterministic (e.g. `CURRENT_DATE`).
- Check constraints cannot work with user-defined functions.
- Check constrains cannot work with pseudo columns such as: `CURRVAL`, `NEXTVAL`, `LEVEL`, or `ROWNUM`.

Example
Create an inline check constraint that uses a regular expression to validate that the email suffix of inserted rows contains “@aws.com”.

```
SQL> CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID   NUMBER PRIMARY KEY,  
  FIRST_NAME    VARCHAR2(20),  
  LAST_NAME     VARCHAR2(25),  
  EMAIL         VARCHAR2(25) CHECK(REGEXP_LIKE (EMAIL, '^[A-Za-z]+@aws.com?{1,3}$'))),  
  DEPARTMENT_ID NUMBER);  
```

Not Null Constraint
The not null constraint prevents a column from containing any null values. In order to enable the not null constraint, the keywords `NOT NULL` must be specified during table creation (inline only). Permitting null values is the default if `NOT NULL` is not specified.

Example

```
SQL> CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID   NUMBER PRIMARY KEY,  
  FIRST_NAME    VARCHAR2(20) NOT NULL,  
  LAST_NAME     VARCHAR2(25) NOT NULL,  
  EMAIL         VARCHAR2(25),  
  DEPARTMENT_ID NUMBER);  
```
REF Constraint
REF constraints define a relationship between a column of type REF and the object it references. The REF constraint can be created both inline and out-of-line. Both methods permit you to define a scope constraint, a rowid constraint, or a referential integrity constraint based on the REF column.

Example
1. Create a new Oracle type object:

   ```sql
   SQL> CREATE TYPE DEP_TYPE AS OBJECT
   2                     (DEP_NAME VARCHAR2(60),
   3                     DEP_ADDRESS VARCHAR2(300));
   ```

2. Create a table based on the previously created type object:

   ```sql
   SQL> CREATE TABLE DEPARTMENTS_OBJ_T OF DEP_TYPE;
   ```

3. Create the EMPLOYEES table with a reference to the previously created DEPARTMENTS table that is based on the DEP_TYPE object:

   ```sql
   SQL> CREATE TABLE EMPLOYEES (EMP_NAME VARCHAR2(60),
   2                     EMP_EMAIL VARCHAR2(60),
   3                     EMP_DEPT REF DEPARTMENT_TYP REFERENCES DEPARTMENTS_OBJ_T);
   ```

Special Constraint States
Oracle provides granular control of database constraint enforcement. For example, you can disable constraints temporarily while making modifications to table data.

Constraint states can be defined using the CREATE TABLE / ALTER TABLE statements. The following constraint states are supported:

- **DEFERRABLE**: Enables the use of the SET CONSTRAINT clause in subsequent transactions until a COMMIT statement is submitted.
- **NOT DEFERRABLE**: Disables the use of the SET CONSTRAINT clause.
- **INITIALLY IMMEDIATE**: Checks the constraint at the end of each subsequent SQL statement (this state is the default).
- **INITIALLY DEFERRED**: Checks the constraint at the end of subsequent transactions.
- **VALIDATE | NO VALIDATE**: These parameters depend on whether the constraint is ENABLED or DISABLED.
- **ENABLE | DISABLE**: Specifies if the constraint should be enforced after creation (ENABLE by default). Several options are available when using ENABLE | DISABLE:
  - **ENABLE VALIDATE**: Enforces that the constraint applies to all existing and new data.
  - **ENABLE NOVALIDATE**: Only new data complies with the constraint.
  - **DISABLE VALIDATE**: A valid constraint is created in disabled mode with no index.
- **DISABLE NOVALIDATE**: The constraint is created in disabled mode without validation of new or existing data.

**Example**

1. Create a unique constraint with a state of DEFERRABLE:

   ```sql
   SQL> CREATE TABLE EMPLOYEES (
       2   EMPLOYEE_ID NUMBER PRIMARY KEY,
       3   FIRST_NAME VARCHAR2(20),
       4   LAST_NAME  VARCHAR2(25),
       5   EMAIL      VARCHAR2(25) CONSTRAINT UNIQ_EMP_EMAIL UNIQUE DEFERRABLE,
       6   DEPARTMENT_ID NUMBER);
   ```

2. Modify the state of the constraint to ENABLE NOVALIDATE:

   ```sql
   SQL> ALTER TABLE EMPLOYEES
   2     ADD CONSTRAINT CHK_EMP_NAME CHECK(FIRST_NAME LIKE 'a%' )
   3     ENABLE NOVALIDATE;
   ```

**Using Existing Indexes to Enforce Constraint Integrity** *(using_index_clause)*

Primary key and unique constraints can be created based on an existing index to enforce the constraint integrity instead of implicitly creating a new index during constraint creation.

**Example**

Create a unique constraint based on an existing index:

```sql
SQL> CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES(EMPLOYEE_ID);
SQL> ALTER TABLE EMPLOYEES
2     ADD CONSTRAINT PK_CON_UNIQ
3     PRIMARY KEY(EMPLOYEE_ID) USING INDEX IDX_EMP_ID;
```

**Required Privileges for Creating Constraints**

You must have privileges on the table in which constraints are created and, in case of foreign key constraints, you must have the REFERENCES privilege on the referenced table.

*For additional details:*

https://docs.oracle.com/cd/B28359_01/server.111/b28286/clauses002.htm#SQLRF52163

https://docs.oracle.com/database/121/SQLRF/clauses002.htm#SQLRF52180
Overview

PostgreSQL supports the following types of table constraints:

- PRIMARY KEY
- FOREIGN KEY
- UNIQUE
- CHECK
- NOT NULL
- EXCLUDE (specific to PostgreSQL)

**Note:** PostgreSQL *does not* support Oracle’s REF constraint.

Similar to constraint deceleration in Oracle, PostgreSQL allows creating constraints in-line or out-of-line during table column specification.

PostgreSQL constraints can be specified using `CREATE / ALTER TABLE`. Views are not supported.

Privileges

You must have privileges on the table in which constrains will be created. With foreign key constraints, you must also have the REFERENCES privilege.

Primary Key Constraint

- Uniquely identifies each record and cannot contain a NULL value.
- Uses the same ANSI SQL syntax as Oracle.
- Can be created on a single column or on multiple columns (“composite primary keys”) as the only PRIMARY KEY in a table.
- Create a PRIMARY KEY constraint creates a unique B-Tree index automatically on the column or group of columns marked as the primary key of the table.
- Constraint names can be generated automatically by PostgreSQL or explicitly specified during constraint creation.

Examples

1. Create an inline primary key constraint with a system-generated constraint name:

```sql
demo=> CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID NUMERIC PRIMARY KEY,  
  FIRST_NAME  VARCHAR(20),  
  LAST_NAME   VARCHAR(25),  
  EMAIL       VARCHAR(25));
```
2. Create an inline primary key constraint with a user-specified constraint name:

```sql
CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID NUMERIC CONSTRAINT PK_EMP_ID PRIMARY KEY,  
  FIRST_NAME VARCHAR(20),  
  LAST_NAME VARCHAR(25),  
  EMAIL VARCHAR(25));
```

3. Create an out-of-line primary key constraint:

```sql
CREATE TABLE EMPLOYEES (  
  EMPLOYEE_ID NUMERIC,  
  FIRST_NAME VARCHAR(20),  
  LAST_NAME VARCHAR(25),  
  EMAIL VARCHAR(25));  
CONSTRAINT PK_EMP_ID PRIMARY KEY (EMPLOYEE_ID));
```

4. Add a primary key constraint to an existing table:

```sql
ALTER TABLE SYSTEM_EVENTS  
ADD CONSTRAINT PK_EMP_ID PRIMARY KEY (EVENT_CODE, EVENT_TIME);
```

5. Drop the primary key:

```sql
ALTER TABLE SYSTEM_EVENTS  
DROP CONSTRAINT PK_EMP_ID;
```

**Foreign Key Constraint**

- Enforces referential integrity in the database. Values in specific columns or group of columns must match the values from another table (or column).
- Creating a FOREIGN KEY constraint in PostgreSQL uses the same ANSI SQL syntax as Oracle.
- Can be created in-line or out-of-line during table creation.
- Use the REFERENCES clause to specify the table referenced by the foreign key constraint.
- When specifying REFERENCES in absence of a column list in the referenced table, the PRIMARY KEY of the referenced table is used as the referenced column or columns.
- A table can have multiple FOREIGN KEY constraints to describe its relationships with other tables.
- Use the ON DELETE clause to handle cases of FOREIGN KEY parent records deletions (such as cascading deletes).
- Foreign key constraint names are generated automatically by the database or specified explicitly during constraint creation.

**Foreign Key and the ON DELETE clause**

PostgreSQL provides three main options to handle cases where data is deleted from the parent table and a child table is referenced by a FOREIGN KEY constraint. By default, without specifying any additional options, PostgreSQL will use the NO ACTION method and raise an error if the referencing rows still exist when the constraint is verified.

- **ON DELETE CASCADE**
  Any dependent foreign key values in the child table are removed along with the referenced values from the parent table.
• **ON DELETE RESTRICT**
  Prevents the deletion of referenced values from the parent table and the deletion of dependent foreign key values in the child table.

• **ON DELETE NO ACTION**
  Performs no action (the default action). The fundamental difference between RESTRICT and NO ACTION is that NO ACTION allows the check to be postponed until later in the transaction; RESTRICT does not.

**Foreign Key and the ON UPDATE clause**
Handling updates on FOREIGN KEY columns is also available using the ON UPDATE clause, which shares the same options as the ON DELETE clause:

• **ON UPDATE CASCADE**
• **ON UPDATE RESTRICT**
• **ON UPDATE NO ACTION**

Note: Oracle does not provide an ON UPDATE clause.

**Examples**
1. Create an inline foreign key with a user-specified constraint name:

   ```sql
demo=> CREATE TABLE EMPLOYEES (  
   EMPLOYEE_ID   NUMERIC PRIMARY KEY,  
   FIRST_NAME    VARCHAR(20),  
   LAST_NAME     VARCHAR(25),  
   EMAIL         VARCHAR(25),  
   DEPARTMENT_ID NUMERIC,  
   CONSTRAINT FK_FEP_ID  
   FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID));
```

*PostgreSQL foreign key columns must have a specified data type while Oracle doesn’t*

2. Create an out-of-line foreign key constraint with a system-generated constraint name:

   ```sql
demo=> CREATE TABLE EMPLOYEES (  
   EMPLOYEE_ID   NUMERIC PRIMARY KEY,  
   FIRST_NAME    VARCHAR(20),  
   LAST_NAME     VARCHAR(25),  
   EMAIL         VARCHAR(25),  
   DEPARTMENT_ID NUMERIC,  
   CONSTRAINT FK_FEP_ID  
   FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID));
```

3. Create a foreign key using the ON DELETE CASCADE clause:

   ```sql
demo=> CREATE TABLE EMPLOYEES (  
   EMPLOYEE_ID   NUMERIC PRIMARY KEY,  
   FIRST_NAME    VARCHAR(20),  
   LAST_NAME     VARCHAR(25),  
   EMAIL         VARCHAR(25),  
   DEPARTMENT_ID NUMERIC,  
   CONSTRAINT FK_FEP_ID  
   FOREIGN KEY(DEPARTMENT_ID) REFERENCES DEPARTMENTS(DEPARTMENT_ID)  
   ON DELETE CASCADE);
4. Add a foreign key to an existing table:

```sql
demo=> ALTER TABLE EMPLOYEES
    ADD CONSTRAINT FK_FEP_ID
        FOREIGN KEY(DEPARTMENT_ID)
        REFERENCES DEPARTMENTS(DEPARTMENT ID);
```

**UNIQUE Constraints**
- Ensures that a value in a column, or a group of columns, is unique across the entire table.
- PostgreSQL UNIQUE constraint syntax is ANSI SQL compatible.
- Automatically creates a B-Tree index on the respective column, or a group of columns, when creating a UNIQUE constraint.
- If duplicate values exist in the column(s) on which the constraint was defined during UNIQUE constraint creation, the UNIQUE constraint creation fails, returning an error message.
- UNIQUE constraints in PostgreSQL will accept multiple NULL values (similar to Oracle).
- UNIQUE constraint naming can be system-generated or explicitly specified.

**Example**
Create an inline unique constraint ensuring uniqueness of values in the email column:

```sql
demo=> CREATE TABLE EMPLOYEES (  
    EMPLOYEE_ID   NUMERIC PRIMARY KEY,  
    FIRST_NAME    VARCHAR(20),  
    LAST_NAME     VARCHAR(25),  
    EMAIL         VARCHAR(25) CONSTRAINT UNIQ_EMP_EMAIL UNIQUE,  
    DEPARTMENT_ID NUMERIC);
```

**CHECK Constraint**
- CHECK constraints enforce that values in a column satisfy a specific requirement.
- CHECK constraints in PostgreSQL use the same ANSI SQL syntax as Oracle.
- Can only be defined using a **Boolean data type** to evaluate the values of a column.
- CHECK constraints naming can be system-generated or explicitly specified by the user during constraint creation.

**Example**
Create an inline CHECK constraint, using a regular expression, to enforce that the email column contains email addresses with an “@aws.com” suffix.

```sql
demo=> CREATE TABLE EMPLOYEES (  
    EMPLOYEE_ID   NUMERIC PRIMARY KEY,  
    FIRST_NAME    VARCHAR(20),  
    LAST_NAME     VARCHAR(25),  
    EMAIL         VARCHAR(25) CHECK(EMAIL ~ '^[A-Za-z]+@[A-Za-z]+.com$'),  
    DEPARTMENT_ID NUMERIC);
```
NOT NULL Constraints

- **NOT NULL** constraints enforce that a column *cannot* accept NULL values. This behavior is different from the default column behavior in PostgreSQL where columns *can* accept NULL values.
- **NOT NULL** constraints can only be defined inline, during table creation (similar to Oracle).
- **NOT NULL** constraints in PostgreSQL use the same ANSI SQL syntax as Oracle.
- You can explicitly specify names for **NOT NULL** constraints when used with a **CHECK** constraint.

Example

Define two not null constraints on the **FIRST_NAME** and **LAST_NAME** columns. Define a check constraint (with an explicitly user-specified name) to enforce not null behavior on the **EMAIL** column.

```
CREATE TABLE EMPLOYEES (  
EMPLOYEE_ID NUMERIC PRIMARY KEY,  
FIRST_NAME  VARCHAR(20) NOT NULL,  
LAST_NAME   VARCHAR(25) NOT NULL,  
EMAIL       VARCHAR(25) CONSTRAINT CHK_EMAIL  
                 CHECK(EMAIL IS NOT NULL));
```

Constraint States

Similarly to Oracle, PostgreSQL provides controls for certain aspects of constraint behavior:

- **DEFERRABLE** | NOT DEFERRABLE
  Using the PostgreSQL **SET CONSTRAINTS** statement, constraints can be defined as:

  - **DEFERRABLE**
    Allows you to use the **SET CONSTRAINTS** statement to set the behavior of constraint checking within the current transaction until transaction commit.

  - **IMMEDIATE**
    Constraints are enforced only at the end of each statement.
    Note: Each constraint has its own **IMMEDIATE** or **DEFERRED** mode (same as Oracle)

  - **NOT DEFERRABLE**
    This statement always runs as **IMMEDIATE** and is not affected by the **SET CONSTRAINTS** command.

PostgreSQL SET CONSTRAINTS Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

- **VALIDATE CONSTRAINT** | NOT VALID

  - **VALIDATE CONSTRAINT**
    Validates foreign key or check constraints (only) that were previously created as **NOT VALID**. This action performs a validation check by scanning the table to ensure that all records satisfy the constraint definition.
- NOT VALID
  Can be used only for foreign key or check constraints. When specified, new records are not validated with the creation of the constraint. Only when the VALIDATE CONSTRAINT state is applied does the constraint state is enforced on all records.

Example

```
demo=> ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_DEPT
      FOREIGN KEY (department_id)
      REFERENCES DEPARTMENTS (department_id) NOT VALID;
demo=> ALTER TABLE EMPLOYEES VALIDATE CONSTRAINT FK_DEPT;
```

Using Existing Indexes During Constraint Creation *(table_constraint_using_index)*
PostgreSQL can add a new primary key or unique constraints based on an existing unique Index. All the index columns are included in the constraint. When creating constraints using this method, the index is owned by the constraint. When dropping the constraint, the index is also dropped.

Example
Use an existing unique Index to create a primary key constraint:

```
demo=> CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES(EMPLOYEE_ID);
demo=> ALTER TABLE EMPLOYEES
      ADD CONSTRAINT PK_CON_UNIQ PRIMARY KEY USING INDEX IDX_EMP_ID;
```
Oracle Constraints Comparison to PostgreSQL

<table>
<thead>
<tr>
<th>Oracle Constraint / Parameter</th>
<th>PostgreSQL Constraint / Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY KEY</td>
<td>PRIMARY KEY</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>FOREIGN KEY</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>CHECK</td>
<td>CHECK</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>REF</td>
<td>Not Supported</td>
</tr>
<tr>
<td>DEFERRABLE</td>
<td>DEFERRABLE</td>
</tr>
<tr>
<td>NOT DEFERRABLE</td>
<td>NOT DEFERRABLE</td>
</tr>
<tr>
<td>SET CONSTRAINTS</td>
<td>SET CONSTRAINTS</td>
</tr>
<tr>
<td>INITIALLY IMMEDIATE</td>
<td>INITIALLY IMMEDIATE</td>
</tr>
<tr>
<td>INITIALLY DEFERRED</td>
<td>INITIALLY DEFERRED</td>
</tr>
<tr>
<td>ENABLE</td>
<td>Default, not supported as keyword</td>
</tr>
<tr>
<td>DISABLE</td>
<td>Not supported as keyword, NOT VALID can use instead</td>
</tr>
<tr>
<td>ENABLE VALIDATE</td>
<td>Default, not supported as keyword</td>
</tr>
<tr>
<td>ENABLE NOVALIDATE</td>
<td>NOT VALID</td>
</tr>
<tr>
<td>DISABLE VALIDATE</td>
<td>Not supported</td>
</tr>
<tr>
<td>DISABLE NOVALIDATE</td>
<td>Not supported</td>
</tr>
<tr>
<td>USING_INDEX_CLAUSE</td>
<td>table_constraint_using_index</td>
</tr>
<tr>
<td>View Constraints</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Metadata:</td>
<td>Metadata:</td>
</tr>
<tr>
<td>DBA_CONSTRAINTS</td>
<td>PG_CONSTRAINT</td>
</tr>
</tbody>
</table>

For additional details:
https://www.postgresql.org/docs/9.6/static/ddl-constraints.html
https://www.postgresql.org/docs/9.6/static/sql-altertable.html
Overview
The purpose of database partitioning is to provide support for very large tables and indexes by splitting them into smaller pieces, known as partitions. Each partition has its own name and definitions and can be managed separately from other partitions, or collectively as one object. From an application perspective, partitions are transparent - partitioned tables act the same as non-partitioned tables allowing your applications to access a partitioned table using unmodified SQL statements. Table partitioning provides several benefits:

- **Performance improvement**
  Table partitions help improve query performance by accessing a subset of a partitions instead of scanning a larger set of data. Additional performance improvements can also be achieved when using partitions and parallel query execution for DML and DDL operations.

- **Data Management**
  Table partitions facilitate easier data management operations (such as data migration), index management (creation, dropping, or rebuilding indexes), and backup/recovery. These operations are also referred to as “Information Lifecycle Management” (ILM) activities.

- **Maintenance Operations**
  Table partitions can significantly reduce downtime caused by table maintenance operations.

Oracle basic Table Partitioning methods

- **Hash Table Partitioning**
  When a partition key is specified (for example, a table column with a number data type), Oracle applies a hashing algorithm to evenly distribute the data (records) among all defined partitions (partitions have approximately the same size).

Example
Create a hash-partitioned Table:

```sql
SQL> CREATE TABLE SYSTEM_LOGS
    (EVENT_NO NUMBER NOT NULL,
     EVENT_DATE DATE NOT NULL,
     EVENT_STR VARCHAR2(500),
     ERROR_CODE VARCHAR2(10))
PARTITION BY HASH (ERROR_CODE)
PARTITIONS 3
STORE IN (TB1, TB2, TB3);
```
• **List Table Partitioning**
You can specify a list of discrete values for the table partitioning key in the description of each partition. This type of table partitioning enables control over partition organization using explicit values. For example, partition “events” by error code values.

**Example**
Create a list-partition table:

```sql
SQL> CREATE TABLE SYSTEM_LOGS
    2 (EVENT_NO NUMBER NOT NULL,
    3 EVENT_DATE DATE NOT NULL,
    4 EVENT_STR VARCHAR2(500),
    5 ERROR_CODE VARCHAR2(10))
PARTITION BY LIST (ERROR_CODE)
    1 (PARTITION warning VALUES ('err1', 'err2', 'err3') TABLESPACE TB1,
    2 PARTITION critical VALUES ('err4', 'err5', 'err6') TABLESPACE TB2);
```

• **Range Table Partitioning**
Partition a table based on a *range* of values. The Oracle database assigns rows to table partitions based on column values falling within a given range. Range table partitioning is one of the most frequently used type of partitioning in the Oracle database, primarily with date values. Range table partitioning can also be implemented with numeric ranges (1-10000, 10001- 20000...).

**Example**
Create a range-partitioned table:

```sql
SQL> CREATE TABLE SYSTEM_LOGS
    2 (EVENT_NO NUMBER NOT NULL,
    3 EVENT_DATE DATE NOT NULL,
    4 EVENT_STR VARCHAR2(500))
PARTITION BY RANGE (EVENT_DATE)
    1 (PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2015',
    2 'DD/MM/YYYY')) TABLESPACE TB1,
    3 PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2016',
    4 'DD/MM/YYYY')) TABLESPACE TB2,
    5 PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2017',
    6 'DD/MM/YYYY')) TABLESPACE TB3);
```

**Composite Table Partitioning**
With composite partitioning, a table can be partitioned by one data distribution method and then each partition can be further subdivided into sub-partitions using the same, or different, data distribution method(s). For example:

- Composite list-range partitioning
- Composite list-list partitioning
- Composite range-hash partitioning
Partitioning Extensions
Oracle provides additional partitioning strategies that enhance the capabilities of basic partitioning. These partitioning strategies are:

- Manageability extensions
  - Interval partitioning
  - Partition advisor

- Partitioning key extensions
  - Reference partitioning
  - Virtual column-based partitioning

Examples
Split and exchange partitions:

- **Split Partitions**
  The `SPLIT PARTITION` statement can be used to redistribute the contents of one partition or sub-partition into multiple partitions or sub-partitions:

```
SQL> ALTER TABLE SPLIT PARTITION p0 INTO
  PARTITION P01 VALUES LESS THAN (100),
  PARTITION p02);
```

- **Exchange Partitions**
  The `EXCHANGE PARTITION` statement is useful to exchange table partitions in, or out, of a partitioned table.

```
SQL> ALTER TABLE orders
  EXCHANGE PARTITION p_ord3 WITH TABLE orders_year_2016;
```

Sub-Partitioning Tables

Sub-Partitions are created within partitions to further split the parent partition:

```
SQL> PARTITION BY RANGE(department_id)
  SUBPARTITION BY HASH(last_name)
  SUBPARTITION TEMPLATE
  (SUBPARTITION a TABLESPACE ts1,
   SUBPARTITION b TABLESPACE ts2,
   SUBPARTITION c TABLESPACE ts3,
   SUBPARTITION d TABLESPACE ts4
  )
  (PARTITION p1 VALUES LESS THAN (1000),
   PARTITION p2 VALUES LESS THAN (2000),
   PARTITION p3 VALUES LESS THAN (MAXVALUE)
  ,
```
Automatic List Partitioning (Oracle 12c only)
Automatic-list partitioning is an enhancement of Oracle list partitioning. Automatic-list partitioning enables the automatic creation of new partitions for new values inserted into the list-partitioned table. An automatic list-partitioned table is created with only one partition. The database creates the additional table partitions automatically.

Example
Create an automatic list-partitioned table:

```sql
SQL> CREATE TABLE SYSTEM_LOGS
    2 (EVENT_NO NUMBER NOT NULL,
    3     EVENT_DATE DATE   NOT NULL,
    4     EVENT_STR VARCHAR2(500),
    5     ERROR_CODE VARCHAR2(10))
    6  PARTITION BY LIST (ERROR_CODE) AUTOMATIC
    7   (PARTITION warning VALUES ('err1', 'err2', 'err3'))
```

For additional information on Oracle Automatic List Partitioning:
Migration to: PostgreSQL Table Inheritance

Overview
The table partitioning mechanism in PostgreSQL differs from Oracle. Partitioning in PostgreSQL is implemented using “table inheritance”. Each table partition is represented by a child table referenced to a single parent table. The parent table should be empty and is only used to represent the entire table data set (as a metadata dictionary and as a query source).

Partitioning management operations are performed directly on the child tables. Querying is performed directly on the parent table.

For additional information on PostgreSQL Table Inheritance, see: https://www.postgresql.org/docs/9.6/static/ddl-inherit.html

Implementing List “Table Partitioning”

1. Create a parent table (“master table”) from which all child tables (“partitions”) will inherit.
2. Create child tables (which act similar to Table Partitions) that inherit from the parent table, the child tables should have an identical structure to the parent table.
3. Create indexes on each child table. Optionally, add constraints (for example, primary keys or check constraints) to define allowed values in each table.
4. Create a database trigger to redirect data inserted into the parent table to the appropriate child table.
5. Ensure the PostgreSQL constraint_exclusion parameter is enabled and set to partition. This parameter insures that the queries are optimized for working with table partitions.

```
demo=# show constraint_exclusion;
constraint_exclusion
----------------------
partition
```

For additional information on PostgreSQL constraint_exclusion parameter: https://www.postgresql.org/docs/9.6/static/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION

PostgreSQL 9.6 does not support “declarative partitioning” as well as several of the table partitioning features available in Oracle. Alternatives, such as for replacing Oracle’s interval table partitioning, include using application-centric methods using PL/pgSQL or other programing languages.

Notes:
- PostgreSQL 9.6 Table Partitioning does not support the creation of foreign keys on the parent table. Alternative solutions include application-centric methods such as using triggers/functions.
- PostgreSQL 9.6 does not support sub-partitions and does not support SPLIT and EXCHANGE of table partitions.
### Oracle versus PostgreSQL Partitioning Comparison Table

<table>
<thead>
<tr>
<th>Oracle Table Partition Type</th>
<th>Build-In PostgreSQL Support</th>
<th>Link To Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Yes</td>
<td><a href="https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html">PostgreSQL List Partitioning</a></td>
</tr>
<tr>
<td>Range</td>
<td>Yes</td>
<td><a href="https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html">PostgreSQL Range Partitioning</a></td>
</tr>
<tr>
<td>Hash</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Composite Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Interval Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Partition Advisor</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Reference Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Virtual Column Based Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Automatic List Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Sub Partitioning</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Split / Exchange Partitions</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

For additional details:
Example
Steps for creating a PostgreSQL “list-partitioned table”:

1. Create the parent table:

   ```
   demo=# CREATE TABLE SYSTEM_LOGS
   (EVENT_NO NUMERIC NOT NULL,
   EVENT_DATE DATE NOT NULL,
   EVENT_STR VARCHAR(500),
   ERROR_CODE VARCHAR(10));
   ```

2. Create child tables (“partitions”) with check constraints:

   ```
   demo=# CREATE TABLE SYSTEM_LOGS_WARNING
   (    CHECK (ERROR_CODE IN('err1', 'err2', 'err3'))
   INHERITS (SYSTEM_LOGS);
   demo=# CREATE TABLE SYSTEM_LOGS_CRITICAL
   (    CHECK (ERROR_CODE IN('err4', 'err5', 'err6'))
   INHERITS (SYSTEM_LOGS);
   ```

3. Create indexes on each of the child tables (“partitions”):

   ```
   demo=# CREATE INDEX IDX_SYSTEM_LOGS_WARNING ON
   SYSTEM_LOGS_WARNING(ERROR_CODE);
   demo=# CREATE INDEX IDX_SYSTEM_LOGS_CRITICAL ON
   SYSTEM_LOGS_CRITICAL(ERROR_CODE);
   ```

4. Create a function to redirect data inserted into the Parent Table:

   ```
   demo=# CREATE OR REPLACE FUNCTION SYSTEM_LOGS_ERR_CODE_INS()
   RETURNS TRIGGER AS
   $$
   BEGIN
   IF (NEW.ERROR_CODE IN('err1', 'err2', 'err3')) THEN
   INSERT INTO SYSTEM_LOGS_WARNING VALUES (NEW.*);
   ELSIF (NEW.ERROR_CODE IN('err4', 'err5', 'err6')) THEN
   INSERT INTO SYSTEM_LOGS_CRITICAL VALUES (NEW.*);
   ELSE
   RAISE EXCEPTION 'Value out of range, check
   SYSTEM_LOGS_ERR_CODE_INS() Function!';
   END IF;
   RETURN NULL;
   END;
   $$
   LANGUAGE plpgsql;
   ```

5. Attach the trigger function created above to log to the table:

   ```
   demo=# CREATE TRIGGER SYSTEM_LOGS_ERR_TRIG
   BEFORE INSERT ON SYSTEM_LOGS
   FOR EACH ROW EXECUTE PROCEDURE SYSTEM_LOGS_ERR_CODE_INS();
   ```
6. Insert data directly into the parent table:

```sql
demo=# INSERT INTO SYSTEM_LOGS VALUES(1, '2015-05-15', 'a...', 'err1');
demo=# INSERT INTO SYSTEM_LOGS VALUES(2, '2016-06-16', 'b...', 'err3');
demo=# INSERT INTO SYSTEM_LOGS VALUES(3, '2017-07-17', 'c...', 'err6');
```

7. View results from across all the different child tables:

```sql
demo=# SELECT * FROM SYSTEM_LOGS;
<table>
<thead>
<tr>
<th>event_no</th>
<th>event_date</th>
<th>event_str</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2015-05-15</td>
<td>a...</td>
</tr>
<tr>
<td>2</td>
<td>2016-06-16</td>
<td>b...</td>
</tr>
<tr>
<td>3</td>
<td>2017-07-17</td>
<td>c...</td>
</tr>
</tbody>
</table>

demo=# SELECT * FROM SYSTEM_LOGS_WARNING;
<table>
<thead>
<tr>
<th>event_no</th>
<th>event_date</th>
<th>event_str</th>
<th>error_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2015-05-15</td>
<td>a...</td>
<td>err1</td>
</tr>
<tr>
<td>2</td>
<td>2016-06-16</td>
<td>b...</td>
<td>err3</td>
</tr>
</tbody>
</table>

demo=# SELECT * FROM SYSTEM_LOGS_CRITICAL;
<table>
<thead>
<tr>
<th>event_no</th>
<th>event_date</th>
<th>event_str</th>
<th>error_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2017-07-17</td>
<td>c...</td>
<td>err6</td>
</tr>
</tbody>
</table>
Example
Steps for creating a PostgreSQL “range-partitioned table”:

1. Create the parent table:

```
demo=# CREATE TABLE SYSTEM_LOGS
     (EVENT_NO    NUMERIC NOT NULL,
      EVENT_DATE  DATE   NOT NULL,
      EVENT_STR   VARCHAR(500));
```

2. Create the child tables (“partitions”) with check constraints:

```
demo=# CREATE TABLE SYSTEM_LOGS_2015
     (CHECK (EVENT_DATE >= DATE '2015-01-01'
              AND EVENT_DATE < DATE '2016-01-01'))
     ) INHERITS (SYSTEM_LOGS);

demo=# CREATE TABLE SYSTEM_LOGS_2016
     (CHECK (EVENT_DATE >= DATE '2016-01-01'
              AND EVENT_DATE < DATE '2017-01-01'))
     ) INHERITS (SYSTEM_LOGS);

demo=# CREATE TABLE SYSTEM_LOGS_2017
     (CHECK (EVENT_DATE >= DATE '2017-01-01'
              AND EVENT_DATE <= DATE '2017-12-31'))
     ) INHERITS (SYSTEM_LOGS);
```

3. Create indexes on each child table (“partitions”):

```
demo=# CREATE INDEX IDX_SYSTEM_LOGS_2015 ON
      SYSTEM_LOGS_2015(EVENT_DATE);

demo=# CREATE INDEX IDX_SYSTEM_LOGS_2016 ON
      SYSTEM_LOGS_2016(EVENT_DATE);

demo=# CREATE INDEX IDX_SYSTEM_LOGS_2017 ON
      SYSTEM_LOGS_2017(EVENT_DATE);
```
4. Create a function to redirect data inserted into the parent table:

```sql
CREATE OR REPLACE FUNCTION SYSTEM_LOGS_INS ()
RETURNS TRIGGER AS
$$
BEGIN
    IF (NEW.EVENT_DATE >= DATE '2015-01-01' AND NEW.EVENT_DATE <
        DATE '2016-01-01') THEN
        INSERT INTO SYSTEM_LOGS_2015 VALUES (NEW.*);
    ELSIF (NEW.EVENT_DATE >= DATE '2016-01-01' AND NEW.EVENT_DATE <
        DATE '2017-01-01') THEN
        INSERT INTO SYSTEM_LOGS_2016 VALUES (NEW.*);
    ELSIF (NEW.EVENT_DATE >= DATE '2017-01-01' AND NEW.EVENT_DATE <=
        DATE '2017-12-31') THEN
        INSERT INTO SYSTEM_LOGS_2017 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. check SYSTEM_LOGS_INS ()
        function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

5. Attach the trigger function created above to log to the SYSTEM_LOGS table:

```sql
CREATE TRIGGER SYSTEM_LOGS_TRIG
BEFORE INSERT ON SYSTEM_LOGS
FOR EACH ROW EXECUTE PROCEDURE SYSTEM_LOGS_INS ();
```

6. Insert data directly to the parent table:

```sql
demo=# INSERT INTO SYSTEM_LOGS VALUES (1, '2015-05-15', 'a...');
demo=# INSERT INTO SYSTEM_LOGS VALUES (2, '2016-06-16', 'b...');
demo=# INSERT INTO SYSTEM_LOGS VALUES (3, '2017-07-17', 'c...');
```

7. Test the solution by selecting data from the parent and child tables:

```sql
demo=# SELECT * FROM SYSTEM_LOGS;
    event_no | event_date | event_str
    ---------+------------+-----------
       1 | 2015-05-15 | a...
       2 | 2016-06-16 | b...
       3 | 2017-07-17 | c...

demo=# SELECT * FROM SYSTEM_LOGS_2015;
    event_no | event_date | event_str
    ---------+------------+-----------
       1 | 2015-05-15 | a...
```
Migrating from: Oracle Temporary Tables

Overview
Oracle enables you to create temporary tables for storing data that should exist only for the duration of a session or transaction.

Oracle uses the `CREATE GLOBAL TEMPORARY TABLE` statement to create a temporary table. This type of table has a persistent DDL structure, but not persistent data, and does not generate redo during DML. Two of the primary use-cases for temporary tables include:

- Processing many rows as part of a batch operation while requiring staging tables to store intermediate results.
- Data is required only for the duration of a specific session. When the session ends, the session data should be cleared.

When using temporary tables, the data is visible only to the session that inserts the data into the table.

Oracle Global Temporary Tables Notes:

- Global Temporary Tables store data inside the Oracle Temporary Tablespace.
- DDL operations on a temporary table are permitted including: `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`.
- Temporary tables cannot be partitioned, clustered, or created as Index-Organized Tables. Also, they do not support parallel `UPDATE`, `DELETE` and `MERGE`.
- Foreign key constraints cannot be created on temporary tables.
- Processing DML operations on a Temporary Table does not generate Redo Data. However, Undo Data for the rows and Redo Data for the Undo Data itself are generated.
- Indexes can be created for a Temporary Table and are treated as Temporary Indexes. Temporary Tables also support Triggers.
- Temporary Tables cannot be named after an existing table object and cannot be dropped while containing records, even from another session.
Session-specific and Transaction-specific Temporary Table syntax:

- **ON COMMIT**
  This clause is associated only with Temporary Tables. It specifies whether the temporary table data persists for the duration of a transaction or a session.

  - **PRESERVE ROWS**
    When the session ends, all data is truncated but persists beyond the end of the transaction.

  - **DELETE ROWS**
    The default behavior. Data is truncated after each commit.

Oracle 12c Temporary Table enhancements:

- **Global Temporary Table statistics**
  Prior to Oracle 12c, statistics on temporary tables were common to all sessions. Oracle 12c introduces session-specific statistics for Temporary Tables. Statistics can be configured using the `DBMS_STATS` preference `GLOBAL_TEMP_TABLE_STATS`, which can be set to `SHARED` or `SESSION`.

- **Global Temporary Table Undo**
  Performing DML operations on a Temporary Table does not generate Redo data, but does generate Undo Data that eventually, by itself, will generate Redo records. Oracle 12c provides an option to store the temporary Undo Data in the Temporary Tablespace itself. This feature is configured using the `temp_undo_enabled` parameter with the options `TRUE` or `FALSE`.

For additional details:
[https://docs.oracle.com/database/121/REFRN/GUID-E2A01A84-2D63-401F-B64E-C96B18C5DCA6.htm#REFRN10326](https://docs.oracle.com/database/121/REFRN/GUID-E2A01A84-2D63-401F-B64E-C96B18C5DCA6.htm#REFRN10326)
Examples
Create an Oracle Global Temporary Table (with \texttt{ON COMMIT PRESERVE ROWS}):

```sql
SQL> CREATE GLOBAL TEMPORARY TABLE EMP_TEMP (
    EMP_ID NUMBER PRIMARY KEY,
    EMP_FULL_NAME VARCHAR2(60) NOT NULL,
    AVG_SALARY NUMERIC NOT NULL)
    ON COMMIT PRESERVE ROWS;

SQL> CREATE INDEX IDX_EMP_TEMP_FN ON EMP_TEMP(EMP_FULL_NAME);

SQL> INSERT INTO EMP_TEMP VALUES(1, 'John Smith', '5000');

SQL> COMMIT;

SQL> SELECT * FROM SCT.EMP_TEMP;

    EMP_ID EMP_FULL_NAME        AVG_SALARY  
---------- -------------------- ----------
        1 John Smith                 5000
```

Create an Oracle Global Temporary Table (with \texttt{ON COMMIT DELETE ROWS}):

```sql
SQL> CREATE GLOBAL TEMPORARY TABLE EMP_TEMP (   EMP_ID NUMBER PRIMARY KEY,
    EMP_FULL_NAME VARCHAR2(60) NOT NULL,
    AVG_SALARY NUMERIC NOT NULL)
    ON COMMIT DELETE ROWS;

SQL> INSERT INTO EMP_TEMP VALUES(1, 'John Smith', '5000');

SQL> COMMIT;

SQL> SELECT * FROM SCT.EMP_TEMP;

```

For additional details:

- [https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_7002.htm#SQLRF01402](https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_7002.htm#SQLRF01402)
- [https://docs.oracle.com/database/121/SQLRF/statements_7002.htm](https://docs.oracle.com/database/121/SQLRF/statements_7002.htm)
Migration to: PostgreSQL Temporary Tables

Overview
PostgreSQL Temporary Tables share many similarities with Oracle Global Temporary Tables.
From a syntax perspective, PostgreSQL Temporary Tables are referred to as “Temporary Tables” (without Oracle’s Global definition). The implementation is mostly identical.

In terms of differences, Oracle stores the temporary table structure (DDL) for repeated use – even after a database restart – but does not store rows persistently. PostgreSQL implements temporary tables differently: the table structure (DDL) is not stored in the database. When a session ends, the temporary table is dropped.

- **Session-specific** - In PostgreSQL, every session is required to create its own Temporary Tables. Each session can create its own “private” Temporary Tables, using identical table names.

- **LOCAL / GLOBAL syntax** - PostgreSQL temporary tables do not support cross-session data access. PostgreSQL does not distinguish between “GLOBAL” and “LOCAL” temporary tables. The use of these keywords *is permitted* in PostgreSQL, but they have *no effect* because PostgreSQL creates Temporary Tables as local and session-isolated tables.

  *Note:* use of the GLOBAL keyword is deprecated.

- **In the Oracle Database, the default behavior when the ON COMMIT clause is omitted is ON COMMIT DELETE ROWS. In PostgreSQL, the default is ON COMMIT PRESERVE ROWS.**

**PostgreSQL Temporary Tables ON COMMIT clause:**

- **ON COMMIT**
The clause specifies the state of the data as it persists for the duration of a transaction or a session.

  - **PRESERVE ROWS**
The PostgreSQL default. When a session ends, all data is truncated but persists beyond the end of the transaction.

  - **DELETE ROWS**
The data is truncated after each commit.
Examples

1. Create a Temporary Table, with ON DELETE PRESERVE ROWS:

   ```sql
demo=> CREATE GLOBAL TEMPORARY TABLE EMP_TEMP (
   EMP_ID       NUMERIC PRIMARY KEY,
   EMP_FULL_NAME VARCHAR(60) NOT NULL,
   AVG_SALARY   NUMERIC NOT NULL
   ON COMMIT PRESERVE ROWS;

   demo=> CREATE INDEX IDX_EMP_TEMP_FN ON EMP_TEMP(EMP_FULL_NAME);

   demo=> INSERT INTO EMP_TEMP VALUES(1, 'John Smith', '5000');

   demo=> COMMIT;

   demo=> SELECT * FROM SCT.EMP_TEMP;
   emp_id | emp_full_name | avg_salary
   --------+---------------+------------
   1 | John Smith    | 5000

   demo=> DROP TABLE EMP_TEMP;
   DROP TABLE
   ```

2. Create and use a Temporary Table, with ON COMMIT DELETE ROWS:

   ```sql
demo=> CREATE GLOBAL TEMPORARY TABLE EMP_TEMP (
   EMP_ID       NUMERIC PRIMARY KEY,
   EMP_FULL_NAME VARCHAR(60) NOT NULL,
   AVG_SALARY   NUMERIC NOT NULL
   ON COMMIT DELETE ROWS;

   demo=> INSERT INTO EMP_TEMP VALUES(1, 'John Smith', '5000');

   demo=> COMMIT;

   demo=> SELECT * FROM SCT.EMP_TEMP;
   emp_id | emp_full_name | avg_salary
   --------+---------------+------------
   l | John Smith   | 5000

   demo=> DROP TABLE EMP_TEMP;
   DROP TABLE
   ```
## Oracle Global Temporary Tables vs. PostgreSQL Temporary Tables:

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Oracle Temporary Tables</th>
<th>PostgreSQL Temporary Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global Temporary Table</td>
<td>Temporary Table / Temp Table</td>
</tr>
<tr>
<td><strong>Create table</strong></td>
<td>CREATE GLOBAL TEMPORARY...</td>
<td>CREATE GLOBAL TEMPORARY...</td>
</tr>
<tr>
<td></td>
<td>CREATE TEMPORARY...</td>
<td>CREATE TEMPORARY...</td>
</tr>
<tr>
<td></td>
<td>CREATE TEMP...</td>
<td>CREATE TEMP...</td>
</tr>
<tr>
<td>Accessible from multiple sessions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Temp table DDL persist after session end / database restart</td>
<td>Yes</td>
<td>No (dropped at the end of the session)</td>
</tr>
<tr>
<td>Create index support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Foreign key support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>ON COMMIT default</strong></td>
<td>COMMIT DELETE ROWS</td>
<td>ON COMMIT PRESERVE ROWS</td>
</tr>
<tr>
<td><strong>ON COMMIT PRESERVE ROWS</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>ON COMMIT DELETE ROWS</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Alter table support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Gather statistics</td>
<td>dbms_stats.gather_table_stats</td>
<td>ANALYZE</td>
</tr>
<tr>
<td>Oracle 12c</td>
<td>dbms_stats.set_table_prefs</td>
<td>ANALYZE</td>
</tr>
</tbody>
</table>

For additional details:
Overview
Oracle provides a method to mark columns as “unused”. Unused columns are not physically dropped, but are treated as if they were dropped. Unused columns cannot be restored. Select statements do not retrieve data from columns marked as unused and are not displayed when executing a `DESCRIBE` table command.

The main advantage of setting a column to `UNUSED` is to reduce possible high database load when dropping a column from a large table. To overcome this issue, a column can be marked as unused and then be physically dropped later.

To set a column as unused, use the `SET UNUSED` clause.

Example

```
SQL> ALTER TABLE EMPLOYEES SET UNUSED (COMMISSION_PCT);
SQL> ALTER TABLE EMPLOYEES SET UNUSED (JOB_ID, COMMISSION_PCT);
```

Display unused columns:

```
SQL> SELECT * FROM USER_UNUSED_COL_TABS;
```

```
<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEES</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Drop the Column Permanently (physically drop the column):

```
SQL> ALTER TABLE EMPLOYEES DROP UNUSED COLUMNS;
```

For additional details:
https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_3001.htm
http://docs.oracle.com/database/121/SQLRF/statements_3001.htm
Migration to: PostgreSQL Alter Table

Overview
PostgreSQL does not support marking table columns as “unused”. However, when executing the `ALTER TABLE… DROP COLUMN` command, the drop column statement does not physically remove the column; it only makes it invisible to SQL operations. As such, dropping a column is a “fast” action, but does not reduce the on-disk size of your table immediately because the space occupied by the dropped column is not reclaimed.

The unused space is reclaimed by new DML actions, as they use the space that once was occupied by the dropped column. To force an immediate reclamation of storage space, the `VACUUM FULL` command should be used. Alternatively, execute an `ALTER TABLE` statement to forces a rewrite.

Example
1. PostgreSQL “crop column” statement:
   ```sql
demo=> ALTER TABLE EMPLOYEES DROP COLUMN COMMISSION_PCT;
```
2. Verify the operation:
   ```sql
demo=> SELECT TABLE_NAME, COLUMN_NAME
   FROM INFORMATION_SCHEMA.COLUMNS
   WHERE TABLE_NAME = 'emps1' AND COLUMN_NAME=LOWER('COMMISSION_PCT');
```
   |

   | table_name | column_name |
   |------------+-------------|
   | (0 rows)   |

3. Use the `VACUUM FULL` command to reclaim unused space from storage:
   ```sql
demo=> VACUUM FULL EMPLOYEES;
```
4. Run the `VACUUM FULL` statement with the `VERBOSE` option to display an activity report of the vacuum process that includes the tables vacuumed and the time taken to perform the vacuum operation:
   ```sql
demo=> VACUUM FULL VERBOSE EMPLOYEES;
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-altertable.html
https://www.postgresql.org/docs/9.6/static/sql-vacuum.html
Migrating from: Oracle Virtual Columns

Overview
Oracle Virtual Columns appear as normal columns but their values are calculated instead of being stored in the database. Virtual Columns cannot be created based on other Virtual Columns and can only reference columns from the same table. When creating a Virtual Column, you can explicitly specify the datatype or let the database choose the datatype based on the expression.

Notes
- Virtual Columns can be used with Constraints, Indexes, Table Partitioning, and Foreign Keys.
- Functions in expressions must be deterministic at the time of table creation.
- Virtual Columns cannot be manipulated by DML operations.
- Virtual Columns can be used in a `WHERE` clause and as part of DML commands.
- When creating an index on a virtual column, a Function Based Index is created.
- Virtual columns do not support Index-Organized Tables, external, objects, Clusters, or Temporary Tables.
- The output of a Virtual Column expression must be a Scalar value.
- The Virtual Column keyword `GENERATED ALWAYS AS` and `VIRTUAL` are not mandatory and provided for clarity only.
- The keyword `AS` after the column name can indicate the column is created as a Virtual Column.
- A Virtual Column does not need to be specified in an `INSERT` statement.

Example
1. Create a table that includes two Virtual Columns:

```
SQL> CREATE TABLE EMPLOYEES (  
EMPLOYEE_ID NUMBER,  
FIRST_NAME VARCHAR2(20),  
LAST_NAME VARCHAR2(25),  
USER_NAME VARCHAR2(25),  
EMAIL AS (LOWER(USER_NAME) || '@aws.com'),  
HIRE_DATE DATE,  
BASE_SALARY NUMBER,  
SALES_COUNT NUMBER,  
FINEL_SALARY NUMBER GENERATED ALWAYS AS  
(CASE WHEN SALES_COUNT >= 10 THEN BASE_SALARY + (BASE_SALARY * (SALES_COUNT * 0.05)) END) VIRTUAL);```
2. Insert a new record into the table without specifying values for the Virtual Column:

```sql
SQL> INSERT INTO EMPLOYEES
       (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, USER_NAME, HIRE_DATE,
        BASE_SALARY, SALES_COUNT)
    VALUES(1, 'John', 'Smith', 'jsmith', '17-JUN-2003', 5000, 21);
```

3. Select the email Virtual Column from the table:

```sql
SQL> SELECT email FROM EMPLOYEES;

EMAIL                FINEL_SALARY
-------------------- ------------
jsmith@aws.com              10250
```

For additional details:
https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_7002.htm#SQLRF01402
https://docs.oracle.com/database/121/SQLRF/statements_7002.htm#SQLRF01402
Migration to: PostgreSQL Virtual Columns

[Back to TOC]

Overview
PostgreSQL does not provide a feature that is directly equivalent to a Virtual Column in Oracle. However, there are workarounds to emulate similar functionality.

Alternatives for Virtual Columns:

- **Views**
  Create a View using the function for the “Virtual Column” as part of the View syntax.

- **Function as a column**
  Create a function that receives column values from table records (as parameters) and returns a modified value according to a specific expression. The function serves as a Virtual Column equivalent. You can create a PostgreSQL Expression Index (equivalent to Oracle’s Function Based index) that is based on the function.

Example
The email address for a user is calculated based on the USER_NAME column that is a physical property of the table.

1. Create a table that includes a USER_NAME column but does not include an email address column:
   ```sql
demo=> CREATE TABLE EMPLOYEES (  
     EMPLOYEE_ID NUMERIC PRIMARY KEY,  
     FIRST_NAME   VARCHAR(20),  
     LAST_NAME    VARCHAR(25),  
     USER_NAME    VARCHAR(25));
```

2. Create a PL/pgSQL function which receives the USER_NAME value and return the full email address:
   ```sql
demo=> CREATE OR REPLACE FUNCTION USER_EMAIL(EMPLOYEES)  
  RETURNS text AS $$  
  SELECT (LOWER($1.USER_NAME) || '@aws.com')  
  $$ STABLE LANGUAGE SQL;
```

3. Insert data to the table, including a value for USER_NAME. During insert, no reference to the USER_EMAIL function is made:
   ```sql
demo=> INSERT INTO EMPLOYEES  
  (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, USER_NAME)  
  VALUES(1, 'John', 'Smith', 'jsmith'),  
  (2, 'Steven', 'King', 'sking');
```
4. Use the `USER_EMAIL` function as part of a `SELECT` statement:

```sql
demo=> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, USER_NAME, USER_EMAIL(EMPLOYEES) FROM EMPLOYEES;
```

<table>
<thead>
<tr>
<th>employee_id</th>
<th>first_name</th>
<th>last_name</th>
<th>user_name</th>
<th>user_email</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Smith</td>
<td>jsmith</td>
<td><a href="mailto:jsmith@aws.com">jsmith@aws.com</a></td>
</tr>
<tr>
<td>2</td>
<td>Steven</td>
<td>King</td>
<td>sking</td>
<td><a href="mailto:sking@aws.com">sking@aws.com</a></td>
</tr>
</tbody>
</table>
```

5. Create a view that incorporates the `USER_EMAIL` function:

```sql
demo=> CREATE VIEW employees_function AS SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, USER_NAME, USER_EMAIL(EMPLOYEES) FROM EMPLOYEES;
```

6. Create an **Expression Based Index** on the `USER_EMAIL` column for improved performance:

```sql
demo=> CREATE INDEX IDX_USER_EMAIL ON EMPLOYEES(USER_EMAIL(EMPLOYEES));
```

7. Verify the Expression Based Index with `EXPLAIN`:

```sql
demo=> SET enable_seqscan = OFF;
demo=> EXPLAIN SELECT * FROM EMPLOYEES WHERE USER_EMAIL(EMPLOYEES) = 'jsmith@aws.com';
```

```
QUERY PLAN
---------------------------------------------------------------------------
Index Scan using idx_user_email on employees  (cost=0.13..8.14 rows=1 width=294)
Index Cond: ((lower((user_name)::text) || '@aws.com'::text) = 'jsmith@aws.com'::text)
```

DML Support

Using triggers, you can populate column values automatically as “Virtual Columns”. For this approach, create two PostgreSQL objects:

- Create a function containing the data modification logic based on table column data.
- Create a trigger to use the function and execute it as part of the DML.

Example

In the following example, the `FULL_NAME` column is automatically populated by the values using data from the `FIRST_NAME` and `LAST_NAME` columns.

1. Create the table:

   ```sql
   CREATE TABLE EMPLOYEES ( 
   EMPLOYEE_ID NUMERIC PRIMARY KEY, 
   FIRST_NAME   VARCHAR(20), 
   LAST_NAME    VARCHAR(25), 
   FULL_NAME    VARCHAR(25));
   ```

2. Create a function to concatenate the `FIRST_NAME` and `LAST_NAME` columns:

   ```sql
   CREATE OR REPLACE FUNCTION FUNC_USER_FULL_NAME ()
   RETURNS trigger as '
   BEGIN
   NEW.FULL_NAME = NEW.FIRST_NAME || '' '' || NEW.LAST_NAME;
   RETURN NEW;
   END;
   ' LANGUAGE plpgsql;
   ```

3. Create a trigger that uses the function created in the previous step. The function will execute before an insert:

   ```sql
   CREATE TRIGGER TRG_USER_FULL_NAME 
   BEFORE INSERT OR UPDATE 
   ON EMPLOYEES FOR EACH ROW 
   EXECUTE PROCEDURE FUNC_USER_FULL_NAME();
   ```

4. Verify the functionality of the trigger:

   ```sql
   INSERT INTO EMPLOYEES (EMPLOYEE_ID, FIRST_NAME, LAST_NAME) 
   VALUES(1, 'John', 'Smith'),(2, 'Steven', 'King');
   ```

   ```sql
   SELECT  * FROM EMPLOYEES;
   ```

<table>
<thead>
<tr>
<th>employee_id</th>
<th>first_name</th>
<th>last_name</th>
<th>full_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Smith</td>
<td>John Smith</td>
</tr>
<tr>
<td>2</td>
<td>Steven</td>
<td>King</td>
<td>Steven King</td>
</tr>
</tbody>
</table>
5. Create an Index based on the “virtual” `FULL_NAME` column:

```sql
demo=> CREATE INDEX IDX_USER_FULL_NAME
      ON EMPLOYEES(FULL_NAME);
```

6. Verify the Expression Based Index with `EXPLAIN`:

```sql
demo=> SET enable_seqscan = OFF;
demo=> EXPLAIN
    SELECT * FROM EMPLOYEES
    WHERE FULL_NAME = 'John Smith';
```

```
QUERY PLAN
---------------------------------------------------------------------------
Index Scan using idx_user_full_name on employees  (cost=0.13..8.14 rows=1
 width=226)
  Index Cond: ((full_name)::text = 'John Smith'::text)
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createtrigger.html
Migrating from: Oracle User Defined Types

Overview
Oracle refers to User Defined Types (UDTs) as OBJECT TYPES. They are managed using PL/SQL.
User Defined Types enable you to create application-dedicated, complex data types that are based on, and extend, the built-in Oracle data types.

The `CREATE TYPE` statement supports creating:
- Objects Types
- Varying Array (varray) types
- Nested Table types
- Incomplete Types
- Additional types such as an SQLJ object type (Java class mapped to SLQ user defined type)

Examples

1. Create an Oracle Object Type to store an employee phone number:

```sql
SQL> CREATE OR REPLACE TYPE EMP_PHONE_NUM AS OBJECT ( 
    PHONE_NUM VARCHAR2(11));
SQL> CREATE TABLE EMPLOYEES ( 
    EMP_ID NUMBER PRIMARY KEY, 
    EMP_PHONE EMP_PHONE_NUM NOT NULL);
SQL> INSERT INTO EMPLOYEES VALUES(1, EMP_PHONE_NUM('111-222-333'));
SQL> SELECT a.EMP_ID, a.EMP_PHONE.PHONE_NUM FROM EMPLOYEES a;
```

```yaml
EMP_ID EMP_PHONE.PHONE_NUM
--------- -------------
1 111-222-333
```
2. Create an Oracle Object Type as a “collection of attributes” for the employees table:

```sql
SQL> CREATE OR REPLACE TYPE EMP_ADDRESS AS OBJECT (
    STATE VARCHAR2(2),
    CITY VARCHAR2(20),
    STREET VARCHAR2(20),
    ZIP_CODE NUMBER);

SQL> CREATE TABLE EMPLOYEES (
    EMP_ID NUMBER PRIMARY KEY,
    EMP_NAME VARCHAR2(10) NOT NULL,
    EMP_ADDRESS EMP_ADDRESS NOT NULL);

SQL> INSERT INTO EMPLOYEES
    VALUES(1, 'John Smith',
           EMP_ADDRESS('AL', 'Gulf Shores', '3033 Joyce Street', '36542'));

SQL> SELECT a.EMP_ID,
            a.EMP_NAME,
            a.EMP_ADDRESS.STATE,
            a.EMP_ADDRESS.CITY,
            a.EMP_ADDRESS.STREET,
            a.EMP_ADDRESS.ZIP_CODE
    FROM EMPLOYEES a;

EMP_ID EMP_NAME    STATE  CITY         STREET             ZIP_CODE
------ ----------- ------ ------------ ------------------ ------
1      John Smith  AL     Gulf Shores  3033 Joyce Street  36542
```

For additional details:
http://docs.oracle.com/cloud/latest/db112/SQLRF/statements_8001.htm#SQLRF01506
http://docs.oracle.com/cloud/latest/db112/LNPLS/create_type.htm#LNPLS01375
Migration to: PostgreSQL User Defined Types

Overview

Similar to Oracle, PostgreSQL enables creation of User Defined Types using the `CREATE TYPE` statement. A User Defined Type is owned by the user who creates it. If a schema name is specified, the type is created under the specified schema.

PostgreSQL supports the creation of several different User Defined Types:

- **Composite Types**
  Stores a single named attribute that is attached to a data type or multiple attributes as an attribute collection. In PostgreSQL, you can also use the `CREATE TYPE` statement standalone with an association to a table.

- **Enumerated Types** (`enum`)
  Stores a static ordered set of values. For example, product categories:

  ```sql
demo=> CREATE TYPE PRODUCT_CATEGORT AS ENUM ('Hardware', 'Software', 'Document');
  ```

- **Range Types**
  Stores a range of values, for example, a range of timestamps used to represent the ranges of time of when a course is scheduled.

  ```sql
demo=> CREATE TYPE float8_range AS RANGE
  (subtype = float8, subtype_diff = float8mi);
  ```

  For more information on PostgreSQL Range Types:
  [https://www.postgresql.org/docs/9.6/static/rangetypes.html](https://www.postgresql.org/docs/9.6/static/rangetypes.html)

- **Base Types**
  These types are the system core types (abstract types) and are implemented in a low-level language such as C.

- **Array Types**
  Support definition of columns as multidimensional arrays. An array column can be created with a built-in type or a user-defined base type, enum type, or composite.

  ```sql
demo=> CREATE TABLE COURSE_SCHEDULE (
    COURSE_ID NUMERIC PRIMARY KEY,
    COURSE_NAME VARCHAR(60),
    COURSE_SCHEDULES text[]);
  ```

  For additional details:
  [https://www.postgresql.org/docs/9.1/static/arrays.html](https://www.postgresql.org/docs/9.1/static/arrays.html)
CREATE TYPE name AS RANGE (  
    SUBTYPE = subtype  
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]  
    [ , COLLATION = collation ]  
    [ , CANONICAL = canonical_function ]  
    [ , SUBTYPE_DIFF = subtype_diff_function ]  
  )

CREATE TYPE name (  
    INPUT = input_function,  
    OUTPUT = output_function  
    [ , RECEIVE = receive_function ]  
    [ , SEND = send_function ]  
    [ , TYPMOD_IN = type_modifier_input_function ]  
    [ , TYPMOD_OUT = type_modifier_output_function ]  
    [ , ANALYZE = analyze_function ]  
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]  
    [ , PASSEDBYVALUE ]  
    [ , ALIGNMENT = alignment ]  
    [ , STORAGE = storage ]  
    [ , LIKE = like_type ]  
    [ , CATEGORY = category ]  
    [ , PREFERRED = preferred ]  
    [ , DEFAULT = default ]  
    [ , ELEMENT = element ]  
    [ , DELIMITER = delimiter ]  
    [ , COLLATABLE = collatable ]  
  )

PostgreSQL syntax differences from Oracle’s CREATE TYPE Statement:
- PostgreSQL does not support: CREATE OR REPLACE TYPE.
- PostgreSQL does not accept: AS OBJECT.
Examples
1. Create a User Define Type as a dedicated type for storing an employee phone number:

```sql
demo=> CREATE TYPE EMP_PHONE_NUM AS (PHONE_NUM VARCHAR(11));
demo=> CREATE TABLE EMPLOYEES (EMP_ID NUMERIC PRIMARY KEY, EMP_PHONE EMP_PHONE_NUM NOT NULL);
demo=> INSERT INTO EMPLOYEES VALUES(1, ROW('111-222-333'));
demo=> SELECT a.EMP_ID, (a.EMP_PHONE).PHONE_NUM FROM EMPLOYEES a;
```

```
emp_id | phone_num
--------+-------------
    1 | 111-222-333
```

(1 row)

2. Create a PostgreSQL Object Type as a collection of Attributes for the employees table:

```sql
demo=> CREATE OR REPLACE TYPE EMP_ADDRESS AS OBJECT (STATE VARCHAR(2), CITY VARCHAR(20), STREET VARCHAR(20), ZIP_CODE NUMERIC);
demo=> CREATE TABLE EMPLOYEES (EMP_ID NUMERIC PRIMARY KEY, EMP_NAME VARCHAR(10) NOT NULL, EMP_ADDRESS EMP_ADDRESS NOT NULL);
demo=> INSERT INTO EMPLOYEES VALUES(1, 'John Smith', ('AL', 'Gulf Shores', '3033 Joyce Street', '36542'));
```

```
emp_name | state |    city     |      street       | zip_code
----------+-------+-------------+-------------------+----------
John Smith | AL    | Gulf Shores | 3033 Joyce Street | 36542
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createtype.html
https://www.postgresql.org/docs/9.6/static/rowtypes.html
Overview
Beginning with Oracle 11g, tables can be marked as “read-only”, which prevents DML operations from altering table data.

Prior to Oracle 11g, the only way to set a table to “read only” mode was by limiting table privileges to SELECT. The table owner was still able to perform read and write operations. Starting with Oracle 11g, users can execute an ALTER TABLE statement and change the table mode to either READ ONLY or READ WRITE.

Oracle 12c Release 2 introduces greater granularity for read-only objects and supports “read only” table partitions. Any attempt to perform a DML operation on a partition, or sub-partition, set to READ ONLY in Oracle 12.2 results in an error.

Notes:
- SELECT FOR UPDATE statements are not allowed.
- DDL operations are permitted if they do not modify table data.
- Operations on indexes are allowed on tables set to READ ONLY mode.

Example
Oracle READ ONLY and READ WRITE Modes:

```
SQL> CREATE TABLE EMP_READ_ONLY (
    EMP_ID    NUMBER PRIMARY KEY,
    EMP_FULL_NAME VARCHAR2(60) NOT NULL);

SQL> INSERT INTO EMP_READ_ONLY VALUES(1, 'John Smith');
1 row created

SQL> ALTER TABLE EMP_READ_ONLY READ ONLY;

SQL> INSERT INTO EMP_READ_ONLY VALUES(2, 'Steven King');
ORA-12081: update operation not allowed on table "SCT"."TBL_READ_ONLY"

SQL> ALTER TABLE EMP_READ_ONLY READ WRITE;

SQL> INSERT INTO EMP_READ_ONLY VALUES(2, 'Steven King');
1 row created

SQL> COMMIT;

SQL> SELECT * FROM EMP_READ_ONLY;

EMP_ID EMP_FULL_NAME
---------- -----------------
1 John Smith
2 Steven King
```
For additional details:
https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_3001.htm
http://docs.oracle.com/database/121/SQLRF/statements_3001.htm
https://docs.oracle.com/database/122/VLDBG/release-changes.htm#GUID-3878B6B7-DBE7-440D-9BCA-E5469E7AE88B__READ-ONLYPARTITIONS-5B55A563
Overview
PostgreSQL does not provide an equivalent to the READ ONLY mode supported in Oracle.

The following alternatives could be used as workarounds:
- “Read-only” User or Role.
- “Read-only” database.
- Creating a “read-only” database trigger or a using a “read-only” constraint.

PostgreSQL “read-only” User or Role
To achieve some degree of protection from unwanted DML operations on table for a specific Database User, you can grant the user only the SELECT privilege on the table and set the user default_transaction_read_only parameter to ON.

Example
Create a PostgreSQL User with READ ONLY privileges:

```
CREATE TABLE EMP_READ_ONLY (
    EMP_ID       NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL);
CREATE USER aws_readonly PASSWORD 'aws_readonly';
ALTER USER aws_readonly SET DEFAULT_TRANSACTION_READ_ONLY=ON;
GRANT SELECT ON EMP_READ_ONLY TO aws_readonly;
-- Open a new session with user “aws_readonly”
SELECT * FROM EMP_READ_ONLY;

emp_id | emp_full_name
--------+---------------
(0 rows)
```

ERROR: cannot execute INSERT in a read-only transaction
PostgreSQL “read-only” database

As an alternative solution for restricting write operations on database objects, a dedicated “read-only” PostgreSQL database can be created to store all “read-only” tables. PostgreSQL supports multiple databases under the same database instance. Adding a dedicated “read-only” database is a simple and straightforward solution.

- Set the `DEFAULT_TRANSACTION_READ_ONLY` to `ON` for a database. If a session attempts to perform DDL or DML operations, an error will be raised.
- The database can be altered back to `READ WRITE` mode when the parameter is set to “OFF”.

Example

Create a PostgreSQL READ ONLY database:

```sql
demo=> CREATE DATABASE readonly_db;
CREATE DATABASE

demo=> ALTER DATABASE readonly_db SET DEFAULT_TRANSACTION_READ_ONLY=ON;
ALTER DATABASE
-- Open a new session connected to the “readonly_db” database

demo=> CREATE TABLE EMP_READ_ONLY (
    EMP_ID        NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
-- In case of an existing table

demo=> INSERT INTO EMP_READ_ONLY VALUES(1, 'John Smith');
ERROR:  cannot execute INSERT in a read-only transaction
```

“Read-only” Database Trigger

An `INSTEAD OF` trigger can be created to prevent data modifications on a specific table, such as restricting INSERT, UPDATE, DELETE and TRUNCATE.

Example

1. Create PostgreSQL function which contains the logic for restricting to “read-only” operations:

```sql
demo=> CREATE OR REPLACE FUNCTION READONLY_TRIGGER_FUNCTION()
RETURNS TRIGGER AS $$
BEGIN
    RAISE EXCEPTION 'THE "%" TABLE IS READ ONLY!',
    TG_TABLE_NAME using hint = 'Operation Ignored';
RETURN NULL;
END;
$$ language 'plpgsql';
```
2. Create a trigger which will execute the function that was previously created:

```sql
demo=> CREATE TRIGGER EMP_READONLY_TRIGGER
    BEFORE INSERT OR UPDATE OR DELETE OR TRUNCATE
    ON EMP_READ_ONLY FOR EACH STATEMENT
    EXECUTE PROCEDURE READONLY_TRIGGER_FUNCTION();
```

3. Test DML & truncate commands against the table with the new trigger:

```sql
demo=> INSERT INTO EMP_READ_ONLY VALUES(1, 'John Smith');
ERROR:  THE "EMP_READ_ONLY" TABLE IS READ ONLY!
HINT:  Operation Ignored
CONTEXT:  PL/pgSQL function readonly_trigger_function() line 3 at RAISE

demo=> TRUNCATE TABLE SRC;
ERROR:  THE "EMP_READ_ONLY" TABLE IS READ ONLY!
HINT:  Operation Ignored
CONTEXT:  PL/pgSQL function readonly_trigger_function() line 3 at RAISE
```

For additional details:
https://www.postgresql.org/docs/9.6/static/ddl-priv.html
https://www.postgresql.org/docs/9.6/static/sql-grant.html
https://www.postgresql.org/docs/9.6/static/runtime-config-client.html
Migration to: PostgreSQL Indexes

Overview
PostgreSQL supports multiple types of Indexes using different indexing algorithms that can provide performance benefits for different types of queries. The built-in PostgreSQL Index types include:

- **B-Tree**
  Default indexes that can be used for equality and range for the majority of queries. These indexes can operate against all datatypes and can be used to retrieve NULL values. B-Tree index values are sorted in ascending order by default.

- **Hash**
  Hash Indexes are practical for equality operators. These types of indexes are rarely used because they are not transaction-safe. They need to be rebuilt manually in case of failures.

- **GIN (Generalized Inverted Indexes)**
  GIN indexes are useful when an index needs to map a large amount of values to one row, while B-Tree indexes are optimized for cases when a row has a single key value. GIN indexes work well for indexing full-text search and for indexing array values.

- **GiST (Generalized Search Tree)**
  GiST indexes are not viewed as a single type of index but rather as an index infrastructure; a base to create different indexing strategies. GiST indexes enable building general B-Tree structures that can be used for operations more complex than equality and range comparisons. They are mainly used to create indexes for geometric data types and they support full-text search indexing.

- **BRIN (Block Range Indexes)**
  BRIN Indexes store summary data for values stored in sequential physical table block ranges. A BRIN index contains only the minimum and maximum values contained in a group of database pages. Its main advantage is that it can rule out the presence of certain records and therefore reduce query run time.

Additional PostgreSQL indexes (such as SP-GiST) exist but are currently not supported because they require a loadable extension not currently available in Amazon Aurora PostgreSQL.
**PostgreSQL CREATE INDEX Synopsis**

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

By default, the CREATE INDEX statement creates a B-Tree index.

**Examples**

**Oracle CREATE/DROP Index:**

```sql
SQL> CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES (EMPLOYEE_ID DESC);
SQL> DROP INDEX IDX_EMP_ID;
```

**PostgreSQL CREATE/DROP Index:**

```sql
demo=> CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES (EMPLOYEE_ID DESC);
demo=> DROP INDEX IDX_EMP_ID;
```

**Oracle ALTER INDEX - RENAME:**

```sql
SQL> ALTER INDEX IDX_EMP_ID RENAME TO IDX_EMP_ID_OLD;
```

**PostgreSQL ALTER INDEX - RENAME:**

```sql
demo=> ALTER INDEX IDX_EMP_ID RENAME TO IDX_EMP_ID_OLD;
```

**Oracle ALTER INDEX - TABLESPACE:**

```sql
SQL> ALTER INDEX IDX_EMP_ID REBUILD TABLESPACE USER_IDX;
```

**PostgreSQL ALTER INDEX - TABLESPACE:**

```sql
demo=> CREATE TABLESPACE PGIDX LOCATION '/data/indexes';
demo=> ALTER INDEX IDX_EMP_ID SET TABLESPACE PGIDX;
```

**Oracle REBUILD INDEX:**

```sql
SQL> ALTER INDEX IDX_EMP_ID REBUILD;
```

**PostgreSQL REINDEX (REBUILD) INDEX:**

```sql
demo=> REINDEX INDEX IDX_EMP_ID;
```
Oracle REBUILD INDEX ONLINE:

```sql
SQL> ALTER INDEX IDX_EMP_ID REBUILD ONLINE;
```

PostgreSQL REINDEX (REBUILD) INDEX ONLINE:

```sql
demo=> CREATE INDEX CONCURRENTLY IDX_EMP_ID1 ON EMPLOYEES(EMPLOYEE_ID);
demo=> DROP INDEX CONCURRENTLY IDX_EMP_ID;
```

For additional information on PostgreSQL Indexes:

- [https://www.postgresql.org/docs/9.6/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY](https://www.postgresql.org/docs/9.6/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY)
- [https://www.postgresql.org/docs/9.6/static/sql-alterindex.html](https://www.postgresql.org/docs/9.6/static/sql-alterindex.html)
- [https://www.postgresql.org/docs/current/static/sql-reindex.html](https://www.postgresql.org/docs/current/static/sql-reindex.html)

Oracle vs. PostgreSQL Indexes

<table>
<thead>
<tr>
<th>Oracle Indexes Types / Features</th>
<th>PostgreSQL Compatibility</th>
<th>PostgreSQL Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-Tree Index</td>
<td>Supported</td>
<td>B-Tree Index</td>
</tr>
<tr>
<td>Index-Organized Tables</td>
<td>Supported</td>
<td>PostgreSQL CLUSTER</td>
</tr>
<tr>
<td>Reverse key indexes</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>Descending indexes</td>
<td>Supported</td>
<td>ASC (default) / DESC</td>
</tr>
<tr>
<td>B-tree cluster indexes</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>Unique / non-unique Indexes</td>
<td>Supported</td>
<td>Syntax is identical</td>
</tr>
<tr>
<td>Function-Based Indexes</td>
<td>Supported</td>
<td>PostgreSQL Expression Indexes</td>
</tr>
<tr>
<td>Application Domain indexes</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>BITMAP Index / Bitmap Join Indexes</td>
<td>Not supported</td>
<td>Consider BRIN index*</td>
</tr>
<tr>
<td>Composite Indexes</td>
<td>Supported</td>
<td>Multicolumn Indexes</td>
</tr>
<tr>
<td>Invisible Indexes</td>
<td>Not supported</td>
<td>Extension “hypopg” is not currently supported*</td>
</tr>
<tr>
<td>Local and Global Indexes</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>Partial Indexes for Partitioned Tables (Oracle 12c)</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>CREATE INDEX... / DROP INDEX...</td>
<td>Supported</td>
<td>High percentage of syntax similarity</td>
</tr>
<tr>
<td>ALTER INDEX... (General Definitions)</td>
<td>Supported</td>
<td>-</td>
</tr>
<tr>
<td>ALTER INDEX... REBUILD</td>
<td>Supported</td>
<td>REINDEX</td>
</tr>
<tr>
<td>ALTER INDEX... REBUILD ONLINE</td>
<td>Limited support</td>
<td>CONCURRENTLY</td>
</tr>
<tr>
<td>Index Metadata</td>
<td>PG_INDEXES (Oracle USER_INDEXES)</td>
<td>-</td>
</tr>
<tr>
<td>Index Tablespace Allocation</td>
<td>Supported</td>
<td>SET TABLESPACE</td>
</tr>
<tr>
<td>Index Parallel Operations</td>
<td>Not supported</td>
<td>-</td>
</tr>
<tr>
<td>Index Compression</td>
<td>Not direct equivalent to Oracle index key compression or advanced index compression</td>
<td>-</td>
</tr>
</tbody>
</table>
Overview

B-Tree indexes ("B" stands for balanced), are the most common index type in Relational Database and are used for a variety of common query performance enhancing tasks. B-Tree indexes can be defined as an ordered list of values divided into ranges. They provide superior performance by associating a key with a row or range of rows.

B-Tree indexes contain two types of blocks: branch blocks for searching and leaf blocks for storing values. The branch blocks also contain the root branch, which points to lower-level index blocks in the B-Tree index structure.

B-Tree indexes are useful for Primary Keys and other high-cardinality columns. They provide excellent data access performance for a variety of query patterns such as exact match searches and range searches. B-Tree indexes serve as the default index type when creating a new index.

Example

Creating an Oracle B-Tree Index:

```sql
SQL> CREATE INDEX IDX_EVENT_ID ON SYSTEM_LOG(EVENT_ID);
```

For additional details:
https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT721
Migration to: PostgreSQL B-Tree Indexes

Overview
When creating an Index in PostgreSQL, a B-Tree Index is created by default, similarly to the behavior in the Oracle Database. PostgreSQL B-Tree indexes have the same characteristics as Oracle and these types of indexes can handle equality and range queries on data. The PostgreSQL optimizer considers using B-Tree indexes especially when using one or more of the following operators in queries: >, >=, <, <=, =

In addition, performance improvements can be achieved when using IN, BETWEEN, IS NULL or IS NOT NULL.

Example
Create a PostgreSQL B-Tree Index:

```
demo=> CREATE INDEX IDX_EVENT_ID ON SYSTEM_LOG(EVENT_ID);
OR
demo=> CREATE INDEX IDX_EVENT_ID1 ON SYSTEM_LOG USING BTREE (EVENT_ID);
```
Overview
An index that is created on multiple table columns is known as a multi-column, concatenated or Composite Index. The main purpose of these indexes is to improve the performance of data retrieval for SELECT statements when filtering on all or some of the Composite Index columns. When using Composite Indexes, it is beneficial to place the most restrictive columns at the first position of the index to improve query performance. Column placement order is crucial when using Composite Indexes as the most prevalent columns are accessed first.

Example
Create a Composite Index on the HR.EMPLOYEES table:

```sql
CREATE INDEX IDX_EMP_COMPI ON EMPLOYEES (FIRST_NAME, EMAIL, PHONE_NUMBER);
```

Drop a Composite Index:

```sql
DROP INDEX IDX_EMP_COMPI;
```

For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28274/data_acc.htm#i2773
https://docs.oracle.com/database/121/CNCPT/indexiot.htm#CNCPT88833
Migration to: PostgreSQL Multi-Column Indexes

Overview
PostgreSQL Multi-Column Indexes are similar to Oracle Composite Indexes.

- Currently, only B-tree, GiST, GIN, and BRIN support Multi-Column Indexes.
- 32 columns can be specified when creating a Multi-Column Index.

PostgreSQL uses the exact same syntax as Oracle to create Multi-Column Indexes.

Example
Create a Multi-Column Index on the EMPLOYEES table:

```
CREATE INDEX IDX_EMP_COMPI
    ON EMPLOYEES (FIRST_NAME, EMAIL, PHONE_NUMBER);
```

Drop a Multi-Column Index:

```
DROP INDEX IDX_EMP_COMPI;
```

For additional details:
https://www.postgresql.org/docs/9.6/static/indexes-multicolumn.html
Overview
BITMAP indexes are task-specific indexes that are best suited for providing fast data retrieval for OLAP workloads. BITMAP Indexes are generally very fast for read-mostly scenarios. BITMAP indexes do not perform well in heavy-DML or OLTP-type workloads.

Unlike B-Tree Indexes where an index entry points to a specific table row, when using BITMAP Indexes, the index stores a BITMAP for each index key.

BITMAP Indexes are ideal for low-cardinality data filtering, where the number of distinct values in a column is relatively small.

Example
Create an Oracle BITMAP Index:

```sql
SQL> CREATE BITMAP INDEX IDX_BITMAP_EMP_GEN ON EMPLOYEES(GENDER);
```

For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_5011.htm#SQLRF01209
https://docs.oracle.com/database/121/SQLRF/statements_5013.htm#SQLRF01209
Migration to: PostgreSQL BRIN Indexes

PostgreSQL Index Overview

PostgreSQL does not provide native support for BITMAP indexes. However, a BRIN index, which splits table records into block ranges with MIN/MAX summaries, can be used as a partial alternative for certain analytic workloads. For example, BRIN indexes are suited for queries that rely heavily on aggregations to analyze large numbers of records.

However, Oracle BITMAP indexes and PostgreSQL BRIN indexes are not implemented in the same way and cannot be used as direct equivalents.

Example

PostgreSQL BRIN Index Creation:

```
CREATE INDEX IDX_BRIN_EMP ON EMPLOYEES USING BRIN(salary);
```

For additional details:

https://www.postgresql.org/docs/9.6/static/indexes-types.html
https://www.postgresql.org/docs/9.6/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY
**Overview**

Function-Based Indexes allow functions to be used in the `WHERE` clause of queries on indexes columns. Function-Based Indexes store the output of a Function applied on the values of a table column. The Oracle Query Optimizer will only use a Function-Based Index when the function itself is used in the query itself. To maintain Function-Based Indexes updated, when the Oracle Database processes DML operations it will also evaluate the output of the Function on updated column values.

**Example**

Creation of a Function-Based Index:

```sql
SQL> CREATE TABLE SYSTEM_EVENTS(
    2    EVENT_ID NUMERIC PRIMARY KEY,
    3    EVENT_CODE VARCHAR2(10) NOT NULL,
    4    EVENT_DESCRIPTION VARCHAR2(200),
    5    EVENT_TIME TIMESTAMP NOT NULL);

SQL> CREATE INDEX EVNT_BY_DAY ON SYSTEM_EVENTS (EXTRACT(DAY FROM EVENT_TIME));
```

For additional details:
- [Oracle Function-Based Indexes](https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT721)
- [SQL for Indexes](https://docs.oracle.com/database/121/SQLRF/statements_5013.htm#SQLRF01209)
Migration to: PostgreSQL Expression Indexes

Overview
PostgreSQL supports Expression Indexes which are similar to Function-Based Indexes in Oracle.

Examples
1. Creating an Expression Index in PostgreSQL:

```sql
CREATE TABLE SYSTEM_EVENTS(
    EVENT_ID NUMERIC PRIMARY KEY,
    EVENT_CODE VARCHAR(10) NOT NULL,
    EVENT_DESCRIPTION VARCHAR(200),
    EVENT_TIME TIMESTAMP NOT NULL);
CREATE INDEX EVNT_BY_DAY ON SYSTEM_EVENTS(EXTRACT(DAY FROM EVENT_TIME));
```

2. Inserting records to the SYSTEM_EVENTS table, gathering table statistics using the ANALYZE statement and verifying that the Expression Index (“EVNT_BY_DAY”) is being used for data access.

```sql
INSERT INTO SYSTEM_EVENTS
SELECT ID AS event_id,
    'EVNT-A'||ID+9'||'-'||ID AS event_code,
    CASE WHEN mod(ID,2) = 0 THEN 'Warning' ELSE 'Critical' END AS event_desc,
    now() + INTERVAL '1 minute' * ID AS event_time
FROM
    (SELECT generate_series(1,1000000) AS ID) A;
ANALYZE SYSTEM_EVENTS;
EXPLAIN
    SELECT * FROM SYSTEM_EVENTS
    WHERE EXTRACT(DAY FROM EVENT_TIME) = '22';
```

QUERY PLAN

```
---
Bitmap Heap Scan on system_events  (cost=729.08..10569.58 rows=33633 width=41)
Recheck Cond: (date_part('day':text, event_time) = '22':double precision)
->  Bitmap Index Scan on evnt_by_day  (cost=0.00..720.67 rows=33633 width=0)
   Index Cond: (date_part('day':text, event_time) = '22':double precision)
```
Partial Indexes
PostgreSQL also offers “partial indexes”, which are indexes that use a `WHERE` clause when created. The biggest benefit of using “partial indexes” is reduction of the overall subset of indexed data allowing users to index relevant table data only. “Partial indexes” can be used to increase efficiency and reduce the size of the index.

Example
Create a PostgreSQL “partial Index”:

```sql
demo=> CREATE TABLE SYSTEM_EVENTS(
    EVENT_ID NUMERIC PRIMARY KEY,
    EVENT_CODE VARCHAR(10) NOT NULL,
    EVENT_DESCRIPTION VARCHAR(200),
    EVENT_TIME DATE NOT NULL);

Demo=> CREATE INDEX IDX_TIME_CODE ON SYSTEM_EVENTS(EVENT_TIME)
    WHERE EVENT_CODE like '01-A%';
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY
Overview
Local and Global Indexes are used for Partitioned Tables:

- **Local Partitioned Index**
  Maintain a one-to-one relationship between the Index Partitions and the Table Partitions. For each Table Partition, a separate Index Partition will be created. This type of index is created using the `LOCAL` clause. Because each Index Partition is independent, index maintenance operations are easier and can be performed independently. Local Partitioned Indexes are managed automatically by the Oracle Database during creation or deletion of Table Partitions.

- **Global Partitioned Index**
  Each Global Index contains keys from multiple table partitions in a single index partition. This type of index is created using the `GLOBAL` clause during index creation. A Global index can be partitioned or non-partitioned (default).

Certain restrictions exist when creating Global Partitioned Indexes on Partitioned Tables, specifically for index management and maintenance. For example, dropping a Table Partition causes the Global Index to become unusable without an index rebuild.

Example
Create a Local and Global Index on a Partitioned Table:

```sql
SQL> CREATE INDEX IDX_SYS_LOGS_LOC ON SYSTEM_LOGS (EVENT_DATE) LOCAL (PARTITION EVENT_DATE_1, PARTITION EVENT_DATE_2, PARTITION EVENT_DATE_3);

SQL> CREATE INDEX IDX_SYS_LOGS_GLOB ON SYSTEM_LOGS (EVENT_DATE) GLOBAL PARTITION BY RANGE (EVENT_DATE) (PARTITION EVENT_DATE_1 VALUES LESS THAN (TO_DATE('01/01/2015','DD/MM/YYYY')), PARTITION EVENT_DATE_2 VALUES LESS THAN (TO_DATE('01/01/2016','DD/MM/YYYY')), PARTITION EVENT_DATE_3 VALUES LESS THAN (TO_DATE('01/01/2017','DD/MM/YYYY')), PARTITION EVENT_DATE_4 VALUES LESS THAN (MAXVALUE));
```

For additional details:
https://docs.oracle.com/cd/E18282_01/server.112/e16541/partition.htm
https://docs.oracle.com/database/121/VLDBG/GUID-81DD6045-A269-4BD2-9EBF-E430F8C3E51B.htm#VLDBG1354
Migration to: PostgreSQL Partitioned Indexes

Overview

The Table Partitioning mechanism in PostgreSQL is different when compared to Oracle. There is no direct equivalent for Oracle Local and Global Indexes. The implementation of partitioning in PostgreSQL (“Table Inheritance”) includes the use of a Parent Table with Child Tables used as the table partitions.

- Indexes created on the Child Tables behave similarly to Local Indexes in the Oracle database, with per-table indexes (“partitions”).
- Creating an index on the parent table, similar to a Global Indexes in Oracle, has no effect.

Example

1. Create the Parent Table:

   ```sql
demo=# CREATE TABLE SYSTEM_LOGS
   (EVENT_NO    NUMERIC NOT NULL,
    EVENT_DATE  DATE   NOT NULL,
    EVENT_STR   VARCHAR(500),
    ERROR_CODE  VARCHAR(10));
```

2. Create Child Tables (“partitions”) with a Check Constraint:

   ```sql
demo=# CREATE TABLE SYSTEM_LOGS_WARNING (CHECK (ERROR_CODE IN('err1', 'err2', 'err3'))) INHERITS (SYSTEM_LOGS);

demo=# CREATE TABLE SYSTEM_LOGS_CRITICAL (CHECK (ERROR_CODE IN('err4', 'err5', 'err6'))) INHERITS (SYSTEM_LOGS);
```

3. Create Indexes on each Child Table (“partitions”)

   ```sql
demo=# CREATE INDEX IDX_SYSTEM_LOGS_WARNING ON SYSTEM_LOGS_WARNING(ERROR_CODE);

demo=# CREATE INDEX IDX_SYSTEM_LOGS_CRITICAL ON SYSTEM_LOGS_CRITICAL(ERROR_CODE);
```

PostgreSQL does not have direct equivalents for Local and Global indexes in Oracle. However, indexes that have been created on the Child Tables behave similarly to Local Indexes in Oracle.

For additional details:
- [https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html](https://www.postgresql.org/docs/9.6/static/ddl-partitioning.html)
Overview
Oracle 12c introduced support for automatic generation of values to populate columns in database tables. The 
IDENTITY type generates a sequence and associates it with a table column without the need to manually 
create a separate Sequence object. The IDENTITY type relies (internally) on Sequences, which can also be 
manually configured.

Example
1. Create a table with an Oracle 12c Identity Column:

   SQL> CREATE TABLE IDENTITY_TST (  
   2       COL1 NUMBER GENERATED BY DEFAULT AS IDENTITY  
   3               (START WITH 100  
   4                   INCREMENT BY 10),  
   5       COL2 VARCHAR2(30));

2. Insert data into the table. The Identity Column automatically generates values for COL1.

   SQL> INSERT INTO IDENTITY_TST(COL2) VALUES('A');
   SQL> INSERT INTO IDENTITY_TST(COL1, COL2) VALUES(DEFAULT, 'B');
   SQL> INSERT INTO IDENTITY_TST(col1, col2) VALUES(NULL, 'C');

   SQL> SELECT * FROM IDENTITY_TST;
   COL1  COL2
   ---------- ------------------------------
   100    A
   110    B

For additional details:
https://docs.oracle.com/database/121/SQLRF/statements_6017.htm#SQLRF01314
Migration to: PostgreSQL SERIAL Type

Overview
PostgreSQL enables you to create a Sequence that is similar to the AUTO_INCREMENT property supported by Oracle 12c’s Identity column feature. When creating a new table using the SERIAL pseudo-type, a Sequence is created. Additional types from the same family are SMALLSERIAL and BIGSERIAL.

By assigning a SERIAL type to a column as part of table creation, PostgreSQL creates a Sequence using default configuration and adds the NOT NULL constraint to the column. The new Sequence can be altered and configured as a regular Sequence.

Example
Using the PostgreSQL SERIAL pseudo-type (with a Sequence that is created implicitly):

```
psql=> CREATE TABLE SERIAL_SEQ_TST(
    COL1 SERIAL PRIMARY KEY,
    COL2 VARCHAR(10));

psql=> \ds

Schema |          Name           |   Type   | Owner
--------+-------------------------+----------+-------
public | serial_seq_tst_col1_seq | sequence | pg_tst_db

psql=> ALTER SEQUENCE SERIAL_SEQ_TST_COL1_SEQ RESTART WITH 100
       INCREMENT BY 10;

psql=> INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
psql=> INSERT INTO SERIAL_SEQ_TST(COL1, COL2) VALUES(DEFAULT, 'B');

psql=> SELECT * FROM SERIAL_SEQ_TST;

   col1 | col2
-------+------
     100 | A
     110 | B
```

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createsequence.html
https://www.postgresql.org/docs/9.6/static/functions-sequence.html
Migrating from: Oracle MVCC

Overview
Two primary lock types exist in the Oracle database: exclusive locks and share locks which implement the following high-level locking semantics:

- Writers never block readers.
- Readers never block writers.
- Oracle never escalates locks from row to page and table level, which reduces potential deadlocks.
- Oracle allows the user to issue an explicit lock on a specific table using the `LOCK TABLE` statement.

Lock types can be divided into four categories:

- **DML Locks**
  Preserving data integrity accessed concurrently by multiple users, DML statements acquire locks automatically both on row and table levels.

  - **Row Locks (TX)** – obtained on a single row of a table by one the following statements: `INSERT`, `UPDATE`, `DELETE`, `MERGE`, and `SELECT ... FOR UPDATE`. If a transaction obtains a row lock, a table lock is also acquired to prevent DDL modifications to the table that might cause conflicts. The lock exists until the transaction ends with a `COMMIT` or `ROLLBACK`.

  - **Table Locks (TM)** - When performing one of the following DML operations: `INSERT`, `UPDATE`, `DELETE`, `MERGE`, and `SELECT ... FOR UPDATE`, a transaction automatically acquires a table lock to prevent DDL modifications to the table that might cause conflicts if the transaction did not issue a `COMMIT` or `ROLLBACK`. 
The following table provides additional information regarding row and table locks:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Row Locks</th>
<th>Table Lock Mode</th>
<th>RS</th>
<th>RX</th>
<th>S</th>
<th>SRX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ... FROM table...</td>
<td>—</td>
<td>none</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>INSERT INTO table...</td>
<td>Yes</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>UPDATE table...</td>
<td>Yes</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>MERGE INTO table ...</td>
<td>Yes</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DELETE FROM table...</td>
<td>Yes</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SELECT ... FROM table FOR UPDATE OF...</td>
<td>Yes</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LOCK TABLE table IN...</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ROW SHARE MODE</td>
<td>SS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ROW EXCLUSIVE MODE</td>
<td>SX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SHARE MODE</td>
<td>S</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SHARE ROWEXCLUSIVE MODE</td>
<td>SSX</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>EXCLUSIVE MODE</td>
<td>X</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- **DDL Locks**
  The main purpose of a DDL lock is to protect the definition of a schema object while it is modified by an ongoing DDL operation such as `ALTER TABLE EMPLOYEES ADD <COLUMN>`.

- **Explicit (Manual) Data Locking**
  The user has the ability to explicitly create a lock to achieve transaction-level read consistency for when an application requires transactional exclusive access to a resource without waiting for other transactions to complete. Explicit data locking can be done at the transaction level or the session level:

  **Transaction Level:**
  - SET TRANSACTION ISOLATION LEVEL
  - LOCK TABLE
  - SELECT ... FOR UPDATE

  **Session Level:**
  - ALTER SESSION SET ISOLATION LEVEL

- **System Locks**
  Oracle lock types such as Latches, Mutexes, and internal locks.
Examples

Explicit data lock using the `LOCK TABLE` command:

```sql
-- Session 1
SQL> LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;

-- Session 2
SQL> UPDATE EMPLOYEES
    SET SALARY=SALARY+1000
    WHERE EMPLOYEE_ID=114;

-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

Explicit data lock using the `SELECT... FOR UPDATE` command. Oracle obtains exclusive row-level locks on all the rows identified by the `SELECT FOR UPDATE` statement:

```sql
-- Session 1
SQL> SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;

-- Session 2
SQL> UPDATE EMPLOYEES
    SET SALARY=SALARY+1000
    WHERE EMPLOYEE_ID=114;

-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

For additional details:

https://docs.oracle.com/cloud/latest/db112/SQLRF/statements_9015.htm#SQLRF01605
http://docs.oracle.com/cd/E18283_01/server.112/e17118/ap_locks002.htm
https://docs.oracle.com/database/121/SQLRF/ap_locks001.htm#SQLRF55502
https://docs.oracle.com/database/121/SQLRF/ap_locks003.htm#SQLRF55513
https://docs.oracle.com/database/121/SQLRF/ap_locks002.htm#SQLRF55509
Overview
PostgreSQL provides various lock modes to control concurrent access to data in tables. Data consistency is maintained using a Multi-Version Concurrency Control (MVCC) mechanism. Most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes.

The MVCC mechanism prevents viewing inconsistent data produced by concurrent transactions performing updates on the same rows. MVCC in PostgreSQL provides strong transaction isolation for each database session and minimizes lock-contention in multiuser environments.

- Similarly, to Oracle, MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data. **Reads will never block writes and writes never blocks reads.**

- Similarly to Oracle, PostgreSQL does not escalate locks to table-level, such as where an entire table is locked for writes when a certain threshold of row locks is exceeded.

Implicit and Explicit Transactions (Auto-Commit Behavior)
Unlike Oracle, PostgreSQL uses auto-commit for transactions by default. However, there are two options to support explicit transactions, which are similar to the default behavior in Oracle (non-auto-commit):

- **Use the** START TRANSACTION (or BEGIN TRANSACTION) **statements and then COMMIT or ROLLBACK.**

- **Set AUTOCOMMIT to OFF at the session level:**
  
  ```bash
  psql=> \set AUTOCOMMIT off
  ```

With explicit transactions:

- Users can explicitly issue a lock similar to the LOCK TABLE statement in Oracle.

- SELECT... FOR UPDATE is supported.
Similarly to Oracle, PostgreSQL automatically acquires the necessary locks to control concurrent access to data. PostgreSQL implements the following types of locks:

1. **Table-level Locks**:

<table>
<thead>
<tr>
<th>Requested Lock Mode VS current</th>
<th>ACCESS SHARE</th>
<th>ROW SHARE</th>
<th>ROW EXCLUSIVE</th>
<th>SHARE UPDATE EXCLUSIVE</th>
<th>SHARE</th>
<th>SHARE ROW EXCLUSIVE</th>
<th>EXCLUSIVE</th>
<th>ACCESS EXCLUSIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS SHARE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ROW SHARE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ROW EXCLUSIVE</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SHARE UPDATE EXCLUSIVE</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SHARE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SHARE ROW EXCLUSIVE</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ACCESS EXCLUSIVE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

2. **Row-level Locks**:

<table>
<thead>
<tr>
<th>Requested Lock Mode VS current</th>
<th>FOR KEY SHARE</th>
<th>FOR SHARE</th>
<th>FOR NO KEY UPDATE</th>
<th>FOR UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR KEY SHARE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FOR SHARE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FOR NO KEY UPDATE</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FOR UPDATE</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

3. **Page-level Locks**: Shared/Exclusive locks used to control read or write access to table pages in the shared buffer pool. They are released immediately after a row is fetched or updated.

4. **Deadlocks**: Occur when two or more transactions are waiting for one another to release each lock.

**Transaction-level locking**:
PostgreSQL does not support session isolation levels, although it can be controlled via transactions:

- SET TRANSACTION ISOLATION LEVEL
- LOCK TABLE
- SELECT ... FOR UPDATE

**PostgreSQL LOCK TABLE Synopsis**

LOCK [ TABLE ] [ ONLY ] name [*] [ , ... ] [ IN lockmode MODE ] [ NOWAIT ]

where lockmode is one of:

- ACCESS SHARE
- ROW SHARE
- ROW EXCLUSIVE
- SHARE UPDATE EXCLUSIVE
- SHARE
- SHARE ROW EXCLUSIVE
- EXCLUSIVE
- ACCESS EXCLUSIVE
Notes:

- If `ONLY` and `[*]` are specified, the command aborts with an error.
- There is no `UNLOCK TABLE` command. Locks are always released at the end of a transaction (`COMMIT / ROLLBACK`).
- The `LOCK TABLE` command can be used inside a transaction and should appear after the `START TRANSACTION` statement.
Examples

1. Obtain an explicit lock on a table using the `LOCK TABLE` command:

   ```
   -- Session 1
   psql=> START TRANSACTION;
   psql=> LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;
   -- Session 2
   psql=> UPDATE EMPLOYEES
          SET SALARY=SALARY+1000
          WHERE EMPLOYEE_ID=114;
   -- Session 2 waits for session 1 to COMMIT or ROLLBACK
   ```

2. Explicit lock via the `SELECT... FOR UPDATE` command. PostgreSQL obtains exclusive row-level locks on rows referenced by the `SELECT FOR UPDATE` statement. Must be executed inside a transaction.

   ```
   -- Session 1
   psql=> START TRANSACTION;
   psql=> SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;
   -- Session 2
   psql=> UPDATE EMPLOYEES
          SET SALARY=SALARY+1000
          WHERE EMPLOYEE_ID=114;
   -- Session 2 waits for session 1 to COMMIT or ROLLBACK
   ```

PostgreSQL Deadlocks

Deadlocks occur when two or more transactions acquired locks on each other’s process resources (table or row). PostgreSQL can detect Deadlocks automatically and resolve the event by aborting one of the transactions, allowing the other transaction to complete.

Example
Simulating a Deadlock:

**Session 1**

- **Step1:**
  
  UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;

- **Step4:**
  
  UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;

**Session 2**

- **Step2:**
  
  UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;

- **Step3:**
  
  UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;

Session 1 is waiting for Session 2 and Session 2 is waiting for Session 1 = deadlock.
Real-time Monitoring of Locks using Catalog Tables

- `pg_locks`
- `pg_stat_activity`

Examples

1. Monitor locks using a SQL query:

   ```sql
   psql=> SELECT
       block.pid AS block_pid,
       block_stm.usename AS blocker_user,
       block.mode AS block_mode,
       block.locktype AS block_locktype,
       block.relation::regclass AS block_table,
       block_stm.query AS block_query,
       block.GRANTED AS block_granted,
       waiting.locktype AS waiting_locktype,
       waiting_stm.usename AS waiting_user,
       waiting.relation::regclass AS waiting_table,
       waiting_stm.query AS waiting_query,
       waiting.mode AS waiting_mode,
       waiting.pid AS waiting_pid
   FROM pg_catalog.pg_locks AS waiting
   JOIN pg_catalog.pg_stat_activity AS waiting_stm
   ON (waiting_stm.pid = waiting.pid)
   JOIN pg_catalog.pg_locks AS block
   ON ((waiting."database" = block."database"
       AND waiting.relation  = block.relation)
       OR waiting.transactionid = block.transactionid)
   JOIN pg_catalog.pg_stat_activity AS block_stm
   ON (block_stm.pid = block.pid)
   WHERE NOT waiting.GRANTED
   AND waiting.pid <> block.pid;
   
   -- Session 1
   psql=> START TRANSACTION;
   psql=> SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;
   
   -- Session 2
   psql=> UPDATE EMPLOYEES
       SET SALARY=SALARY+1000
       WHERE EMPLOYEE_ID=114;
   -- Session 2 waits for session 1 to COMMIT or ROLLBACK
   ```
3. Run the SQL query from step #1 monitoring locks while distinguishing between the “blocking” and “waiting” session:

```
BEGIN;
LOCK TABLE employees IN SHARE ROW EXCLUSIVE MODE;
SELECT * FROM employees WHERE employee_id=102 FOR UPDATE;
```

### Comparing “Locks”, Oracle vs. PostgreSQL

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Dictionary” tables to obtain information about locks</td>
<td>v$lock; v$locked_object; v$session_blockers;</td>
<td>pg_locks pg_stat_activity</td>
</tr>
</tbody>
</table>
| Lock a table                    | BEGIN;
|                                | LOCK TABLE employees IN SHARE ROW EXCLUSIVE MODE; | LOCK TABLE employees IN SHARE ROW EXCLUSIVE MODE; |
| Explicit Locking                | SELECT * FROM employees WHERE employee_id=102 FOR UPDATE; | BEGIN;
|                                | SELECT * FROM employees WHERE employee_id=102 FOR UPDATE; | |
| Explicit Locking, options       | SELECT...FOR UPDATE                        | SELECT...FOR UPDATE                        |
|                                | KEY SHARE SHARE NO KEY UPDATE              | KEY SHARE SHARE NO KEY UPDATE              |

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-lock.html
https://www.postgresql.org/docs/9.6/static/explicit-locking.html
Overview
Oracle supports most national and international encoded character set standards including extensive support for Unicode character sets.

Oracle provides two scalar string-specific data types:

1. **VARCHAR2**: stores variable-length character strings with a string length between 1 and 4000 bytes. The Oracle database can be configured to use the VARCHAR2 data type to store either Unicode or Non-Unicode characters.

2. **NVARCHAR2**: scalar data type used to store Unicode data. Supports AL16UTF16 or UTF8, specified during database creation.

Character sets in the Oracle database are defined at Instance (Oracle 11g) or Pluggable Database level (Oracle 12c R2) level. In Pre-12cR2 Oracle databases, the character set for the root Container and all Pluggable Databases were required to be identical.

**UTF8 Unicode**
Oracle’s implementation is done using the AL32UTF8 Character Set and offers encoding of ASCII characters as single-byte for Latin characters, two-bytes for some European and Middle-Eastern languages, and three-bytes for certain South and East-Asian characters. Therefore, Unicode storage requirements are usually higher when compared non-Unicode character sets.

**Character Set Migration**
Two options exist for modifying existing Instance-level or database-level character sets:

1. Export/Import from the source Instance/PDB to a new Instance/PDB with a modified CS.
2. Database Migration Assistant for Unicode (DMU) which simplifies the migration process to the Unicode CS.

As of 2012, using the CSALTER utility for CS migrations is deprecated.

**Notes:**
1. Oracle Database 12c Release 1 (12.1.0.1) complies with version 6.1 of the Unicode Standard.
2. Oracle Database 12c Release 2 (12.1.0.2) extends the compliance to version 6.2 of the Unicode standard.
3. UTF-8 is supported through the AL32UTF8 CS and is valid as both the client and database character sets.
4. UTF-16BE is supported through AL16UTF16 and is valid as the national (NCHAR) character set.
For additional details:
https://docs.oracle.com/database/121/SQLRF/ap_standard_sql015.htm#SQLRF55539
https://docs.oracle.com/database/121/NLSPG/applocaledata.htm#NLSPG584
https://docs.oracle.com/database/121/NLSPG/ch11charsetmig.htm
Overview
PostgreSQL supports a variety of different character sets, also known as encoding, including support for both single-byte and multi-byte languages. The default character set is specified when initializing your PostgreSQL database cluster with initdb. Each individual database created on the PostgreSQL cluster supports individual character sets defined as part of database creation.

Notes:
1. All supported character sets can be used by clients. However, some client-side only characters are not supported for use within the server.
2. Unlike Oracle, PostgreSQL does not natively support an NVARCHAR data type and does not offer support for UTF-16.

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Implementation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding</td>
<td>Defines the basic rules on how alphanumeric characters are represented in binary format, for example – Unicode Encoding.</td>
<td>Database</td>
</tr>
<tr>
<td>Locale</td>
<td>Superset which include LC_COLLATE and LC_CTYPE, among others. For example, LC_COLLATE defines how strings are sorted and needs to be a subset supported by the database Encoding.</td>
<td>Table-Column</td>
</tr>
</tbody>
</table>

Example
1. Create a database named test01 which uses the Korean EUC_KR Encoding the and the ko_KR locale.

```sql
CREATE DATABASE test01 WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

2. View the character sets configured for each database by querying the System Catalog:

```sql
select datname, datcollate, datctype from pg_database;
```
Changing Character Sets / Encoding

In-place modification of the database encoding is not recommended nor supported. You must export all data, create a new database with the new encoding, and import the data.

Example

1. Export the data using the `pg_dump` utility:

   ```
   pg_dump mydb1 > mydb1_export.sql
   ```

2. Rename (or delete) your current database:

   ```
   ALTER DATABASE mydb1 TO mydb1_backup;
   ```

3. Create a new database using the modified encoding:

   ```
   CREATE DATABASE mydb1_new_encoding WITH ENCODING 'UNICODE'
   TEMPLATE=template0;
   ```

4. Import the data using the `pg_dump` file previously created. Verify that you set your client encoding to the encoding of your “old” database.

   ```
   PGCLIENTENCODING=OLD_DB_ENCODING psql -f mydb1_export.sql
   mydb1_new_encoding
   ```

Note: Using the `client_encoding` parameter overrides the use of `PGCLIENTENCODING`.

Client/Server Character Set Conversions

PostgreSQL supports conversion of character sets between server and client for specific character set combinations as described in the `pg_conversion` system catalog.

PostgreSQL includes predefined conversions, for a complete list:

https://www.postgresql.org/docs/current/static/multibyte.html#MULTIBYTE-TRANSLATION-TABLE

You can create a new conversion using the SQL command `CREATE CONVERSION`. By using `CREATE CONVERSION`.

Examples

1. Create a conversion from UTF8 to LATIN1 using a custom-made `myfunc1` function:

   ```
   CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc1;
   ```

2. Configure the PostgreSQL client character set:

   ```
   Method 1
   ========
   psql \encoding SJIS
   Method 2
   ========
   SET CLIENT_ENCODING TO 'value';
   ```
3. View the client character set and reset it back to the default value.

```
SHOW client_encoding;
RESET client_encoding;
```

### Table Level Collation
PostgreSQL supports specifying the sort order and character classification behavior on a per-column level.

### Example
Specify specific collations for individual table columns:

```
CREATE TABLE test1 (  
col1 text COLLATE "de_DE",  
col2 text COLLATE "es_ES";
```

### Oracle vs. PostgreSQL Character Sets

<table>
<thead>
<tr>
<th></th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>View database character set</strong></td>
<td>SELECT * FROM NLS_DATABASE_PARAMETER S;</td>
<td>select datname, pg_encoding_to_char(encoding ), datcollate, datctype from pg_database;</td>
</tr>
</tbody>
</table>
| **Modify the database character set** | 1. Full Export/Import.  
2. When converting to Unicode, use the Oracle DMU utility. | • Export the database.  
• Drop or rename the database.  
• Re-create the database with the desired new character set.  
• Import database data from the exported file into the new database. |
| **Character set granularity** | Instance (11g + 12cR1) Database (Oracle 12cR2) | Database |
| **UTF8**               | Supported via VARCHAR2 and NVARCHAR data types     | Supported via VARCHAR datatype                                  |
| **UTF16**              | Supported via NVARCHAR2 datatype                  | Not Supported                                                  |
| **NCHAR/NVARCHAR data types** | Supported                                   | Not supported                                                  |

*For additional details:*
[https://www.postgresql.org/docs/9.6/static/multibyte.html](https://www.postgresql.org/docs/9.6/static/multibyte.html)
Overview
Database transactions are a logical, atomic units of processing that contains one or more SQL statements and may run concurrently alongside other transactions. The primary purpose of a transaction is to ensure the ACID model is enforced, including:

- **Atomicity**
  Every statement in a transaction is processed as one logical unit or none are processed. If a single part of a transaction fails, the entire transaction is aborted and no changes are persisted (“all or nothing”).

- **Consistency**
  All data integrity constraints are checked and all triggers are processed before a transaction is processed. If any of the constraint are violated, the entire transaction fails.

- **Isolation**
  One transaction is not affected by the behavior of other concurrently-running transactions. The effect of a transaction is not visible to other transactions until the transaction is committed.

- **Durability**
  Once a transaction commits, its results will not be lost, regardless of subsequent failures. After a transaction completes, changes made by committed transactions are permanent. The database ensures that committed transactions cannot be lost.

Database Transaction Isolation Levels
Four levels of isolation are defined by the ANSI/ISO SQL standard (SQL92). Each level offers a different approach to handle the concurrent execution of database transactions. Transaction isolation levels are meant to manage the visibility of the changed data, as seen by other running transactions. In addition, when accessing the same data with several concurrent transactions, the selected level of transaction isolation affects the way different transactions interact with each other.
Example
If a bank account is shared by two individuals, what will happen if both parties attempt to perform a transaction on the shared account at the same time? One checks the account balance while the other withdraws money.

- **Read Uncommitted Isolation Level**
  A currently processed transaction can see uncommitted data made by the other transaction. If a rollback is performed, all data is restored to its previous state.

- **Read Committed Isolation Level**
  A currently processed transaction only sees data changes that were committed; “dirty reads” (uncommitted changes) are not possible.

- **Repeatable Read Isolation Level**
  A currently processed transaction can view changes made by the other transaction only after both transactions issue a Commit or both are rolled-back.

- **Serializable Isolation Level**
  The strictest isolation level. Any concurrent execution of a set of serializable transactions is guaranteed to produce the same effect as running them, one at a time, in the same order.

Using different isolation levels will affect the following in terms of database behavior:

- **Dirty Reads**
  A transaction can read data that was written by another transaction, but is not yet committed.

- **Non-Repeatable (fuzzy) Reads**
  When reading the same data several times, a transaction can find that the data has been modified by another transaction that has just committed. The same query executed twice can return different values for the same rows.

- **Phantom Reads**
  Similar to a non-repeatable read, but it is related to new data created by another transaction. The same query executed twice can return a different numbers of records.
### Isolation Levels Supported by the Oracle Database

Oracle supports the *Read Committed* and *Serializable* isolation levels. Oracle also provides a *Read-Only* isolation level which is not a part of the ANSI/ISO SQL standard (SQL92). *Read Committed* is the default isolation level in the Oracle Database.

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
<th>Phantom read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Permitted</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Not permitted</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not permitted</td>
<td>Not permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not permitted</td>
<td>Not permitted</td>
<td>Not permitted</td>
</tr>
</tbody>
</table>

#### Read Committed
Default transaction isolation level in the Oracle database. Each query, executed inside a transaction, only sees data that was committed before the query itself. The Oracle database will never allow reading “dirty pages” and uncommitted data.

#### Serializable
Serializable transactions do not experience non-repeatable reads or phantom reads because they are only able to “see” changes that were committed at the time the transaction began, in addition to the changes made by the transaction itself performing DML operations.

#### Read-Only
As implemented, the read-only isolation level does not allow any DML operations during the transaction and only sees data committed at the time the transaction began.

### Oracle Multiversion Concurrency Control (MVCC)
Oracle uses the MVCC mechanism to provide automatic read consistency across the entire database and all sessions. Using MVCC, database sessions “see” data based on a single point in time, ensuring that only committed changes is viewable. Oracle relies on using the SCN (System Change Number) of the current transaction to obtain a consistent view of the database. Therefore, every database query only returns data which was committed with respect to the SCN that is taken at the time of query execution.
Setting Isolation Levels in Oracle
Oracle database isolation levels can be altered. The isolation level can be changed at the transaction-level and at the session-level.

Example
Altering the isolation-level at the transaction-level:

```
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SQL> SET TRANSACTION READ ONLY;
```

Altering the isolation-level at a session-level:

```
SQL> ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
SQL> ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED;
```

For additional details:
http://docs.oracle.com/cd/E25054_01/server.1111/e25789/transact.htm
https://docs.oracle.com/database/121/CNCPT/transact.htm#CNCPT041
Overview
The same ANSI/ISO SQL (SQL92) isolation levels apply to PostgreSQL, with several similarities and some differences:

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Reads</th>
<th>Non-Repeatable Reads</th>
<th>Phantom Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Permitted but not implemented in PostgreSQL</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Not permitted</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not permitted</td>
<td>Not permitted</td>
<td>Permitted but not implemented in PostgreSQL</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not permitted</td>
<td>Not permitted</td>
<td>Not permitted</td>
</tr>
</tbody>
</table>

PostgreSQL technically supports the use of any of the above four transaction isolation levels, but only three can practically be used. The Read-Uncommitted isolation level serves as Read-Committed.

The way the Repeatable-Read isolation-level is implemented does not allow for phantom reads, which is similar to the Serializable isolation-level. The primary difference between Repeatable-Read and Serializable is that Serializable guarantees that the result of concurrent transactions will be precisely the same as if they were executed serially, which is not always true for Repeatable-Reads.

**Isolation Levels Supported by PostgreSQL**
PostgreSQL supports the Read-Committed, Repeatable-Reads, and Serializable isolation levels. Read-Committed is the default isolation level (similar to the default isolation level in the Oracle database).

- **Read-Committed**
  The default PostgreSQL transaction isolation level. Preventing sessions from “seeing” data from concurrent transactions until it is committed. Dirty reads are not permitted.

- **Repeatable-Read**
  Queries can only see rows committed before the first query or DML statement was executed in the transaction.
• **Serializable**
  Provides the strictest transaction isolation level. The Serializable isolation level assures that the result of the concurrent transactions will be the same as if they were executed serially. This is not always the case for the Repeatable-Read isolation level.

**Multiversion Concurrency Control (MVCC)**
PostgreSQL implements a similar MVCC mechanism when compared to Oracle. In PostgreSQL, the MVCC mechanism allows transactions to work with a consistent snapshot of data ignoring changes made by other transactions which have not yet committed or rolled back. Each transaction “sees” a snapshot of accessed data accurate to its execution start time, regardless of what other transactions are doing concurrently.

**Setting Isolation Levels in Aurora PostgreSQL**
Isolation levels can be configured at several levels:
- Session level.
- Transaction level.
- Instance level using Aurora “Parameter Groups”.

**Example**
Configure the isolation level for a specific transaction:
```
Demo=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Demo=> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Demo=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Configure the isolation level for a specific session:
```
Demo=> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
Demo=> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Demo=> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

View the current isolation level:
```
Demo=> SELECT CURRENT_SETTING('TRANSACTION_ISOLATION');  -- Session
Demo=> SHOW DEFAULT_TRANSACTION_ISOLATION;               -- Instance
```

Modifying instance-level parameters for Aurora PostgreSQL is done using “Parameter Groups”. For example altering the `default_transaction_isolation` parameter using the AWS Console or the AWS CLI.

For additional details:

**PostgreSQL Transaction Synopsis**
```
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
```
<table>
<thead>
<tr>
<th>Database Feature</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoCommit</td>
<td>Off</td>
<td>On</td>
<td>Can be set to OFF</td>
</tr>
<tr>
<td>MVCC</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Default Isolation Level</td>
<td>Read Committed</td>
<td>Read Committed</td>
<td></td>
</tr>
<tr>
<td>Supported Isolation Levels</td>
<td>Serializable Read-only</td>
<td>Repeatable Reads Serializable Read-only</td>
<td></td>
</tr>
<tr>
<td>Configure Session Isolation Levels</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Configure Transaction Isolation Levels</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Nested Transaction Support</td>
<td>Yes</td>
<td>No</td>
<td>Consider using SAVEPOINT instead</td>
</tr>
<tr>
<td>Support for Transaction SAVEPOINTs</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Example
Read-Committed Isolation Level

<table>
<thead>
<tr>
<th>TX1</th>
<th>TX2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>Same results returned from both sessions</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td>employee_id</td>
</tr>
<tr>
<td>100</td>
<td>24000.00</td>
<td>100</td>
</tr>
<tr>
<td>begin; update employees set salary=27000 where employee_id=100;</td>
<td>begin; set transaction isolation level read committed;</td>
<td>TX1 starts a transaction; performs an update. TX2 starts a transaction with read-committed isolation level</td>
</tr>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>TX1 will “see” the modified results (27000.00) while TX2 “sees” the original data (24000.00)</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td>employee_id</td>
</tr>
<tr>
<td>100</td>
<td>27000.00</td>
<td>100</td>
</tr>
<tr>
<td>update employees set salary=29000 where employee_id=100;</td>
<td>wait as TX2 is blocked by TX1</td>
<td></td>
</tr>
<tr>
<td>commit;</td>
<td></td>
<td>TX1 issues a commit, and the lock is released</td>
</tr>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>Both queries return the value - 29000.00</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td>employee_id</td>
</tr>
<tr>
<td>100</td>
<td>29000.00</td>
<td>100</td>
</tr>
</tbody>
</table>
## Example
### Serializable Isolation Level

<table>
<thead>
<tr>
<th>TX1</th>
<th>TX2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>Same results returned from both sessions</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>24000.00</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>24000.00</td>
<td></td>
</tr>
<tr>
<td>begin; update employees set salary=27000 where employee_id=100;</td>
<td>begin; set transaction isolation level serializable;</td>
<td>TX1 starts a transaction; performs an update. TX2 starts a transaction with isolation level of read committed</td>
</tr>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>TX1 will “see” the modified results (27000.00) while TX2 “sees” the original data (24000.00)</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>27000.00</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>24000.00</td>
<td></td>
</tr>
<tr>
<td>update employees set salary=29000 where employee_id=100;</td>
<td></td>
<td>Waits as TX2 is blocked by TX1</td>
</tr>
<tr>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>select employee_id, salary from EMPLOYEES where employee_id=100;</td>
<td>Both queries will return the data updated according to TX1</td>
</tr>
<tr>
<td>employee_id</td>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>27000.00</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>27000.00</td>
<td></td>
</tr>
</tbody>
</table>

Commit;

| | ERROR: could not serialize access due to concurrent update |
| | TX2 received an error message |
| | Commit; ROLLBACK |
| | TX2 trying to issue a commit but receives a rollback message, the transaction failed due to the serializable isolation level |

Commit;

For additional details:
- [https://www.postgresql.org/docs/9.6/static/tutorial-transactions.html](https://www.postgresql.org/docs/9.6/static/tutorial-transactions.html)
- [https://www.postgresql.org/docs/9.6/static/transaction-iso.html](https://www.postgresql.org/docs/9.6/static/transaction-iso.html)
Overview
Oracle Materialized Views (also known as MViews) are table segments where the contents are periodically refreshed based on the results of a stored query. Oracle Materialized Views are defined with a specific user-supplied query and can be manually or automatically refreshed based on user-supplied configuration. A Materialized View will run its associated query and store the results as a table segment.

Oracle Materialized Views are especially useful for:

- Replication of data across multiple databases.
- Data warehouse use-cases.
- Performance enhancements by persistently storing the results of complex queries, as database tables.

Like ordinary views, Materialized Views are created with a SELECT query. The FROM clause of the MView query can reference other tables, views, and other Materialized Views. The source objects the Mview uses as data sources are also called “master tables” (replication terminology) or “detail tables” (data warehouse terminology).

Examples

1. Create a simple Materialized View named `mv1` which executes a simple SELECT statement on the employees table:

```sql
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM hr.employees;
```

2. Create a more complex Materialized View using a Database Link (remote) to obtain data from a table located in a remote database. This Materialized View also contains a subquery. The FOR UPDATE clause enables the Materialized View to be updated.

```sql
CREATE MATERIALIZED VIEW foreign_customers FOR UPDATE
AS SELECT * FROM sh.customers@remote cu
WHERE EXISTS (SELECT * FROM sh.countries@remote co
WHERE co.country_id = cu.country_id);
```

Immediate vs. Deferred Refresh
When creating Materialized Views, the BUILD IMMEDIATE option can be specified to instruct Oracle to immediately update the contents of the Materialized View by running the underlying query. This is different from the deferred update where the Materialized View is populated only on the first requested refresh.
Fast and Complete Refresh

1. **REFRESH FAST** – incremental data refresh. Only updates rows that have changed since the last refresh of the Materialized View instead of performing a complete refresh. This type of refresh fails if Materialized View Logs have not been created.
2. **COMPLETE** - the table segment used by the Materialized View is truncated (data is cleared) and repopulated entirely by running the associated query.

Materialized View Logs

When creating Materialized Views, a Materialized View Log can be used to instruct Oracle to store any changes performed by DML commands on the “master tables” that are used to refresh the Materialized View thus providing faster Materialized View refreshes. Without Materialized View Logs, Oracle must re-execute the query associated with the Materialized View each time (also known as a “complete refresh”). This process is slower compared with using Materialized View Logs.

Materialized View Refresh Strategy

1. **ON COMMIT** – refreshes the Materialized View upon any commit made on the underlying associated tables.
2. **ON DEMAND** – the refresh is initiated via a scheduled task or manually by the user.

Example

1. Create a Materialized View on two source tables – *times* and *products*. This approach enables **FAST** refresh of the Materialized View instead of the slower **COMPLETE** refresh.
2. Create a new Materialized View named *sales_mv* which will be refreshed incrementally (**REFRESH FAST**) each time changes in data are detected (**ON COMMIT**) on one, or more, of the tables associated with the Materialized View query.

```sql
CREATE MATERIALIZED VIEW LOG ON times
    WITH ROWID, SEQUENCE (time_id, calendar_year)
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON products
    WITH ROWID, SEQUENCE (prod_id)
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sales_mv
    BUILD IMMEDIATE
    REFRESH FAST ON COMMIT
    AS SELECT t.calendar_year, p.prod_id,
            SUM(s.amount_sold) AS sum_sales
    FROM times t, products p, sales s
    WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
    GROUP BY t.calendar_year, p.prod_id;
```

For additional details:
https://docs.oracle.com/database/121/DWHSG/basicmv.htm
https://docs.oracle.com/database/121/REPLN/repmview.htm#REPLN003
Migration to: PostgreSQL Materialized Views

Overview
PostgreSQL supports Materialized Views with associated queries similar to the Oracle implementation. The query associated with the Materialized View is executed and used to populate the Materialized View at the time the \texttt{REFRESH} command is issued. The PostgreSQL implementation of Materialized Views has three primary limitations when compared to Oracle Materialized Views:

1. PostgreSQL Materialized Views may be refreshed either manually or using a job running the \texttt{REFRESH MATERIALIZED VIEW} command. Automatic refresh of Materialized Views require the creation of a trigger.
2. PostgreSQL Materialized Views only support complete (full) refresh.
3. DML on Materialized Views is not supported.

Examples

1. Create a materialized view named \texttt{sales_summary} using the sales table as the source for the Materialized View:

   ```sql
   CREATE MATERIALIZED VIEW sales_summary AS
   SELECT
   seller_no,
   sale_date,
   sum(sale_amt)::numeric(10,2) as sales_amt
   FROM sales
   WHERE sale_date < CURRENT_DATE
   GROUP BY
   seller_no,
   sale_date
   ORDER BY
   seller_no,
   sale_date;
   ```

2. Execute a manual refresh of the Materialized View:

   ```sql
   REFRESH MATERIALIZED VIEW sales_summary;
   ```

Note: The Materialized View data will not be refreshed automatically if changes occur to its underlying tables. For automatic refresh of materialized view data, a trigger on the underlying tables must be created.

Creating a Materialized View
When you create a Materialized View in PostgreSQL, it uses a regular database table underneath. You can create database indexes on the Materialized View directly and improve performance of queries that access the Materialized View.
Example
Create an index on the `sellerno` and `sale_date` columns of the `sales_summary` Materialized View.

```sql
CREATE UNIQUE INDEX sales_summary_seller
ON sales_summary (seller_no, sale_date);
```

Oracle vs. PostgreSQL Materialized Views

<table>
<thead>
<tr>
<th></th>
<th>ORACLE</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Materialized View</td>
<td><code>CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;</code></td>
<td><code>CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;</code></td>
</tr>
<tr>
<td>Manual refresh of a View</td>
<td><code>DBMS_MVIEW.REFRESH('mv1', 'cf');</code></td>
<td><code>REFRESH MATERIALIZED VIEW mv1;</code></td>
</tr>
</tbody>
</table>
| Online refresh of a View  | `CREATE MATERIALIZED VIEW mv1 REFRESH FAST ON COMMIT AS SELECT * FROM employees;` | Create a trigger that will initiate a refresh after every DML command on the underlying tables:
|                           |                                                                        | `CREATE OR REPLACE FUNCTION refresh_mv1() returns trigger language plpgsql as $$
|                           |                                                                        |     begin
|                           |                                                                        |     refresh materialized view mv1;
|                           |                                                                        |     return null;
|                           |                                                                        |     end $$;
|                           |                                                                        |     create trigger refresh_mv1 after insert or update or delete or truncate on employees for each statement
|                           |                                                                        |     execute procedure refresh_mv1();
| Automatic incremental    | `CREATE MATERIALIZED VIEW LOG ON employees... INCLUDING NEW VALUES;`    | Not Supported                                                              |
| refresh of a View         | `CREATE MATERIALIZED VIEW mv1 REFRESH FAST AS SELECT * FROM employees;` |                                                                           |
| DML on Materialized View  | Supported                                                              | Not Supported                                                              |
| data                      |                                                                        |                                                                           |

For additional information on PostgreSQL materialized views:

https://www.postgresql.org/docs/current/static/rules-materializedviews.htm
Overview
A trigger in Oracle is a “named program” written in PL/SQL that is stored in the database and fired (or executed) when a specified event occurs. The associated event that causes a trigger to execute can either be tied to a specific database table, database view, database schema, or the database itself.

Triggers can be executed after:

- Data Manipulation Language (DML) statements (DELETE, INSERT, or UPDATE).
- Data Definition Language (DDL) statements (CREATE, ALTER, or DROP).
- Certain database events and operations (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Oracle trigger types:

1. **DML Trigger** – can be created on Tables or Views. The trigger fires when one of the following events occur on the object on which the trigger was created: inserting data, updating data, and deleting data.
   a. Triggers can be fired **before** or **after** a DML command occurred.

2. **INSTEAD OF Trigger** – a special type of DML trigger that is created on a non-editable view. INSTEAD OF triggers provide an application-transparent method to modify views that cannot be modified via DML statements.

3. **SYSTEM Event Triggers** – these are defined at the database level or schema level. These include triggers that fire after specific events:
   a. User logon and logoff.
   b. Database events (startup/shutdown), DataGuard events, server errors.

Examples
1. Create a trigger that is executed after a row is deleted from the PROJECTS table or if the primary key of a project is updated:

   ```sql
   SQL> CREATE OR REPLACE TRIGGER PROJECTS_SET_NULL
   2       AFTER DELETE OR UPDATE OF PROJECTNO ON PROJECTS
   3       FOR EACH ROW
   4       BEGIN
   5           IF UPDATING AND :OLD.PROJECTNO != :NEW.PROJECTNO OR DELETING THEN
   6               UPDATE EMP SET EMP.PROJECTNO = NULL
   7                   WHERE EMP.PROJECTNO = :OLD.PROJECTNO;
   8           END IF;
   9       END;
  
  
   Trigger created.
   SQL> DELETE FROM PROJECTS WHERE PROJECTNO=123;
   SQL> SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;
   PROJECTNO
   -------------
   NULL
   ```
2. Create a SYSTEM/Schema trigger on a table. The trigger fires if a DDL DROP command is executed for an object in the HR schema and prevents dropping of the object while raising an application error.

SQL> CREATE OR REPLACE TRIGGER PREVENT_DROP_TRIGGER
    BEFORE DROP ON HR.SCHEMA
    BEGIN
        RAISE_APPLICATION_ERROR (
            num => -20000,
            msg => 'Cannot drop object');
    END;
/
Trigger created.

SQL> DROP TABLE HR.EMP

ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: Cannot drop object
ORA-06512: at line 2

For additional details:
https://docs.oracle.com/database/121/LNPLS/create_trigger.htm#LNPLS01374
Overview
PostgreSQL triggers can be associated with a specific table, view, or foreign table and will invoke execution of a function when a certain events occur. DML triggers in PostgreSQL share much of the functionality that exists in Oracle triggers.

1. DML triggers (triggers that fire based on table related events such as DML).
2. Event triggers (triggers that fire after certain database events such as running DDL commands).

Unlike Oracle triggers, PostgreSQL triggers must call a function and do not support anonymous blocks of PL/pgSQL code as part of the trigger body. The user-supplied function is declared with no arguments and has a return type of `trigger`.

**PostgreSQL DML Triggers**

1. PostgreSQL triggers can run **BEFORE** or **AFTER** a DML operation.
   a. Fire before the operation is attempted on a row.
      i. Before constraints are checked and the `INSERT`, `UPDATE`, or `DELETE` is attempted.
      ii. If the trigger fires before or instead of the event, the trigger can skip the operation for the current row or change the row being inserted (for `INSERT` and `UPDATE` operations only).
   b. After the operation was completed, after constraints are checked and the `INSERT`, `UPDATE`, or `DELETE` command completed.
      i. If the trigger fires after the event, all changes, including the effects of other triggers, are "visible" to the trigger.

2. PostgreSQL triggers can run **INSTEAD OF** a DML command when created on views.

3. PostgreSQL triggers can run **FOR EACH ROW** affected by the DML statement or **FOR EACH STATEMENT** running only once as part of a DML statement.

<table>
<thead>
<tr>
<th>When Fired</th>
<th>Database Event</th>
<th>Row-Level Trigger (FOR EACH ROW)</th>
<th>Statement-Level Trigger (FOR EACH STATEMENT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE</td>
<td>INSERT, UPDATE, DELETE</td>
<td>Tables and foreign tables</td>
<td>Tables, views, and foreign tables</td>
</tr>
<tr>
<td></td>
<td>TRUNCATE</td>
<td>—</td>
<td>Tables</td>
</tr>
<tr>
<td>AFTER</td>
<td>INSERT, UPDATE, DELETE</td>
<td>Tables and foreign tables</td>
<td>Tables, views, and foreign tables</td>
</tr>
<tr>
<td></td>
<td>TRUNCATE</td>
<td>—</td>
<td>Tables</td>
</tr>
<tr>
<td>INSTEAD OF</td>
<td>INSERT, UPDATE, DELETE</td>
<td>Views</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TRUNCATE</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
**PostgreSQL Event Triggers**

An event trigger executes when a specific event that is associated with the trigger occurs in the database. Supported events include: `ddl_command_start`, `ddl_command_end`, `table_rewrite` and `sql_drop`.

1. **`ddl_command_start`** - occurs before the execution of a `CREATE`, `ALTER`, `DROP`, `SECURITY` `LABEL`, `COMMENT`, `GRANT`, `REVOKE` or `SELECT INTO` command.
2. **`ddl_command_end`** – occurs after the command completed and before the transaction commits.
3. **`sql_drop`** – fired only for the `DROP DDL command`. Fires before `ddl_command_end` trigger fire.

Full list of supported PostgreSQL event trigger types:  

**Example**

Create a DML trigger:

1. In order to create an equivalent version of the Oracle DML trigger in PostgreSQL, first create a function trigger which will store the execution logic for the trigger:

```sql
psql=> CREATE OR REPLACE FUNCTION PROJECTS_SET_NULL()
    RETURNS TRIGGER
    AS $$
    BEGIN
        IF TG_OP = 'UPDATE' AND OLD.PROJECTNO != NEW.PROJECTNO OR
            TG_OP = 'DELETE' THEN
            UPDATE EMP
            SET PROJECTNO = NULL
            WHERE EMP.PROJECTNO = OLD.PROJECTNO;
        END IF;

        IF TG_OP = 'UPDATE' THEN RETURN NULL;
        ELSIF TG_OP = 'DELETE' THEN RETURN NULL;
        END IF;
    END;
    $$
    LANGUAGE PLPGSQL;

CREATE FUNCTION
```

2. Create the trigger itself:

```sql
psql=> CREATE TRIGGER TRG_PROJECTS_SET_NULL
    AFTER UPDATE OF PROJECTNO OR DELETE
    ON PROJECTS
    FOR EACH ROW
    EXECUTE PROCEDURE PROJECTS_SET_NULL();
```

CREATE TRIGGER
3. Test the trigger by deleting a row from the `PROJECTS` table:

```
psql=> DELETE FROM PROJECTS WHERE PROJECTNO=123;
pql=> SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;

<table>
<thead>
<tr>
<th>projectno</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
</tbody>
</table>
```

---

**Example**

Create a DDL trigger:

1. In order to create an equivalent version of the Oracle DDL System/Schema level triggers, such as a trigger that prevent running a DDL `DROP` on objects in the `HR` schema: first create an event trigger function. Note that trigger functions are created with no arguments and must have a return type of `TRIGGER` or `EVENT_TRIGGER`:

```sql
psql=> CREATE OR REPLACE FUNCTION ABORT_DROP_COMMAND()
RETURNS EVENT_TRIGGER
AS $$
BEGIN
   RAISE EXCEPTION 'The % Command is Disabled', tg_tag;
END; $$
LANGUAGE PLPGSQL;
```

CREATE FUNCTION

2. Create the event trigger, which will fire before the start of a DDL `DROP` command:

```sql
psql=> CREATE EVENT TRIGGER trg_abort_drop_command
ON DDL_COMMAND_START
WHEN TAG IN ('DROP TABLE', 'DROP VIEW', 'DROP FUNCTION', 'DROP SEQUENCE', 'DROP MATERIALIZED VIEW', 'DROP TYPE')
EXECUTE PROCEDURE abort_drop_command();
```

3. Test the trigger by attempting to drop the `EMPLOYEES` table:

```
psql=> DROP TABLE EMPLOYEES;
```

```
ERROR: The DROP TABLE Command is Disabled
CONTEXT: PL/pgSQL function abort_drop_command() line 3 at RAISE
```

---

For additional details:  
<table>
<thead>
<tr>
<th><strong>Oracle</strong></th>
<th><strong>PostgreSQL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&quot;Before update&quot; trigger, row level</strong></td>
<td>CREATE OR REPLACE TRIGGER check_update BEFORE UPDATE ON projects FOR EACH ROW BEGIN /<em>Trigger body</em>/ END; / CREATE TRIGGER check_update BEFORE UPDATE ON employees FOR EACH ROW EXECUTE PROCEDURE myproc();</td>
</tr>
<tr>
<td><strong>&quot;Before update&quot; trigger, statement level</strong></td>
<td>CREATE OR REPLACE TRIGGER check_update BEFORE UPDATE ON projects BEGIN /<em>Trigger body</em>/ END; / CREATE TRIGGER check_update BEFORE UPDATE ON employees FOR EACH STATEMENT EXECUTE PROCEDURE myproc();</td>
</tr>
<tr>
<td><strong>System / event trigger</strong></td>
<td>CREATE OR REPLACE TRIGGER drop_trigger BEFORE DROP ON hr.SCHEMA BEGIN RAISE_APPLICATION_ERROR ( num =&gt; -20000, msg =&gt; 'Cannot drop object'); END; / CREATE EVENT TRIGGER trg_drops ON ddl_command_start EXECUTE PROCEDURE trg_drops();</td>
</tr>
<tr>
<td><strong>Referencing :old and :new values in triggers</strong></td>
<td>Use &quot;:NEW&quot; and &quot;:OLD&quot; in trigger body: CREATE OR REPLACE TRIGGER UpperNewDeleteOld BEFORE INSERT OR UPDATE OF first_name ON employees FOR EACH ROW BEGIN :NEW.first_name := UPPER(:NEW.first_name); :NEW.salary := :OLD.salary; END; / Use &quot;:NEW&quot; and &quot;:OLD&quot; in trigger Procedure body: CREATE OR REPLACE FUNCTION log_emp_name_upd() RETURNS trigger LANGUAGE plpgsql AS $$ BEGIN IF NEW.last_name &lt;&gt; OLD.last_name THEN INSERT INTO employee_audit (employee_id,last_name,changed_on) VALUES (OLD.id,OLD.last_name ,now()); END IF; RETURN NEW; END; $$ CREATE TRIGGER last_name_change_trg BEFORE UPDATE ON employees FOR EACH ROW EXECUTE PROCEDURE log_last_emp_name_upd();</td>
</tr>
<tr>
<td><strong>Database event level trigger</strong></td>
<td>CREATE TRIGGER register_shutdown ON DATABASE SHUTDOWN BEGIN Insert into logging values ('DB was</td>
</tr>
<tr>
<td></td>
<td>Oracle</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Drop a trigger</td>
<td>shutdown’, sysdate); commit; END;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify logic executed by a trigger</td>
<td>Can be used with create or replace CREATE OR REPLACE TRIGGER UpperNewDeleteOld BEFORE INSERT OR UPDATE OF first_name ON employees FOR EACH ROW BEGIN &lt;&lt;NEW CONTENT&gt;&gt; END;</td>
</tr>
<tr>
<td>Enable a trigger</td>
<td>ALTER TRIGGER UpperNewDeleteOld ENABLE;</td>
</tr>
<tr>
<td>Disable a trigger</td>
<td>ALTER TRIGGER UpperNewDeleteOld DISABLE;</td>
</tr>
</tbody>
</table>
Overview
Database Views store a named SQL query in the Oracle Data Dictionary with a predefined structure. A view does not store actual data and may be considered as a “virtual table” or a “logical table” and is based on the data from one or more “physical” database tables.

Oracle view main privileges as a prerequisite for Creation
- A user must have the CREATE VIEW privilege to create a view in their own schema.
- A user must have the CREATE ANY VIEW privilege to create a view in any schema.
- The owner of a view must have all the necessary privileges on the source tables or views on which the view is based (SELECT or DML privileges).

Oracle views CREATE (OR REPLACE) statements
- CREATE VIEW to create a new view.
- CREATE OR REPLACE to overwrite an existing view and change the view definition without having to manually drop and re-create the original view and without deleting the previously granted privileges.

Example:
```
CREATE VIEW "HR"."EMP_DETAILS_VIEW"...

CREATE OR REPLACE VIEW "HR"."EMP_DETAILS_VIEW"...
```

Oracle common view parameters

<table>
<thead>
<tr>
<th>Oracle View Parameter</th>
<th>Description</th>
<th>PostgreSQL Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE OR REPLACE</td>
<td>Re-create an existing view (if one exists) or create a new view.</td>
<td>Yes</td>
</tr>
<tr>
<td>FORCE</td>
<td>Create the view regardless the existence of the source tables or views and regardless to view privileges.</td>
<td>No</td>
</tr>
<tr>
<td>VISIBLE</td>
<td>INVISIBLE</td>
<td>Specify if a column based on the view will be visible or invisible.</td>
</tr>
<tr>
<td>WITH READ ONLY</td>
<td>Disable DML commands.</td>
<td>No</td>
</tr>
<tr>
<td>WITH CHECK OPTION</td>
<td>Specifies the level of enforcement when performing DML commands on the view</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Running DML Commands On views

Views are classified as follows:

- **Simple View**
  A view having a single source table with no aggregate functions. DML operations can be performed on simple views and affect the base table(s).

  **Example:** Simple view + update operation

  ```sql
  SQL> CREATE OR REPLACE VIEW VW_EMP
         AS
         SELECT EMPLOYEE_ID, LAST_NAME, EMAIL, SALARY
         FROM EMPLOYEES
         WHERE DEPARTMENT_ID BETWEEN 100 AND 130;

         UPDATE VW_EMP
         SET EMAIL=EMAIL||'.org'
         WHERE EMPLOYEE_ID=110;

         1 rows updated.
  ```

- **Complex View**
  A view with several source tables or views containing joins, aggregate (group) functions, or an order by clause. Performing DML operations on complex views cannot be done directly, but INSTEAD OF triggers can be used as a workaround.

  **Example:** Complex view + update operation

  ```sql
  SQL> CREATE OR REPLACE VIEW VW_DEP
         AS
         SELECT B.DEPARTMENT_NAME, COUNT(A.EMPLOYEE_ID) AS CNT
         FROM EMPLOYEES A JOIN DEPARTMENTS B USING(DEPARTMENT_ID)
         GROUP BY B.DEPARTMENT_NAME;

         UPDATE VW_DEP
         SET CNT=CNT +1
         WHERE DEPARTMENT_NAME=90;

         ORA-01732: data manipulation operation not legal on this view
  ```

For additional details:
- [https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_8004.htm#SQLRF01504](https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_8004.htm#SQLRF01504)
- [https://docs.oracle.com/database/121/SQLRF/statements_8004.htm](https://docs.oracle.com/database/121/SQLRF/statements_8004.htm)
Migration to: PostgreSQL Views

Overview
PostgreSQL views share functionality with Oracle views. Creating a view defines a stored query based on one or more physical database tables which executes every time the view is accessed.

PostgreSQL View Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

PostgreSQL View Privileges
A Role or user must be granted SELECT and DML privileges on the bases tables or views in order to create a view.

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-grant.html

PostgreSQL View Parameters

- **CREATE [OR REPLACE] VIEW**
  Similar to the Oracle syntax. Note that when re-creating an existing view, the new view must have the same column structure as generated by the original view (column names, column order and data types). As such, it is sometimes preferable to drop the view and use the `CREATE VIEW` statement instead.

  ```
  hr=# CREATE [ OR REPLACE ] VIEW VW_NAME AS
  SELECT COLUMNS
  FROM TABLE(s)
  [WHERE CONDITIONS];
  ```

  ```
  hr=# DROP VIEW [IF EXISTS] VW_NAME;
  ```

  * The **IF EXISTS** parameter is optional
• WITH [ CASCADED | LOCAL ] CHECK OPTION
DML INSERT and UPDATE operations are verified against the view based tables to ensure that new rows satisfy the original structure conditions or the view-defining condition. If a conflict is detected, the DML operation fails.

CHECK OPTION
- LOCAL - Verifies against the view without a hierarchical check.
- CASCADED - Verifies all underlying base views using a hierarchical check.

• Executing DML Commands On views
PostgreSQL simple views are automatically updatable. Unlike Oracle views, no restrictions exist when performing DML operations against views. An updatable view may contain a combination of updatable and non-updatable columns. A column is updatable if it references an updatable column of the underlying base table. If not, the column is read-only and an error is raised if an INSERT or UPDATE statement is attempted on the column.

Example 1
Creating and updating a view without the CHECK OPTION parameter:

```
hr=# CREATE OR REPLACE VIEW VW_DEP AS
    SELECT DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
    FROM DEPARTMENTS
    WHERE LOCATION_ID=1700;

view VW_DEP created.

hr=# UPDATE VW_DEP
    SET LOCATION_ID=1600;

21 rows updated.
```

Example 2
Creating and updating a view with the LOCAL CHECK OPTION parameter:

```
hr=# CREATE OR REPLACE VIEW VW_DEP AS
    SELECT DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
    FROM DEPARTMENTS
    WHERE LOCATION_ID=1700
    WITH LOCAL CHECK OPTION;

view VW_DEP created.

hr=# UPDATE VW_DEP
    SET LOCATION_ID=1600;

SQL Error: ERROR: new row violates check option for view "vw dep"
```

For additional details:
https://www.postgresql.org/docs/9.6/static/tutorial-views.html
https://www.postgresql.org/docs/9.6/static/sql-createview.html
Overview
Sequences are database objects that serve as a unique identity value generators, such as for automatically generating primary key values. Oracle treats sequences as independent objects and the same sequence can generate values for multiple tables.

Sequences can be configured with multiple parameters which control their value-generating behavior. For example, the INCREMENT BY sequence parameter defines the interval between each generated sequence value. If more than one database user is generating incremented values from the same sequence, each user may encounter gaps in the generated values that are visible to them.

Oracle Sequence Options
By default, the initial and increment values for a sequence are both 1, with no upper limit.

- **INCREMENT BY**
  Controls the sequence interval value of the increment or decrement (if a negative value is specified). If the INCREMENT BY parameter is not specified during sequence creation, the value is set to 1. The increment cannot be assigned a value of 0.

- **START WITH**
  Defines the initial value of a sequence. The default value is 1.

- **MAXVALUE | NOMAXVALUE**
  Allows controlling the maximum limit for values generated by a sequence. Must be equal or greater than the START WITH parameter and must be greater in value than the MINVALUE parameter. The default for NOMAXVALUE is $10^{27}$ for an ascending sequence.

- **MINVALUE | NOMINVALUE**
  Allows controlling the minimum limit for values generated by a sequence. Must be less than or equal to the START WITH parameter and must be less than the MAXVALUE parameter. The default for NOMINVALUE is $-10^{26}$ for a descending sequence.

- **CYCLE | NOCYCLE**
  Instructs a sequence to continue generating values despite reaching the maximum value or the minimum value. If the sequence reaches one of the defined ascending limits, it generates a new value according to the minimum value. If it reaches a descending limit, it generates a new value according to the maximum value. NOCYCLE is the default.
• **CACHE | NOCACHE**

The `CACHE` parameter enables controlling the number of sequence values to keep cached in memory for improved performance. `CACHE` has a minimum value of “2”.

Using the `NOCACHE` parameter will cause a sequence not to cache any values in memory. Specifying neither `CACHE` or `NOCACHE` will cache 20 values to memory. In the event of a database failure, all cached sequence values that have not been used, will be lost and gaps in sequence values may occur.

**Example**

Creating a sequence:

```sql
SQL> CREATE SEQUENCE SEQ_EMP
    2      START WITH 100
    3      INCREMENT BY 1
    4      MAXVALUE 99999999999
    5      CACHE 20
    6      NOCYCLE;
```

Dropping a sequence:

```sql
SQL> DROP SEQUENCE SEQ_EMP;
```

Viewing sequences created for the current schema/user:

```sql
SQL> SELECT * FROM USER_SEQUENCES;
```

Using sequence as part of an `INSERT INTO` statement:

```sql
SQL> CREATE TABLE EMP_SEQ_TST (
    2  COL1 NUMBER PRIMARY KEY,
    3  COL2 VARCHAR2(30));
SQL> INSERT INTO EMP_SEQ_TST VALUES(SEQ_EMP.NEXTVAL, 'A');
```

```text
<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A</td>
</tr>
</tbody>
</table>
```

Query the current value of a sequence:

```sql
SQL> SELECT SEQ_EMP.CURRVAL FROM DUAL;
```

Manually increment the value of a sequence, according to the `INCREMENT BY` specification:

```sql
SQL> SELECT SEQ_EMP.NEXTVAL FROM DUAL;
```
Altering an existing sequence:

```sql
SQL> ALTER SEQUENCE SEQ_EMP MAXVALUE 1000000;
```

**Oracle 12c Default Values Using Sequences**

Starting with Oracle 12c, you can assign a sequence to a table column with the `CREATE TABLE` statement and specify the `NEXTVAL` configuration of the sequence during table creation.

**Example**

Generating DEFAULT values using sequences in Oracle 12c:

```sql
SQL> CREATE TABLE SEQ_TST (  
    COL1 NUMBER DEFAULT SEQ_1.NEXTVAL PRIMARY KEY,  
    COL2 VARCHAR(30));

SQL> INSERT INTO SEQ_TST(COL2) VALUES('A');

SQL> SELECT * FROM SEQ_TST;

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A</td>
</tr>
</tbody>
</table>

**Oracle 12c Session Sequences (Session/Global)**

Starting with Oracle 12c, sequences can be created as session-level or global-level sequences. By adding the `SESSION` parameter to `CREATE SEQUENCE`, the sequence will be created as a session-level sequence. Optionally, the `GLOBAL` keyword can be used to create a sequence as a global sequence to provide consistent results across sessions in the database. Global sequences are the default. Session sequences return a unique range of sequence numbers only within a session.

**Example**

Oracle 12c SESSION and GLOBAL sequences:

```sql
SQL> CREATE SEQUENCE SESSION_SEQ SESSION;

SQL> CREATE SEQUENCE SESSION_SEQ GLOBAL;
```

**Oracle 12c Identity Columns**

Sequences can be used as an `IDENTITY` type, which automatically creates a sequence and associates it with the table column. The main difference is that there is no need to create a sequence manually; the `IDENTITY` type does that for you. An `IDENTITY` type is a sequence that can be configured.

**Example**

Oracle 12c Identity Columns:

Inserting records using an Oracle 12c `IDENTITY` column explicitly/implicitly:

```sql
SQL> INSERT INTO IDENTITY_TST(COL2) VALUES('A');

SQL> INSERT INTO IDENTITY_TST(COL1, COL2) VALUES(DEFAULT, 'B');

SQL> INSERT INTO IDENTITY_TST(col1, col2) VALUES(NULL, 'C');

SQL> SELECT * FROM IDENTITY_TST;

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>A</td>
</tr>
<tr>
<td>130</td>
<td>B</td>
</tr>
</tbody>
</table>
```
For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_6015.htm#SQLRF01314
https://docs.oracle.com/database/121/SQLRF/statements_6017.htm#SQLRF01314
Migration to: PostgreSQL Sequences

Overview
The PostgreSQL CREATE SEQUENCE command is mostly compatible with the Oracle CREATE SEQUENCE command. Sequences in PostgreSQL serve the same purpose as in Oracle; they generate numeric identifiers automatically. A sequence object is owned by the user that created it.

PostgreSQL Sequence Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

Most Oracle CREATE SEQUENCE parameters are compatible with PostgreSQL. Similar to Oracle 12c, in PostgreSQL you can create a sequence and use it directly as part of a CREATE TABLE statement.

Sequence Parameters

- **TEMPORARY or TEMP**
  PostgreSQL can create a temporary sequence within a session. Once the session ends, the sequence is automatically dropped.

- **IF NOT EXISTS**
  Creates a sequence even if a sequence with an identical name already exists. Replaces the existing sequence.

- **INCREMENT BY**
  Optional parameter with a default value of 1. Positive values generate sequence values in ascending order. Negative values generate sequence values in descending sequence.

- **START WITH**
  The same as Oracle. This is an optional parameter having a default of 1. It uses the MINVALUE for ascending sequences and the MAXVALUE for descending sequences.

- **MAXVALUE | NO MAXVALUE**
  Defaults are between $2^{63}$ for ascending sequences and -1 for descending sequences.

- **MINVALUE | NO MINVALUE**
  Defaults are between 1 for ascending sequences and $-2^{63}$ for descending sequences.
• **CYCLE | NO CYCLE**
  If the sequence value reaches MAXVALUE or MINVALUE, the CYCLE parameter instructs the sequence to return to the initial value (MINVALUE or MAXVALUE). The default is NO CYCLE.

• **CACHE**
  Note that in PostgreSQL, the NOCACHE is not supported. By default, when not specifying the CACHE parameter, no sequence values will be pre-cached into memory, which is equivalent to the Oracle NOCACHE parameter. The minimum value is 1.

• **OWNED BY | OWNBY NON**
  Specifies that the sequence object is to be associated with a specific column in a table, which is not supported by Oracle. When dropping this type of sequence, an error will be returned because of the sequence/table association.

### Example

#### Create a sequence:
```sql
demo=> CREATE SEQUENCE SEQ_1
     START WITH 100
     INCREMENT BY 1
     MAXVALUE 99999999999
     CACHE 20
     NO CYCLE;
```

*Identical to Oracle syntax, except for the whitespace in the NO CYCLE parameter.*

#### Drop a sequence:
```sql
demo=> DROP SEQUENCE SEQ_1;
```

#### View sequences created in the current schema and sequence specifications:
```sql
demo=> SELECT * FROM INFORMATION_SCHEMA.SEQUENCES;
OR
demo=> \ds
```

#### Use a PostgreSQL sequence as part of a CREATE TABLE and an INSERT statement:
```sql
demo=> CREATE TABLE SEQ_TST(
    COL1 NUMERIC DEFAULT NEXTVAL('SEQ_1') PRIMARY KEY,
    COL2 VARCHAR(30));

demo=> INSERT INTO SEQ_TST (COL2) VALUES('A');

demo=> SELECT * FROM SEQ_TST;
```

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A</td>
</tr>
</tbody>
</table>
Use the `OWNED BY` parameter to associate the sequence with a table:

```sql
demo=> CREATE SEQUENCE SEQ_1
    START WITH 100
    INCREMENT BY 1
    OWNED BY SEQ_TST.COL1;
```

Query the current value of a sequence:

```sql
demo=> SELECT CURRVAL('SEQ_1');
```

Manually increment a sequence value according to the `INCREMENT BY` value:

```sql
demo=> SELECT NEXTVAL('SEQ_1');
OR
demo=> SELECT SETVAL('SEQ_1', 200);
```

Alter an existing sequence:

```sql
demo=> ALTER SEQUENCE SEQ_1 MAXVALUE 1000000;
```

**Generating Sequence by SERIAL Type**

PostgreSQL enables you to create a sequence that is similar to the `AUTO_INCREMENT` property supported by identity columns in Oracle 12c. When creating a new table, the sequence is created through the `SERIAL` pseudo-type. Other types from the same family are `SMALLSERIAL` and `BIGSERIAL`.

By assigning a `SERIAL` type to a column on table creation, PostgreSQL creates a sequence using the default configuration and adds a `NOT NULL` constraint to the column. The newly created sequence behaves as a regular sequence.

**Example**

**Using a SERIAL Sequence**:

```sql
demo=> CREATE TABLE SERIAL_SEQ_TST(
    COL1 SERIAL PRIMARY KEY,
    COL2 VARCHAR(10));
demo=> INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
demo=> SELECT * FROM SERIAL_SEQ_TST;

col1 | col2
-----+------
1    | A
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>serial_seq_tst_col1_seq</td>
<td>sequence</td>
<td>pg_tst_db</td>
</tr>
</tbody>
</table>
### Oracle Sequences vs. PostgreSQL Sequences:

<table>
<thead>
<tr>
<th>Parameter/Feature</th>
<th>Compatibility with PostgreSQL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create sequence syntax</td>
<td>Full, with minor differences</td>
<td>See Exceptions</td>
</tr>
<tr>
<td>INCREMENT BY</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>START WITH</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>NOMAXVALUE</td>
<td>Full</td>
</tr>
<tr>
<td>MINVALUE</td>
<td>NOMINVALUE</td>
<td>Full</td>
</tr>
<tr>
<td>CYCLE</td>
<td>NOCYCLE</td>
<td>Full</td>
</tr>
<tr>
<td>CACHE</td>
<td>NOCACHE</td>
<td></td>
</tr>
</tbody>
</table>

**Default values using sequences (Oracle 12c)**

Supported by PostgreSQL

**Session sequences (session / global), Oracle 12c**

Supported by PostgreSQL by using the **TEMPORARY** sequence parameter to Oracle **SESSION sequence**

**Oracle 12c identity columns**

Supported by PostgreSQL by using the **SERIAL** data type as sequence

---

For additional details:

- [https://www.postgresql.org/docs/9.6/static/functions-sequence.html](https://www.postgresql.org/docs/9.6/static/functions-sequence.html)
Migrating from: Oracle Database Links

Overview
Database Links are schema objects that are used to interact with remote database objects such as tables. Common use cases for database links include selecting data from tables that reside in a remote database.

**Note:** To use database links, Oracle net services must be installed on both the local and remote database servers to facilitate communications.

Example:
1. Create a database link named `remote_db`. When creating a database link, you have the option to either specify the remote database destination using a TNS Entry or specify the full TNS Connection string.

   ```sql
   CREATE DATABASE LINK remote_db
   CONNECT TO username IDENTIFIED BY password
   USING 'remote';
   
   CREATE DATABASE LINK remotenoTNS
   CONNECT TO username IDENTIFIED BY password
   USING '(DESCRIPTION=(ADDRESS_LIST=(ADDRESS = (PROTOCOL = TCP)(HOST =192.168.1.1)(PORT = 1521)))(CONNECT_DATA =(SERVICE_NAME = orcl)))';
   ```

2. After the database link is created, you can use the database link directly as part of a SQL query using the database link name (`@remote_db`) as a postfix to the table name.

   ```sql
   SELECT * FROM employees@remote_db;
   ```

3. Database links also support DML commands:

   ```sql
   INSERT INTO employees@remote_db
   (employee_id, last_name, email, hire_date, job_id)
   VALUES (999, 'Claus', 'sclaus@example.com', SYSDATE, 'SH_CLERK');

   UPDATE jobs@remote_db SET min_salary = 3000
   WHERE job_id = 'SH_CLERK';

   DELETE FROM employees@remote_db
   WHERE employee_id = 999; 
   ```

For additional details:
https://docs.oracle.com/database/121/SQLRF/statements_5006.htm#SQLRF01205
Migration to: PostgreSQL DBLink and FDWrapper

Overview
Querying data in remote databases in PostgreSQL is available via two primary options:

1. **dblink** database link function.
2. **postgresql_fdw** (Foreign Data Wrapper, FDW) extension.

The Postgres foreign data wrapper extension is new to PostgreSQL and offers functionality that is similar to **dblink**. However, the Postgres foreign data wrapper aligns closer with the SQL standard and can provide improved performance.

Example using the **dblink** function

1. **Load the dblink extension into PostgreSQL:**
   ```sql
   CREATE EXTENSION dblink;
   ```

2. **Create a persistent connection to a remote PostgreSQL database using the dblink_connect function specifying a connection name (myconn), database name (postgresql), port (5432), host (hostname), user (username) and password (password).**
   ```sql
   SELECT dblink_connect('myconn', 'dbname=postgres port=5432 host=hostname user=username password=password');
   ```

   The connection can be used to execute queries against the remote database.

3. **Execute a query using the previously created connection (myconn) via the dblink function.**
   The query returns the id and name columns from the employees table. On the remote database, you must specify the connection name and the SQL query to execute as well as parameters and datatypes for selected columns (id and name in this example).
   ```sql
   SELECT * 
   from dblink('myconn', 'SELECT id, name FROM EMPLOYEES') AS p(id int, fullname text);
   ```

4. **Close the connection using the dblink_disconnect function.**
   ```sql
   SELECT dblink_disconnect('myconn');
   ```
5. Alternatively, you can use the `dblink` function specifying the full connection string to the remote PostgreSQL database, including: database name, port, hostname, username, and password. This can be done instead of using a previously defined connection. You must also specify the SQL query to execute as well as parameters and datatypes for the selected columns (`id` and `name`, in this example).

```
SELECT *
FROM dblink('dbname=postgres port=5432
don't=hostname user=username password=password',
            'SELECT id, name FROM EMPLOYEES')
AS p(id int, fullname text);
```

6. DML commands are supported on tables referenced via the `dblink` function. For example, you can insert a new row and then delete it from the remote table.

```
SELECT * FROM dblink('myconn',$$INSERT into employees VALUES (3,'New Employees No. 3!')$$) AS t(message text);
```

```
SELECT * FROM dblink('myconn',$$DELETE FROM employees WHERE id=3$$) AS t(message text);
```

7. Create a new local table (`new_employees_table`) by querying data from a remote table.

```
SELECT emps.* INTO new_employees_table FROM dblink('myconn','SELECT *
FROM employees') AS emps(id int, name varchar);
```

8. Join remote data with local data.

```
SELECT local_emps.id , local_emps.name, s.sale_year, s.sale_amount,
FROM local_emps INNER JOIN dblink('myconn','SELECT * FROM
working_hours') AS s(id int, hours_worked int) ON
local_emps.id = s.id;
```

9. Execute DDL statements in the remote database.

```
SELECT * FROM dblink('myconn',$$CREATE table new_remote_tbl
(a int, b text)$$) AS t(a text);
```

For additional details:
https://www.postgresql.org/docs/9.6/static/dblink.html
Example using the PostgreSQL Foreign Data Wrapper

1. Load the `fdw` Extension into PostgreSQL.
   
   ```sql
   CREATE EXTENSION postgres_fdw;
   ```

2. Create a connection to the remote PostgreSQL database specifying the remote server (`hostname`), database name (`postgresql`) and the port (5432).
   
   ```sql
   CREATE SERVER remote_db
   FOREIGN DATA WRAPPER postgres_fdw
   OPTIONS ('host' 'hostname', 'dbname' 'postgresql', 'port' '5432');
   ```

3. Create the user mapping, specifying:
   - The `local_user` is a user with permissions in the `current` database.
   - Specify the server connection created in the previous command (`remote_db`).
   - The `user` and `password` arguments specified in the options clause must have the required permissions in the `remote` database.

   ```sql
   CREATE USER MAPPING FOR local_user
   SERVER remote_db
   OPTIONS (user 'remote_user', password 'remote_password');
   ```

After the connection with login credentials for the remote database was created, we can either import individual tables or the entire schema containing all, or some, of the tables and views.

4. Create a FOREIGN TABLE named `foreign_emp_tbl` using the `remote_db` remote connection created earlier specifying both the schema name and table name in the remote database to be queried. For example, the `hr.employees` table.

   ```sql
   CREATE FOREIGN TABLE foreign_emp_tbl (
   id int, name text)
   SERVER remote_db
   OPTIONS (schema_name 'hr', table_name 'employees');
   ```

5. Queries running on the local `foreign_emp_tbl` table will actually query data directly from the remote `hr.employees` table.

   ```sql
   SELECT * FROM foreign_emp_tbl;
   ```

6. You can also “import” an entire schema, or specific tables, without specifying a specific table name:

   ```sql
   IMPORT FOREIGN SCHEMA hr LIMIT TO (employees)
   FROM SERVER remote_db INTO local_hr;
   ```
Notes:

1. Both dblink and FDW store the remote database username and password as plain-text, in two locations:
   a. The pg_user_mapping view, accessible only to “super users” in the database.
   b. When using the dblink function, passwords can be stored in your code or procedures inside the database.

2. Any changes to PostgreSQL user passwords require changing the FDW/dblink specifications as well.

3. When using FDW, if columns in the remote tables have been dropped or renamed, the queries will fail. The FDW tables must be re-created.

For additional details: https://www.postgresql.org/docs/current/static/postgres-fdw.html

Oracle Database Links vs. PostgreSQL DBLink

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>PostgreSQL DBLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a permanent “named” database link</td>
<td>CREATE DATABASE LINK remote</td>
<td>Not Supported.</td>
</tr>
<tr>
<td></td>
<td>CONNECT TO username IDENTIFIED BY password</td>
<td>You have to manually open the connection to the remote database in your sessions / queries:</td>
</tr>
<tr>
<td></td>
<td>USING 'remote';</td>
<td>SELECT dblink_connect('myconn', 'dbname=postgres</td>
</tr>
<tr>
<td></td>
<td></td>
<td>port=5432</td>
</tr>
<tr>
<td></td>
<td></td>
<td>host=hostname</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user=username</td>
</tr>
<tr>
<td></td>
<td></td>
<td>password=password');</td>
</tr>
<tr>
<td>Query using a database link</td>
<td>SELECT * FROM employees@remote;</td>
<td>SELECT * FROM dblink('myconn', 'SELECT * FROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>employees') AS p(id int, fullname text, address</td>
</tr>
<tr>
<td>DML using database link</td>
<td>INSERT INTO employees@remote</td>
<td>text);</td>
</tr>
<tr>
<td></td>
<td>(employee_id, last_name, email, hire_date, job_id)</td>
<td>SELECT * FROM dblink('myconn',$$INSERT into</td>
</tr>
<tr>
<td></td>
<td>VALUES (999, 'Claus', '<a href="mailto:sclaus@example.com">sclaus@example.com</a>',</td>
<td>employees VALUES (45, 'Dan', 'South side</td>
</tr>
<tr>
<td></td>
<td>SYSDATE, 'SH_CLERK');</td>
<td>7432, NY')$$) AS t(id int, name text, address</td>
</tr>
<tr>
<td>Heterogeneous database link connections, such</td>
<td>Supported.</td>
<td>create extension oracle_fdw not supported by</td>
</tr>
<tr>
<td>as Oracle to</td>
<td></td>
<td>Amazon RDS.</td>
</tr>
<tr>
<td>Description</td>
<td>Oracle</td>
<td>PostgreSQL Dblink</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><strong>PostgreSQL or vice-versa</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run DDL via a database link</td>
<td>Not supported directly, but you can run a procedure or create a job on the remote database and executes the desired DDL commands.</td>
<td>SELECT * FROM dblink('myconn', $$CREATE table my_remote_tbl (a int, b text)$$) AS t(a text);</td>
</tr>
<tr>
<td>Delete a database link</td>
<td>drop database link remote;</td>
<td>Not supported. Close the DBLink connection instead.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SELECT dblink_disconnect('myconn');</td>
</tr>
</tbody>
</table>

**PostgreSQL DBLink vs. FDW**

<table>
<thead>
<tr>
<th>Description</th>
<th>PostgreSQL Dblink</th>
<th>PostgreSQL FDW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a permanent reference to a remote table using a database link</td>
<td>Not supported</td>
<td>After creating:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DFW Server definition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- User Mapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Run:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CREATE FOREIGN TABLE foreign_emp_tbl (id int, name text, address text )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SERVER foreign_server OPTIONS (schema_name 'hr', table_name 'employees');</td>
</tr>
<tr>
<td>Query remote data</td>
<td>SELECT * FROM dblink('myconn', 'SELECT * FROM employees') AS p(id int,fullname text, address text);</td>
<td>SELECT * FROM foreign_emp_tbl;</td>
</tr>
<tr>
<td>DML on remote data</td>
<td>SELECT * FROM dblink('myconn', $$INSERT into employees VALUES (45,'Dan','South side 7432, NY')$$) AS t(id int, name text, address text);</td>
<td>INSERT into foreign_emp_tb VALUES (45,'Dan','South side 7432, NY'); (Regular DML)</td>
</tr>
<tr>
<td>Run DDL on remote objects</td>
<td>SELECT * FROM dblink('myconn', $$CREATE table my_remote_tbl $$)</td>
<td>Not Supported</td>
</tr>
<tr>
<td>(a int, b text) AS t(a text);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Migrating from: Oracle Inline Views

Overview
Inline views refer to a SELECT statement located in the FROM clause of secondary (or more) SELECT statement. Inline views can help make complex queries simpler by removing compound calculations or eliminating join operations while condensing several separate queries into a single simplified query.

Example
Inline View in the Oracle database:

```sql
SELECT A.LAST_NAME, A.SALARY, A.DEPARTMENT_ID, B.SAL_AVG
FROM EMPLOYEES A,
     (SELECT DEPARTMENT_ID, ROUND(AVG(SALARY)) AS SAL_AVG
      FROM EMPLOYEES
      GROUP BY DEPARTMENT_ID)
WHERE A.DEPARTMENT_ID = B.DEPARTMENT_ID;
```

The SQL statement marked in red represents the inline view code. In our example above, the query will return each employee matched to their salary and department id. In addition, the query will return the average salary for each department, using the inline view column - SAL_AVG.

AWS Migration to: PostgreSQL Inline Views

Overview
PostgreSQL semantics may refer to inline views as “Subselect” or as “Subquery”. In either case, the functionality is the same. Running the Oracle inline view example above, as is, will result in an error: “ERROR: subquery in FROM must have an alias”. This is because Oracle supports omitting aliases for the inner statement while in PostgreSQL the use of aliases is mandatory. “B” will be used as an alias in the example provided below.

Mandatory aliases are the only major difference when migrating Oracle inline views to PostgreSQL.

Example

```sql
SELECT A.LAST_NAME, A.SALARY, A.DEPARTMENT_ID, B.SAL_AVG
FROM EMPLOYEES A,
     (SELECT DEPARTMENT_ID, ROUND(AVG(SALARY)) AS SAL_AVG
      FROM EMPLOYEES
      GROUP BY DEPARTMENT_ID) B
WHERE A.DEPARTMENT_ID = B.DEPARTMENT_ID;
```
Migrating from: Oracle Database Hints

Overview
Oracle provides users with the ability to influence how the query optimizer behaves and the decisions made to generate query execution plans. Controlling the behavior of a database optimizer is done via the use of special “Database Hints”. These can be defined as a directive operation to the optimizer and as such, alter the decisions on how execution plans are generated.

The Oracle Database supports over 60 different database hints and each database hint can receive 0 or more arguments. Database hints are divided into different categories such as optimizer hints, join order hints, parallel execution hints, etc.

Note: Database hints are embedded directly into the SQL queries immediately following the SELECT keyword using the following format: /* <DB_HINT> */

Example

1. Force the Query Optimizer to use a specific index for data access using a database hint embedded into the query:

   SQL> SELECT /* INDEX(EMP, IDX_EMP_HIRE_DATE)*/ * FROM EMPLOYEES EMP
          WHERE HIRE_DATE >= '01-JAN-2010';

   Execution Plan
   ----------------------------------------------------------
   Plan hash value: 3035503638
   "--------------------------------------------------------------------------------------------
   | Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time
   |-------------------------------|----------------|-------|-------|------------|-------
   | 0   | SELECT STATEMENT            |               |   1   | 62    |   2 (0)   | 00:00:01
   | 1   |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |   1   | 62    |   2 (0)   | 00:00:01
   |*  2 |   INDEX RANGE SCAN          | IDX_HIRE_DATE |   1   |       |   1 (0)   | 00:00:01
   "--------------------------------------------------------------------------------------------

   Predicate Information (identified by operation id):
   "--------------------------------------------------------------------------------------------
   2 - access("HIRE_DATE">=TO_DATE(' 2010-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

For additional details:
http://docs.oracle.com/cd/E25178_01/server.1111/e16638/hintsref.htm#CHDIDIDI
https://docs.oracle.com/database/121/TGSQL/tgsql_influence.htm#TGSQ246
Migration to: PostgreSQL DB Query Planning

Overview

PostgreSQL does not support “database hints” to influence the behavior of the query planner and we cannot influence how execution plans are generated from within SQL queries. Although database hints are not directly supported, session parameters (also known as “Query Planning Parameters”) can influence the behavior of the query optimizer at a session level.

Example

1. Set the query planner to use indexes instead of full table scans (disable `SEQSCAN`):

   ```sql
   psql=> SET ENABLE_SEQSCAN = FALSE;  
   psql=>
   ```

2. Sets the query planner’s estimated “cost” of a disk page fetch that is part of a series of sequential fetches (`SEQ_PAGE_COST`) and set the planner's estimate of the cost of a non-sequentially-fetched disk page (`RANDOM_PAGE_COST`). Reducing the value of `RANDOM_PAGE_COST` relative to `SEQ_PAGE_COST` will cause the query planner to prefer index scans, while raising the value will make index scans more “expensive”.

   ```sql
   psql=> SET SEQ_PAGE_COST to 4;  
   psql=> SET RANDOM_PAGE_COST to 1;
   ```

3. Enables or disables the query planner's use of nested-loops when performing joins. While it is impossible to completely disable the usage of nested-loop joins, setting the `ENABLE_NESTLOOP` to an `OFF` value discourages the query planner from choosing nested-loop joins compared to alternative join methods.

   ```sql
   psql=> SET ENABLE_NESTLOOP to FALSE;
   ```

For additional details:

https://www.postgresql.org/docs/9.6/static/runtime-config-query.html
Overview
RMAN, or Oracle Recovery Manager, is Oracle’s primary backup and recovery tool. RMAN provides its own scripting syntax, which can be used to take full or incremental backups of your Oracle database.

1. **Full RMAN backup** – you can take a full backup of an entire database or individual Oracle data files. For example, a level 0 full backup.
2. **Differential incremental RMAN backup** – performs a backup of all database blocks that have changed from the previous level 0 or 1 backup.
3. **Cumulative incremental RMAN backup** – perform a backup all of the blocks that have changed from the previous level 0 backup.

Notes
- RMAN supports online backups of your Oracle database if your database has been configured to run in Archived Log Mode.
- RMAN is used to take backups of the following files:
  - Database data files.
  - Database control file.
  - Database parameter file.
  - Database Archived Redo Logs.

Examples

1. Connect using the RMAN CLI to the Oracle database you wish to back-up:
   
   ```
   export ORACLE_SID=ORCL
   rman target=/
   ```

2. Perform a full backup of the database and the database archived redo logs:
   
   ```
   BACKUP DATABASE PLUS ARCHIVELOG;
   ```

3. Perform an incremental level 0 or level 1 backup of the database:
   
   ```
   BACKUP INCREMENTAL LEVEL 0 DATABASE;
   BACKUP INCREMENTAL LEVEL 1 DATABASE;
   ```
4. Restore the database using RMAN:

```sql
RUN {
    SHUTDOWN IMMEDIATE;
    STARTUP MOUNT;
    RESTORE DATABASE;
    RECOVER DATABASE;
    ALTER DATABASE OPEN;
}
```

5. Restore a specific pluggable database (Oracle 12c):

```sql
RUN {
    ALTER PLUGGABLE DATABASE pdbA, pdbB CLOSE;
    RESTORE PLUGGABLE DATABASE pdbA, pdbB;
    RECOVER PLUGGABLE DATABASE pdbA, pdbB;
    ALTER PLUGGABLE DATABASE pdbA, pdbB OPEN;
}
```

6. Restore a database to a specific point in time:

```sql
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;
}
```

7. Report (list) on all current database backups created via RMAN:

```sql
LIST BACKUP OF DATABASE;
```

For additional details:
https://docs.oracle.com/database/121/BRADV/toc.htm
Migration to: Amazon Aurora Snapshots

Overview
The primary backup mechanism for Amazon Aurora is using snapshots. Taking a snapshot is an extremely fast and non-intrusive operation for your database. Both taking snapshots and restoring your database from a snapshot can be done using the Amazon RDS Management Console or using the AWS CLI. Unlike RMAN, there is no need for incremental backups. You can choose to restore your database to the exact time when a snapshot was taken, or to any other point in time.

- **Automated backups.** Always enabled on Amazon Aurora. Backups do not impact database performance.
- **Manual backups.** You can create a snapshot at any given time. There is no performance impact when taking snapshots of your Aurora database. Restoring data from snapshots requires you to create a new instance. Up to 100 manual snapshots are supported per database.

Example:

Enable Aurora automatic backups and configure the backup retention window as part of the database creation process. Doing this is equivalent to setting the Oracle RMAN backup retention policy (using the “configure retention policy to recovery window of X days” command).

1. Go to the Amazon RDS page in your AWS Management Console:

2. Click *Instances*
3. Select **Launch DB Instance**.
4. Select the **Aurora PostgreSQL-compatible** database engine.

5. Configure your database settings and parameters.

6. On the next page, you can configure a backup retention policy for your Aurora database, defined as the number of days for Amazon RDS to automatically retain your snapshots:
Example:

Perform a manual snapshot backup of your Aurora database, equivalent to creating a full Oracle RMAN backup ("BACKUP DATABASE PLUS ARCHIVELOG").

1. Go to the Amazon RDS page in your AWS Management Console:

2. Click Instances.
3. Select your Aurora PostgreSQL instance, click **Instance actions** and select **Take snapshot**:

![Instance actions and Take snapshot](image)

**Example**

Restore an Aurora database from a snapshot. Similar to Oracle RMAN “RESTORE DATABASE” and “RECOVER DATABASE” commands. However, note that instead of running in-place, restoring an Amazon Aurora database will create a new cluster.

1. Navigate to the Amazon RDS page in your AWS Management Console:

![AWS services](image)

2. Click on the **Snapshots** link on the left-hand menu to see the list of snapshots you have available across your database instances.
3. Select Snapshots.

4. Select the snapshot to restore.

5. Click Snapshot Actions on the context menu and select Restore snapshot.

Note: This creates a new instance.

6. You will be presented with a wizard for creating your new Aurora database instance from the selected snapshot. Fill the required configuration options and click Restore DB Instance.
Example

Restore an Aurora PostgreSQL database backup to a specific previous point in time, similar to running an Oracle RMAN "SET UNTIL TIME "TO_DATE('XXX')" command, before running RMAN RESTORE DATABASE and RECOVER DATABASE.

1. Navigate to the Amazon RDS page in your AWS Management Console.
2. Click **Instances**.

3. Select your Aurora instance and click **Instance Actions**.

4. Select **Restore to Point in Time** on the context menu.

5. This process will launch a new instance. Select the date and time to which you want to restore your database. The selected date and time must be within the configured backup retention for this instance.
Example

Modify your Aurora backup retention policy after a database was already created. You need to configure how long your Aurora database backups should be stored. When restoring an Aurora database to a previous point in time, the selected time must be within the configured backup retention window.

1. Navigate to the Amazon RDS page in your AWS Management Console.

2. Click Instances.
3. Select your Aurora instance, click **Instance Actions**.

4. Select **Modify** from the menu.

5. Configure the desired backup retention policy (maximum retention allowed is up to 35 days).
Using the AWS CLI for Amazon Aurora backup and restore operations

In addition to using the AWS web console user-interface to backup and restore your Aurora instance to a previous point in time or using a specific snapshot, you can also use the AWS CLI to perform the same actions. This is especially useful in case you need to convert your existing automated Oracle RMAN scripts to an Amazon Aurora environment.

Examples

1. Use `describe-db-cluster-snapshots` to view all current Aurora PostgreSQL snapshots.
2. Use `create-db-cluster-snapshot` to create a snapshot (“Restore Point”).
3. Use `restore-db-cluster-from-snapshot` to restore a new cluster from an existing database snapshot.
4. Use `create-db-instance` to add new instances to the restored cluster.

```
aws rds describe-db-cluster-snapshots
aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name
aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql
aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier NewCluster --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large
```

5. Use `restore-db-instance-to-point-in-time` to perform point-in-time recovery.

```
aws rds restore-db-cluster-to-point-in-time --db-cluster-identifier clustername-restore --source-db-cluster-identifier clustername --restore-to-time 2017-09-19T23:45:00.000Z
aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large
```
## Oracle RMAN vs. Aurora snapshot backups

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>Amazon Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheduled backups</strong></td>
<td>Create <code>DBMS_SCHEDULER</code> job that will execute your RMAN script on a scheduled basis.</td>
<td><strong>Automatic</strong></td>
</tr>
<tr>
<td><strong>Manual full database backups</strong></td>
<td><code>BACKUP DATABASE PLUS ARCHIVELOG;</code></td>
<td>Use Amazon RDS dashboard or the AWS CLI command to take a snapshot on the cluster:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name</code></td>
</tr>
<tr>
<td><strong>Manual full database backups</strong></td>
<td></td>
<td>Create new cluster from a cluster snapshot:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add a new instance to the new/restored cluster:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large</code></td>
</tr>
<tr>
<td><strong>Incremental differential</strong></td>
<td><code>BACKUP INCREMENTAL LEVEL 0 DATABASE;</code></td>
<td><strong>N/A</strong></td>
</tr>
<tr>
<td></td>
<td><code>BACKUP INCREMENTAL LEVEL 1 DATABASE;</code></td>
<td></td>
</tr>
<tr>
<td><strong>Incremental cumulative</strong></td>
<td><code>BACKUP INCREMENTAL LEVEL 0 CUMULATIVE DATABASE;</code></td>
<td><strong>N/A</strong></td>
</tr>
<tr>
<td></td>
<td><code>BACKUP INCREMENTAL LEVEL 1 CUMULATIVE DATABASE;</code></td>
<td></td>
</tr>
<tr>
<td><strong>Restore database to a specific point-in-time</strong></td>
<td><code>RUN { SHUTDOWN IMMEDIATE; STARTUP MOUNT; RESTORE DATABASE; RECOVER DATABASE; ALTER DATABASE OPEN; }</code></td>
<td>Create new cluster from a cluster snapshot by given custom time to restore:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>aws rds restore-db-cluster-to-point-in-time --db-cluster-identifier clustername-restore --source-db-cluster-identifier clustername-restore --restore-to-time 2017-09-19T23:45:00.000Z</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add a new instance to the new/restored cluster:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Oracle</td>
<td>Amazon Aurora</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Description</td>
<td>Oracle</td>
<td>Amazon Aurora</td>
</tr>
<tr>
<td>Backup database</td>
<td>BACKUP ARCHIVELOG ALL;</td>
<td>N/A</td>
</tr>
<tr>
<td>Archive logs</td>
<td>CROSSCHECK BACKUP; DELETE EXPIRED BACKUP;</td>
<td>N/A</td>
</tr>
<tr>
<td>Delete old database</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Archive logs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restore a single Pluggable database (12c)</td>
<td>RUN { ALTER PLUGGABLE DATABASE pdb1, pdb2 CLOSE; RESTORE PLUGGABLE DATABASE pdb1, pdb2; RECOVER PLUGGABLE DATABASE pdb1, pdb2; ALTER PLUGGABLE DATABASE pdb1, pdb2 OPEN; }</td>
<td>Create new cluster from a cluster snapshot: aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql Add a new instance to the new/restored cluster: aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large Use pg_dump and pg_restore to copy the database to the original instance: pgdump -F c -h hostname.rds.amazonaws.com -U username -d hr -p 5432 &gt; c:\Export\hr.dmp pg_restore -h restoredhostname.rds.amazonaws.com -U hr -d hr_restore -p 5432 c:\Export\hr.dmp Optionally, replace with the old database using ALTER DATABASE RENAME</td>
</tr>
</tbody>
</table>
Overview

Oracle 12c introduces a new multitenant architecture which provides the ability to create additional independent “pluggable” databases under a single Oracle instance. Prior to Oracle 12c, a single Oracle database instance only supported running single Oracle database, as shown in the picture below.

Oracle 12c introduces a new multi-container database, or CDB, that supports one or more “Pluggable Databases”, or PDBs. The CDB can be thought of as a single “superset” database with multiple pluggable database. The relationship between The Oracle instance and databases is now 1:N.
Advantages of the Oracle 12c multitenant architecture

- PDBs can be used to isolate applications from one another.
- PDBs can be used as portable collection of schemas.
- PDBs can cloned and transported to different CDBs/Oracle instances.
- Management of many databases (individual PDBs) as a whole.
- Separate security, users, permissions, and resource management per PDB provides greater application isolation.
- Enables a consolidated database model of many individual applications sharing a single Oracle server.
- Provide an easier way to patch and upgrade individual clients and/or applications, using PDBs.
- Backups are supported at both a multitenant container-level as well as at an individual PDB-level (both for physical and logical backups).

The Oracle multitenant architecture

- A multitenant Container Database (CDB) can support one or more “pluggable databases” (PDBs).
- Each pluggable database contains its own copy of SYSTEM and application tablespaces.
- The PDBs will share the Oracle Instance memory and background processes. The use of PDBs enables consolidation of many databases and applications into individual containers under the same Oracle instance.
- A single “Root Container” (or CDB$ROOT) exists in a CDB and contains the Oracle Instance Redo Logs, undo tablespace (unless Oracle 12.2 local undo mode is enabled) and control files.
- A single Seed PDB exists in a CDB and used as a template for creating new PDBs.
CDB & PDB Semantics

- **CDB (Container Database)**
  - A “Super” database that contains the Root Container – `cdb$root` (one per instance) and one or more Pluggable Databases (with user-provided naming).
  - Created as part of the Oracle 12c software installation.
  - Contains the Oracle control files, its own set of system tablespaces, the instance undo tablespaces (unless Oracle 12.2 local undo mode is enabled), and the instance redo logs.
  - Holds the data dictionary for the root container and for all of the PDBs.

- **PDB (Pluggable Database)**
  - Independent database that exists under a CDB. Also known as a “container”.
  - Used to store application-specific data.
  - Can be created from a the `pdb$seed` (template database) or as a clone of an existing PDB
  - Stores metadata information specific to its own objects (data-dictionary)
  - Has its own set of application and system data files and tablespaces along with temporary files to manage objects.

Examples

1. List existing PDBs created in an Oracle CDB instance:

   ```sql
   SQL> SHOW PDBS;
   CON_ID CON_NAME              OPEN MODE RESTRICTED
   ---------- ------------------------------ ---------- ----------
   2 PDB$SEED                       READ ONLY NO
   3 PDB1                           READ WRITE NO
   ```

2. Provisioning of a new PDB from the template `seed$pdb`:

   ```sql
   SQL> CREATE PLUGGABLE DATABASE PDB2 admin USER ora_admin IDENTIFIED BY ora_admin FILE_NAME_CONVERT=('/pdb$seed','/pdb2');
   ```

3. Alter a specific PDB to READ/WRITE and verify:

   ```sql
   SQL> ALTER PLUGGABLE DATABASE PDB2 OPEN READ WRITE;
   SQL> SHOW PDBS;
   CON_ID CON_NAME              OPEN MODE RESTRICTED
   ---------- ------------------------------ ---------- ----------
   2 PDB$SEED                       READ ONLY NO
   3 PDB1                           READ WRITE NO
   4 PDB2                           READ WRITE NO
   ```
4. Clone a PDB from an existing PDB:

```
SQL> CREATE PLUGGABLE DATABASE PDB3 FROM PDB2 FILE_NAME_CONVERT=('/pdb2/','/pdb3/');
SQL> SHOW PDBS;
CON_ID CON_NAME                   OPEN MODE RESTRICTED
---------- ------------------------- ----------- ----------
   2 PDB$SEED                      READ ONLY  NO
   3   PDB1                       READ WRITE NO
   4    PDB2                      READ WRITE NO
   5     PDB3                     MOUNTED
```

For additional details:
http://docs.oracle.com/database/122/CNCPT/overview-of-the-multitenant-architecture.htm#CNCPT89250
http://docs.oracle.com/database/122/ADMIN/managing-a-multitenant-environment.htm#ADMIN13506
Migration to: PostgreSQL Databases

Overview
Amazon Aurora PostgreSQL offers a different and simplified architecture to manage and create a multitenant database environment. Using Aurora PostgreSQL, it is possible to provide levels of functionality similar (but not identical) to those offered by Oracle PDBs by creating multiple databases under the same Aurora PostgreSQL cluster and/or using separate Aurora clusters, when total isolation of workloads is required.

- Multiple PostgreSQL databases can be created under a single Amazon Aurora PostgreSQL Cluster.

- Each Amazon Aurora cluster contains a primary instance that can accept both reads and writes for all cluster databases.
- Up to 15 read-only nodes can be created which provide both scale-out functionality for application reads as well as for high availability proposes.

In theory, an Oracle CDB/Instance can be considered as the high-level equivalent to an Amazon Aurora cluster, and an Oracle Pluggable Database (PDB) would be equivalent to PostgreSQL database created inside the Amazon Aurora cluster. Not all features are comparable between Oracle 12c PDBs and Amazon Aurora.
Examples

1. Create a new database in PostgreSQL using the `CREATE DATABASE` statement:

   ```sql
   psql=> CREATE DATABASE pg_db1;
   CREATE DATABASE
   psql=> CREATE DATABASE pg_db2;
   CREATE DATABASE
   psql=> CREATE DATABASE pg_db3;
   CREATE DATABASE
   ```

2. List all databases created under an Amazon Aurora PostgreSQL cluster:

   ```sql
   psql=> \l
   Name     |    Owner     | Encoding |   Collate   |    Ctype    |
   --------------------------
   admindb     | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   pg_db1      | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   pg_db2      | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   pg_db3      | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   postgres    | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   rdsadmin    | rdsadmin     | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   template0   | rdsadmin     | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   template1   | rds_pg_admin | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
   ```

Independent database backups in Amazon Aurora PostgreSQL

Oracle 12c provides the ability to perform both logical backups (via DataPump) and physical backups (via RMAN) at both CDB and PDB levels. Similarly, Amazon Aurora PostgreSQL provides the ability to perform logical backups on all or a specific database(s) using `pg_dump`. However, for physical backups when using snapshots, the entire cluster and all databases are included in the snapshot, backing up a specific database within the cluster is not supported.

This is usually not a concern as volume snapshots are extremely fast operations that occur at the storage-infrastructure layer and thus incur minimal overhead and operate at extremely fast speeds. However, you the process of restoring a single PostgreSQL database from an Aurora snapshot requires additional steps, such as exporting the specific database after a snapshot restore and importing it back to the original Aurora cluster.
Examples

Physical backup: take an Amazon Aurora PostgreSQL snapshot.

1. On the AWS Management Console, navigate to RDS > Instances > Instance Actions and choose “Take Snapshot”.

Logical backup: Use PostgreSQL pg_dump (installed on your client machine) to create a logical backup for a specific PostgreSQL database:

```
$ pg_dump -h hostname.rds.amazonaws.com -U username -d db_name -f dump_file_name.sql
```

For additional information on PostgreSQL databases:
https://www.postgresql.org/docs/current/static/sql-createdatabase.html
Migrating from: Oracle Tablespaces & Data Files

Overview
The storage structure of an Oracle database contains both physical and logical elements.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablespaces</td>
<td>Each Oracle database contains one or more <em>tablespaces</em>, which are logical storage groups, that are used as “containers” when creating new tables and indexes.</td>
</tr>
<tr>
<td>Data files</td>
<td>Each tablespace is made up of one or more <em>data files</em>, which are the physical elements that make up an Oracle database tablespace. Datafiles can be located on the local file system, raw partitions, managed by Oracle ASM or files located on network file system.</td>
</tr>
</tbody>
</table>

Storage Hierarchy
- **Database**: each Oracle database is composed from one or more tablespaces.
- **Tablespace**: each Oracle tablespace is composed from one or more datafiles. Tablespaces are logical entities that have no physical manifestation on the file system.
- **Data files**: physical files, located on a file-system. Each Oracle tablespace is made from one or more data files.
  - **Segments**: each represents a single database object that consumes storage, such as tables, indexes, undo segments etc.
- **Extent**: each segment is made from one or more extents. Oracle uses extents as a form of allocating contiguous sets of database blocks on disk.
- **Block**: the smallest unit of I/O that can be used by a database for reads and writes. In case of blocks that store table data, each block can store one or more table rows.

Types of Oracle Database Tablespace
- **Permanent Tablespaces**: designated to store persistent schema objects for your applications.
- **Undo Tablespace**: a special type of system permanent tablespace that is used by Oracle to manage UNDO data when running the database in automatic undo management mode.
- **Temporary Tablespace**: contains schema objects that are valid for the duration of a session. It is also used for spilling sorts that cannot fit into memory.

Tablespace Privileges
In order to create a tablespace:

- The database user must have the *CREATE TABLESPACE* system privilege.
- Create a database and the database must be in open mode.
1. Create the **USERS** tablespace comprised of a single data file.

```sql
SQL> CREATE TABLESPACE USERS
    2   DATAFILE '/u01/app/oracle/oradata/orcl/users01.dbf' SIZE 5242880
    3   AUTOEXTEND ON NEXT 1310720 MAXSIZE 32767M
    4   LOGGING ONLINE PERMANENT BLOCKSIZE 8192
    5   EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT
    6   NOCOMPRESS SEGMENT SPACE MANAGEMENT AUTO;
```

Drop a tablespace:

```sql
SQL> DROP TABLESPACE USERS;
SQL> DROP TABLESPACE USERS INCLUDING CONTENTS AND DATAFILES;
```

*For additional details:*
https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_7003.htm#SQLRF01403
https://docs.oracle.com/database/121/SQLRF/statements_7003.htm#SQLRF01403
https://docs.oracle.com/cd/E11882_01/server.112/e41084/clauses004.htm#SQLRF01602
https://docs.oracle.com/database/121/SQLRF/clauses004.htm#SQLRF01602
https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_9004.htm#SQLRF01807
https://docs.oracle.com/database/121/SQLRF/statements_9004.htm#SQLRF01807

---

**Migration to:** PostgreSQL Tablespaces & Data Files

[Back to TOC]

### Overview

The logical storage structure in PostgreSQL shares similar concepts as Oracle, utilizing tablespaces for storing database objects. Tablespaces in PostgreSQL are made from datafiles and are used to store different databases and database object.

- **Tablespace** - the directory where datafiles are stored.
- **Data files** - file-system files that are placed inside a tablespace (directory) and are used to store database objects such as tables or indexes. Created automatically by PostgreSQL. Similar to how Oracle-Managed-Files (OMF) behave.

**Notes:**

- Unlike Oracle, a PostgreSQL tablespace does not have user-configured segmentation into multiple and separate data files. When you create the tablespace, PostgreSQL automatically creates the necessary files to store the data.
- Each table and index are stored in a separate O/S file, named after the table or index's *filenode* number.
Tablespaces in Amazon Aurora PostgreSQL

After an Amazon Aurora PostgreSQL cluster is created, two system tablespaces are automatically provisioned and cannot be modified or dropped:

- **pg_global tablespace**
  - Used for the shared system catalogs.
  - Stores objects that are visible to all Cluster databases.

- **pg_default tablespace**
  - The default tablespace of the `template1` and `template0` databases.
  - Serves as the default tablespace for other databases, by default, unless a different tablespace was explicitly specified during database creation.

One of the main advantages when using Amazon Aurora PostgreSQL is the absence of complexity for storage management. Therefore, creating tablespaces in Aurora PostgreSQL is simplified and has several advantages over a “vanilla” PostgreSQL database deployment:

- When creating tablespaces, the superuser can specify an OS path (location) that does not currently exist. The directory will be implicitly created.

- A user-specified tablespace directory will be created under an embedded Amazon RDS/Aurora path. For example, every path specified in the **LOCATION** clause when creating a new tablespace will be created under the Amazon RDS path of: `/rdsdbdata/tablespaces/`

- Amazon Aurora PostgreSQL leverages a unique self-managed shared storage architecture. The DBA does not need to micro-manage most storage aspects of the database.

**Examples**

1. Creating a Tablespace via Amazon Aurora PostgreSQL and view its associated directory:

   ```sql
   demo=> CREATE TABLESPACE TBS_01 LOCATION '/app_data/tbs_01';
   CREATE TABLESPACE
   
   demo=> \du
    Name    |  Owner   |                   Location
    ------------+----------+----------------------------------------------
     pg_default | rdsadmin |                                
     pg_global  | rdsadmin |                                
     tbs_01     | rdsadmin | /rdsdbdata/tablespaces/app_data/tbs_01
   ```

   * Notice that the newly specified path was created under the embedded base path for Amazon Aurora: `/rdsdbdata/tablespaces/`
2. View current tablespaces and associated directories:

```
select spcname, pg_tablespace_location(oid) from pg_tablespace;
```

3. Drop the PostgreSQL TBS_01 tablespace:

```
demo=> DROP TABLESPACE TBS_01;
DROP TABLESPACE
```

4. Alter a tablespace:

```
demo=> ALTER TABLESPACE TBS_01 RENAME TO IDX_TBS_01;
ALTER TABLESPACE

demo=> ALTER TABLESPACE TBS_01 OWNER TO USER1;
ALTER TABLESPACE
```

5. Assign a database with a specific tablespace:

```
demo=> CREATE DATABASE DB1 TABLESPACE TBS_01;
CREATE DATABASE

demo=> SELECT DATNAME, PG_TABLESPACE_LOCATION(DATTABLESPACE) FROM PG_DATABASE
WHERE DATNAME='db1';

<table>
<thead>
<tr>
<th>datname</th>
<th>pg_tablespace_location</th>
</tr>
</thead>
<tbody>
<tr>
<td>db1</td>
<td>/rdsdbdata/tables/spaces/app_data/tbs_01</td>
</tr>
</tbody>
</table>
```

6. Assign a table with a specific tablespace:

```
demo=> CREATE TABLE TBL(
    COL1 NUMERIC, COL2 VARCHAR(10))
    TABLESPACE TBS_01;
CREATE TABLE

demo=> SELECT SCHEMANAME, TABLENAME, TABLESPACE FROM PG_TABLES
WHERE TABLENAME='tbl';

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>tablespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>tbl</td>
<td>tbs_01</td>
</tr>
</tbody>
</table>
7. Assign an index with a specific tablespace:

```sql
demo=> CREATE INDEX IDX_TBL ON TBL(COL1) TABLESPACE TBS_01;
CREATE INDEX

demo=> SELECT SCHEMANAME, TABLENAME, INDEXNAME, TABLESPACE FROM PG_INDEXES
    WHERE INDEXNAME='idx_tbl';

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>indexname</th>
<th>tablespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>tbl</td>
<td>idx_tbl</td>
<td>tbs_01</td>
</tr>
</tbody>
</table>

demo=>
```

8. Alter a table to use a different tablespace:

```sql
demo=> ALTER TABLE TBL SET TABLESPACE TBS_02;
ALTER TABLE
```

**Tablespace Exceptions**

- `CREATE TABLESPACE` cannot be executed inside a transaction block.
- A tablespace cannot be dropped until all objects in all databases using the tablespace have been removed/moved.

**Privileges**

- The creation of a tablespace in the PostgreSQL database must be performed by a database superuser.
- Once a tablespace has been created, it can be used from any database, provided that the requesting user has sufficient privileges.

**Tablespace Parameters**

The `default_tablespace` parameter controls the system default location for newly created database objects. By default, this parameter is set to an empty value and any newly created database object will be stored in the default tablespace (`pg_default`).

The `default_tablespace` parameter can be altered by using the cluster parameter group.
To verify and to set the `default_tablespace` variable:

```sql
demo=> SHOW DEFAULT_TABLESPACE; -- No value
default_tablespace
------------------
demo=> SET DEFAULT_TABLESPACE=TBS_01;
demo=> SHOW DEFAULT_TABLESPACE;
default_tablespace
------------------
tbs_01
```

### Oracle vs. PostgreSQL tablespaces

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle</th>
<th>Aurora PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tablespace</strong></td>
<td>Exists as a logical object and made from one or more user-specified or system-generated data files.</td>
<td>Logical object that is tied to a specific directory on the disk where datafiles will be created.</td>
</tr>
</tbody>
</table>
| **Data file**                        | 1. Can be explicitly created and resized by the user. Oracle-Managed-Files (OMF) allows for automatically created data files.  
                                           2. Each data file can contain one or more tables and/or indexes. | Behavior is more akin to Oracle Managed Files (OMF).  
                                           1. Created automatically in the directory assigned to the tablespace.  
                                           2. Single data file stores information for a specific table or index. Multiple data files can exist for a table or index.  
                                           Additional files are created:  
                                           1. Freespace map file  
                                              Exists in addition to the datafiles themselves.  
                                              The free space map is stored as a file named with the filenode number plus the _fsm suffix.  
                                           2. Visibility Map File  
                                              Stored with the _vm suffix and used to track which pages are known to have no dead tuples. |
<p>| Creates a new tablespace with system-managed datafiles | CREATE TABLESPACE sales_tbs DATAFILE SIZE 400M; | create tablespace sales_tbs LOCATION '/postgresql/data'; |
| Create a new tablespace with user-managed datafiles | CREATE TABLESPACE sales_tbs DATAFILE '/oradata/sales01.dbf' SIZE 1M AUTOEXTEND ON NEXT 1M; | N/A |
| Alter the size of a datafile          | ALTER DATABASE DATAFILE '/oradata/sales01.dbf' RESIZE 100M; | N/A |</p>
<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle</th>
<th>Aurora PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a datafile to an existing tablespace</td>
<td>ALTER TABLESPACE sales_tbs ADD DATAFILE '/oradata/sales02.dbf' SIZE 10M;</td>
<td>N/A</td>
</tr>
<tr>
<td>Per-database tablespace</td>
<td>Supported as part of the Oracle 12c Multi-Tenant architecture. Different dedicated tablespaces can be created for different pluggable databases and set as the default tablespace for a PDB: ALTER SESSION SET CONTAINER = 'sales'; CREATE TABLESPACE sales_tbs DATAFILE '/oradata/sales01.dbf' SIZE 1M AUTOEXTEND ON NEXT 1M; ALTER DATABASE sales TABLESPACE sales_tds;</td>
<td>Tablespaces are shared across all databases but a default tablespace can be created and configured for the database: create tablespace sales_tbs LOCATION '/postgresql/data'; CREATE DATABASE sales OWNER sales_app TABLESPACE sales_tbs;</td>
</tr>
<tr>
<td>Metadata tables</td>
<td>Data Dictionary tables are stored in the SYSTEM tablespace</td>
<td>System Catalog tables are stored in the pg_global tablespace</td>
</tr>
<tr>
<td>Tablespace data encryption</td>
<td>Supported 1. Using transparent data encryption. 2. Encryption and decryption are handled seamlessly so the user does not have to modify the application to access the data.</td>
<td>Supported 1. Encrypt using keys managed through KMS. 2. Encryption and decryption are handled seamlessly so the user does not have to modify the application to access the data 3. Enable encryption while deploying a new cluster via the AWS Management Console or API actions. For additional details: <a href="http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Encryption.html">http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Encryption.html</a></td>
</tr>
</tbody>
</table>

For additional details:
https://www.postgresql.org/docs/9.6/static/manage-ag-tablespaces.html
https://www.postgresql.org/docs/9.6/static/sql-createtablespace.html
https://www.postgresql.org/docs/9.6/static/storage-fsm.html
https://www.postgresql.org/docs/9.6/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE
https://www.postgresql.org/docs/9.6/static/sql-droptablespace.html
https://www.postgresql.org/docs/current/static/sql-altertablespace.html
Overview
Oracle Data Pump is a utility for exporting/importing data from/to an Oracle database. Data Pump can be used to copy an entire database, an entire schema(s) or specific objects in a schema. Oracle Data Pump is commonly used as a part of backup strategy for restoring individual database objects (specific records, tables, views, stored procedures, etc.) as opposed to snapshots or Oracle RMAN, which provides backup and recovery capabilities at the database-level. By default, and without using the sqlfile parameter during export, the “dumpfile” generated by Oracle Data Pump is a binary file and cannot be opened using a text editor.

Oracle Data Pump Supports:

1. Export of data from an Oracle database:
The Data Pump EXPDP command creates a binary dump file containing the exported database objects. Objects can be exported with data or, as an alternative, containing metadata only. Support for cross-object consistent exports can be accomplished by requesting the export to be done according to a specific timestamp or Oracle SCN.

2. Import data to an Oracle database:
The Data Pump IMPDP command will import objects and data from specific dump file created with the EXPDP Data Pump command. The IMPDP command can filter on import (only import certain objects) and remap object and schema names during import.

Notes:
• The term “Logical backup” refers to a dump file created by Oracle Data Pump.
• Both EXPDP and IMPDP can only read/write “dumpfiles” from filesystem paths that were pre-configured in the Oracle database as “directories”. During export/import, users will need to specify the logical “directory” name where the dumpfile should be created, and not the actual filesystem path.

Examples
Export - using EXPDP to export the Oracle HR schema:

```bash
$ expdp system/**** directory=expdp_dir schemas=hr dumpfile=hr.dmp logfile=hr.log
```

* The command contains the credentials to run Data Pump, logical Oracle directory name for the dump file location (which maps in the database to a physical filesystem location), schema name to export and dump file and log files names.
Import - Using the IMPDP to import the HR schema and rename to HR_COPY:

```
$ impdp system/**** directory=expdp_dir schemas=hr dumpfile=hr.dmp logfile=hr.log REMAP_SCHEMA=hr:hr_copy
```

* The command contains the database credentials to run Data Pump, logical Oracle directory for where the export dumpfile is located, dump file name, schema to export, name for the dump file and log file name and the REMAP_SCHEMA parameter.

For additional details:
https://docs.oracle.com/cloud/latest/db112/SUTIL/part_dp.htm

AWS Migration to: PostgreSQL pg_dump & pg_restore

[Back to TOC]

Overview
PostgreSQL provides native utilities - pg_dump and pg_restore can be used to perform logical database exports and imports with a degree of comparable functionality to the Oracle Data Pump utility. Such as for moving data between two databases and creating logical database backups.

- pg_dump equivalent to Oracle expdp
- pg_restore equivalent to Oracle impdp

Amazon Aurora PostgreSQL supports data export and import using both pg_dump and pg_restore, but the binaries for both utilities will need to be placed on your local workstation or on an Amazon EC2 server as part of the PostgreSQL client binaries.

PostgreSQL dump files created using pg_dump can be copied, after export, to an Amazon S3 bucket as cloud backup storage or for maintaining the desired backup retention policy. Later, when dump files are needed for database restore, the dump files should be copied back to the desktop/server that has a PostgreSQL client (such as your workstation or an Amazon EC2 server) to issue the pg_restore command.

Notes:
- pg_dump will create consistent backups even if the database is being used concurrently.
- pg_dump does not block other users accessing the database (readers or writers).
- pg_dump only exports a single database, in order to backup global objects that are common to all databases in a cluster, such as roles and tablespaces, use pg_dumpall.
- Unlike Data Pump, PostgreSQL dump files are plain-text files.
Examples

1. Export data using `pg_dump`:

   Use a workstation or server with the PostgreSQL client installed in order to connect to the Aurora PostgreSQL instance in AWS; providing the hostname (-h), database user name (-U) and database name (-d) while issuing the `pg_dump` command:

   ```
   $ pg_dump -h hostname.rds.amazonaws.com -U username -d db_name -f dump_file_name.sql
   ```

   Note:
The output file, `dump_file_name.sql`, will be stored on the server where the `pg_dump` command executed. You can later copy the outfile file to an S3 Bucket, if needed.

2. Run `pg_dump` and copy the backup file to an Amazon S3 bucket using pipe and the AWS CLI:

   ```
   $ pg_dump -h hostname.rds.amazonaws.com -U username -d db_name -f dump_file_name.sql | aws s3 cp - s3://pg-backup/pg_bck-$date
   ```

3. Restore data - `pg_restore`:

   Use a workstation or server with the PostgreSQL client installed to connect to the Aurora PostgreSQL instance providing the hostname (-h), database user name (-U), database name (-d) and the dump file to restore from while issuing the `pg_restore` command:

   ```
   $ pg_restore -h hostname.rds.amazonaws.com -U username -d db_name_restore dump_file_name.sql
   ```

4. Copy the output file from the local server to an Amazon S3 Bucket using the AWS CLI:

   Upload the dump file to S3 bucket:

   ```
   $ aws s3 cp /usr/Exports/hr.dmp s3://my-bucket/backup-$date
   ```

   * Note that the `-$date` format will only work on Linux servers.

   Download the output file from S3 bucket:

   ```
   $ aws s3 cp s3://my-bucket/backup-2017-09-10-01-10-10 /usr/Exports/hr.dmp
   ```

   Note:

   You can create a copy of an existing database without having to use `pg_dump` or `pg_restore`. Instead, use the `template` keyword to signify the database used as the source:

   ```
   psql> CREATE DATABASE mydb_copy TEMPLATE mydb;
   ```
## Oracle Data Pump vs. PostgreSQL pg_dump and pg_restore

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle data pump</th>
<th>PostgreSQL SQL Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export data to a local file</td>
<td><code>expdp system/**** schemas=hr</code>&lt;br&gt;dumpsfile=hr.dmp&lt;br&gt;logfile=hr.log</td>
<td><code>pgdump -F c -h</code>&lt;br&gt;hostname.rds.amazonaws.com -U username -d hr -p 5432 &gt; c:\Export\hr.dmp</td>
</tr>
<tr>
<td>Export data to a remote file</td>
<td>• Create Oracle directory on remote storage mount or NFS directory called EXP_DIR&lt;br&gt;• Use export command:&lt;br&gt;<code>expdp system/**** schemas=hr</code>&lt;br&gt;directory=EXP_DIR&lt;br&gt;dumpsfile=hr.dmp&lt;br&gt;logfile=hr.log</td>
<td><code>Export-</code>&lt;br&gt;<code>pgdump -F c -h</code>&lt;br&gt;hostname.rds.amazonaws.com -U username -d hr -p 5432 &gt; c:\Export\hr.dmp&lt;br&gt;<code>Upload to S3-</code>&lt;br&gt;<code>aws s3 cp c:\Export\hr.dmp s3://my-bucket/backup-$(date &quot;+%Y-%m-%d-%H-%M-%S&quot;)</code></td>
</tr>
<tr>
<td>Import data to a new database with a new name</td>
<td><code>impdp system/**** schemas=hr</code>&lt;br&gt;dumpsfile=hr.dmp&lt;br&gt;logfile=hr.log&lt;br&gt;REMAP_SCHEMA=hr:hr_copy&lt;br&gt;TRANSFORM=OID:N</td>
<td><code>pg_restore -h</code>&lt;br&gt;hostname.rds.amazonaws.com -U hr -d hr_restore -p 5432&lt;br&gt;c:\Expires\hr.dmp</td>
</tr>
</tbody>
</table>

For additional details:<br><br>https://www.postgresql.org/docs/current/static/backup-dump.html<br>https://www.postgresql.org/docs/9.6/static/app-pgrestore.html
Migrating from: Oracle Resource Manager

Overview
Oracle’s Resource Manager enables enhanced management of multiple concurrent workloads running under a single Oracle database. Using Oracle Resource Manager, you can partition server resources for different workloads. Resource Manager helps with sharing server and database resources without causing excessive resource contention and helps to eliminate scenarios involving inappropriate allocation of resources across different database sessions.

Oracle Resource Manager enables you to:

• Guarantee a minimum amount of CPU cycles for certain sessions regardless of other running operations.
• Distribute available CPU by allocating percentages of CPU time to different session groups.
• Limit the degree of parallelism of any operation performed by members of a user group.
• Manage the order of parallel statements in the parallel statement queue.
• Limit the number of parallel execution servers that a user group can use.
• Create an active session pool. An active session pool consists of a specified maximum number of user sessions allowed to be concurrently active within a user group.
• Monitor used database/server resources by dictionary views.
• Manage runaway sessions or calls and prevent them from overloading the database.
• Prevent the execution of operations that the optimizer estimates will run for a longer time than a specified limit.
• Limit the amount of time that a session can be connected but idle, thus forcing inactive sessions to disconnect and potentially freeing memory resources.
• Allow a database to use different resource plans, based on changing workload requirements.
• Manage CPU allocation when there is more than one instance on a server in an Oracle Real Application Cluster environment (also called instance caging).

Oracle Resource Manager introduces three concepts:

Consumer Group – A collection of sessions grouped together based on resource requirements. The Oracle Resource Manager allocates server resources to resource consumer groups, not to the individual sessions.

Resource Plan – Specifies how the database allocates its resources to different Consumer Groups. You will need to specify how the database allocates resources by activating a specific resource plan.

Resource Plan Directive – Associates a resource consumer group with a plan and specifies how resources are to be allocated to that resource consumer group.

Notes:
• Only one Resource Plan can be active at any given time.
• Resource Directives control the resources allocated to a Consumer Group belong to a Resource Plan.
• The Resource Plan can refer to Subplans to create even more complex Resource Plans.
Creating a Simple Resource Plan

1. To enable the Oracle Resource Manager, you need to assign a plan name to the `RESOURCE_MANAGER_PLAN` parameter. Using an empty string will disable the Resource Manager.

```sql
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN = 'mydb_plan';
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN = '';
```

We can also create complex Resource Plans. A complex Resource Plan is one that is not created with the `CREATE_SIMPLE_PLAN` PL/SQL procedure and provides more flexibility and granularity.

```sql
BEGIN
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
        PLAN => 'DAYTIME',
        GROUP_OR_SUBPLAN => 'OLTP',
        COMMENT => 'OLTP group',
        MGMT_P1 => 75);
END;
/
```

For additional details:
https://docs.oracle.com/database/121/ADMIN/dbrm.htm#ADMIN027
Overview
PostgreSQL does not have built-in resource management capabilities that are equivalent to the functionality provided by Oracle Resource Manager. However, due to the elasticity and flexibility provided by “cloud economics”, workarounds could be applicable and such capabilities might not be as of similar importance to monolithic on-premises databases.

The Oracle Resource Manager primarily exists because traditionally, Oracle databases were installed on very powerful monolithic servers that powered multiple applications simultaneously. The monolithic model made the most sense in an environment where the licensing for the Oracle database was per-CPU and where Oracle databases were deployed on physical hardware. In these scenarios, it made sense to consolidate as many workloads as possible into few servers. In cloud databases, the strict requirement to maximize the usage of each individual “server” is often not as important and a different approach can be employed:

Individual Amazon Aurora clusters can be deployed, with varying sizes, each dedicated to a specific application or workload. Additional read-only Aurora Replica servers can be used to offload any reporting-style workloads from the master instance.

The traditional Oracle model where maximizing the usage of each physical Oracle server was essential due to physical hardware constraints and the per-CPU core licensing model.
With Amazon Aurora, separate and dedicated database clusters can be deployed, each dedicated to a specific application/workload creating isolation between multiple connected sessions and applications.

Each Amazon Aurora instance (Primary/Replica) can be scaled independently in terms of CPU and memory resources using the different “instance types”. Because multiple Amazon Aurora Instances can be instantly deployed and much less overhead is associated with the deployment and management of Aurora instances when compared to physical servers, separating different workloads to different instance classes could be a suitable solution for controlling resource management.

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>vCPU</th>
<th>Memory (GiB)</th>
<th>PIOPS-Optimized</th>
<th>Network Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db.m4.large</td>
<td>2</td>
<td>8</td>
<td>Yes</td>
<td>Moderate</td>
</tr>
<tr>
<td>db.m4.xlarge</td>
<td>4</td>
<td>16</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.m4.2xlarge</td>
<td>8</td>
<td>32</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.m4.4xlarge</td>
<td>16</td>
<td>64</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.m4.10xlarge</td>
<td>40</td>
<td>160</td>
<td>Yes</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td>db.m3.medium</td>
<td>1</td>
<td>3.75</td>
<td>-</td>
<td>Moderate</td>
</tr>
<tr>
<td>db.m3.large</td>
<td>2</td>
<td>7.5</td>
<td>-</td>
<td>Moderate</td>
</tr>
<tr>
<td>db.m3.xlarge</td>
<td>4</td>
<td>15</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.m3.2xlarge</td>
<td>8</td>
<td>30</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td><strong>Memory Optimized</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db.r3.large</td>
<td>2</td>
<td>15</td>
<td>-</td>
<td>Moderate</td>
</tr>
<tr>
<td>db.r3.xlarge</td>
<td>4</td>
<td>30.5</td>
<td>Yes</td>
<td>Moderate</td>
</tr>
<tr>
<td>db.r3.2xlarge</td>
<td>8</td>
<td>61</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.r3.4xlarge</td>
<td>16</td>
<td>122</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>db.r3.8xlarge</td>
<td>32</td>
<td>244</td>
<td>-</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td><strong>Micro instances</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db.t2.micro</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>Low</td>
</tr>
<tr>
<td>db.t2.small</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>Low</td>
</tr>
</tbody>
</table>
In addition, each Amazon Aurora primary/replica instance can also be directly accessed from your applications using its own endpoint. This capability is especially useful if you have multiple Aurora read-replicas for a given cluster and you wish to utilize different Aurora replicas to segment your workload.

**Example**

Suppose that you were using a single Oracle Database for multiple separate applications and used Oracle Resource Manager to enforce a workload separation, allocating a specific amount of server resources for each application. With Amazon Aurora, you might want to create multiple separate databases for each individual application. Adding additional replica instances to an existing Amazon Aurora cluster is easy.

1. In the AWS Management Console, select the **Amazon RDS service** and click the **DB Instances** link from the Resources section of the RDS Dashboard window (highlighted).
2. Select the Amazon Aurora cluster that you want to scale-out by adding an additional read Replica.

3. Click on the Instance Actions button.

4. Select Create Aurora Replica.

5. Select the instance class depending on the amount of compute resources your application requires.
6. Once completed, click **Create Aurora Replica**.

Oracle Resource Manager vs. Dedicated Aurora PostgreSQL Instances

<table>
<thead>
<tr>
<th>Oracle Resource Manager</th>
<th>Amazon Aurora Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set the maximum CPU usage for a resource group</strong></td>
<td>Create a dedicated Aurora Instance for a specific application.</td>
</tr>
<tr>
<td><strong>Limit the degree of parallelism for specific queries</strong></td>
<td>SET <code>max_parallel_workers_per_gather</code> TO <code>x</code>;</td>
</tr>
<tr>
<td><strong>Limit parallel execution</strong></td>
<td><em>Setting the PostgreSQL</em></td>
</tr>
<tr>
<td><strong>Limit the number of active sessions</strong></td>
<td><code>max_parallel_workers_per_gather</code> parameter should be done as part of your application database connection.*</td>
</tr>
<tr>
<td><strong>Restrict maximum runtime of queries</strong></td>
<td>Manually detect the number of connections that are open from a specific application and restrict connectivity either via database procedures or within the application DAL itself.</td>
</tr>
<tr>
<td></td>
<td>select <code>pid</code> from <code>pg_stat_activity</code> where <code>usename</code> in(</td>
</tr>
<tr>
<td></td>
<td>select <code>usename</code> from <code>pg_stat_activity</code> where <code>state</code> = 'active' group by <code>usename</code> having <code>count(*)</code> &gt; 10 and <code>state</code> = 'active' order by <code>query_Start</code>;</td>
</tr>
<tr>
<td></td>
<td>Manually terminate sessions that exceed the required threshold. You can detect the length of running queries using SQL commands and restrict max execution duration using either</td>
</tr>
<tr>
<td><strong>Oracle Resource Manager</strong></td>
<td><strong>Amazon Aurora Instances</strong></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>database procedures or within the application DAL itself. SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE now()-pg_stat_activity.query_start &gt; interval '5 minutes';</td>
<td>Manually terminate sessions that exceed the required threshold. You can detect the length of your idle sessions using SQL queries and restrict maximum execution using either database procedures or within the application DAL itself. SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname = 'regress' AND pid &lt;&gt; pg_backend_pid() AND state = 'idle' AND state_change &lt; current_timestamp - INTERVAL '5' MINUTE;</td>
</tr>
<tr>
<td><strong>Limit the maximum idle time for sessions</strong></td>
<td><strong>Limit the time that an idle session holding open locks can block other sessions</strong></td>
</tr>
<tr>
<td></td>
<td>Manually terminate sessions that exceed the required threshold. You can detect the length of blocking idle sessions using SQL queries and restrict max execution duration using either database procedures or within the application DAL itself. SELECT pg_terminate_backend(blocking_locks.pid) FROM pg_catalog.pg_locks AS blocked_locks JOIN pg_catalog.pg_stat_activity AS blocked_activity ON blocked_activity.pid = blocked_locks.pid JOIN pg_catalog.pg_locks AS 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</td>
</tr>
<tr>
<td>Oracle Resource Manager</td>
<td>Amazon Aurora Instances</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td></td>
<td>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 AS blocking_activity ON blocking_activity.pid = blocking_locks.pid WHERE NOT blocked_locks.granted and blocked_activity.state_change &lt; current_timestamp - INTERVAL '5' minute;</td>
</tr>
</tbody>
</table>

| Use “instance caging” in a multi-node Oracle RAC Environment | Similar capabilities can be achieved by separating different applications to different Aurora clusters or, for read-only workloads, separate Aurora read replicas within the same Aurora cluster. |

For additional detail:
https://www.postgresql.org/docs/9.6/static/runtime-config-resource.html
Overview
Database user accounts are used for authenticating connecting sessions as well as authorizing access for individual users to specific database objects. The Database Administrator grants privileges to database user accounts that are used by applications to authenticate with the Oracle database.

Steps for Providing Database Access to Applications

1. Create a user account in the database, typically authenticated by using a password. Additional methods of authenticating users also exist.
2. Assign permissions to the database user account enabling access certain database objects and system permissions.
3. Connecting applications will use the database username and password combination to authenticate with the database.

Oracle Database Users Common Properties

1. Granting privileges or roles (collection of privileges) to the database user.
2. Defining the default database tablespace for the user.
3. Assigning tablespace quotas for the database user.
4. Configuring password policy, password complexity, lock or unlock the account.

Authentication Mechanisms

1. Username and password combination (default).
2. External - using OS or third-party (Kerberos).
3. Global - enterprise directory service (such as Active Directory or Oracle Internet Directory).

Oracle Schemas Compared to Users

In the Oracle database, a user equals a schema. This relationship is special because in Oracle, users and schemas are essentially the same thing. Consider an Oracle database user as the account you use to connect to a database while a database schema is the set of objects (tables, views, etc.) that belong to that account.

- You cannot create schemas and users separately. When you create a database user, you also create a database schema with the same name.
- When you run the `CREATE USER` command in Oracle, you create a user for login and a schema to store database objects in.
- The schema created will be initially empty, but objects, such as tables, can be created inside it.
Database Users in Oracle 12c
Two kinds of users exist in the Oracle 12c database:
- Common users - created in all database containers, root and PDBs. Common user must have the C## prefix in the username.
- Local users – created only in a specific PDB. Different database users with identical usernames can be created in multiple PDBs.

Examples
1. Create a common database user, using the default tablespace.
2. Grant privileges and roles to the user.
3. Assign a profile to the user, unlock the account and force the user to change the password (PASSWORD EXPIRE).
4. Create a local database user in the my_pdb1 pluggable database.

CREATE USER c##test_user IDENTIFIED BY password DEFAULT TABLESPACE USERS;
GRANT CREATE SESSION TO c##test_user;
GRANT RESOURCE TO c##test_user;
ALTER USER c##test_user ACCOUNT UNLOCK;
ALTER USER c##test_user PASSWORD EXPIRE;
ALTER USER c##test_user PROFILE ORA_STIG_PROFILE;
ALTER SESSION SET CONTAINER = my_pdb1;
CREATE USER app_user1 IDENTIFIED BY password DEFAULT TABLESPACE USERS;

For additional details:
https://docs.oracle.com/database/121/DBSEG/users.htm
Migration to: PostgreSQL Roles

Overview

In a PostgreSQL database, users and roles are identical. Roles with connect permissions are essentially database users.

- A role is a database entity that can own objects and have database privileges.
- A role can be considered a "user", a "group", or both depending on how it is used.
- Roles are defined at the root level and are valid in all databases in the Amazon Aurora cluster. In terms of database scope, roles in PostgreSQL can be compared to common users in Oracle 12c as they are global for all the databases and are not created in the individual scope of a specific database.
- Schemas are created separately from roles/users in PostgreSQL.

<table>
<thead>
<tr>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common database user (12c)</td>
<td>Database role with Login</td>
</tr>
<tr>
<td>Local database user (12c)</td>
<td>N/A</td>
</tr>
<tr>
<td>Database user (11g)</td>
<td>Database role with Login</td>
</tr>
<tr>
<td>Database role</td>
<td>Database role without Login</td>
</tr>
<tr>
<td>Database users are identical to schema</td>
<td>Database users and schemas are created separately</td>
</tr>
</tbody>
</table>

The `CREATE USER` command in PostgreSQL is an alias for the `CREATE ROLE` command with one important difference: the `CREATE USER` command it automatically adds the `LOGIN` argument so that the role can access the database and act as a “database user”.

Examples

1. Create a role that can log-in to the database and specify a password:

```
CREATE USER test_user1 WITH PASSWORD 'password';
CREATE ROLE test_user2 WITH LOGIN PASSWORD 'password';
```

Notes

- CREATE USER is identical to CREATE ROLE, except that it implies a log-in to the database.
- When we provision a new Amazon Aurora cluster, a master user is created as the most powerful user in the database:

```
Settings

DB Instance Identifier*  my_aurora01
Master Username*         super_user
Master Password*         ************
Confirm Password*        ************
```

2. Create a role that can log in to the database and assign a password that has an expiration date:

```
CREATE ROLE test_user3 WITH LOGIN PASSWORD 'password' VALID UNTIL '2018-01-01';
```
3. Create a powerful role `db_admin` that will allow users to which this role is assigned to create new databases. Note that this role will not be able to log in to the database. Assign this role to the `test_user1` database user.

```
CREATE ROLE db_admin WITH CREATEDB;
GRANT db_admin TO test_user1;
```

4. Create a new schema `hello_world` and create a new table inside that schema:

```
CREATE SCHEMA hello_world;
CREATE TABLE hello_world.test_table1 (a int);
```

**Oracle vs. PostgreSQL Database Users**

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>Amazon Aurora PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>List all database users</td>
<td><code>SELECT * FROM dba_users;</code></td>
<td><code>SELECT * FROM pg_user;</code></td>
</tr>
<tr>
<td>Create a database user</td>
<td><code>CREATE USER c##test_user IDENTIFIED BY test_user;</code></td>
<td><code>CREATE ROLE test_user WITH LOGIN PASSWORD 'test_user';</code></td>
</tr>
<tr>
<td>Change the password for a database user</td>
<td><code>ALTER USER c##test_user IDENTIFIED BY test_user;</code></td>
<td><code>ALTER ROLE test_user WITH LOGIN PASSWORD 'test_user';</code></td>
</tr>
<tr>
<td>External authentication</td>
<td>Supported via Externally Identified Users</td>
<td>Currently not supported; future support for AWS Identity and Access Management (IAM) users is possible</td>
</tr>
<tr>
<td>Tablespace quotas</td>
<td><code>ALTER USER c##test_user QUOTA UNLIMITED ON TABLESPACE users;</code></td>
<td>Not supported</td>
</tr>
<tr>
<td>Grant role to user</td>
<td><code>GRANT my_role TO c##test_user;</code></td>
<td><code>GRANT my_role TO test_user;</code></td>
</tr>
<tr>
<td>Lock user</td>
<td><code>ALTER USER c##test_user ACCOUNT LOCK;</code></td>
<td><code>ALTER ROLE test_user WITH NOLOGIN;</code></td>
</tr>
<tr>
<td>Unlock user</td>
<td><code>ALTER USER c##test_user ACCOUNT UNLOCK;</code></td>
<td><code>ALTER ROLE test_user WITH LOGIN;</code></td>
</tr>
<tr>
<td>Grant privileges</td>
<td><code>GRANT CREATE TABLE TO c##test_user;</code></td>
<td><code>GRANT create ON DATABASE postgres_to_test_user;</code></td>
</tr>
<tr>
<td>Default tablespace</td>
<td><code>ALTER USER c##test_user default_tablespace users;</code></td>
<td><code>ALTER ROLE test_user SET default_tablespace = 'pg_global';</code></td>
</tr>
<tr>
<td>Grant select privilege on a table</td>
<td><code>GRANT SELECT ON hr.employees TO c##test_user;</code></td>
<td><code>GRANT SELECT ON hr.employees_to_test_user;</code></td>
</tr>
<tr>
<td>Grant DML privileges on a table</td>
<td><code>GRANT INSERT,DELETE ON hr.employees TO c##test_user;</code></td>
<td><code>GRANT INSERT,DELETE ON hr.employees_to_test_user;</code></td>
</tr>
<tr>
<td>Grant execute</td>
<td><code>GRANT EXECUTE ON hr.procedure_name TO c##test_user;</code></td>
<td><code>grant execute on function &quot;newdate&quot;() to test_user; inside the brackets &quot;)&quot; - specify the arguments types for the function</code></td>
</tr>
<tr>
<td>Limits user connection</td>
<td><code>CREATE PROFILE app_users LIMIT SESSIONS_PER_USER 5;</code></td>
<td><code>ALTER ROLE test_user WITH CONNECTION LIMIT 5;</code></td>
</tr>
<tr>
<td>Create a new database schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER USER C##TEST_USER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROFILE app_users;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE USER my_app_schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDENTIFIED BY password;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE SCHEMA my_app_schema;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For additional details:
https://www.postgresql.org/docs/9.6/static/sql-createrole.html
Overview
The Oracle instance allocates several individual “pools” of server RAM used as various caches for the database. These include the Buffer Cache, Redo Buffer, Java Pool, Shared Pool, Large Pool and others. These caches are contained in the SGA, or System Global Area. These caches are shared across all Oracle sessions.

In addition to the SGA, each connecting Oracle session is granted an additional area of memory used for session-private operations (sorting, private SQL cursors elements, etc.) called PGA, or Private Global Area.

The size of the various caches is controlled via parameters, either individually on per-cache level or managed as a whole automatically by the Oracle database by setting a unified “memory size” parameter and allowing Oracle to take care of individual cache sizes.

- All Oracle memory parameters are set using an ALTER SYSTEM command.
- Some changes to memory parameters may require an instance restart.

Some of the common Oracle parameters that control memory allocations include:

- `db_cache_size` – size of the cache used for database data.
- `log_buffer` – cache used to store Oracle redo log buffers, until written to disk.
- `shared_pool_size` – cache used to store shared cursors, stored procedures, control structures, and other structures.
- `large_pool_size` – cached used for parallel queries and RMAN backup / restore operations.
- `Java_pool_size` – cached used to store Java code and JVM context.

While these parameters can be configured individually, most Database Administrators choose to let Oracle automatically manage available RAM. The Database Administrator configures the size of the overall size of the SGA only, and Oracle sizes individual caches based on workload characteristics.

- `sga_max_size` – specify the hard-limit, maximum size of the SGA as a whole.
- `sga_target` – sets the required soft-limit for the SGA and within it, the individual caches.

Oracle also grants control over how much private memory is dedicated for each connecting session. The Database Administrator will configure the total size of memory available for all connecting sessions, and Oracle will allocate individual dedicated “chunks” from the total amount of available memory for each session.

- `pga_aggregate_target` – a soft-limit controlling the total amount of memory available for all sessions, combined.
- `pga_aggregate_limit` – (Oracle 12c only) a hard-limit for the total amount of memory available for all sessions, combined.

In addition, instead of manually configuring the SGA and PGA memory areas, we can also configure one overall memory limit for both the SGA and PGA and let Oracle automatically balance memory between the various memory pools. This behavior can be enabled using the `memory_target` and `memory_max_target` parameters.
Migration to: PostgreSQL Memory Buffers

Overview
PostgreSQL provides us with control over how server RAM is allocated. Some of the most important PostgreSQL memory parameters include:

<table>
<thead>
<tr>
<th>Memory Pool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_buffers</td>
<td>Used to cache database data read from disk.</td>
</tr>
<tr>
<td></td>
<td>Approximate Oracle Database Buffer Cache equivalent.</td>
</tr>
<tr>
<td>walBuffers</td>
<td>Used to store WAL (Write-Ahead-Log) records before they are written to disk.</td>
</tr>
<tr>
<td></td>
<td>Approximate Oracle Redo Log Buffer equivalent.</td>
</tr>
<tr>
<td>work_mem</td>
<td>Used for parallel queries and SQL sort operations.</td>
</tr>
<tr>
<td></td>
<td>Approximate Oracle PGA equivalent and/or the Large Pool (for parallel workloads)</td>
</tr>
<tr>
<td>maintenance_work_mem</td>
<td>Memory used for certain backend database operations such as: VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY</td>
</tr>
<tr>
<td>temp_buffers</td>
<td>Memory buffers used by each database session for reading data from temporary tables.</td>
</tr>
</tbody>
</table>

Total memory available for PostgreSQL Cluster
Controlled by choosing the “DB Instance Class” during instance creation:

![DB Instance Class](image)

Notes:
Cluster level parameters, such as shared_buffers and wal_buffers, are configured using “parameter groups” in the Amazon RDS Management Console.

Examples
1. View the configured value for database parameters:

```plaintext
show shared_buffers
show work_mem
show temp_buffers
```
2. View current configured values for all database parameters:

```
SELECT * FROM pg_settings;
```

3. Use of the `SET SESSION` command to modify the value of parameters that support session-specific settings. Changing the value using the `SET SESSION` command for one session will have no effect on other sessions.

```
SET SESSION work_mem='100MB';
```

**Note**

If a `SET SESSION` command is issued within a transaction that is aborted or rolled back, the effects of the `SET SESSION` command disappear. Once the transaction is committed, the effects will become persistent until the end of the session, unless overridden by another execution of `SET SESSION`.

4. Use of the `SET LOCAL` command to modify the current value of those parameters that can be set locally to a single transaction. Changing the value using the `SET LOCAL` command for one transaction will have no subsequent effect on other transactions from the same session. After issuing a `COMMIT` or `ROLLBACK`, the session-level settings will take effect.

```
SET LOCAL work_mem='100MB';
```

5. Reset a value of a run-time parameter to its default value:

```
RESET work_mem;
```

6. Changing parameter values can also be done via a direct update to the `pg_settings` table:

```
UPDATE pg_settings SET setting = '100MB' WHERE name = 'work_mem';
```
Common Oracle vs. PostgreSQL Memory Caches
Please note that the table provided below should be used as a general reference only and functionality might not be identical across Oracle and PostgreSQL.

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory for caching table data</td>
<td><code>db_cache_size</code></td>
<td><code>shared_buffers</code></td>
</tr>
<tr>
<td>Memory for transaction log records</td>
<td><code>log_buffer</code></td>
<td><code>wal_buffers</code></td>
</tr>
<tr>
<td>Memory for parallel queries</td>
<td><code>large_pool_size</code></td>
<td><code>work_mem</code></td>
</tr>
<tr>
<td>Java code and JVM</td>
<td><code>Java_pool_size</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Maximum amount of physical memory available for the Instance</td>
<td><code>sga_max_size</code> or <code>memory_max_size</code></td>
<td>Configured via the Amazon RDS/Aurora Instance class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>db.r3.large: 15.25GB</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>db.r3.xlarge: 30.5GB</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Etc.</td>
</tr>
<tr>
<td>Total amount of private memory for all sessions</td>
<td><code>pga_aggregate_target +</code> <code>pga_aggregate_limit</code></td>
<td><code>temp_buffers</code> (for reading data from temp tables)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>work_mem</code> (for sorts)</td>
</tr>
<tr>
<td>View values for all database parameters</td>
<td><code>Select * from v$parameter;</code></td>
<td><code>Select * from pg_settings;</code></td>
</tr>
<tr>
<td>Configure a session-level parameter</td>
<td><code>ALTER SESSION SET ...</code></td>
<td><code>SET SESSION work_mem='100MB';</code></td>
</tr>
<tr>
<td>Configure instance-level parameter</td>
<td><code>ALTER SYSTEM SET ...</code></td>
<td>Configured via “Parameter Groups” in the Amazon RDS Management Console.</td>
</tr>
</tbody>
</table>

For additional details:
https://www.postgresql.org/docs/current/static/runtime-config-resource.html
https://www.postgresql.org/docs/current/static/runtime-config-wal.html
Overview
Oracle roles are groups of privileges that can be granted to database users. A database role can contain individual system and object permissions as well as other roles. Database roles enable you to grant multiple database privileges to users in one go. It is convenient to group permissions together to ease the management of privileges.

Oracle 12c introduces a new multi-tenant database architecture that supports the creation of both common as well as local roles:

1. **Common roles** – these are roles created at the container database (CDB) level. A common role is a database role that exists in the root and in every existing, and future, pluggable database (PDB) that will be created. Common roles are useful for cross-container operations, such as ensuring that a common user has a role in every container.

2. **Local roles** – these are roles created in a specific pluggable database (PDB). A local role exists only in a single pluggable database and can only contain roles and privileges that apply within the pluggable database in which the role exists.

Notes:
- Common role names must start with a `c##` prefix. Starting with Oracle 12.1.0.2, these prefixes can be change using the `COMMON_USER_PREFIX` parameter.
- A `CONTAINER` clause can be added to `CREATE ROLE` statement to choose the container applicable for the role.

Examples

1. Create a common role:

   ```sql
   SQL> show con_name
   CON_NAME
   ------------------------
   CDB$ROOT
   SQL> CREATE ROLE c##common_role;
   Role created.
   ```
2. Create a local role:

```
SQL> show con_name
CON_NAME                  
-------------------------
ORCLPDB                    

SQL> CREATE ROLE local_role;
Role created.
```

3. Grant privileges and roles to the `local_role` database role.

```
GRANT RESOURCE, ALTER SYSTEM, SELECT ANY DICTIONARY TO local_role;
```

Any database user to which the `local_role` role will be granted, will now hold all privileges that were granted to the role.

4. Revoke privileges and roles from the `local_role` database role:

```
REVOKE RESOURCE, ALTER SYSTEM, SELECT ANY DICTIONARY FROM local_role;
```

For additional details:
https://docs.oracle.com/database/121/DBSEG/authorization.htm
https://docs.oracle.com/database/121/DBSEG/authorization.htm
**Migration to:** PostgreSQL Roles

**Overview**

In PostgreSQL, roles without login permissions are similar to database roles in Oracle. PostgreSQL roles are most similar to common roles in Oracle 12c as they are global in scope for all the databases in the instance.

- Roles are defined at the database cluster level and are valid in all databases in the PostgreSQL cluster. In terms of database scope, roles in PostgreSQL can be compared to common roles in Oracle 12c as they are global for all the databases and are not created in the individual scope of each database.
- The `CREATE USER` command in PostgreSQL is an alias for the `CREATE ROLE` command with one important difference: when using `CREATE USER` command, it automatically adds `LOGIN` so the role can access to the database as a “database user”. As such, for creating PostgreSQL roles that are similar in function to Oracle roles, be sure to use the `CREATE ROLE` command.

**Example**

Create a new database role called `myrole1` that will allow users (to which the role is assigned) to create new databases in the PostgreSQL cluster. Note that this role will not be able to login to the database and act as a “database user”. In addition, grant `SELECT`, `INSERT` and `DELETE` privileges on the `hr.employees` table to the role:

```sql
CREATE ROLE hr_role;
GRANT SELECT, INSERT, DELETE ON hr.employees TO hr_role;
```

Typically, a role being used as a group of permissions would not have the `LOGIN` attribute, as with the example above.

**Comparing Oracle to PostgreSQL database roles**

<table>
<thead>
<tr>
<th>Description</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>List all roles</td>
<td><code>SELECT * FROM dba_roles;</code></td>
<td><code>SELECT * FROM pg_roles;</code></td>
</tr>
<tr>
<td>Create a new role</td>
<td><code>CREATE ROLE c##common_role;</code></td>
<td><code>CREATE ROLE test_role;</code></td>
</tr>
<tr>
<td>Or <code>CREATE ROLE local_role1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grant one role privilege to another database role</td>
<td><code>GRANT local_role1 TO local_role2;</code></td>
<td><code>grant myrole1 to myrole2;</code></td>
</tr>
<tr>
<td>Grant privileges on a database object to a database role</td>
<td><code>GRANT CREATE TABLE TO local_role;</code></td>
<td><code>GRANT create ON DATABASE postgresdb TO test_user;</code></td>
</tr>
<tr>
<td>Grant DML permissions on a database object to a role</td>
<td><code>GRANT INSERT, DELETE ON hr.employees TO myrole1;</code></td>
<td><code>GRANT INSERT, DELETE ON hr.employees TO myrole1;</code></td>
</tr>
</tbody>
</table>

For additional details:  
Overview
Oracle provides several built-in views that are used to monitor the database and query its operational state. These views can be used to track the status of the database, view information about database schema objects and more.

The data dictionary is a collection of internal tables and views that supply information about the state and operations of the Oracle database including: database status, database schema objects (tables, views, sequences, etc.), users and security, physical database structure (datafiles), and more. The contents of the data dictionary are persistent to disk.

Examples for data dictionary views include:

- **DBA_TABLES** – information about all of the tables in the current database.
- **DBA_USERS** – information about all the database users.
- **DBA_DATA_FILES** – information about all of the physical datafiles in the database.
- **DBA_TABLESPACES** – information about all tablespaces in the database.
- **DBA_TABLES** – information about all tables in the database.
- **DBA_TAB_COLS** – information about all columns, for all tables, in the database.

**Note:** data dictionary view names can start with `DBA_`*, `ALL_`*, `USER_`*, depending on the level and scope of information presented (user-level versus database-level).

For the complete list of dba_* data dictionary views:
https://docs.oracle.com/database/121/nav/catalog_views-dba.htm

Dynamic performance views (V$ Views) are a collection of views that provide real-time monitoring information about the current state of the database instance configuration, runtime statistics and operations. These views are continuously updated while the database is running.

Information provided by the dynamic performance views includes session information, memory usage, progress of jobs and tasks, SQL execution state and statistics and various other metrics.

Common dynamic performance views include:

- **V$SESSION** – information about all current connected sessions in the instance.
- **V$LOCKED_OBJECT** – information about all objects in the instance on which active “locks” exist.
- **V$INSTANCE** – dynamic instance properties.
- **V$SESSION_LONG_OPS** – information about certain “long running” operations in the database such as queries currently executing.
- **V$MEMORY_TARGET_ADVICE** – advisory view on how to size the instance memory, based on instance activity and past workloads.

For additional details:
https://docs.oracle.com/database/121/nav/catalog_views.htm
Migration to: PostgreSQL System Catalog & The Statistics Collector

Overview
PostgreSQL provides three different sets of metadata tables that are used to retrieve information about the state of the database and current activities. These tables are similar in nature to the Oracle data dictionary tables and V$ performance views. In addition, Amazon Aurora PostgreSQL provides a “Performance Insights” console for monitoring and analyzing database workloads and troubleshooting performance issues.

2. System Catalog Tables
These are a set of tables used to store dynamic and static metadata for the PostgreSQL database and can be thought of as the “data dictionary” for the database. These tables are used for internal “bookkeeping”-type activities. All System catalog tables start with the `pg_*` prefix and can be found in the `pg_catalog` schema. Both system catalog tables and statistics collector views can be found on the `pg_catalog` schema.

Example
Display all tables in the `pg_catalog` schema:

```
select * from pg tables where schemaname='pg catalog';
```

Some of the common system catalog tables include:

<table>
<thead>
<tr>
<th>Table name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_database</code></td>
<td>Contains information and properties about each database in the PostgreSQL cluster, such as the database encoding settings as well as others.</td>
</tr>
<tr>
<td><code>pg_tables</code></td>
<td>Information about all tables in the database, such as indexes and the tablespace for each database table.</td>
</tr>
<tr>
<td><code>pg_index</code></td>
<td>Contains information about all indexes in the database</td>
</tr>
<tr>
<td><code>pg_cursors</code></td>
<td>List of currently available/open cursors</td>
</tr>
</tbody>
</table>
3. **Statistics Collector**

Special subsystem which collects runtime dynamic information about the current activities in the database instance. For example, statistics collector views are useful to determine how frequently a particular table is accessed and if the table is scanned or accessed using an index.

```sql
SELECT * FROM pg_stat_activity WHERE STATE = 'active';
```

Common statistics collector views include:

<table>
<thead>
<tr>
<th>Table name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_stat_activity</td>
<td>Statistics of currently sessions in the database. Useful for identifying long running queries</td>
</tr>
<tr>
<td>pg_stat_all_tables</td>
<td>Performance statistics on all tables in the database, such as identifying table size, write activity, full scans vs. index access, etc.</td>
</tr>
<tr>
<td>pg_statio_all_tables</td>
<td>Performance statistics and I/O metrics on all database tables</td>
</tr>
<tr>
<td>pg_stat_database</td>
<td>One row for each database showing database-wide statistics such as blocks read from the buffer cache vs. blocks read from disk (buffer cache hit ratio).</td>
</tr>
<tr>
<td>pg_stat_bgwriter</td>
<td>Important performance information on PostgreSQL checkpoints and background writes</td>
</tr>
<tr>
<td>pg_stat_all_indexes</td>
<td>Performance and usage statistics on indexes, for example, useful for identifying unused indexes</td>
</tr>
</tbody>
</table>

4. **Information Schema Tables**

The information schema consists of views which contain information about objects that were created in the current database.

- The information schema is specified by the SQL standard and as such, supported by PostgreSQL.
- The owner of this schema is the initial database user.
- Since the information schema is defined as part of the SQL standard, it can be expected to remain stable across PostgreSQL versions. This is unlike the system catalog tables, which are specific to PostgreSQL, and subject to changes across different PostgreSQL versions.
- The information schema views do not display information about PostgreSQL-specific features.

```sql
select * from information_schema.tables;
```
Note
By default, all database users (public) can query both the system catalog tables, the statistics collector views and the information schema.

Common Oracle vs. PostgreSQL system metadata tables

<table>
<thead>
<tr>
<th>Information</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database properties</td>
<td>V$DATABASE</td>
<td>PG DATABASE</td>
</tr>
<tr>
<td>Database sessions</td>
<td>V$SESSION</td>
<td>PG_STAT_ACTIVITY</td>
</tr>
<tr>
<td>Database users</td>
<td>DBA_USERS</td>
<td>PG_USER</td>
</tr>
<tr>
<td>Database tables</td>
<td>DBA_TABLES</td>
<td>PG_TABLES</td>
</tr>
<tr>
<td>Database roles</td>
<td>DBA_ROLES</td>
<td>PG_ROLES</td>
</tr>
<tr>
<td>Table columns</td>
<td>DBA_TAB_COLS</td>
<td>PG_ATTRIBUTE</td>
</tr>
<tr>
<td>Database locks</td>
<td>V$LOCKED_OBJECT</td>
<td>PG_LOCKS</td>
</tr>
<tr>
<td>Currently configured runtime parameters</td>
<td>V$PARAMETER</td>
<td>PG_SETTINGS</td>
</tr>
<tr>
<td>All system statistics</td>
<td>V$SYSSTAT</td>
<td>PG_STAT_DATABASE</td>
</tr>
<tr>
<td>Privileges on tables</td>
<td>DBA_TAB_PRIVS</td>
<td>TABLE_PRIVILEGES</td>
</tr>
<tr>
<td>Information about IO operations</td>
<td>V$SEGSTAT</td>
<td>PG_STATIO_ALL_TABLES</td>
</tr>
</tbody>
</table>

5. Amazon RDS performance Insights

In addition to monitoring database status and activity using queries on metadata tables, Aurora PostgreSQL provides a visual performance monitoring and status information via the “Performance Insights” feature accessible as part of the Amazon RDS Management Console.

Performance insights monitors your Amazon RDS/Aurora databases and captures workloads so that you can analyze and troubleshoot database performance. Performance insights visualizes the database load and provides advanced filtering using various attributes such as: waits, SQL statements, hosts, or users.

Example
Accessing the Amazon Aurora Performance Insights Console

1. Navigate to the RDS section of the AWS Console.
• Select **Performance Insights**.

Once you have accessed the **Performance insights** console, you will be presented with a visualized dashboard of your current and past database performance metrics. You can choose the period of time of the displayed performance data (5m, 1h, 6h or 24h) as well as different criteria to filter and slice the information presented such as waits, SQL, Hosts or Users, etc.

**Enabling Performance Insights**
Performance Insights is enabled by default for Amazon Aurora clusters. If you have more than one database created in your Aurora cluster, performance data for all of the databases is aggregated. Database performance data is kept for 24 hours.

For additional details:  
Migrating from: Oracle Flashback Database

Overview
Oracle flashback database is a special mechanism built into the Oracle database that can help protect against human errors by providing the capabilities to revert the entire database back to a previous point in time using SQL commands. Flashback database implements a self-logging mechanism that captures all the changes applied to the database and to data, essentially storing previous versions of database modifications in the configured database “fast recovery area” destination.

When using Oracle flashback database, you can choose to restore your entire database to either a user-created restore point, a timestamp value or to a specific Oracle System Change Number (SCN).

Examples

- Create a database restore point to which you can flashback your database to:

  ```sql
  CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE;
  ```

- Flashback your database to a previously created restore point:

  ```sql
  SQL> shutdown immediate;
  SQL> startup mount;
  SQL> flashback database to restore point before_update;
  ```

- Flashback your database to a specific time:

  ```sql
  SQL> shutdown immediate;
  SQL> startup mount;
  SQL> FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2017','MM/DD/YY')";
  ```

For additional details: https://docs.oracle.com/database/121/RCMRF/rcmsynta023.htm#RCMRF194
Migration to: Amazon Aurora Snapshots

Overview
The primary backup mechanism for Amazon Aurora are snapshots. Taking a database snapshot is an extremely fast and non-intrusive operation for your database. Database snapshots can be used in a similar way to flashback database in Oracle.

Amazon Aurora provides two types of snapshots:

- **Automated** - enabled by default.
- **Manual** – User-initiated backup of the database which can be done at any given time.

Restoring a snapshot will result in creating a new database instance. Up to 100 manual snapshots are supported for each Amazon Aurora database.

Similarly, to Oracle flashback, Amazon Aurora snapshots support two options for specifying how to restore your database:

1. Restore your database to a specific snapshot, similar to Oracle flashback database “restore points”.
2. Restore your database to a previous point in time, similar to Oracle Flashback database “restore to timestamp”.

Example
Enable automatic snapshots for your Amazon Aurora database and set the backup retention window during database creation (equivalent to setting the `DB_FLASHBACK_RETENTION_TARGET` parameter in Oracle).

1. Navigate to the Amazon RDS page in your AWS console:

   ![Amazon RDS page](image)

2. Select Instances.
3. Click on **Launch DB Instance**.

4. Select the **Amazon Aurora** with the **PostgreSQL** database engine.

5. Configure your database settings and parameters.
6. Configure your Amazon Aurora cluster backup retention policy as the number of days ("retention period") to automatically to store your snapshots:

Example

Perform a manual snapshot backup of your database, equivalent to creating a “guaranteed flashback database restore point” in Oracle (CREATE RESTORE POINT xxxx GUARANTEE FLASHBACK DATABASE;).

1. Navigate to the Amazon RDS page in your AWS Console:

2. Select Instances.
3. Select your Amazon Aurora PostgreSQL instance.
4. Click Instance actions.
5. Select Take Snapshot in the context menu.

Example
Restore an Amazon Aurora database backup from an existing snapshot, similar to using “flashback database to restore point xxx;” in Oracle.

1. Navigate to the Amazon RDS page in your AWS Console:
2. Choose Snapshots on the left-hand menu to see the list of snapshots available for your database instances:

![Amazon RDS Snapshots menu]

3. Select the snapshot to restore. Choose the snapshot, click on Snapshot Actions and select Restore snapshot in the context menu.

![Snapshot Actions menu]

Note: The restore process will create a new instance.

4. You will be presented with a wizard for creating your new Amazon Aurora instance from the snapshot you selected. Complete all the required properties for creating your newly restored database instance.

5. Click Restore DB Instance.
Example

Restore an Amazon Aurora database to a specific (previous) point in time, similar to the "FLASHBACK DATABASE TO TIME "TO_DATE('xxxx')" command in Oracle.

1. Navigate to the Amazon RDS page in your AWS Console.
2. Click **Instances**.

4. Select your Amazon Aurora instance and click on **Instance Actions**. Select **Restore to Point in Time** on the context menu.

5. This process will launch a new instance. Select the **date and time** to which you want to restore your new instance. The selected time must be within the configured backup retention for this instance.
Example

Modify the backup retention policy for an Amazon Aurora database, after a database was created. This process is similar to setting the `DB_FLASHBACK_RETENTION_TARGET` parameter in Oracle.

This process allows you to control for how long your Aurora database snapshots will be retained. When restoring an Amazon Aurora database to a previous point in time, the specified date/time must be within the configured backup retention window.

1. Navigate to the Amazon RDS page in your AWS Console.

2. Click Instances.

3. Select your Aurora instance, click Instance Actions.
4. Select **Modify** in the context menu.

5. Configure the desired **backup retention period**. Maximum supported retention is 35 days.
AWS CLI commands for Aurora database backup and database restore

In addition to using the AWS management console to restore your Amazon Aurora database to a previous point in time or to a specific snapshot, you can also use the AWS CLI to perform the same actions. Some examples include:

1. Use `describe-db-cluster-snapshots` to view all current Amazon Aurora snapshots.
2. Use `create-db-cluster-snapshot` to create a new snapshot (“restore point”).
3. Use `restore-db-cluster-from-snapshot` to restore a new cluster from an existing snapshot.
4. Use `create-db-instance` to add new instances to the newly restored Amazon Aurora cluster.

```bash
aws rds describe-db-cluster-snapshots
aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name
aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql
aws rds create-db-instance --region us-east-1 --db-subnet-group default - -engine aurora-postgresql --db-cluster-identifier NewCluster --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large
```

5. Use `restore-db-instance-to-point-in-time` to perform point-in-time recovery.

```bash
aws rds restore-db-cluster-to-point-in-time --db-cluster-identifier clustername-restore --source-db-cluster-identifier clustername --restore-to-time 2017-09-19T23:45:00.000Z
aws rds create-db-instance --region us-east-1 --db-subnet-group default - -engine aurora-postgresql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large
```

Oracle Flashback database vs. Amazon Aurora Snapshots

<table>
<thead>
<tr>
<th>Create a “restore point”</th>
<th>Oracle</th>
<th>Amazon Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE;</td>
<td>aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configure flashback “retention period”</th>
<th>Oracle</th>
<th>Amazon Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER SYSTEM SET db_flashback_retention_target=2880;</td>
<td>Configure the “Backup retention window” setting using the AWS management console or using the AWS CLI.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flashback database to a previous “restore point”</th>
<th>Oracle</th>
<th>Amazon Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td>shutdown immediate; startup mount; flashback database to restore point before update;</td>
<td>1. Create new cluster from a snapshot: aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier</td>
<td></td>
</tr>
<tr>
<td><strong>Oracle</strong></td>
<td><strong>Amazon Aurora</strong></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td><strong>SnapshotToRestore --engine aurora-postgresql</strong></td>
<td><strong>2. Add new instance to the cluster:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clusternname-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large</code></td>
<td></td>
</tr>
</tbody>
</table>

**Flashback database to a previous point in time**

- shutdown immediate;
- startup mount;
- `FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2017','MM/DD/YY')";`
- `aws rds restore-db-cluster-to-point-in-time --db-cluster-identifier clusternname-restore --source-db-cluster-identifier clustername --restore-to-time 2017-09-19T23:45:00.000Z`

2. **Add a new instance to the cluster:**

- `aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large`

For additional details:
Overview
Oracle Log Miner is a tool that enables you to query the database Redo Logs and the Archived Redo Logs using a SQL interface. Using Log Miner, you can analyze the content of database “transaction logs” (online and archived redo logs) and provide historical insight on past database activity, such as reviewing individual DML statements which have modified data in the database.

Examples
Use Log Miner to view DML statements executed on the employees table:

1. Find current redo log file to analyze:

   ```sql
   SQL> SELECT V$LOG.STATUS, MEMBER
           FROM V$LOG, V$LOGFILE
           WHERE V$LOG.GROUP# = V$LOGFILE.GROUP#
           AND V$LOG.STATUS = 'CURRENT';
   ``

<table>
<thead>
<tr>
<th>STATUS</th>
<th>MEMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT</td>
<td>/u01/app/oracle/oradata/orcl/redo02.log</td>
</tr>
</tbody>
</table>

2. Use the DBMS_LOGMNR.ADD_LOGFILE procedure, pass the file path as a parameter to the Log Miner API:

   ```sql
   SQL> BEGIN
       DBMS_LOGMNR.ADD_LOGFILE('/u01/app/oracle/oradata/orcl/redo02.log');
       END;
   /
   PL/SQL procedure successfully completed.
   ``

3. Start Log Miner using the DBMS_LOGMNR.START_LOGMNR procedure:

   ```sql
   SQL> BEGIN
       DBMS_LOGMNR.START_LOGMNR(options=>
       dbms_logmnr.dict_from_online_catalog);
       END;
   /
   PL/SQL procedure successfully completed.
   ``

4. Run a DML statement as an example which we will analyze using Log Miner:

   ```sql
   SQL> UPDATE HR.EMPLOYEES
       SET SALARY=SALARY+1000
       WHERE EMPLOYEE_ID=116;
   COMMIT;
   ```
5. Querying the V$LOGMNR_CONTENTS table to view the DML commands captured using Log Miner:

```sql
SQL> SELECT TO_CHAR(TIMESTAMP,'mm/dd/yy hh24:mi:ss') TIMESTAMP,
    2    SEG_NAME, OPERATION, SQL_REDO, SQL_UNDO
FROM V$LOGMNR_CONTENTS
WHERE TABLE_NAME = 'EMPLOYEES'
AND OPERATION = 'UPDATE';
```

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>SEG_NAME</th>
<th>OPERATION</th>
<th>SQL_REDO</th>
<th>SQL_UNDO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/09/17 06:43:44</td>
<td>EMPLOYEES</td>
<td>UPDATE</td>
<td>update &quot;HR&quot;.&quot;EMPLOYEES&quot; set &quot;SALARY&quot; = '3900' where &quot;SALARY&quot; = '2900' and ROWID = 'AAAViUA AEAAABVvAAQ';</td>
<td>update &quot;HR&quot;.&quot;EMPLOYEES&quot; set &quot;SALARY&quot; = '2900' where &quot;SALARY&quot; = '3900' and ROWID = 'AAAABVvAAQ';</td>
</tr>
</tbody>
</table>

For additional information on Oracle LogMiner:
https://docs.oracle.com/cd/E11882_01/server.112/e22490/logminer.htm#SUTIL019

---

Migration to: PostgreSQL Logging Options

[Back to TOC]

Overview
PostgreSQL does not provide a feature that is directly equivalent to Oracle Log Miner. However, several alternatives exist which allow viewing historical database activity in PostgreSQL.

Using PG_STAT_STATEMENTS
Extension module for tracking query execution details with statistical information. The PG_STAT_STATEMENTS view presents a single row for each database operation that was logged, including information about the user, query, number of row retrieved by the query and more.

Examples
Configure and use PG_STAT_STATEMENTS to view past database activity:

1. On the AWS Management Console, navigate to RDS > Parameter Groups.
2. Select the current database parameter group:

3. Set the following parameters:

   shared_preload_libraries = 'pg_stat_statements'
   pg_stat_statements.max = 10000
   pg_stat_statements.track = all

   Note: A database reboot may be required for the updated values to take effect.

4. Connect to the and run the following command:

   psql=> CREATE EXTENSION PG_STAT_STATEMENTS;
5. **Test the** PG_STAT_STATEMENTS **view to see captured database activity:**

```sql
psql=> UPDATE EMPLOYEES
    SET SALARY=SALARY+1000
    WHERE EMPLOYEE_ID=116;

psql=> SELECT *
   FROM PG_STAT_STATEMENTS
  WHERE LOWER(QUERY) LIKE '%update%';
```

```
- [ RECORD 1 ]------------------------------
  userid              | 16393
  dbid                | 16394
  queryid             | 2339248071
  query               | UPDATE EMPLOYEES +
                      |     SET SALARY=SALARY+?? +
                      |     WHERE EMPLOYEE_ID=??
  calls               | 1
  total_time          | 11.989
  min_time            | 11.989
  max_time            | 11.989
  mean_time           | 11.989
  stddev_time         | 0
  rows                | 1
  shared_blks_hit     | 15
  shared_blks_read    | 10
  shared_blks_dirtied | 0
  shared_blks_written | 0
  local_blks_hit      | 0
  local_blks_read     | 0
  local_blks_dirtied  | 0
  local_blks_written  | 0
  temp_blks_read      | 0
  temp_blks_written   | 0
  blk_read_time       | 0
  blk_write_time      | 0
```

**Note:** PostgreSQL PG_STAT_STATEMENTS does not provide a feature that is equivalent to LogMiner’s SQL_UNDO column.

**DML / DDL Database Activity Logging**

DML and DML operations can be tracked inside the PostgreSQL log file (postgres.log) and viewed using AWS console.
Example

1. On the AWS Console, navigate to RDS > Parameter Groups.

2. Set the following parameters:

   ```
   log_statement = 'ALL'
   log_min_duration_statement = 1
   ```

   ![Parameter group settings]

   Note: A reboot may be required for the parameters to take effect.

3. Test DDL/DML logging:

   - On the AWS Management Console, navigate to RDS > Instances > Select Instance > Logs
   - Sort via the Last Written column to show recent logs (click on column header).
   - Click View on the relevant log. For example, the PostgreSQL log file shown here with a logged UPDATE command:

   ```
   2017-10-05 07:44:39 UTC:217.132.162.150(63545):aurora_admin@nayadb:[12069]:LOG: execute
   <unnamed>: UPDATE EMPLOYEES
   SET SALARY=SALARY+1000
   WHERE EMPLOYEE_ID=118
   2017-10-05 07:44:39 UTC:217.132.162.150(63545):aurora_admin@nayadb:[12069]:LOG: duration:
   12.054 ms
   2017-10-09 07:44:44 UTC:217.132.162.150(63545):aurora_admin@nayadb:[12069]:LOG: duration:
   0.134 ms parse <unnamed>: select * from pg_stat_statements where lower(query) like 'UPDATE%'
   ```
Amazon Aurora Performance Insights
The Amazon Aurora Performance Insights dashboard provides information about current and historical SQL statements, executions and workloads. Note, enhanced monitoring should be enabled during Amazon Aurora instance configuration.

Example

1. On the AWS Management Console, navigate to RDS > Instances.
2. Select the relevant instance and choose Instance Actions > Modify.
3. Ensure that Enable Enhanced Monitoring option is set to Yes.
4. Mark the checkbox for Apply Immediately.
5. Click Continue.
7. Select the relevant instance to monitor.
8. Select the timeframe and monitor scope (Waits, SQL, Hosts and Users).

For additional information:
https://www.postgresql.org/docs/9.6/static/runtime-config-logging.html
https://www.postgresql.org/docs/current/static/pgstatstatements.html
Overview
Oracle Instance and database-level parameters can be configured via ALTER SYSTEM commands. Certain parameters can be configured dynamically and take immediate effect, while other parameters require an instance restart.

1. All Oracle instance and database-level parameters are stored in a binary file known as the SPFILE (or Server Parameter FILE).
2. The binary server parameter file (SPFILE) can be exported to a text file via the following command:
   CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';

When modifying parameters, the DBA can choose the persistency of the changed values with one of the three following options:

- Make the change applicable only after a restart by specifying scope=spfile.
- Make the change dynamically but not persistent after a restart by specifying scope=memory.
- Make the change both dynamically and persistent by specifying scope=both.

Example

Use the ALTER SYSTEM SET command for configuring a value for an Oracle parameter

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE SCOPE=both;
```

For additional details about Oracle initialize Parameters and ALTER SYSTEM command:
https://docs.oracle.com/database/121/SQLRF/statements_2017.htm#SQLRF00902
**Overview**

When running your PostgreSQL databases as Amazon Aurora Clusters, changes to cluster-level and database-level parameters are performed via *Parameter Groups*.

Most of the PostgreSQL parameters are configurable in an Amazon Aurora PostgreSQL cluster, but some are disabled and non-modifiable. Since Amazon Aurora clusters restrict access to the underlying operating system, modification to PostgreSQL parameters are done using *parameter groups*.

Amazon Aurora is a cluster of DB instances and, as a direct result, some of the PostgreSQL parameters apply to the *entire cluster*, while other parameters apply only to a particular *database instance* in the cluster.

<table>
<thead>
<tr>
<th>Aurora PostgreSQL Parameter Class</th>
<th>Controlled Via</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cluster-level parameters</strong></td>
<td>Managed via <em>cluster parameter groups</em></td>
</tr>
<tr>
<td><em>Single cluster parameter group per Amazon Aurora Cluster</em></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>• The PostgreSQL <em>wal_buffers</em> parameter is controlled via a cluster parameter group.</td>
</tr>
<tr>
<td></td>
<td>• The PostgreSQL <em>autovacuum</em> parameter is controlled via a cluster parameter group.</td>
</tr>
<tr>
<td></td>
<td>• The <em>client_encoding</em> parameter is controlled via a cluster parameter group.</td>
</tr>
<tr>
<td><strong>Database Instance-Level parameters</strong></td>
<td>Managed via <em>database parameter groups</em></td>
</tr>
<tr>
<td><em>Every instance in your Amazon Aurora cluster can be associated with a unique database parameter group</em></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>• The PostgreSQL <em>shared_buffers</em> memory cache configuration parameter is controlled via a database parameter group with an AWS-optimized default value based on the configured database class: <code>{DBInstanceClassMemory/10922}</code></td>
</tr>
<tr>
<td></td>
<td>• The PostgreSQL <em>max_connections</em> parameter which controls maximum number of client connections allowed to the PostgreSQL instance, is controlled via a database parameter group. Default value is optimized by AWS based on the configured database class: <code>LEAST(({DBInstanceClassMemory/9531392},5000))</code></td>
</tr>
<tr>
<td></td>
<td>• The PostgreSQL <em>effective_cache_size</em> which informs the query optimizer how much cache is present in the kernel and helps control how expensive large index scans will be, is controlled via a database level parameter group. The default value is optimized by AWS based on database class (RAM): <code>{DBInstanceClassMemory/10922}</code></td>
</tr>
</tbody>
</table>
### Aurora PostgreSQL Parameter Class | Controlled Via

- The `authentication_timeout` parameter, which controls the maximum time to complete client authentication, in seconds, is controlled via a database parameter group.

- The `superuser_reserved_connections` parameter which determines the number of reserved connection "slots" for PostgreSQL superusers, is configured via a database parameter group.

### Examples

Create and configure the Amazon Aurora database and cluster parameter groups:

1. Navigate to the **RDS** Service section of the AWS Console.

2. Click **Parameters Group** on the left-hand navigation menu and select **Create Parameter Group**. **Note:** you cannot edit the default parameter group, you will need to create a custom parameter group to apply changes to your Amazon Aurora cluster and its database instances.

3. Complete all the required configuration and click **Create**.
• **Parameter group family** – select the database engine type for this group. For example – “aurora-postgresql9.6” should be selected for Amazon Aurora PostgreSQL clusters.
• **Type** – cluster or database-level parameter group.
• **Specify a custom name for your new parameter group.**
• **Specify a description for your parameter group.**

4. Once the new parameter group is created, you can configure its parameters by clicking **Edit Parameters**:

5. Setting the values for specific parameters inside the parameter groups is performed by searching for the parameter name (for example, the **authentication_timeout** parameter) and specifying a new value (for example, 3 minutes). Once the modification is complete, click **Save Changes**.
6. To associate an Aurora PostgreSQL Cluster with a specific parameter group, do the following:

- Navigate to the Instances List page.
- Select your desired Amazon Aurora instance.
- Click Instance Actions.
- In the context menu, click Modify.

7. In the configuration page, select the desired parameter group.
8. To apply the changes:

- Navigate to the Amazon Aurora instance list page.
- Expand the instance properties (1).
- Click the Details button (2). If the parameter group is listed as “pending reboot”, an instance restart is required.

9. To restart your Aurora instance, select Instance Actions and click Reboot.
Overview
Certain parameters and configuration options in the Oracle database are modifiable on a per-session level. This is accomplished using the `ALTER SESSION` command, which configures parameters for the scope of the connected session only.

Note:
Not all Oracle configuration options and parameters can be modified on a per-session basis. To view a list of all configurable parameters that can be set for the scope of a specific session, you will need to query the `v$parameter` view:

```
SELECT NAME, VALUE FROM V$PARAMETER WHERE ISSES_MODIFIABLE='TRUE';
```

Example
1. Change the `NLS_LANGUAGE` (codepage) parameter of the current session using an `ALTER SESSION` command:

   ```
   SQL> alter session set nls_language='SPANISH';
   Session modificada.
   SQL> alter session set nls_language='ENGLISH';
   Session altered.
   SQL> alter session set nls_language='FRENCH';
   Session modificado.
   SQL> alter session set nls_language='GERMAN';
   Session wurde geändert.
   ```

2. Specify the format of dates values returned from the database using the `NLS_DATE_FORMAT` session parameter:
Overview

PostgreSQL provides session-modifiable parameters that are configured using the \texttt{SET SESSION} command. Configuration of parameters using \texttt{SET SESSION} will only be applicable in the current session. To view the list of parameters that can be set with \texttt{SET SESSION}, you can query \texttt{pg_settings}:

\begin{verbatim}
SELECT * FROM pg_settings where context = 'user';
\end{verbatim}

Examples of commonly-used session parameters:

1. \texttt{client_encoding} - configures the connected client character set.
2. \texttt{force_parallel_mode} - forces use of parallel query for the session.
3. \texttt{lock_timeout} - sets the maximum allowed duration of time to wait for a database lock to release.
4. \texttt{search_path} - sets the schema search order for object names that are not schema-qualified.
5. \texttt{transaction_isolation} - sets the current Transaction Isolation Level for the session.

Example

\begin{verbatim}
SQL> select sysdate from dual;
SYSDATE 
--------
SEP-09-17

SQL> alter session set nls_date_format='DD-MON-RR';
Session altered.

SQL> select sysdate from dual;
SYSDATE 
--------
09-SEP-17

SQL> alter session set nls_date_format='MM-DD-YYYY';
Session altered.

SQL> select sysdate from dual;
SYSDATE 
--------
09-09-2017

SQL> alter session set nls_date_format='DAY-MON-RR';
Session altered.
\end{verbatim}
Change the Date format of the connected session:

```
mydb=> set session DateStyle to POSTGRES, DMY;
SET
mydb=> select now();
    now
-----------------------------
  Sat 09 Sep 11:03:43.597202 2017 UTC
(1 row)
```

```
mydb=> set session DateStyle to ISO, MDY;
SET
mydb=> select now();
    now
-----------------------------
  2017-09-09 11:04:01.3859+00
(1 row)
```

**Oracle vs. PostgreSQL Session parameter examples**

Please note that the list below is partial and is meant to highlight various session-level configuration parameters in both Oracle and PostgreSQL. Not all parameters are directly comparable.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure time and date format</td>
<td><code>ALTER SESSION SET nls_date_format = 'dd/mm/yyyy hh24:mi:ss';</code></td>
<td><code>SET SESSION datestyle to 'SQL, DMY';</code></td>
</tr>
<tr>
<td>Configure the current default schema/database</td>
<td><code>ALTER SESSION SET current_schema='schema_name'</code></td>
<td><code>SET SESSION SEARCH_PATH TO schemaname;</code></td>
</tr>
<tr>
<td>Generate traces for specific errors</td>
<td><code>ALTER SESSION SET events '10053 trace name context forever';</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Run trace for a SQL statement</td>
<td><code>ALTER SESSION SET sql_trace=TRUE;</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Modify query optimizer cost for index access</td>
<td><code>ALTER SESSION SET optimizer_index_cost_adj = 50</code></td>
<td><code>SET SESSION random_page_cost TO 6;</code></td>
</tr>
<tr>
<td>Modify query optimizer row access strategy</td>
<td><code>ALTER SESSION SET optimizer_mode=all_rows;</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Memory allocated to sort operations</td>
<td><code>ALTER SESSION SET sort_area_size=6321;</code></td>
<td><code>SET SESSION work_mem TO '6MB';</code></td>
</tr>
<tr>
<td>Memory allocated to hash-joins</td>
<td><code>ALTER SESSION SET hash_area_size=1048576000;</code></td>
<td><code>SET SESSION work_mem TO '6MB';</code></td>
</tr>
</tbody>
</table>

For additional details: [https://www.postgresql.org/docs/9.6/static/sql-set.html](https://www.postgresql.org/docs/9.6/static/sql-set.html)
Migrating from: Oracle Alert.log and logs files

Overview
The primary error log file for the Oracle database is known as the “Alert Log” with a file name that in the following format: “alert<SID>.log”. The Alert Log contains verbose information regarding the activity of the Oracle database including informational messages and errors. Each event includes a timestamp indicating when the event occurred.

When encountering database issues, the Oracle Alert Log is the first place to look for troubleshooting and to investigate errors, failures or for any other messages that might indicate a potential database problem.

Example

1. Partial contents of the Oracle database Alert Log File:

```
Sun Sep 03 13:27:23 2017
Starting ORACLE instance (normal)
********************** Large Pages Information **********************
Per process system memlock (soft) limit = 64 KB
Total Shared Global Region in Large Pages = 0 KB (0%)
Large Pages used by this instance: 0 (0 KB)
Large Pages unused system wide = 0 (0 KB)
Large Pages configured system wide = 0 (0 KB)
Large Page size = 2048 KB
```

Common events logged in the Alert Log include:

1. Database startup or shutdown.
2. Database redo log switch.
3. Database errors and warnings, starting with ORA- and followed by an Oracle error number.
4. Network and connection issues
5. Links for a detailed trace files regarding a specific database event

The Oracle Alert Log can be found inside the database Automatic Diagnostics Repository (ADR), a hierarchical file-based repository for diagnostic information:

```
$ADR_BASE/diag/rdbms/{DB-name}/{SID}/trace
```

In addition, several other Oracle server components have their own unique log files, such as the database listener, Automatic Storage Manager (ASM), etc.

For additional details:
https://docs.oracle.com/cd/B28359_01/server.111/b28310/diag005.htm#ADMIN11267
https://docs.oracle.com/database/121/SUTIL/GUID-E0FF3013-2EBF-4110-88BF-69E7DD2BBB7C.htm#SUTIL1474
PostgreSQL provides detailed logging and reporting of errors that occur during the database and connected sessions lifecycle. In an Amazon Aurora deployment, these informational and error messages are accessible using the Amazon RDS console.

**PostgreSQL vs. Oracle error codes**

Oracle error codes start with the “ORA-” prefix. PostgreSQL messages expressed by assigning five-character error codes divided by message class such as: successful completion, warning, no data and more.

<table>
<thead>
<tr>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-00001:</td>
<td>SQLSTATE[23505]: Unique violation: 7 ERROR: duplicate key value violates unique constraint &quot;constraint_name&quot;</td>
</tr>
<tr>
<td>unique constraint (string.string) violated</td>
<td></td>
</tr>
</tbody>
</table>

For additional details about PostgreSQL Error Codes:
https://www.postgresql.org/docs/9.6/static/errcodes-appendix.html

**Example**

Access the PostgreSQL error log using the Amazon RDS/Aurora Management Console:

- Navigate to: **Services > RDS > Instances > Select Instance**
  - Click **Logs**.
  - Select a specific PostgreSQL log file and select **View** to review a static version of log file. Optionally, select **Watch** for a dynamic (updating) view of log file.
Notes
1. You can use the search box to search for a specific log file.
2. You can click on the download button to download the log file to your local machine.

Partial contents of a PostgreSQL database error log as viewed from the Amazon RDS Management Console:

```
2017-09-14 07:03:40 UTC: started
2017-09-14 07:03:40 UTC: checkpoint starting: time
2017-09-14 07:03:40 UTC: checkpoint complete: wrote 1 buffers (0.0%); 0
transaction log file(s) added, 0 removed, 1 recycled; write=0.101 s, sync=0.008 s,
total=0.132 s; sync files=1, longest=0.008 s, average=0.008 s; distance=16384 kB,
estimate=16385 kB
2017-09-14 07:08:40 UTC: started
2017-09-14 07:08:40 UTC: checkpoint starting: time
2017-09-14 07:08:40 UTC: checkpoint complete: wrote 1 buffers (0.0%); 0
transaction log file(s) added, 0 removed, 1 recycled; write=0.101 s, sync=0.002 s,
total=0.114 s; sync files=1, longest=0.002 s, average=0.002 s; distance=16384 kB,
estimate=16385 kB
2017-09-14 07:13:41 UTC: started
2017-09-14 07:13:41 UTC: checkpoint starting: time
2017-09-14 07:13:41 UTC: checkpoint complete: wrote 1 buffers (0.0%); 0
transaction log file(s) added, 0 removed, 1 recycled; write=0.101 s, sync=0.002 s,
```

PostgreSQL error log configuration
Several parameters control how and where PostgreSQL log and errors files will be placed:

Common Amazon Aurora configuration options

<table>
<thead>
<tr>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_filename</td>
<td>Sets the file name pattern for log files.</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
<tr>
<td>log_rotation_age</td>
<td>(min) Automatic log file rotation will occur after N minutes.</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
<tr>
<td>log_rotation_size</td>
<td>(kB) Automatic log file rotation will occur after N kilobytes.</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
<tr>
<td>log_min_messages</td>
<td>Sets the message levels that are logged (DEBUG, ERROR, INFO, etc....).</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
<tr>
<td>log_min_error_statement</td>
<td>Causes all statements generating error at or above this level to be logged (DEBUG, ERROR, INFO, etc...).</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
<tr>
<td>log_min_duration_statement</td>
<td>Sets the minimum execution time above which statements will be logged (ms).</td>
</tr>
<tr>
<td></td>
<td><em>Modifiable via an Aurora Database Parameter Group</em></td>
</tr>
</tbody>
</table>

**Note**
Modifications to certain parameters, such as `log_directory` (which sets the destination directory for log files) or `logging_collector` (which start a subprocess to capture stderr output and/or csvlogs into log files) are disabled for Aurora PostgreSQL instance.

**Log severity levels supported by PostgreSQL:**

<table>
<thead>
<tr>
<th>Severity</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG1…DEBUG5</td>
<td>Provides successively-more-detailed information for use by developers</td>
</tr>
<tr>
<td>INFO</td>
<td>Provides information implicitly requested by the user</td>
</tr>
<tr>
<td>NOTICE</td>
<td>Provides information that might be helpful to users</td>
</tr>
<tr>
<td>WARNING</td>
<td>Provides warnings of likely problems</td>
</tr>
<tr>
<td>ERROR</td>
<td>Reports an error that caused the current command to abort</td>
</tr>
<tr>
<td>LOG</td>
<td>Reports information of interest to administrators</td>
</tr>
<tr>
<td>FATAL</td>
<td>Reports an error that caused the current session to abort</td>
</tr>
<tr>
<td>PANIC</td>
<td>Reports an error that caused all database sessions to abort</td>
</tr>
</tbody>
</table>

*For additional details about PostgreSQL Error Reporting and Logging:*

[https://www.postgresql.org/docs/9.6/static/runtime-config-logging.html](https://www.postgresql.org/docs/9.6/static/runtime-config-logging.html)
Overview
Table statistics are one of the important aspects that can affect SQL query performance. Table Statistics allow the query optimizer to make informed assumptions when deciding how to generate the execution plan for each query. Oracle provides the `DBMS_STATS` package to manage and control the table statistics which can be collected automatically or manually.

The following statistics are usually collected on database tables and indexes:
- Number of table rows.
- Number of table blocks.
- Number of distinct values or nulls.
- Data distribution histograms.

Automatic Optimizer Statistics Collection
By default, Oracle will collect table and index statistics by using automated maintenance tasks leveraging the database scheduler to automatically collect statistics at predefined maintenance windows. Using the data modification monitoring feature in Oracle, which is responsible for tracking the approximate number of INSERTs, UPDATEs, and DELETEs for that table, the automatic statistics collection mechanism knows which table statistics should be collected.

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:

1. `GATHER_INDEX_STATS` Index statistics
2. `GATHER_TABLE_STATS` Table, column, and index statistics
3. `GATHER_SCHEMA_STATS` Statistics for all objects in a schema
4. `GATHER_DICTIONARY_STATS` Statistics for all dictionary objects
5. `GATHER_DATABASE_STATS` Statistics for all objects in a database

Example
1. Collecting statistics at the table level (schema - HR, table - EMPLOYEES):

```sql
SQL> BEGIN
    DBMS_STATS.GATHER_TABLE_STATS('HR','EMPLOYEES');
END;
/
PL/SQL procedure successfully completed.
```
2. Collecting statistics at a specific column-level (schema - HR, table - EMPLOYEES, column - DEPARTMENT_ID):

SQL> BEGIN  
    DBMS_STATS.GATHER_TABLE_STATS('HR','EMPLOYEES',  
      METHOD_OPT=>'FOR COLUMNS department_id');  
END;  
/

PL/SQL procedure successfully completed.

For additional information on Oracle Collecting Table Statistics:
http://docs.oracle.com/cd/E25054_01/server.1111/e16638/stats.htm#i41448
https://docs.oracle.com/database/121/TGSQL/tgsql_stats.htm#TGSQL390

AWS Migration to: PostgreSQL Table Statistics

Overview
Use the ANALYZE command to collect statistics about a database, a table or a specific table column. The PostgreSQL ANALYZE command collects table statistics which support generation of efficient query execution plans by the query planner.

1. Histograms - ANALYZE will collect statistics on table columns values and create a histogram of the approximate data distribution in each column.
2. Pages and rows - ANALYZE will collect statistics on the number of database pages and rows from which each table is comprised.
3. Data sampling - for large tables, the ANALYZE command will take random samples of values rather than examining each and every single row. This allows the ANALYZE command to scan very large tables in a relatively small amount of time.
3. Statistic collection granularity - executing the ANALYZE command without any parameter will instruct PostgreSQL to examine every table in the current schema. Supplying the table name or column name to the ANALYZE, will instruct the database to examine a specific table or table column.

Automatic Statistics Collection
By default, PostgreSQL is configured with an “autovacuum daemon” which automates the execution of statistics collection via the ANALYZE commands (in addition to automation of the VACUUM command). The “autovacuum daemon” scans for tables which show signs of large modifications in data to collect the current statistics. Autovacuum is controlled by several parameters.

For additional details:
https://www.postgresql.org/docs/9.6/static/runtime-config-autovacuum.html
Manual Statistics Collection

PostgreSQL allows collecting statistics on-demand using the `ANALYZE` command at a database level, table-level or table column-level.

1. `ANALYZE` on indexes is not currently supported.
2. `ANALYZE` requires only a read-lock on the target table, so it can run in parallel with other activity on the table.
3. For large tables, `ANALYZE` takes a random sample of the table contents. Configured via the `show default_statistics_target` parameter. The default value is 100 entries. Raising the limit might allow more accurate planner estimates to be made at the price of consuming more space in the `pg_statistic` table.

Examples

1. Gather statistics for the entire database:
   
   ```
   psql=> ANALYZE;
   ```

2. Gather statistics for a specific table. The `VERBOSE` keyword displays progress.
   
   ```
   psql=> ANALYZE VERBOSE EMPLOYEES;
   ```

3. Gather statistics for a specific column:
   
   ```
   psql=> ANALYZE EMPLOYEES (HIRE_DATE);
   ```

4. Specify the `default_statistics_target` parameter for an individual table column and reset it back to default:
   
   ```
   psql=> ALTER TABLE EMPLOYEES ALTER COLUMN SALARY SET STATISTICS 150;
   psql=> ALTER TABLE EMPLOYEES ALTER COLUMN SALARY SET STATISTICS -1;
   ```

Larger values will increase the time needed to complete an `ANALYZE`, but, will improve the quality of the collected planner's statistics which can potentially lead to better execution plans.

5. View the current (session / global) `default_statistics_target`, modify it to 150 and analyze the `EMPLOYEES` table:
   
   ```
   psql=> SHOW default_statistics_target ;
   psql=> SET default_statistics_target to 150;
   psql=> ANALYZE EMPLOYEES ;
   ```

6. View the last time statistics were collected for a table:
   
   ```
   select relname, last_analyze from pg_stat_all_tables;
   ```
## Comparing Oracle and PostgreSQL Statistics Collection

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analyze a specific database table</strong></td>
<td>BEGIN dbms_stats.gather_table_stats(ownname =&gt; 'hr', tabname =&gt; 'employees', ... ); END;</td>
<td>ANALYZE EMPLOYEES;</td>
</tr>
<tr>
<td><strong>Analyze a database table while only sampling certain rows</strong></td>
<td>Configure via percentage of table rows to sample:</td>
<td>Configure via number of entries for the table:</td>
</tr>
<tr>
<td></td>
<td>BEGIN dbms_stats.gather_table_stats(ownname=&gt;'HR', ... ESTIMATE_PERCENT=&gt;100) ; END;</td>
<td>SET default_statistics_target to 150; ANALYZE EMPLOYEES ;</td>
</tr>
<tr>
<td><strong>Collect statistics for a schema</strong></td>
<td>BEGIN EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS(ownname =&gt; 'HR'); END</td>
<td>ANALYZE;</td>
</tr>
<tr>
<td><strong>View last time statistics were collected</strong></td>
<td>select owner,table_name,last_analyzed;</td>
<td>select relname, last_analyze from pg_stat_all_tables;</td>
</tr>
</tbody>
</table>

*For additional information on PostgreSQL Collecting Table Statistics:*

https://www.postgresql.org/docs/9.6/static/sql-analyze.html

https://www.postgresql.org/docs/9.6/static/routine-vacuuming.html#AUTOVACUUM
Overview

Execution plans represent the choices made by the query optimizer of which actions to perform in order to access data in the database. Execution Plans are generated by the database optimizer for SELECT, INSERT, UPDATE and DELETES statements.

Users and DBAs can request the database to present the execution plan for any specific query or DML operation providing an extensive view on the optimizer’s method of accessing data. Execution Plans are especially useful for performance tuning of queries, including deciding if new indexes should be created. Execution plans can be affected by data volumes, data statistics and instance parameters (global or session parameters).

Execution plans are displayed as a structured tree that presents the following information:
1. Tables access by the SQL statement and the referenced order for each table.
2. Access method for each table in the statement (full table scan vs. index access).
3. Algorithms used for joins operations between tables (hash vs. nested loop joins).
4. Operations that are performed on retrieved data as filtering, sorting and aggregations.
5. Information about rows begin processed (cardinality) and the cost for each operation.
6. Table partitions begin accessed.
7. Information about parallel executions.

Examples

1. Review the potential execution plan for a query using the EXPLAIN PLAN statement:

   ```
   SQL> SET AUTOT TRACE EXP
   SQL> SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
              WHERE LAST_NAME='King' AND FIRST_NAME='Steven';
   Execution Plan
   ----------------------------------------------------------
   Plan hash value: 2077747057
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMPLOYEES</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_NAME_IX</td>
<td>1</td>
<td></td>
<td>1</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

   Predicate Information (identified by operation id):
   -----------------------------------------------
   2 - access("LAST_NAME"='King' AND "FIRST_NAME"='Steven')
   ```

* SET AUTOT TRACE EXP instructs SQL*PLUS to show the execution plan without actually running the query itself.
The EMPLOYEES tables contains indexes for both the LAST_NAME and the FIRST_NAME columns, we can see that in step 2 of the execution plan above, the optimizer is performing an INDEX RANGE SCAN in order to retrieve the filtered employee name.

2. View a different execution plan, this time showing a FULL TABLE SCAN:

```sql
SQL> SET AUTOT TRACE EXP
SQL> SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
     WHERE SALARY > 10000;
```

Execution Plan
-------------------------------
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>72</td>
<td>1368</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 1</td>
<td>TABLE ACCESS FULL</td>
<td>EMPLOYEES</td>
<td>72</td>
<td>1368</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
-----------------------------------------------
1 - filter("SALARY">10000)

For additional details:
http://docs.oracle.com/cd/E25178_01/server.1111/e16638/ex_plan.htm
https://docs.oracle.com/database/121/TGSQL/tgsql_genplan.htm#TGSQL271
Overview

The PostgreSQL equivalent to EXPLAIN PLAN in the Oracle database is the EXPLAIN keyword which is used to display the execution plan for a supplied SQL statement. In similar manner to Oracle, the query planner in PostgreSQL will generate the estimated execution plan for actions such as: SELECT, INSERT, UPDATE and DELETE and will build a structured tree of plan nodes representing the different actions taken (the sign “–>” represent a root line in the PostgreSQL execution plan). In addition, the EXPLAIN statement will provide statistical information regarding each action such as: cost, rows, time and loops.

When using the EXPLAIN command as part of a SQL statement, the statement will not execute and the execution plan would be an estimation. However, by using the EXPLAIN ANALYZE command, the statement will actually be executed in addition to displaying the execution plan itself.

PostgreSQL EXPLAIN Synopsis:

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

where option can be one of:

- ANALYZE [ boolean ]
- VERBOSE [ boolean ]
- COSTS [ boolean ]
- BUFFERS [ boolean ]
- TIMING [ boolean ]
- FORMAT { TEXT | XML | JSON | YAML }

Examples

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

   ```
   psql=> EXPLAIN
   SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
   WHERE LAST_NAME='King' AND FIRST_NAME='Steven';
   ------------------------------------------------------------------------------------------
   Index Scan using idx_emp_name on employees  (cost=0.14..8.16 rows=1 width=18)
   Index Cond: (((last_name)::text = 'King'::text) AND ((first_name)::text = 'Steven'::text))
   (2 rows)
   ```

2. Running the same statement with the ANALYZE keyword:

   ```
   psql=> EXPLAIN ANALYZE
   SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
   WHERE LAST_NAME='King' AND FIRST_NAME='Steven';
   ------------------------------------------------------------------------------------------
   Index Scan using idx_emp_name on employees  (cost=0.14..8.16 rows=1 width=18)
   Index Cond: (((last_name)::text = 'King'::text) AND ((first_name)::text = 'Steven'::text))
   (2 rows)
   ```
By adding the `ANALYZE` keyword and executing the statement, we get additional information in addition to the execution plan.

### 3. Viewing a PostgreSQL execution plan showing a FULL TABLE SCAN:

```
psql=> EXPLAIN ANALYZE
    SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
    WHERE LAST_NAME='King' AND FIRST_NAME='Steven';
```

```
Seq Scan on employees  (cost=0.00..3.60 rows=1 width=18) (actual time=0.012..0.024 rows=1 loops=1)
  Filter: (((last_name)::text = 'King'::text) AND ((first_name)::text = 'Steven'::text))
  Rows Removed by Filter: 106
  Planning time: 0.073 ms
  Execution time: 0.037 ms
(5 rows)
```

```
psql=> EXPLAIN ANALYZE
    SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
    WHERE SALARY > 10000;
```

```
Seq Scan on employees  (cost=0.00..3.34 rows=15 width=18) (actual time=0.012..0.036 rows=15 loops=1)
  Filter: (salary > '10000'::numeric)
  Rows Removed by Filter: 92
  Planning time: 0.069 ms
  Execution time: 0.052 ms
(5 rows)
```

PostgreSQL can perform several scan types for processing and retrieving data from tables including: sequential scans, index scans, and bitmap index scans. The sequential scan (“Seq Scan”) is PostgreSQL equivalent for Oracle “Table access full” (full table scan).

For additional information on PostgreSQL Execution Plans:
[https://www.postgresql.org/docs/9.6/static/sql-explain.html](https://www.postgresql.org/docs/9.6/static/sql-explain.html)
Migrating from: Oracle SecureFile LOBs

Overview

LOBs – or Large Objects is a mechanism for storing binary data inside the Oracle database. Oracle 11g introduced a new data type for storing Large Objects (LOBs) binary files directly inside the database using more efficient storage. This feature is known as Secure File Lobs and implemented using the `SECUREFILE` keyword as part of the `CREATE TABLE` statement.

Primary benefits of using `SECUREFILE` lobs include:

- **Compression**
  With Oracle advanced compression utilized to analyze the SecureFiles LOB data to save disk space.

- **De-Duplication**
  Automatically detect duplicate LOB data within a LOB column or partition and by removing duplicates of repeating binary data, reduce storage space.

- **Encryption**
  Combined with Transparent Data Encryption (TDE).

Examples

1. Create a table using a SecureFiles LOB column:

   ```sql
   SQL> CREATE TABLE sf_tab (
           COL1    NUMBER,
           COL2_CLOB   CLOB)
       LOB(COL2_CLOB) STORE AS SECUREFILE;
   ```

2. Provide additional options for LOB compression during table creation:

   ```sql
   SQL> CREATE TABLE sf_tab (
           COL1    NUMBER,
           COL2_CLOB   CLOB)
       LOB(COL2_CLOB) STORE AS SECUREFILE COMPRESS_LOB(COMPRESS HIGH);
   ```

For additional details:

https://docs.oracle.com/cd/E11882_01/appdev.112/e18294/adlob_smart.htm#ADLOB45944

https://docs.oracle.com/database/121/ADLOB/adlob_smart.htm#ADLOB4444
Overview
PostgreSQL does not support the advanced storage, security, and encryption options of Oracle SecureFile LOBs. Regular Large Objects datatypes (LOBs) are supported by PostgreSQL and provides stream-style access.

Although not designed specifically from LOB columns, for compression PostgreSQL utilizes an internal TOAST mechanism (The Oversized-Attribute Storage Technique).

For more details about PostgreSQL please use the following link: https://www.postgresql.org/docs/9.4/static/storage-toast.html

Supported large objected Data Types by PostgreSQL are:

- **BYTEA**
  - Stores a LOB within the table limited to 1GB.
  - The storage is octal and supports non printable characters.
  - The input / output format is HEX.
  - Can be used to store a URL references to an AWS S3 objects used by the database. For example: storing the URL for pictures stored on AWS S3 on a database table.

- **TEXT**
  - Data type for storing strings with unlimited length.
  - When not specifying the (n) integer for specifying the varchar data type, the TEXT datatype behaves as the text data type.

For data encryption purposes (not only for LOB columns), consider using AWS KMS: https://aws.amazon.com/kms/

For additional information on PostgreSQL LOB Support: https://www.postgresql.org/docs/current/static/largeobjects.html