



**Oracle Database 19c
To Amazon Aurora with PostgreSQL Compatibility (12.4)**

Migration Playbook

2.4 April 2021

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved.

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.

Table of Contents

Introduction	9
Tables of Feature Compatibility	11
What's New	20
AWS Migration Tools and Service	22
AWS Schema Conversion Tool (SCT)	23
SCT Action Code Index	32
AWS Database Migration Service (DMS)	45
Amazon RDS on Outposts	47
Amazon Aurora Backtrack	48
Amazon RDS Proxy	52
Amazon Aurora Serverless v1	53
Migration Quick Tips	57
SQL & PL/SQL	59
OracleSingle-Row and Aggregate Functions vs. PostgreSQL Single-Row and Aggregate Functions	60
PostgreSQL Usage	60
Oracle Create Table as Select (CTAS) vs. PostgreSQL Create Table as Select (CTAS)	68
PostgreSQL Usage	68
Oracle Common Table Expression (CTE) vs. PostgreSQL Common Table Expressions (CTE)	70
PostgreSQL Usage	70
Oracle Identity Columns vs. PostgreSQL SERIAL Type	72
PostgreSQL Usage	72
Oracle Insert From Select vs. PostgreSQL Insert From Select	75
PostgreSQL Usage	76
Oracle Multi-Version Concurrency Control (MVCC) vs. PostgreSQL Multi-Version Concurrency Control (MVCC)	78
PostgreSQL Usage	80
Oracle Merge vs PostgreSQL Merge	87
PostgreSQL Usage	88
Oracle OLAP Functions vs. PostgreSQL Window Functions	90
PostgreSQL Usage	90
Oracle Sequences vs PostgreSQL Sequences	95
PostgreSQL Usage	98

Oracle Transaction Model vs. PostgreSQL Transactions	102
PostgreSQL Usage	104
Oracle Anonymous Block vs. PostgreSQL DO	109
PostgreSQL Usage	110
Oracle PL/SQL Cursors vs. PostgreSQL Cursors	112
PostgreSQL Usage	113
Oracle DBMS_OUTPUT vs. PostgreSQL RAISE	120
PostgreSQL Usage	121
Oracle DBMS_RANDOM vs. PostgreSQL RANDOM Function	123
PostgreSQL Usage	124
Oracle DBMS_SQL Package (11g & 12c) vs. PostgreSQL Dynamic Execution	126
PostgreSQL Usage	127
Oracle EXECUTE IMMEDIATE vs. PostgreSQL EXECUTE and PREPARE	129
PostgreSQL Usage	130
Oracle Procedures and Functions vs. PostgreSQL Stored Procedures	133
PostgreSQL Usage	135
Oracle UDFs vs. PostgreSQL User Defined Functions	141
PostgreSQL Usage	142
Oracle UTL_FILE	144
PostgreSQL Usage	144
Oracle UTL_MAIL & UTL_SMTP (11g & 12c) vs. PostgreSQL Scheduled Lambda with SES	145
PostgreSQL Usage	146
Tables & Indexes	155
Case Sensitivity Differences for Oracle and PostgreSQL	156
Oracle Common Data Types vs. PostgreSQL Common Data Types	157
PostgreSQL Usage	160
Oracle Read-Only Tables and Partitions vs. PostgreSQL Aurora Replicas	165
PostgreSQL Usage	166
Oracle Table Constraints vs. PostgreSQL Table Constraints	168
PostgreSQL Usage	175
Oracle Temporary Tables vs. PostgreSQL Temporary Tables	182
PostgreSQL Usage	184
Oracle Triggers vs. PostgreSQL Trigger Procedure	187

PostgreSQL Usage	188
Oracle Tablespaces & Data Files vs. PostgreSQL Tablespaces and Data files	195
PostgreSQL Usage	196
Oracle User Defined Types vs. PostgreSQL User Defined Types	201
PostgreSQL Usage	202
Oracle Unused Columns vs. PostgreSQL Alter Table	205
PostgreSQL Usage	205
Oracle Virtual Columns vs. PostgreSQL Views and Functions	207
PostgreSQL Usage	208
Overall Indexes Summary	212
Oracle Bitmap Indexes vs. PostgreSQL Bitmap	215
PostgreSQL Usage	215
Oracle B-Tree Indexes vs. PostgreSQL B-Tree Indexes	216
PostgreSQL Usage	216
Oracle Composite Indexes vs. PostgreSQL Multi-Column Indexes	218
PostgreSQL Usage	218
Oracle Function-Based Indexes vs. PostgreSQL Expression Indexes	220
PostgreSQL Usage	220
Oracle Invisible Indexes vs. PostgreSQL Invisible Indexes	222
PostgreSQL Usage	223
Oracle Index-Organized Table (IOT) vs. PostgreSQL “Cluster” Table	224
PostgreSQL Usage	225
Oracle Local and Global Partitioned Indexes vs. PostgreSQL Partitioned Indexes	227
PostgreSQL Usage	228
Oracle Automatic Indexing vs. Self-Managed PostgreSQL	230
PostgreSQL Usage	231
Special Features & Future Content	233
Oracle Character Sets vs. PostgreSQL Encoding	234
PostgreSQL Usage	235
Oracle Database Links vs. PostgreSQL DBLink and FDWrapper	239
PostgreSQL Usage	240
Oracle DBMS_SCHEDULER vs. PostgreSQL Scheduled Lambda	245
PostgreSQL Usage	248

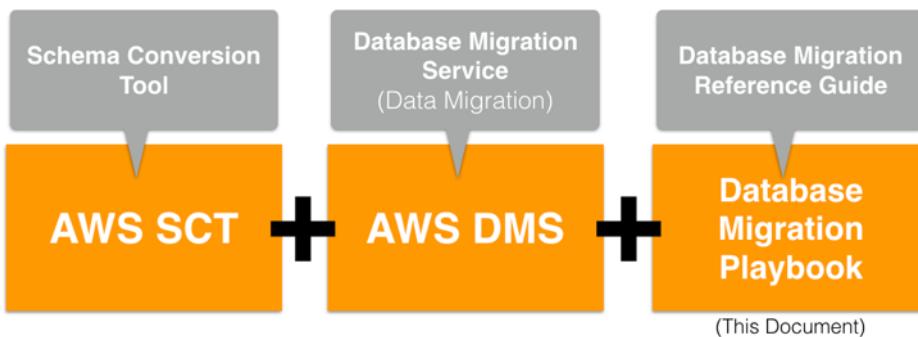
Oracle External tables vs. PostgreSQL Integration with Amazon S3	249
PostgreSQL Usage	250
Oracle Inline Views vs. PostgreSQL Inline Views	253
PostgreSQL Usage	253
Oracle JSON Document Support vs. PostgreSQL JSON Support	254
PostgreSQL Usage	255
Oracle Materialized Views vs. PostgreSQL Materialized Views	259
PostgreSQL Usage	261
Oracle Multitenant vs. PostgreSQL Databases	263
PostgreSQL Usage	266
Oracle Resource Manager vs. PostgreSQL Dedicated Amazon Aurora Clusters	269
PostgreSQL Usage	270
Oracle SecureFile LOBs vs. PostgreSQL Large Objects (LOBs)	275
PostgreSQL Usage	275
Oracle Views vs. PostgreSQL Views	277
PostgreSQL Usage	278
Oracle XML DB vs. PostgreSQL XML Type & Functions	281
PostgreSQL Usage	286
Oracle Log Miner vs. PostgreSQL Logging Options	290
PostgreSQL Usage	291
High Availability and Disaster Recovery (HADR)	296
Oracle Active Data Guard vs. PostgreSQL Replicates	297
PostgreSQL Usage	298
Oracle Real Application Clusters (RAC) vs. PostgreSQL Aurora Architecture	302
PostgreSQL Usage	304
Oracle Traffic Director vs. Amazon RDS Proxy for Amazon Aurora PostgreSQL	309
PostgreSQL Usage	309
Oracle Data Pump vs. PostgreSQL pg_dump & pg_restore	311
PostgreSQL Usage	312
Oracle Flashback Database vs. PostgreSQL Amazon Aurora Snapshots	315
PostgreSQL Usage	315
Oracle Flashback Table vs. PostgreSQL Amazon Aurora Snapshots (Flashback Tables)	321
PostgreSQL Usage	322

Oracle Recovery Manager (RMAN) vs. Amazon RDS Snapshots	324
PostgreSQL Usage	325
Oracle SQL*Loader vs. PostgreSQL pg_dump/pg_restore	328
PostgreSQL Usage	329
Configuration	330
Oracle Upgrades vs. Aurora for PostgreSQL upgrades	331
Aurora for PostgreSQL Usage	332
Oracle Alert.log and logs files vs. PostgreSQL Error Log via Amazon RDS Console	337
PostgreSQL Usage	338
Oracle SGA & PGA Memory Sizing vs. PostgreSQL Memory Buffers	342
PostgreSQL Usage	343
Oracle Instance Parameters vs. AWS RDS Parameter Groups	346
PostgreSQL Usage	346
Oracle Session Parameters vs. PostgreSQL Session Parameters	351
PostgreSQL Usage	352
Performance Tuning	354
Oracle Database Hints vs. PostgreSQL DB Query Planning	355
PostgreSQL Usage	356
Oracle Execution Plans vs. PostgreSQL Execution Plans	357
PostgreSQL Usage	358
Oracle Table Statistics vs. PostgreSQL Table Statistics	363
PostgreSQL Usage	364
Security	367
Oracle Encryption - TDE vs. PostgreSQL AWS Aurora Encryption	368
PostgreSQL Usage	371
Oracle Roles vs. PostgreSQL Roles	376
PostgreSQL Usage	377
Oracle Database Users vs. PostgreSQL Users	381
PostgreSQL Usage	382
Physical Storage	383
Oracle Table Partitioning vs. PostgreSQL Partitions and Table Inheritance	384
PostgreSQL Usage	387
Oracle Sharding vs. PostgreSQL Sharding	401

Monitoring	402
Oracle V\$ Views and the Data Dictionary vs. PostgreSQL System Catalog & The Statistics Collector	403
PostgreSQL Usage	404
Migration Quick Tips	410
Migration Quick Tips	411
Glossary	413

Introduction

The migration process from a source database (Oracle or SQL Server) to Amazon Aurora (PostgreSQL or MySQL) typically involves several stages. The first stage is to use the AWS Schema Conversion Tool (SCT) and the AWS Database Migration Service (DMS) to convert and migrate the schema and data. While most of the migration work can be automated, some aspects require manual intervention and adjustments to both database schema objects and database code.



The purpose of this Playbook is to assist administrators tasked with migrating from source databases to Aurora with the aspects that can't be automatically migrated using the Amazon Web Services Schema Conversion Tool (AWS SCT). It focuses on the differences, incompatibilities, and similarities between the source database and Aurora in a wide range of topics including T-SQL, Configuration, High Availability and Disaster Recovery (HADR), Indexing, Management, Performance Tuning, Security, and Physical Storage.

The first section of this document provides [an overview of AWS SCT](#) and the [AWS Data Migration Service \(DMS\)](#) tools for automating the migration of schema, objects and data. The remainder of the document contains individual sections for the source database features and their Aurora counterparts. Each section provides a short overview of the feature, examples, and potential workaround solutions for incompatibilities.

You can use this playbook either as a reference to investigate the individual action codes generated by the AWS SCT tool, or to explore a variety of topics where you expect to have some incompatibility issues. When using the AWS SCT, you may see a report that lists *Action codes*, which indicates some manual conversion is required, or that a manual verification is recommended. For your convenience, this Playbook includes an [SCT Action Code Index](#) section providing direct links to the relevant topics that discuss the manual conversion tasks needed to address these action codes. Alternatively, you can explore the [Tables of Feature Compatibility](#) section that provides high-level graphical indicators and descriptions of the feature compatibility between the source database and Aurora. It also includes a graphical compatibility indicator and links to the actual sections in the playbook.

There is appendix at the end of this playbook: [Appendix: Migration Quick Tips](#) provides a list of tips for administrators or developers who have little experience with Aurora (PostgreSQL or MySQL). It briefly highlights key differences between the source database and Aurora that they are likely to encounter.

Note that not all of the source database features are fully compatible with Aurora or have simple workarounds. From a migration perspective, this document does not yet cover all source database features and capabilities. This first release focuses on some of the most important features and will be expanded over time.

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. 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 may occur from the information contained in this document.

Tables of Feature Compatibility

Feature Compatibility Legend

Compatibility Score Symbol	Description
	Very high compatibility: None or minimal low-risk and low-effort rewrites needed
	High compatibility: Some low-risk rewrites needed, easy workarounds exist for incompatible features
	Medium compatibility: More involved low-medium risk rewrites needed, some redesign may be needed for incompatible features
	Low compatibility: Medium to high risk rewrites needed, some incompatible features require redesign and reasonable-effort workarounds exist
	Very low compatibility: High risk and/or high-effort rewrites needed, some features require redesign and workarounds are challenging
	Not compatible: No practical workarounds yet, may require an application level architectural solution to work around incompatibilities

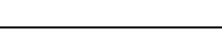
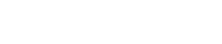
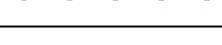
SCT/DMS Automation Level Legend

SCT/DMS Automation Level Symbol	Description
	Full Automation: Performs fully automatic conversion, no manual conversion needed.
	High Automation: Minor, simple manual conversions may be needed.
	Medium Automation: Low-medium complexity manual conversions may be needed.
	Low Automation: Medium-high complexity manual conversions may be needed.
	Very Low Automation: High risk or complex manual conversions may be needed.
	No Automation: Not currently supported, manual conversion is required for this feature.

SQL Language

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Aggregative Functions	Aggregative Functions	<ul style="list-style-type: none"> Not all functions are supported by PostgreSQL and may require to create manually 	 
Create Table As select (CTAS)	Create Table As select (CTAS)		 
Common Table Expression (CTE)	Common Table Expression (CTE)		 
Identity Column	Identity Column	<ul style="list-style-type: none"> Since PostgreSQL 10, there are no differences besides the data types 	 
Insert As Select	Insert As Select	<ul style="list-style-type: none"> ERROR LOG and subpartition options are not supported by PostgreSQL 	 
Locking	Locking	<ul style="list-style-type: none"> PostgreSQL uses auto commit by default (can be changed) 	
MERGE	MERGE	<ul style="list-style-type: none"> MERGE is not supported by PostgreSQL, workaround available 	 
OLAP Functions	OLAP Functions	<ul style="list-style-type: none"> GREATEST and LEAST functions might get different results in PostgreSQL CONNECT BY is not supported by PostgreSQL, workaround available 	 
Sequences	Sequences	<ul style="list-style-type: none"> Different syntax for few options in PostgreSQL 	 
Transactions	Transactions	<ul style="list-style-type: none"> PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions 	 

Tables

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Data Types	Data Types	<ul style="list-style-type: none"> BFILE, ROWID, UROWID are not supported by PostgreSQL 	 
Read Only	Read Only	<ul style="list-style-type: none"> READ ONLY mode is not supported by PostgreSQL, should use workaround 	 
Constraints	Constraints	<ul style="list-style-type: none"> REF, ENABLE/DISABLE are not supported by PostgreSQL Constraints on views are not supported by PostgreSQL 	 
Temp Table	Temp Table	<ul style="list-style-type: none"> GLOBAL temporary table is not supported by PostgreSQL Can't read from multiple sessions in PostgreSQL Table dropped after session ends in PostgreSQL 	 
Triggers	Triggers	<ul style="list-style-type: none"> Different paradigm and syntax System triggers are not supported by PostgreSQL 	 
Tablespaces	Tablespaces	<ul style="list-style-type: none"> All supported by PostgreSQL except managing the physical data files 	 
User Defined Types	User Defined Types	<ul style="list-style-type: none"> FORALL statement and DEFAULT option are not supported by PostgreSQL PostgreSQL doesn't support constructors of the "collection" type 	 
Unused Columns	Unused Columns	<ul style="list-style-type: none"> PostgreSQL does not support unused columns 	 
Virtual Columns	Virtual Columns		 

Configuration

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Alerting	Alerting	<ul style="list-style-type: none"> Use Event Notifications Subscription with Amazon Simple Notification Service (SNS) 	
Cache and Pools	Cache and Pools	<ul style="list-style-type: none"> Different cache names, similar usage 	
Database Parameters	Database Parameters	<ul style="list-style-type: none"> Use Cluster and Database/Cluster Parameters 	
Session Parameters	Session Parameters	<ul style="list-style-type: none"> SET options are significantly different in PostgreSQL 	

Special Features

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Character Set	Character Set	<ul style="list-style-type: none"> UTF16 character and NCHAR/NVARCHAR data types are not supported 	
Database Links	Database Links	<ul style="list-style-type: none"> Different paradigm and syntax 	
DBMS_SCHEDULER	DBMS_SCHEDULER	<ul style="list-style-type: none"> Use Scheduler AWS Lambda 	
External Tables	External Tables	<ul style="list-style-type: none"> PostgreSQL doesn't support EXTERNAL TABLEs 	
Inline Views	Inline Views		
JSON	JSON	<ul style="list-style-type: none"> Different paradigm and syntax will require application/drivers rewrite 	
Materialized Views	Materialized Views	<ul style="list-style-type: none"> PostgreSQL does not support automatic or incremental REFRESH 	

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Multi-tenant	Multi-tenant	<ul style="list-style-type: none"> Distribute load/applications/ users across multiple instances 	
Resource Manager	Resource Manager	<ul style="list-style-type: none"> Distribute load/applications/ users across multiple instances 	
Secure File and LOBs	Secure File and LOBs	<ul style="list-style-type: none"> SecureFiles are not supported by PostgreSQL, automation and compatibility refer only to LOBs 	
Views	Views		
XML	XML	<ul style="list-style-type: none"> Different paradigm and syntax will require application / drivers rewrite 	

High Availability and Disaster Recovery (HADR)

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Active Data Guard	Active Data Guard	<ul style="list-style-type: none"> Distribute load/applications/ users across multiple instances 	
RAC	RAC	<ul style="list-style-type: none"> Distribute load/applications/ users across multiple instances 	
Traffic Director	Traffic Director	<ul style="list-style-type: none"> Some features may be replaced by AWS RDS Proxy 	

Indexes

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Automatic Indexing	Automatic Indexing	<ul style="list-style-type: none"> PostgreSQL does not support Automatic Indexing 	
BITMAP	BITMAP	<ul style="list-style-type: none"> PostgreSQL does not support BITMAP index - BRIN index can be used in some cases 	

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
BTree	BTree		
Composite	Composite		
Function Based Index (FBI)	Function Based Index (FBI)	<ul style="list-style-type: none"> PostgreSQL doesn't support functional indexes that aren't single-column 	
Invisible Indexes	Invisible Indexes	<ul style="list-style-type: none"> PostgreSQL does not support Invisible Indexes 	
Index-Organized Table (IOT)	Index-Organized Table (IOT)	<ul style="list-style-type: none"> PostgreSQL does not support Index-Organized Tables, partially workaround available 	
Local and Global Indexes	Local and Global Indexes	<ul style="list-style-type: none"> PostgreSQL does not support domain index but behavior is like in Oracle 	

PL/SQL

Oracle	Aurora Post- greSQL	Key Differences	Compatibility
Anonymous Block	Anonymous Block	<ul style="list-style-type: none"> Different syntax may require code rewrite. 	
Conversion Functions	Conversion Functions	<ul style="list-style-type: none"> Not all functions are supported by PostgreSQL and may require to create manually 	
Cursors	Cursors	<ul style="list-style-type: none"> TYPE ... IS REF CURSOR is not supported by PostgreSQL Minor different in syntax may require some code rewrite 	

Oracle	Aurora Post-greSQL	Key Differences	Compatibility
		<ul style="list-style-type: none"> %ISOPEN, %BULK_EXCEPTIONS, and %BULK_ROWCOUNT are not supported 	
DBMS_OUTPUT	DBMS_OUTPUT	<ul style="list-style-type: none"> Different paradigm and syntax will require application/drivers rewrite. 	 
DBMS_RANDOM	DBMS_RANDOM	<ul style="list-style-type: none"> Different syntax may require code rewrite. 	 
DBMS_SQL	DBMS_SQL	<ul style="list-style-type: none"> Different paradigm and syntax will require application/drivers rewrite 	 
EXECUTE IMMEDIATE	EXECUTE IMMEDIATE	<ul style="list-style-type: none"> Different syntax may require code rewrite Remove IMMEDIATE keyword 	 
Procedures and Functions	Procedures and Functions	<ul style="list-style-type: none"> Syntax and option differences 	 
User Defined Functions	User Defined Functions	<ul style="list-style-type: none"> Syntax and option differences 	 
UTL_FILE	UTL_FILE	<ul style="list-style-type: none"> PostgreSQL does not have UTL_FILE equivalent 	 
UTL_MAIL & UTL_SMTP	UTL_MAIL & UTL_SMTP	<ul style="list-style-type: none"> Use Lambda Integration 	 

Performance Tuning

Oracle	Aurora Post-greSQL	Key Differences	Compatibility
Database Hints	Database Hints	<ul style="list-style-type: none"> Very limited set of hints in PostgreSQL- Index hints and optimizer 	

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
		<ul style="list-style-type: none"> hints as comments Syntax differences 	
Execution Plan	Execution Plan	<ul style="list-style-type: none"> Syntax differences Completely different optimizer with different operators and rules in PostgreSQL 	
Statistics	Statistics	<ul style="list-style-type: none"> Syntax and option differences, similar functionality 	

Physical Storage

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Partitions	Partitions	<ul style="list-style-type: none"> Foreign keys referencing to/from partitioned tables are supported on the individual tables in PostgreSQL Some partition types are not supported by PostgreSQL 	
Sharding	Sharding	<ul style="list-style-type: none"> PostgreSQL doesn't support sharding 	

Security

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Encryption	Encryption	<ul style="list-style-type: none"> Use AWS Aurora Encryption 	
Roles	Roles	<ul style="list-style-type: none"> Syntax and option differences, similar functionality There are no users - only roles in PostgreSQL 	
Users	Users	<ul style="list-style-type: none"> Syntax and option differences, similar functionality There are no users - only roles in PostgreSQL 	

Future content

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
LogMiner	LogMiner	<ul style="list-style-type: none"> PostgreSQL does not support LogMiner, workaround available 	

Backup and Recovery

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
DataPump	DataPump	<ul style="list-style-type: none"> • Non-compatible tool 	
Flashback Database	Flashback Database	<ul style="list-style-type: none"> • Storage level backup managed by Amazon RDS 	
Flashback Table	Flashback Table	<ul style="list-style-type: none"> • Storage level backup managed by Amazon RDS 	
RMAN	RMAN	<ul style="list-style-type: none"> • Storage level backup managed by Amazon RDS 	

Monitoring

Oracle	Aurora PostgreSQL	Key Differences	Compatibility
Information Views	Information Views	<ul style="list-style-type: none"> • Table names in queries need to be changed in PostgreSQL 	

What's New

The previous playbook covered the Aurora PostgreSQL 9.6 compatible and Oracle version 12c, this playbook will cover the main updates for Aurora PostgreSQL 12 compatible and Oracle version 19c.

AWS is keep on updating all the services, for the open-source databases, AWS is trying to keep up with the releases of the brand.

In order to provide the complete picture to all users, RDS related information is being mentioned in this playbook as well. Sections formatted in the following way are relevant for RDS only:

RDS ONLY: This paragraph is about the latest db engine version which is supported only in RDS (and not Aurora)

Update	Aspect	Link
Updated screen shots	AWS	AWS SCT
Updated SCT warning lists	AWS	SCT Error Code
Amazon RDS on Outposts	AWS	RDS Outposts
Amazon RDS Proxy	AWS	RDS Proxy
Amazon Aurora Serverless	AWS	Aurora Serverless
AWS RDS Backtrack	AWS	RDS Backtrack
New features for pg_dump support for extension dependencies pg_dump dump files can not only be plain text files but also with custom and compressed format support for WHERE in COPY	PostgreSQL	pg_dump
Online partition merging and partition strategy modification Oracle 19 Hybrid partitioned tables	Oracle	Partitions
Rewrite partitioning topic to present declarative partitions	PostgreSQL	Partitions
Monitoring index creation	PostgreSQL	Overall Indexes
Monitoring VACUUM FULL and CLUSTER operations Monitoring index creation New system views to monitor shared memory usage and ANALYZE progress	PostgreSQL	Information Views
New option AND CHAIN for COMMIT / ROLLBACK	PostgreSQL	Transactions
Generated columns	PostgreSQL	Virtual Columns
Unicode 9.0	Oracle	Character Set
New supported collation versions	PostgreSQL	Character Set
New ability to rename view columns	PostgreSQL	Views
Oracle private temporary tables	Oracle	Temporary Tables

Update	Aspect	Link
Oracle sharding	New topic	Sharding
Inline external tables	Oracle	External Tables
Traffic Director Mode of Connection Manager	New topic	Connection Manager
New Oracle (18c + 19c) PDB / CDB features like: CDB fleets, DBCA PDB Clone, PDB Clone switcher and more	Oracle	Container Databases
Copying databases using templates	PostgreSQL	Container Databases
Scalable sequences	Oracle	Sequences
New Oracle 19 function, JSON_SERIALIZE	Oracle	JSON
New Oracle 19 adding DISTINCT option for LISTAGG function and new bitmap functions to speed up COUNT DISTINCT	Oracle	Aggregate Functions
New topic of Automatic Indexing	Oracle	Automatic Indexing
Oracle 19 real-time statistics collection and High-Frequency Automatic Optimizer Statistics Collection	Oracle	Statistics
Table storage parameters that can trigger autovacuum	PostgreSQL	Statistics
Oracle 19 SQL Quarantine	Oracle	Execution Plans
Cancel queries automatically	PostgreSQL	Execution Plans
SQL*Loader	New topic	SQL*Loader
Case sensitive topic to call out the difference between the two engines	New topic	Case Sensitive
Upgrades differences	New topic	Upgrades
Amazon RDS Proxy	New topic	Connection Manager PostgreSQL

General migration tips topic has been added here: [link](#)

Use the following links to find out about other new Aurora specifics features [AWS Aurora News](#).

For additional details, see: <https://www.postgresql.org/docs/13/release-13.html>

AWS Migration Tools and Service

AWS Schema Conversion Tool (SCT)

Usage

The AWS Schema Conversion Tool (SCT) is a Java utility that connects to source and target databases, scans the source database schema objects (tables, views, indexes, procedures, etc.), and converts them to target database objects.

This section provides a step-by-step process for using AWS SCT to migrate an Oracle database to an Aurora PostgreSQL database cluster. Since AWS SCT can automatically migrate most of the database objects, it greatly reduces manual effort.

It is recommended to start every migration with the process outlined in this section and then use the rest of the Playbook to further explore manual solutions for objects that could not be migrated automatically. For more information, see

<http://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/Welcome.html>

Migrating a Database

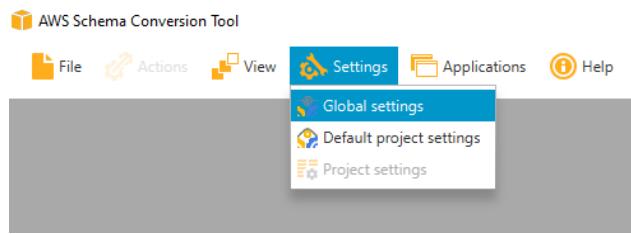
Note: This walkthrough uses the AWS DMS Sample Database. You can download it from <https://github.com/aws-samples/aws-database-migration-samples>.

Download the Software and Drivers

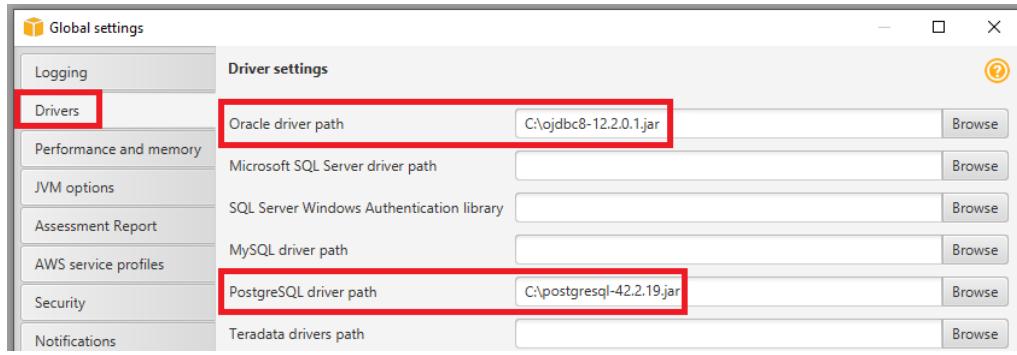
1. Download and install the AWS SCT from
https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Installing.html.
2. Download the relevant drivers:
 - Oracle
<http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html>
 - SQL Server
<https://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=11774>
 - PostgreSQL
<https://jdbc.postgresql.org/>
 - MySQL
<https://www.mysql.com/products/connector/>
 - Other link to supported drivers can be found in here:
https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Installing.html#CHAP_Installing.JDBCDrivers

Configure SCT

Launch SCT. Click the **Settings** button and select **Global Settings**.

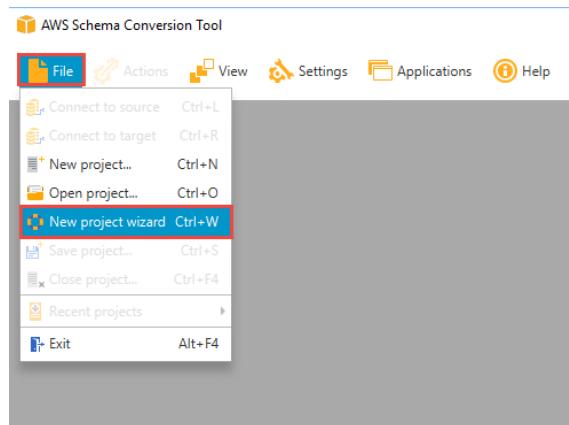


On the left navigation bar, click **Drivers**. Enter the paths for the Oracle and PostgreSQL drivers downloaded in the first step. Click **Apply** and then **OK**.

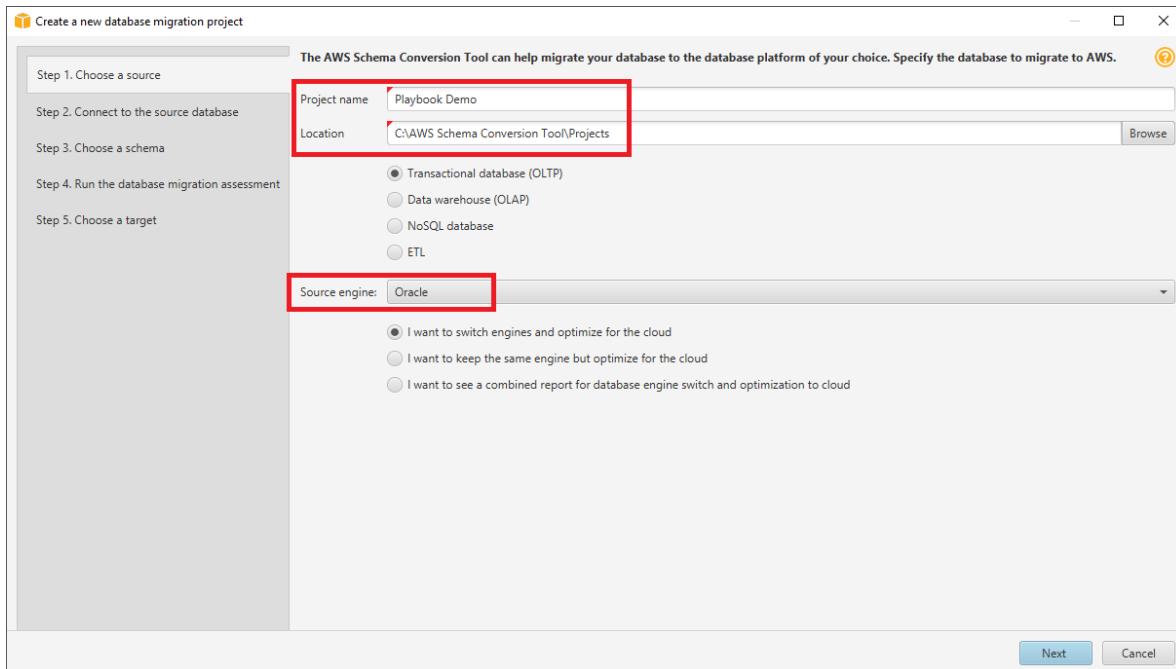


Create a New Migration Project

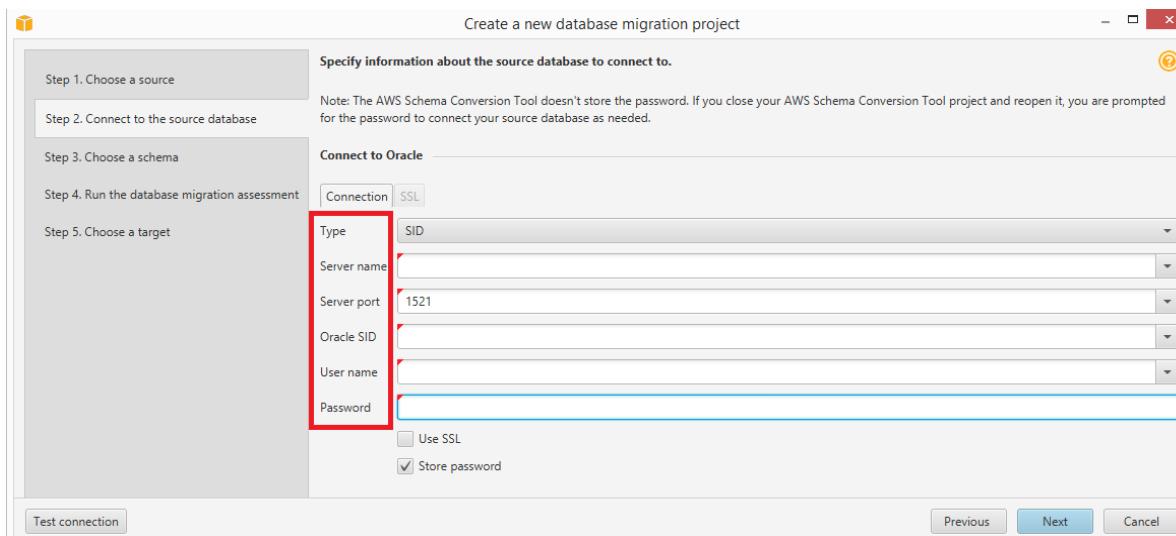
Click **File > New project wizard**. Alternatively, use the keyboard shortcut **<Ctrl+W>**.



Enter a project name and select a location for the project files. Click **Next**.



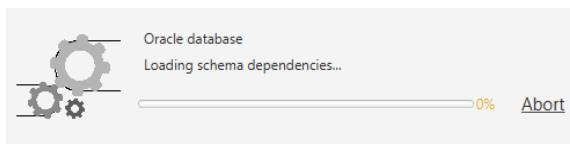
Enter connection details for the source Oracle database and click **Test Connection** to verify. Click **Next**.



Select the schema or database to migrate and click **Next**.

The screenshot shows the AWS Database Migration Service interface. On the left, a vertical navigation bar lists steps: Step 1. Choose a source, Step 2. Connect to the source database, Step 3. Choose a schema, Step 4. Run the database migration assessment, and Step 5. Choose a target. Step 3 is currently selected. The main panel title is "Choose the schema that you want to analyze." Below this, a tree view shows the "postgres" schema expanded, listing 24 schemas: ANONYMOUS, APPQOSSYS, AUDSYS, CTXSYS, DBSFWUSER, DBSNMP, DIP, DMS_SAMPLE, GSMADMIN_INTERNAL, GSMCATUSER, and GSMUSER. Each schema name is preceded by a checkmark.

The progress bar displays the objects being analyzed.



The Database Migration Assessment Report is displayed when the analysis completes. Read the Executive summary and other sections. Note that the information on the screen is only partial. To read the full report, including details of the individual issues, click **Save to PDF** and open the PDF document.

The screenshot shows the "Database Switch Assessment" report. At the top, there are "Save to CSV" and "Save to PDF" buttons. The main section is titled "Executive summary". It features a table comparing migration options across different target platforms. The table has four main columns: "Target platform", "Auto or minimal changes", and "Complex actions" (which further divides into "Storage objects" and "Code objects"). The "Auto or minimal changes" column includes "Storage objects" (e.g., 63 for MySQL) and "Code objects" (e.g., 12 for MySQL). The "Complex actions" column includes "Conversion actions" (e.g., 12 for MySQL), "Storage objects count" (e.g., 0 for MySQL), "Conversion actions" (e.g., 0 for MySQL), "Code objects count" (e.g., 11 for MySQL), and "Conversion actions" (e.g., 55 for MySQL). The "Complex actions" section also includes "Storage objects count" (e.g., 0 for MySQL), "Conversion actions" (e.g., 0 for MySQL), "Code objects count" (e.g., 3 for MySQL), and "Conversion actions" (e.g., 1 for MySQL). The "Complex actions" section also includes "Storage objects count" (e.g., 0 for MySQL), "Conversion actions" (e.g., 0 for MySQL), "Code objects count" (e.g., 3 for MySQL), and "Conversion actions" (e.g., 1 for MySQL). The "Complex actions" section also includes "Storage objects count" (e.g., 0 for MySQL), "Conversion actions" (e.g., 0 for MySQL), "Code objects count" (e.g., 10 for MySQL), and "Conversion actions" (e.g., 50 for MySQL).

Scroll down to the section **Database objects with conversion actions for Amazon Aurora (PostgreSQL compatible)**.

Of the total 63 database storage object(s) and 23 database code object(s) in the source database, we identified 63 (100%) database storage object(s) and 20 (87%) database code object(s) that can be converted to Amazon Aurora (PostgreSQL compatible) automatically or with minimal changes.

3 (13%) database code object(s) require 1 complex user action(s) to complete the conversion.

Figure: Conversion statistics for database storage objects

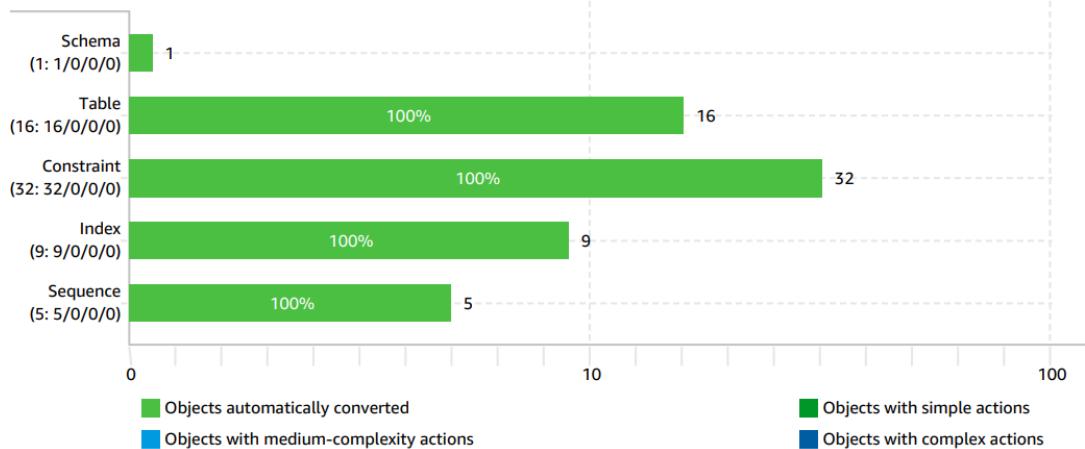
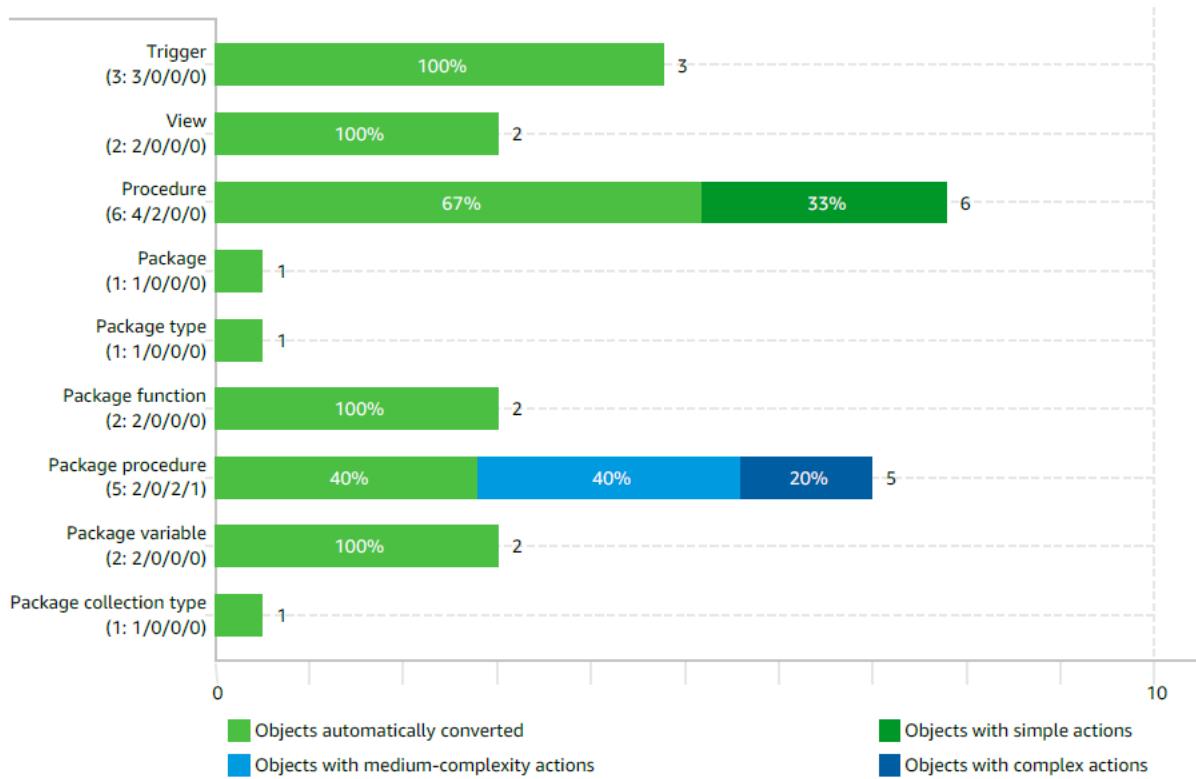
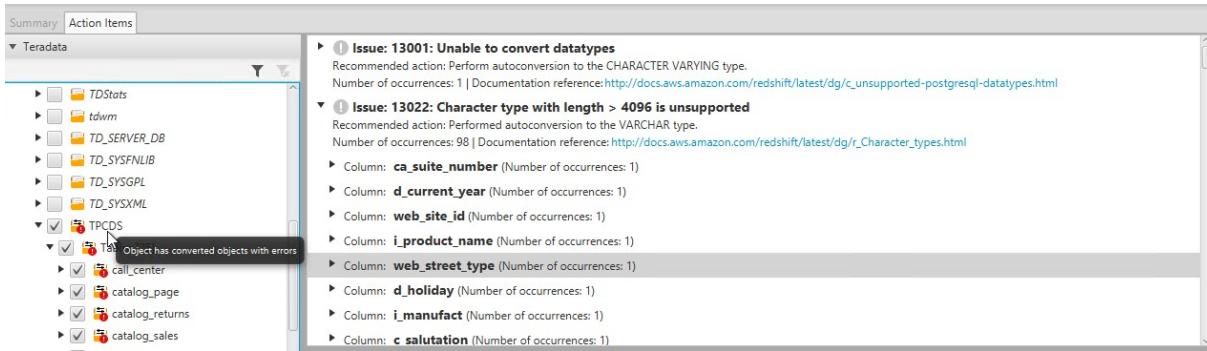


Figure: Conversion statistics for database code objects



Scroll further down to the section **Detailed recommendations for Amazon Aurora (PostgreSQL compatible) migrations**.



Return to AWS SCT and click **Next**. Enter the connection details for the target Aurora PostgreSQL database and click **Finish**.

Note: The changes have not yet been saved to the target.

This screenshot shows the 'Step 5. Choose a target' screen of the AWS SCT wizard. On the left, a sidebar lists steps: Step 1. Choose a source, Step 2. Connect to the source database, Step 3. Choose a schema, Step 4. Run the database migration assessment, and Step 5. Choose a target. The main panel is titled 'Specify the target database engine and the connection information.' It shows the 'Target engine:' set to 'Amazon Aurora (PostgreSQL compatible)'. Below this, under 'Connect to Amazon Aurora (PostgreSQL compatible)', the 'Connection' tab is selected. The connection details are as follows:

Server name	pg-playbooks-instance-1.cr77a85iv8n.eu-central-1.rds.amazonaws.com
Server port	5432
Database	pgplaybooks
User name	postgres
Password	[REDACTED]

Below the password field are two checkboxes: 'Use SSL' (unchecked) and 'Store password' (unchecked). At the bottom of the panel are 'Test connection' and 'Finish' buttons.

When the connection is complete, AWS SCT displays the main window. In this interface, you can explore the individual issues and recommendations discovered by AWS SCT.

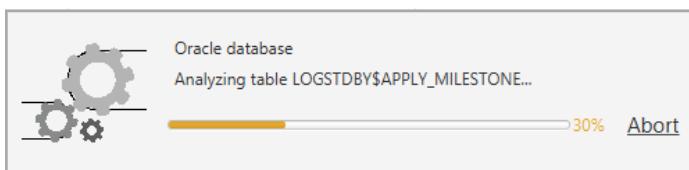
For example, expand **sample database > dms sample > Procedures > generate_tickets**. This issue has a red marker indicating it could not be automatically converted and requires a manual code change (issue 811 above). Select the object to highlight the incompatible code section.

The screenshot shows the AWS Schema Conversion Tool interface. On the left, the Oracle schema tree includes objects like DIP, DMS_SAMPLE, Tables, Views, Packages, Procedures, GENERATESEATS, and GENERATETICKETS. In the center, the Oracle procedure 'GENERATE_TICKETS' is displayed with its SQL code. On the right, the Amazon Aurora (PostgreSQL compatible) schema tree includes Schemas, Foreign Servers, Server Level Objects, Contexts, and SQL Scripts.

Right-click the schema and then click **Create Report** to create a report tailored for the target database type. It can be viewed in AWS SCT.

The screenshot shows the Oracle schema tree with the 'DMS SAMPLE' schema selected. A context menu is open, and the 'Create report' option is highlighted. The central panel shows the 'Properties' tab for the 'DMS_SAMPLE' schema, and the bottom panel shows the 'Amazon Aurora (PostgreSQL compatible) category: Schemas' properties.

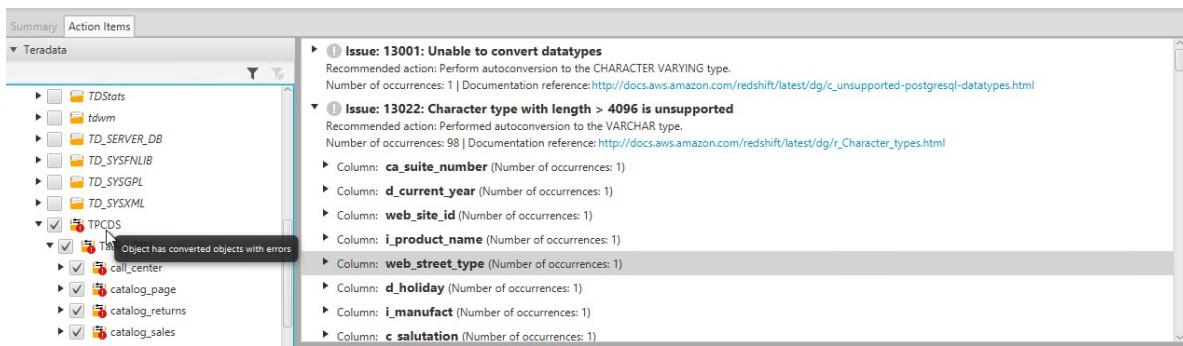
The progress bar updates while the report is generated.



The executive summary page displays. Click the **Action Items** tab.

The screenshot shows the 'Database migration assessment report' page. It includes sections for 'Executive summary', 'Database objects with conversion actions for Amazon Aurora (PostgreSQL compatible)', and a note about objects that could not be converted. The AWS logo is in the top right corner.

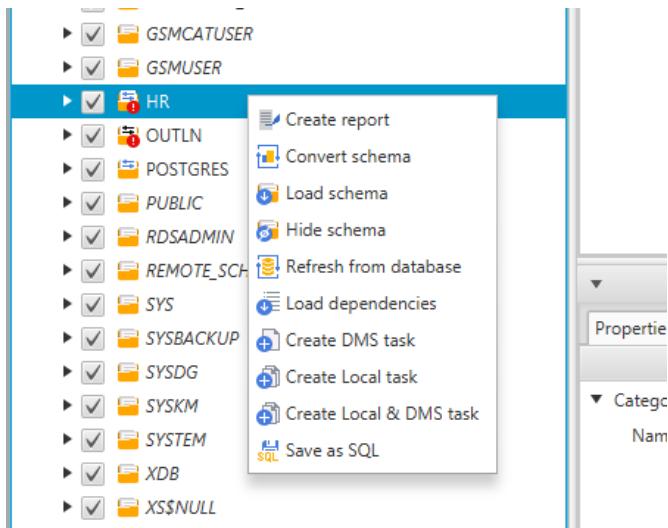
In this window, you can investigate each issue in detail and view the suggested course of action. For each issue, drill down to view all instances of that issue.



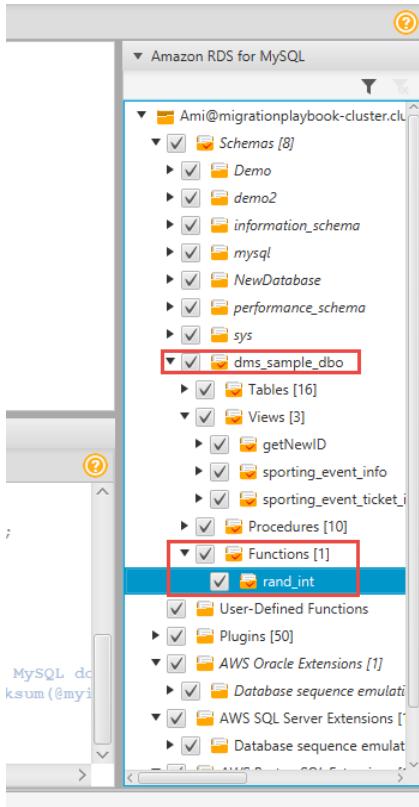
Right-click the database name and click **Convert Schema**.

Be sure to uncheck the **sys** and **information_schema** system schemas. Aurora PostgreSQL already has an **information_schema** schema.

This step does not make any changes to the target database.

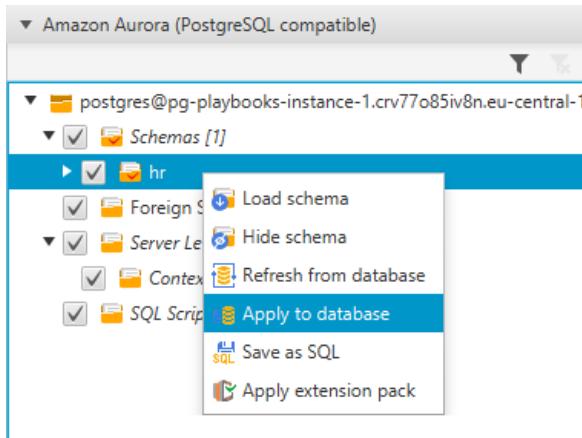


On the right pane, the new virtual schema is displayed as if it exists in the target database. Drilling down into individual objects displays the actual syntax generated by AWS SCT to migrate the objects.



Right-click the database on the right pane and choose either **Apply to database** to automatically execute the conversion script against the target database, or click **Save as SQL** to save to an SQL file.

Saving to an SQL file is recommended because it allows you to verify and QA the SCT code. Also, you can make the adjustments needed for objects that could not be automatically converted.



For more information, see https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Welcome.html

SCT Action Code Index

Legend

SCT/DMS Automation Level Symbol	Description
	Full Automation SCT performs fully automatic conversion, no manual conversion needed.
	High Automation: Minor, simple manual conversions may be needed.
	Medium Automation: Low-medium complexity manual conversions may be needed.
	Low Automation: Medium-high complexity manual conversions may be needed.
	Very Low Automation: High risk or complex manual conversions may be needed.
	No Automation: Not currently supported by SCT, manual conversion is required for this feature.

The following sections list the [Schema Conversion Tool](#) Action codes for topics that are covered in this playbook.

Note: The links in the table point to the Oracle topic pages, which are immediately followed by the PostgreSQL pages for the same topics.

SQL



AWS SCT automatically converts the most commonly used SQL statements as both Oracle and Aurora PostgreSQL support the entry level ANSI compliance, some changes may be required for DML related to ERROR LOG, subquery, or partitions

Action Code	Action Message
5064	PostgreSQL doesn't support the UPDATE statement with the ERROR LOG option
5065	PostgreSQL doesn't support the UPDATE statement for a subquery
5067	PostgreSQL doesn't support the DELETE statement with the ERROR LOG option
5068	PostgreSQL doesn't support the DELETE statement for a subquery
5098	PostgreSQL doesn't support the DELETE statement for a PARTITION
5126	PostgreSQL doesn't support the MODEL statement
5140	PostgreSQL doesn't support BULK COLLECT INTO clause
5334	Unable to convert statements with dynamic SQL statement
5024	PostgreSQL doesn't support the INSERT statement with partition_extension_clause
5070	PostgreSQL doesn't support the INSERT statement with the ERROR LOG option
5071	PostgreSQL doesn't support the INSERT statement for a subquery
5087	PostgreSQL doesn't support the RETURNING BULK COLLECT INTO clause
5088	PostgreSQL doesn't support the RETURNING BULK COLLECT INTO clause
5090	PostgreSQL doesn't support the INSERT statement for a SUBPARTITION
5139	PostgreSQL doesn't support FOR UPDATE SKIP LOCKED
5144	PostgreSQL doesn't support FOR UPDATE WAIT clauses
5578	Unable to automatically transform the SELECT statement
5585	Automatic conversion outer join into correlation query not supported
5558	PostgreSQL doesn't support the UPDATE statement for a PARTITION
5608	Unable to convert the UPDATE statement with multiple-column subquery in SET clause
5352	PostgreSQL doesn't support synonyms
5353	PostgreSQL doesn't support synonyms
5557	PostgreSQL doesn't support GROUPING SETS, CUBE, and ROLLUP functions
5663	PostgreSQL doesn't explicitly support autonomous transactions.

Creating Tables



AWS SCT automatically converts the most commonly used constructs of the CREATE TABLE statement as both Oracle and Aurora PostgreSQL support the entry level ANSI compliance. These items include table names, containing security schema (or database), column names, basic column data types, column and table constraints, column default values, primary, candidate (UNIQUE), and foreign keys. Some changes may be required for computed columns and global temporary tables.

Action Code	Action Message
5326	PostgreSQL doesn't support constraints with the status DISABLED
5196	PostgreSQL doesn't support OBJECT TABLE
5198	PostgreSQL doesn't support GLOBAL TEMPORARY TABLE
5199	PostgreSQL doesn't support CLUSTERED TABLE
5200	PostgreSQL doesn't support EXTERNAL TABLES
5201	PostgreSQL doesn't support this partition type
5212	PostgreSQL doesn't have a data type BFILE
5213	PostgreSQL expands fractional seconds support for TIME, DATETIME, and TIMESTAMP values, with up to microseconds (6 digits) of precision
5348	PostgreSQL doesn't support NESTED TABLES
5550	PostgreSQL doesn't have a data type ROWID
5551	PostgreSQL doesn't have a data type UROWID
5552	PostgreSQL expands fractional seconds support for TIME, DATETIME, and TIMESTAMP values, with up to microseconds (6 digits) of precision
5553	PostgreSQL expands fractional seconds support for TIME, DATETIME, and TIMESTAMP values, with up to microseconds (6 digits) of precision
5554	PostgreSQL doesn't support virtual columns
5581	PostgreSQL doesn't support index-organized table
5620	Extension pack doesn't support ON COMMIT DELETE ROWS clause for global temporary table
5635	Unable convert function with Oracle specific formatting settings parameters in optimistic mode
5659	Table contains column with %s datatype
5621	Please check unique(exclude) constraint existence on field %s
5298	PostgreSQL doesn't support the DROP STORAGE clause in the TRUNCATE statement
5299	PostgreSQL doesn't support the REUSE STORAGE clause in the TRUNCATE statement
5300	PostgreSQL doesn't support the PRESERVE clause in the TRUNCATE statement
5301	PostgreSQL doesn't support the PURGE clause in the TRUNCATE statement

Data Types



Data type syntax is very similar between Oracle and Aurora PostgreSQL and most are converted automatically by AWS SCT. Note that date and time handling paradigms are different for Oracle and Aurora PostgreSQL and require manual verifications and/or conversion. Also note that due to differences in data type behavior between Oracle and Aurora PostgreSQL , manual verification and strict testing are highly recommended.

For more details, see [Data Types](#).

Action Code	Action Message
5212	PostgreSQL doesn't have a data type BFILE
5550	PostgreSQL doesn't have a data type ROWID
5551	PostgreSQL doesn't have a data type UROWID
5636	Unable to convert VARRAY OF VARRAY
5613	Unable to convert Multidimensional arrays
5644	Unable automatically convert assign operation of array or global nested table, because of nested record
5609	PostgreSQL doesn't have a datatype %s
5572	PostgreSQL doesn't support object type methods
5028	Unable to convert definition of object with unsupported %s datatype
5029	Unable to convert simple usage of unsupported %s datatype object
5030	Unable to convert complex usage of unsupported %s datatype object
5595	This object uses a ROWID from the %s table
5597	This object uses a ROWID from the %s table
5598	PostgreSQL doesn't support a ROWID

Character Set



The character set granularity in Oracle and Aurora PostgreSQL are significantly different, in some cases

For more details, see [Character Set](#).

Action Code	Action Message
5623	UUE format encoding is not supported yet

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. There are specific options which are not supported for automatic conversion by SCT.

For more details, see [Cursors](#).

Action Code	Action Message
5225	PostgreSQL doesn't support TYPE ... IS REF CURSOR declaration
5226	PostgreSQL doesn't support TYPE ... IS REF CURSOR usage
5599	PostgreSQL doesn't support a reference to SQLERRM outside an exception handler
5600	PostgreSQL doesn't support a reference to SQLERRM with any specified parameter value
5601	PostgreSQL doesn't support a reference to SQLCODE outside an exception handler
5602	PostgreSQL error code type is not a number and is incompatible with the number type variables
5560	PostgreSQL doesn't support the PROGRAM_ERROR exception
5561	PostgreSQL doesn't support the %s exception
5580	The exception block was emptied after conversion
5559	PostgreSQL doesn't support the RETURN TYPE for the cursor
5604	Global cursor's fetching was prep-ended with the cursor's declaration and opening
5612	Unable to convert FETCH of global parametrized cursor without preceding OPEN within the routine
5117	PostgreSQL doesn't support the %s attribute
5330	PostgreSQL doesn't support global cursors
5031	Unable to convert CURSOR expression

Action Code	Action Message
5040	PostgreSQL doesn't support %s sharing option
5042	PostgreSQL doesn't support typed cursor specification

Flow Control



Although the flow control syntax of Oracle differs from Aurora PostgreSQL , the AWS SCT can convert most constructs automatically including loops, command blocks, and delays. Aurora PostgreSQL does not support the GOTO command nor Conditional Compilation command, which require manual conversion.

Action Code	Action Message
5603	Conditional Compilation is not supported
5335	PostgreSQL doesn't support the GOTO operator

Transaction Isolation



Aurora PostgreSQL supports the four transaction isolation levels specified in the SQL:92 standard: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE, all of which are automatically converted by AWS SCT. AWS SCT also converts BEGIN / COMMIT and ROLLBACK commands that use slightly different syntax. Manual conversion is required for named, marked, and delayed durability transactions that are not supported by Aurora PostgreSQL .

For more details, see [Transaction Isolation](#).

Action Code	Action Message
5350	Unable automatically convert statements that explicitly apply or cancel a transaction
5611	PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions

Stored Procedures



Aurora PostgreSQL Stored Procedures (functions) provides very similar functionality to Oracle stored procedures and can be automatically converted by AWS SCT. Manual conversion is required for procedures that use RETURN values and some less common EXECUTE options such as the RECOMPILE and RESULTS SETS options.

For more details, see [Stored Procedures](#).

Action Code	Action Message
5624	Converted code might be incorrect because of the bind variable names
5625	PostgreSQL doesn't support parameters in an anonymous block. Converted code might be incorrect because of use of bind variables
5627	Converted code might be incorrect because of the use of UDF
5628	Converted code might be incorrect because of the dynamic SQL
5629	Not all features of the original DBMS_SQL package function are fully supported. Converted code behaviour and execution results might differ from the expected ones.
5630	Unable to convert DBMS_SQL package function (procedure).
5633	Converted code behaviour and execution results might differ from the expected ones.
5634	Unable convert the use of UDF with OUT/IN OUT parameters
5607	Unable to convert Java stored routine
5027	Unable to convert object since there is no package body source code provided
5626	Unable to convert function %s
5340	PostgreSQL doesn't support the %s function
5579	The second parameter of %s function can be processed incorrectly
5584	The function %s depends on the time zone setting
5617	Values 'm', 'x' are not fully supported as "match_parameter" arguments, as well as a "subexpression" parameter treating. Please notice that the same regular expression in Oracle and PostgreSQL might produce different results
5616	Unable to convert TABLE function. Try to use UNNEST function

Triggers



Aurora PostgreSQL supports BEFORE and AFTER triggers for INSERT, UPDATE, and DELETE. However, Aurora PostgreSQL triggers differ substantially from Oracle's triggers, but most common use cases can be migrated with minimal code changes.

For more details, see [Triggers](#).

Action Code	Action Message
5238	PostgreSQL doesn't support the REFERENCING clauses
5240	PostgreSQL doesn't support triggers on nested table columns in views
5241	PostgreSQL doesn't support the FOLLOWS PRECEDES clause
5242	PostgreSQL doesn't support COMPOUND TRIGGER
5243	Converted trigger was owned by another schema. PostgreSQL always create trigger under the table's schema
5306	Transformation from invalid trigger
5311	PostgreSQL doesn't support system triggers
5313	PostgreSQL doesn't support the action-type clause
5317	PostgreSQL doesn't support the PARENT referencing clause
5415	PostgreSQL doesn't support system triggers
5556	PostgreSQL doesn't support conditional predicates

Sequences



Although the syntax for Oracle IDENTITY and Aurora PostgreSQL SERIAL auto-enumeration columns differs significantly, it can be automatically converted by AWS SCT.

For more details, see [Sequences](#) and [Identity](#).

Action Code	Action Message
5574	PostgreSQL doesn't support the sequence's status

Views



Although the basic syntax for creating a view in Oracle and Aurora PostgreSQL is almost identical there are some sub-options that can differ significantly and this can add manual needed tasks to the migration process.

For more details, see [Views](#).

Action Code	Action Message
5075	PostgreSQL doesn't support VIEW with the READ ONLY option
5077	PostgreSQL doesn't support the PIVOT clause for the SELECT statement
5245	PostgreSQL doesn't support views with nested table columns
5320	PostgreSQL doesn't support the view with invalid status
5321	PostgreSQL doesn't support the object view
5322	PostgreSQL doesn't support the typed view
5583	PostgreSQL doesn't support constraints for view
5614	PostgreSQL doesn't support DML operations with non updatable views

User Defined Types



User defined type are not supported, AWS SCT can convert standard user defined types by replacing it with their base types. More complicated User Defined Type may require manual intervention

For more details, see [User Defined Types](#).

Action Code	Action Message
5099	Unable convert object due to %s not created
5332	The object has referenced on the object in %s schema
5582	Encrypted Objects
5638	Unable to convert using of global variable of Nested Table as a function or procedure argument
5569	PostgreSQL support session variables just SQL standard date and time types
5032	Unable to convert %s datatype due to its or its member types have an incomplete definition. PostgreSQL doesn't support creation of types based on missed types
5062	The %s type constructor call converted to direct assignment
5575	PostgreSQL doesn't support DEFAULT for creating TYPE
5577	PostgreSQL doesn't support a UDT with body
5118	Unable to convert the %s datatype declaration - PostgreSQL doesn't support associative arrays
5120	PostgreSQL doesn't support constructors of the "collection" type
5121	PostgreSQL doesn't support the FORALL statement

Action Code	Action Message
5587	PostgreSQL doesn't support EXTEND method with parameters

MERGE



MERGE statement is not supported and it cannot be automatically converted by AWS SCT. Manual conversion is straight-forward in most cases.

For more details and potential workarounds, see [MERGE](#).

Action Code	Action Message
5102	PostgreSQL doesn't support the MERGE statement
5618	PostgreSQL doesn't support the MERGE statement with the ERROR LOG clause
5621	Please check unique(exclude) constraint existence on field %s

Materialized Views



Materialized Views are not supported, some features such as incremental refresh or DML commands on Materialized Views are not supported

For more details and potential workarounds, see [Materialized Views](#)

Action Code	Action Message
5093	Unable to convert a query of the materialized view
5094	Unable to convert a materialized view
5095	PostgreSQL doesn't support using a materialized VIEW

Query hints



Basic query hints such as index hints can be converted automatically by AWS SCT, except for DML statements. Note that specific optimizations used for Oracle may be completely inapplicable to a new query optimizer. It is recommended to start migration testing with all hints removed. Then, selectively apply hints as a last resort if other means such as schema, index, and query optimizations have failed. Plan guides are not supported by Aurora PostgreSQL .

For more details, see [Query hints and Plan Guides](#).

Action Code	Action Message
5103	PostgreSQL doesn't support the %s hint

Database Links



Migrating database links from Oracle to Aurora PostgreSQL requires a full rewrite the mechanism that managed the database links. This cannot be automatically converted by AWS SCT.

For more details, see: [Database Links](#)

Action Code	Action Message
5605	PostgreSQL doesn't support use of dblinks
5639	postgres_fdw extension must be installed, connection options must be rechecked manually.
5640	Define foreign table structure manually
5641	PostgreSQL FCW does not support UDF usage
5657	Unable to create view on undefined foreign table

Indexes



Basic non-clustered indexes, which are the most commonly used type of indexes are automatically migrated by AWS SCT. In addition, filtered indexes, indexes with included columns, and some Oracle specific index options such as bitmap or domain can not be migrated automatically and require manual conversion.

For more details, see [Indexes topics](#).

Action Code	Action Message
5206	PostgreSQL doesn't support bitmap indexes
5208	PostgreSQL doesn't support domain indexes
5555	PostgreSQL doesn't support functional indexes that aren't single-column

Partitioning



Aurora PostgreSQL uses "table inheritance", some of the physical aspects of partitioning in Oracle do not apply to Aurora PostgreSQL . For example, the concept of file groups and assigning partitions to file groups. Aurora PostgreSQL supports a much richer framework for table partitioning than Oracle, with many additional options such as hash partitioning, and sub partitioning.

For more details, see [Partitioning](#).

Action Code	Action Message
5652	NULL columns not supported for partitioning
5653	PostgreSQL does not support foreign keys referencing partitioned tables
5654	PostgreSQL does not support foreign key references from a partitioned table to some other table
5655	Update of the partitioned table, its partition or subpartition may lead to errors
5656	Partition by TIMESTAMP is not supported
5658	Partition with DEFAULT is not supported

OLAP Functions



Aurora PostgreSQL does provide native support for almost all OLAP Functions.

For more details, see [OLAP Functions](#).

Action Code	Action Message
5271	The Oracle GREATEST function and PostgreSQL GREATEST function might give different results
5272	The Oracle LEAST function and PostgreSQL LEAST function might give different results
5622	PostgreSQL txid_current() always returning a new one transaction id

AWS Database Migration Service (DMS)

Usage

The AWS Database Migration Service (DMS) helps you migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. The AWS Database Migration Service can migrate your data to and from most widely-used commercial and open-source databases.

The service supports homogenous migrations such as Oracle to Oracle as well as heterogeneous migrations between different database platforms such as Oracle to Amazon Aurora or Microsoft SQL Server to MySQL. It also allows you to stream data to Amazon Redshift, Amazon DynamoDB, and Amazon S3 from any of the supported sources, which are Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, SAP ASE, SQL Server, IBM DB2 LUW, and MongoDB, enabling consolidation and easy analysis of data in a petabyte-scale data warehouse. The AWS Database Migration Service can also be used for continuous data replication with high-availability.

When migrating databases to Aurora, Redshift or DynamoDB, you can [use DMS free for six months](#).

For all supported sources for DMS, see

https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Source.html

For all supported targets for DMS, see

https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Target.html

Migration Tasks Performed by AWS DMS

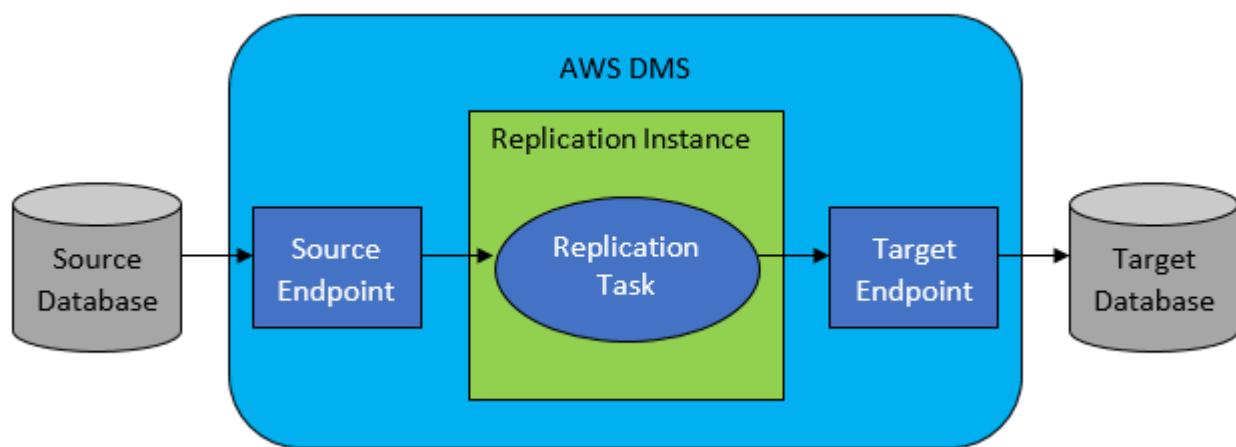
- In a traditional solution, you need to perform capacity analysis, procure hardware and software, install and administer systems, and test and debug the installation. AWS DMS automatically manages the deployment, management, and monitoring of all hardware and software needed for your migration. Your migration can be up and running within minutes of starting the AWS DMS configuration process.
- With AWS DMS, you can scale up (or scale down) your migration resources as needed to match your actual workload. For example, if you determine that you need additional storage, you can easily increase your allocated storage and restart your migration, usually within minutes. On the other hand, if you discover that you aren't using all of the resource capacity you configured, you can easily downsize to meet your actual workload.
- AWS DMS uses a pay-as-you-go model. You only pay for AWS DMS resources while you use them as opposed to traditional licensing models with up-front purchase costs and ongoing maintenance charges.
- AWS DMS automatically manages all of the infrastructure that supports your migration server including hardware and software, software patching, and error reporting.
- AWS DMS provides automatic failover. If your primary replication server fails for any reason, a backup replication server can take over with little or no interruption of service.
- AWS DMS can help you switch to a modern, perhaps more cost-effective database engine than the one you are running now. For example, AWS DMS can help you take advantage of the managed database services provided by Amazon RDS or Amazon Aurora. Or, it can help you move to the managed data warehouse service provided by Amazon Redshift, NoSQL platforms like Amazon DynamoDB, or low-cost storage platforms like Amazon S3. Conversely, if you want to migrate away from old infrastructure but continue to use the same database engine, AWS DMS also supports that process.

- AWS DMS supports nearly all of today's most popular DBMS engines as data sources, including Oracle, Microsoft SQL Server, MySQL, MariaDB, PostgreSQL, Db2 LUW, SAP, MongoDB, and Amazon Aurora.
- AWS DMS provides a broad coverage of available target engines including Oracle, Microsoft SQL Server, PostgreSQL, MySQL, Amazon Redshift, SAP ASE, S3, and Amazon DynamoDB.
- You can migrate from any of the supported data sources to any of the supported data targets. AWS DMS supports fully heterogeneous data migrations between the supported engines.
- AWS DMS ensures that your data migration is secure. Data at rest is encrypted with AWS Key Management Service (AWS KMS) encryption. During migration, you can use Secure Socket Layers (SSL) to encrypt your in-flight data as it travels from source to target.

How AWS DMS Works

At its most basic level, AWS DMS is a server in the AWS Cloud that runs replication software. You create a source and target connection to tell AWS DMS where to extract from and load to. Then, you schedule a task that runs on this server to move your data. AWS DMS creates the tables and associated primary keys if they don't exist on the target. You can pre-create the target tables manually if you prefer. Or you can use AWS SCT to create some or all of the target tables, indexes, views, triggers, and so on.

The following diagram illustrates the AWS DMS process.



Latest updates

DMS is continuously evolving and supporting more and more options, find below some of the updates add since last edition of this playbook:

- CDC tasks and Oracle source tables created using CREATE TABLE AS AWS -DMS now supports both full-load and CDC and CDC-only tasks running against Oracle source tables created using the CREATE TABLE AS statement.

- New MySQL version AWS DMS now supports MySQL version 8.0 as a source except when the transaction payload is compressed.
- Support for AWS Secrets Manager integration You can store the database connection details (user credentials) for supported endpoints securely in AWS Secrets Manager. You can then submit the corresponding secret instead of plain-text credentials to AWS DMS when you create or modify an endpoint. AWS DMS then connects to the endpoint databases using the secret. For more information on creating secrets for AWS DMS endpoints see Using secrets to access AWS Database Migration Service endpoints.
- Support for Oracle extended data types Oracle extended data types for both Oracle source and targets are now supported.
- TLS 1.2 support for MySQL AWS DMS now supports TLS 1.2 for MySQL endpoints.
- TLS 1.2 support for SQL Server AWS DMS now supports TLS 1.2 for SQL Server endpoints.

For a complete guide with a step-by-step walkthrough including all the latest notes for migrating SQL Server to Aurora MySQL (which is very similar to the Oracle-PostgreSQL migration process) with DMS, see https://docs.aws.amazon.com/dms/latest/sbs/CHAP_SQLServer2Aurora.html

For more information about DMS, see:

- <https://docs.aws.amazon.com/dms/latest/userguide/Welcome.html>
- https://docs.aws.amazon.com/dms/latest/userguide/CHAP_BestPractices.html

Amazon RDS on Outposts

PLEASE NOTE, ENTIRE TOPIC IS RELATED TO RDS AND IS NOT SUPPORTED WITH AURORA

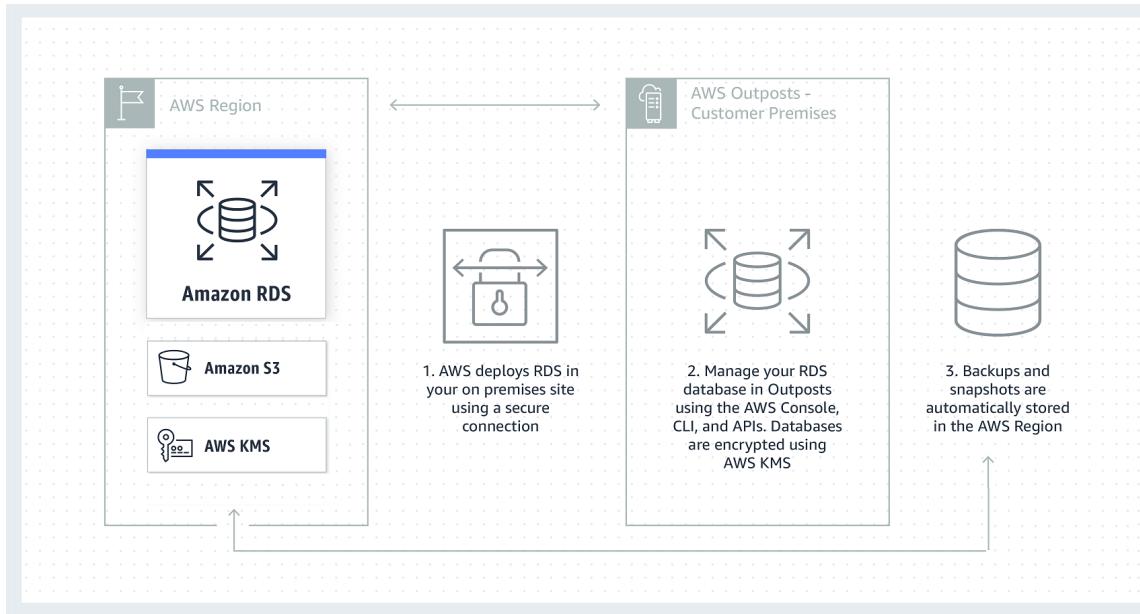
Usage

Amazon RDS on Outposts is a fully managed service that offers the same AWS infrastructure, AWS services, APIs, and tools to virtually any data center, co-location space, or on-premises facility for a truly consistent hybrid experience. Amazon RDS on Outposts is ideal for workloads that require low latency access to on-premises systems, local data processing, data residency, and migration of applications with local system inter-dependencies.

When you deploy Amazon RDS on Outposts, you can run RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications. Amazon RDS on Outposts also enables automatic backup to an AWS Region. You can manage RDS databases both in the cloud and on premises using the same AWS Management Console, APIs, and CLI. Amazon RDS on Outposts supports Microsoft SQL Server, MySQL, and PostgreSQL database engines, with support for additional database engines coming soon.

How it works

Amazon RDS on Outposts lets you run Amazon RDS in your on-premises or co-location site. You can deploy and scale an RDS database instance in Outposts just as you do in the cloud, using the AWS console, APIs, or CLI. RDS databases in Outposts are encrypted at rest using AWS KMS keys. RDS automatically stores all automatic backups and manual snapshots in the AWS Region.



This option is helpful when you need to run RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications

For more information, see:

- <https://aws.amazon.com/outposts/>
- <https://aws.amazon.com/rds/outposts/>
- <https://aws.amazon.com/blogs/aws/new-create-amazon-rds-db-instances-on-aws-outposts/>

Amazon Aurora Backtrack

Usage

We've all been there, you need to make a quick, seemingly simple fix to an important production database. You compose the query, give it a once-over, and let it run. Seconds later you realize that you forgot the WHERE clause, dropped the wrong table, or made another serious mistake, and interrupt the query, but the damage has been done. You take a deep breath, whistle through your teeth, wish that reality came with an Undo option.

Backtracking "rewinds" the DB cluster to the time you specify. Backtracking is not a replacement for backing up your DB cluster so that you can restore it to a point in time. However, backtracking provides the following advantages over traditional backup and restore:

- You can easily undo mistakes. If you mistakenly perform a destructive action, such as a DELETE without a WHERE clause, you can backtrack the DB cluster to a time before the destructive action with minimal interruption of service.
- You can backtrack a DB cluster quickly. Restoring a DB cluster to a point in time launches a new DB cluster and restores it from backup data or a DB cluster snapshot, which can take hours. Backtracking a DB cluster doesn't require a new DB cluster and rewinds the DB cluster in minutes.
- You can explore earlier data changes. You can repeatedly backtrack a DB cluster back and forth in time to help determine when a particular data change occurred. For example, you can backtrack a DB cluster three hours and then backtrack forward in time one hour. In this case, the backtrack time is two hours before the original time.

Aurora uses a distributed, log-structured storage system (read Design Considerations for High Throughput Cloud-Native Relational Databases to learn a lot more); each change to your database generates a new log record, identified by a Log Sequence Number (LSN). Enabling the backtrack feature provisions a FIFO buffer in the cluster for storage of LSNs. This allows for quick access and recovery times measured in seconds.

When you create a new Aurora MySQL DB cluster, backtracking is configured when you choose Enable Backtrack and specify a Target Backtrack window value that is greater than zero in the Backtrack section.

To create a DB cluster, follow the instructions in Creating an Amazon Aurora DB cluster. The following image shows the Backtrack section.

Backtrack

Backtrack lets you quickly rewind the DB cluster to a specific point in time, without having to create another DB cluster. [Info](#)

Enable Backtrack
Enabling Backtrack will charge you for storing the changes you make for backtracking.

Target Backtrack window
The Backtrack window determines how far back in time you could go. Aurora will try to retain enough log information to support that window of time. [Info](#)

12	hours (up to 72)
----	---------------------

Typical user cost
The cost of Backtrack depends on how often you are updating your database. This is an estimate based on typical workloads for your selected instance size (db.r5.large). [Info](#)

\$ 14.38 USD / month

After a production error, you can simply pause your application, open up the Aurora Console, select the cluster, and click Backtrack DB cluster

Then you select Backtrack and choose the point in time just before your epic fail, and click Backtrack DB cluster:

Backtrack DB cluster

Rewinds the DB cluster to a previous point in time without creating a new DB cluster.

Earliest restorable time is June 16, 2021 at 8:53:02 PM UTC-4 (Local) ⓘ

Date	Time
June 16, 2021	20 : 53 : 02 UTC-4

The next available time will be used if the specified time is not available.

⚠️ Your DB cluster is unavailable during the Backtrack process, which typically takes a few minutes.

Cancel **Backtrack DB cluster**

Then you wait for the rewind to take place, unpause your application and proceed as if nothing had happened. When you initiate a backtrack, Aurora will pause the database, close any open connections, drop uncommitted writes, and wait for the backtrack to complete. Then it will resume normal operation and be able to accept requests. The instance state will be backtracking while the rewind is underway.

Backtrack window

With backtracking, there is a target backtrack window and an actual backtrack window:

- The target backtrack window is the amount of time you want to be able to backtrack your DB cluster. When you enable backtracking, you specify a target backtrack window. For example, you might specify a target backtrack window of 24 hours if you want to be able to backtrack the DB cluster one day.
- The actual backtrack window is the actual amount of time you can backtrack your DB cluster, which can be smaller than the target backtrack window. The actual backtrack window is based on your workload and the storage available for storing information about database changes, called change records.

As you make updates to your Aurora DB cluster with backtracking enabled, you generate change records. Aurora retains change records for the target backtrack window, and you pay an hourly rate for storing them. Both the target backtrack window and the workload on your DB cluster determine the number of change records you store. The workload is the number of changes you make to your DB cluster in a given amount of time. If your workload is heavy, you store more change records in your backtrack window than you do if your workload is light.

You can think of your target backtrack window as the goal for the maximum amount of time you want to be able to backtrack your DB cluster. In most cases, you can backtrack the maximum amount of time that you specified. However, in some cases, the DB cluster can't store enough change records to backtrack the maximum amount of time, and your actual backtrack window is smaller than your target. Typically, the actual backtrack window is smaller than the target when you have extremely heavy workload on your DB cluster. When your actual backtrack window is smaller than your target, we send you a notification.

When backtracking is enabled for a DB cluster, and you delete a table stored in the DB cluster, Aurora keeps that table in the backtrack change records. It does this so that you can revert back to a time before you deleted the table. If you don't have enough space in your backtrack window to store the table, the table might be removed from the backtrack change records eventually.

Backtracking limitations

The following limitations apply to backtracking:

- Backtracking an Aurora DB cluster is available in certain AWS Regions and for specific Aurora MySQL versions only. For more information, see Backtracking in Aurora.
- Backtracking is only available for DB clusters that were created with the Backtrack feature enabled. You can enable the Backtrack feature when you create a new DB cluster or restore a snapshot of a DB cluster. For DB clusters that were created with the Backtrack feature enabled, you can create a clone DB cluster with the Backtrack feature enabled. Currently, you can't perform backtracking on DB clusters that were created with the Backtrack feature disabled.
- The limit for a backtrack window is 72 hours.
- Backtracking affects the entire DB cluster. For example, you can't selectively backtrack a single table or a single data update.
- Backtracking isn't supported with binary log (binlog) replication. Cross-Region replication must be disabled before you can configure or use backtracking.
- You can't backtrack a database clone to a time before that database clone was created. However, you can use the original database to backtrack to a time before the clone was created. For more information about database cloning, see Cloning an Aurora DB cluster volume.
- Backtracking causes a brief DB instance disruption. You must stop or pause your applications before starting a backtrack operation to ensure that there are no new read or write requests. During the backtrack operation, Aurora pauses the database, closes any open connections, and drops any uncommitted reads and writes. It then waits for the backtrack operation to complete.
- Backtracking isn't supported for the following AWS Regions:
 - Africa (Cape Town)
 - China (Ningxia)
 - Asia Pacific (Hong Kong)
 - Europe (Milan)
 - Europe (Stockholm)
 - Middle East (Bahrain)
 - South America (São Paulo)
- You can't restore a cross-Region snapshot of a backtrack-enabled cluster in an AWS Region that doesn't support backtracking.
- You can't use Backtrack with Aurora multi-master clusters.
- If you perform an in-place upgrade for a backtrack-enabled cluster from Aurora MySQL version 1 to version 2, you can't backtrack to a point in time before the upgrade happened

For more information, see: <https://aws.amazon.com/blogs/aws/amazon-aurora-backtrack-turn-back-time/>

Amazon RDS Proxy

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon Relational Database Service (RDS) that makes applications more scalable, more resilient to database failures, and more secure.

Many applications, including those built on modern server-less architectures, can have many open connections to the database server, and may open and close database connections at a high rate, exhausting database memory and compute resources. Amazon RDS Proxy allows applications to pool and share connections established with the database, improving database efficiency and application scalability. With RDS Proxy, fail-over times for Aurora and RDS databases are reduced by up to 66% and database credentials, authentication, and access can be managed through integration with AWS Secrets Manager and AWS Identity and Access Management (IAM).

Amazon RDS Proxy can be enabled for most applications with no code changes, and you don't need to provision or manage any additional infrastructure. Pricing is simple and predictable: you pay per vCPU of the database instance for which the proxy is enabled. Amazon RDS Proxy is now generally available for Aurora MySQL, Aurora PostgreSQL, RDS MySQL and RDS PostgreSQL.

Benefits

- Improved application performance

Amazon RDS proxy manages a connection pooling which helps with reducing the stress on database compute and memory resources that typically occurs when new connections are established and it is useful to efficiently support a large number and frequency of application connections

- Increase application availability

By automatically connecting to a new database instance while preserving application connections Amazon RDS Proxy can reduce fail-over time by 66%

- Manage application security

RDS Proxy also enables you to centrally manage database credentials using AWS Secrets Manager

- Fully managed

Amazon RDS Proxy gives you the benefits of a database proxy without requiring additional burden of patching and managing your own proxy server.

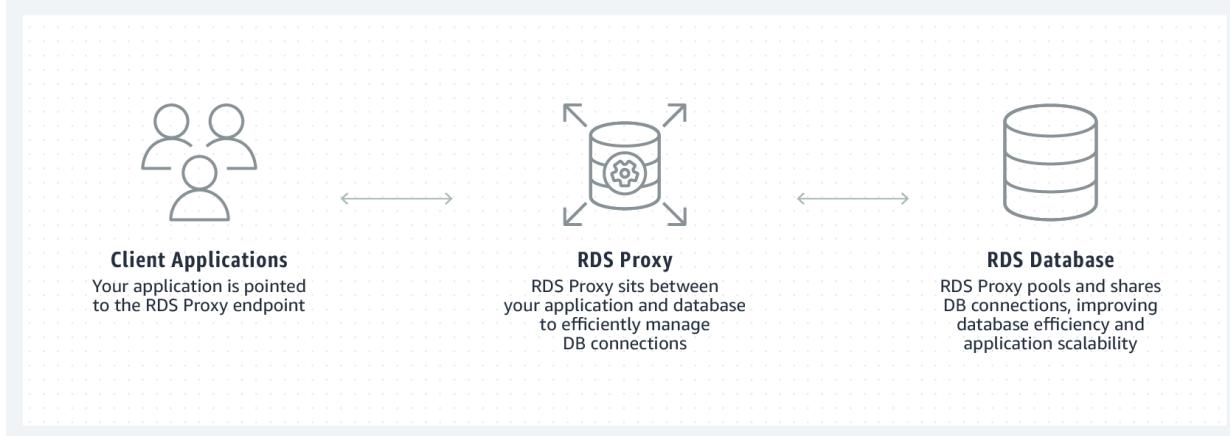
- Fully compatible with your database

Amazon RDS Proxy is fully compatible with the protocols of supported database engines, so you can deploy RDS Proxy for your application without making changes to your application code.

- Available and durable

Amazon RDS Proxy is highly available and deployed over multiple Availability Zones (AZs) to protect you from infrastructure failure

How it works



For more information, see:

- <https://aws.amazon.com/blogs/aws/amazon-rds-proxy-now-generally-available/>
- <https://aws.amazon.com/rds/proxy/>

Amazon Aurora Serverless v1

Usage

Amazon Aurora Serverless v1 (Amazon Aurora Serverless version 1) is an on-demand autoscaling configuration for Amazon Aurora. An Aurora Serverless DB cluster is a DB cluster that scales compute capacity up and down based on your application's needs. This contrasts with Aurora provisioned DB clusters, for which you manually manage capacity. Aurora Serverless v1 provides a relatively simple, cost-effective option for infrequent, intermittent, or unpredictable workloads. It is cost-effective because it automatically starts up, scales compute capacity to match your application's usage, and shuts down when it's not in use.

To learn more about pricing, see Serverless Pricing under MySQL-Compatible Edition or PostgreSQL-Compatible Edition on the Amazon Aurora pricing page.

Aurora Serverless v1 clusters have the same kind of high-capacity, distributed, and highly available storage volume that is used by provisioned DB clusters. The cluster volume for an Aurora Serverless v1 cluster is always encrypted. You can choose the encryption key, but you can't disable encryption. That means that you can perform the same operations on an Aurora Serverless v1 that you can on encrypted snapshots. For more information, see Aurora Serverless v1 and snapshots.

Aurora Serverless v1 provides the following advantages:

- **Simpler than provisioned** - Aurora Serverless v1 removes much of the complexity of managing DB instances and capacity.

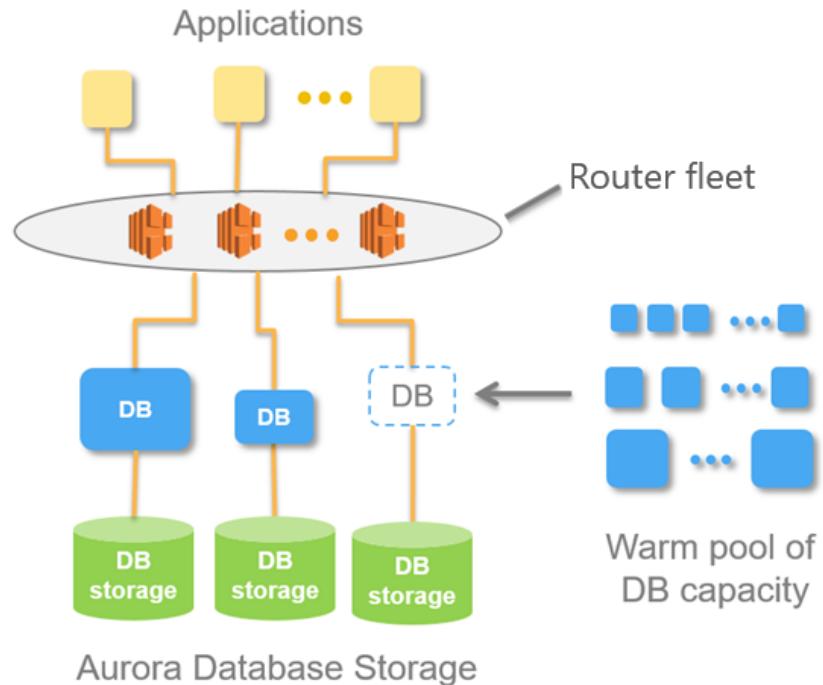
- **Scalable** - Aurora Serverless v1 seamlessly scales compute and memory capacity as needed, with no disruption to client connections.
- **Cost-effective** - When you use Aurora Serverless v1, you pay only for the database resources that you consume, on a per-second basis.
- **Highly available storage** - Aurora Serverless v1 uses the same fault-tolerant, distributed storage system with six-way replication as Aurora to protect against data loss.

Aurora Serverless v1 is designed for the following use cases:

- **Infrequently used applications** - You have an application that is only used for a few minutes several times per day or week, such as a low-volume blog site. With Aurora Serverless v1, you pay for only the database resources that you consume on a per-second basis.
- **New applications** - You're deploying a new application and you're unsure about the instance size you need. By using Aurora Serverless v1, you can create a database endpoint and have the database auto-scale to the capacity requirements of your application.
- **Variable workloads** - You're running a lightly used application, with peaks of 30 minutes to several hours a few times each day, or several times per year. Examples are applications for human resources, budgeting, and operational reporting applications. With Aurora Serverless v1, you no longer need to provision for peak or average capacity.
- **Unpredictable workloads** - You're running daily workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. With Aurora Serverless v1, your database autoscales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over.
- **Development and test databases** - Your developers use databases during work hours but don't need them on nights or weekends. With Aurora Serverless v1, your database automatically shuts down when it's not in use.
- **Multi-tenant applications** - With Aurora Serverless v1, you don't have to individually manage database capacity for each application in your fleet. Aurora Serverless v1 manages individual database capacity for you.

This process takes almost no time and since the storage is shared between nodes Aurora can scale up or down in seconds for most workloads. The service currently has autoscaling thresholds of 1.5 minutes to scale up and 5

minutes to scale down. That means metrics must exceed the limits for 1.5 minutes to trigger a scale up or fall below the limits for 5 minutes to trigger a scale down. The cool-down period between scaling activities is 5 minutes to scale up and 15 minutes to scale down. Before scaling can happen the service has to find a “scaling point” which may take longer than anticipated if you have long-running transactions. Scaling operations are transparent to the connected clients and applications since existing connections and session state are transferred to the new nodes. The only difference with pausing and resuming is a higher latency for the first connection, typically around 25 seconds. You can find more details in the documentation.



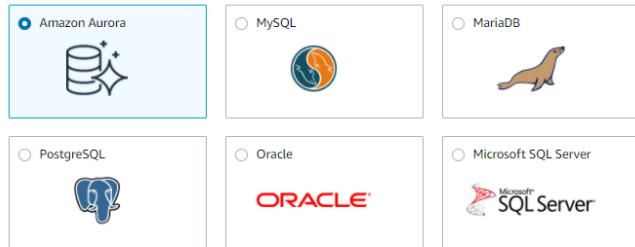
How to provision

Browse to the [Databases page](#) and click on "Create database"

Pick "Serverless" capacity type

Engine options

Engine type [Info](#)



Edition

- Amazon Aurora with MySQL compatibility
- Amazon Aurora with PostgreSQL compatibility

Capacity type [Info](#)

- Provisioned

You provision and manage the server instance sizes.

- Serverless

You specify the minimum and maximum amount of resources needed, and Aurora scales the capacity based on database load. This is a good option for intermittent or unpredictable workloads.

Version

Aurora PostgreSQL (compatible with PostgreSQL 10.14)

To see more versions, modify the capacity types. [Info](#)

Aurora PostgreSQL engine versions earlier than 11.9 don't support the newest r6g generation instance classes.

Choose the capacity properties suite for you use case

Capacity settings

This billing estimate is based on published prices. [Learn more](#)

Minimum Aurora capacity unit [Info](#)

1
2GB RAM

Maximum Aurora capacity unit [Info](#)

64
122GB RAM

▼ Additional scaling configuration

- Force scaling the capacity to the specified values when the timeout is reached [Info](#)
Enable to force capacity scaling as soon as possible. Disable to cancel the capacity changes when a timeout is reached
- Pause compute capacity after consecutive minutes of inactivity [Info](#)
You are only charged for database storage while the compute capacity is paused

For more information, see:

- <https://aws.amazon.com/rds/aurora/serverless/>
- <https://aws.amazon.com/blogs/aws/aurora-serverless-ga/>
- <https://aws.amazon.com/blogs/aws/amazon-aurora-postgresql-serverless-now-generally-available/>

Migration Quick Tips

This section provides migration tips that can help save time as you transition from Oracle to Aurora PostgreSQL. They address many of the challenges faced by administrators new to Aurora PostgreSQL. Some of these tips describe functional differences in similar features between Oracle and Aurora PostgreSQL.

Management

- In Aurora PostgreSQL , the term "Database Snapshot" is equivalent to Oracle RMAN backup
- Partitioning in Aurora PostgreSQL is called "INHERITS" tables and act completely different in terms of management
- Unlike Oracle's statistics, Aurora PostgreSQL does not collect detailed key value distribution; it relies on selectivity only. When troubleshooting execution, be aware that parameter values are insignificant to plan choices.
- Many missing features such as sending emails can be achieved with quick implementations of Amazon's services (like Lambda).
- Parameters and backups are managed by Amazon's RDS,. It is very useful in terms of checking parameter's value against its default and comparing them to another parameter group.
- High Availability can be implemented in few clicks to create Replicas.
- With Database Links, there are two options. The db_link extension is similar to Oracle and the postgres_fdw extension for using Foreign Data Wrapper.

SQL

- Triggers work differently in Aurora PostgreSQL. The syntax for inserted and deleted for each row is NEW and OLD.
- Aurora PostgreSQL does not support many cursors status checks. When declaring cursors in Aurora PostgreSQL , you must create an explicit HANDLER object.
- To execute a stored procedure (functions), use SELECT instead of EXECUTE.
- To execute a string as a query, use Aurora PostgreSQL Prepared Statements instead of EXECUTE (<String>) syntax.
- In Aurora PostgreSQL , IF blocks must be terminated with END IF. WHILE..LOOP loops must be terminated with END LOOP.
- Unlike Oracle, in Aurora PostgreSQL auto commit is ON, make sure to set it to OFF if you want to make the database behavior more similar to Oracle.
- Aurora PostgreSQL does not use special data types for UNICODE data. All string types may use any character set and any relevant collation.
- Collations can be defined at the server, database, and column level, similar to Oracle. They cannot be defined at the table level.
- Oracle's DELETE <Table Name> syntax, which allows omitting the FROM keyword, is invalid in Aurora PostgreSQL . Add the FROM keyword to all delete statements.
- Aurora PostgreSQL SERIAL column property is similar to IDENTITY in Oracle.
- Error handling in Aurora PostgreSQL has less features, but for special requirements, you can log or send alerts by inserting into tables or catching errors.

- Aurora PostgreSQL does not support the MERGE statement. Use the REPLACE statement and the INSERT... ON DUPLICATE KEY UPDATE statement as alternatives.
- You can concatenate strings in Aurora PostgreSQL using the "||" operator, like in Oracle
- Aurora PostgreSQL is much stricter than Oracle in terms of statement terminators. Be sure to always use a semicolons at the end of statements.
- There is no CREATE PROCEDURE syntax; only CREATE FUNCTION. You can create a function that returns void.
- Keep in mind that the window functions GREATEST and LEAST might get different results than the results that might be returned in Oracle from using these functions.
- PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions
- Aurora PostgreSQL does not support BFILE, ROWID, UROWID data types, try to use other data types.
- Temporary tables are being kept only for the session level and only the session that created the table can query the temporary table.
- PostgreSQL does not support unused or virtual columns, there is no workaround for replacing unused columns, for using similar functionality to the virtual columns, views and functions can be combined
- PostgreSQL does not support automatic or incremental REFRESH for Materialized Views, use triggers instead.
- Explore AWS to locate which features can be replaced with Amazon's services, this can help you with managing your database and decrease costs.
- The architecture in PostgreSQL allows you have multiple databases in a single instance, this can be relevant for consolidation projects.
- Beware of control characters when copying and pasting a script to Aurora PostgreSQL clients. Aurora PostgreSQL is much more sensitive to control characters than Oracle and they result in frustrating syntax errors that are hard to find.

SQL & PL/SQL

Oracle Single-Row and Aggregate Functions vs. PostgreSQL Single-Row and Aggregate Functions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Not all functions are supported by PostgreSQL and may require to create manually

Oracle Usage

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. They can be used 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 data types such as NUMERIC functions, CHAR functions, and 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 [PostgreSQL Usage](#) for a comparison of Oracle and PostgreSQL single-row functions.

Oracle 19 adds ability to eliminate duplicate items in LISTAGG function results with new DISTINCT keyword.

Oracle 19 introduces several new bitmap SQL aggregate functions (BITMAP_BUCKET_NUMBER, BITMAP_BIT_POSITION and BITMAP_CONSTRUCT_ AGG) that help to speed up COUNT DISTINCT operations.

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Single-Row-Functions.html#GUID-B93F789D-B486-49FF-B0CD-0C6181C5D85C>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Aggregate-Functions.html#GUID-62BE676B-AF18-4E63-BD14-25206FEA0848>

PostgreSQL Usage

PostgreSQL provides an extensive list of single-row and aggregation functions. Some 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 exhibit different functionality. In the table below, the Equivalent column indicates functional equivalency.

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
NUMERIC FUNCTIONS				
ABS	Absolute value of n: abs (-11.3) --> 11.3	ABS(n)	Absolute value of n: abs (-11.3) --> 11.3	Yes
CEIL	Returns the smallest integer	CEIL /	Returns the smallest integer that is	Yes

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
	that is greater than or equal to n: ceil (-24.9) --> -24	CEILING	greater than or equal to n: ceil (-24.9) --> -24	
FLOOR	Returns the largest integer equal to or less than n: floor (-43.7) --> -44	FLOOR	Returns the largest integer equal to or less than n: floor (-43.7) --> -44	Yes
MOD	Remainder of n2 divided by n1: mod(10,3) --> 1	MOD	Remainder of n2 divided by n1: mod(10,3) --> 1	Yes
ROUND	Returns n rounded to integer places to the right of the decimal point: round (3.49, 1) --> 3.5	ROUND	Returns n rounded to integer places to the right of the decimal point: round (3.49, 1) --> 3.5	Yes
TRUNC (Number)	Returns n1 truncated to n2 decimal places: trunc(13.5) --> 13	TRUNC (Number)	Returns n1 truncated to n2 decimal places: trunc(13.5) --> 13	Yes

CHARACTER FUNCTIONS

CONCAT	Returns char1 concatenated with char2: concat('a', 1) --> a1	CONCAT	Concatenate the text representations of all the arguments: concat('a', 1) --> a1 Also, can use the () operators: select 'a' 'b' --> a b	Partly
LOWER / UPPER	Returns char, with all letters lowercase or uppercase: lower ('MR. Smith') --> mr. smith	LOWER / UPPER	Returns char, with all letters lowercase or uppercase: lower ('MR. Smith') --> mr. smith	Yes
LPAD / RPAD	Returns expr1, left or right padded to length n characters with the sequence of characters in expr2: LPAD('Log-1', 10, '*') --> *****Log-1	LPAD / RPAD	Returns expr1, left or right padded to length n characters with the sequence of characters in expr2: LPAD('Log-1', 10, '*') --> *****Log-1	Yes
REGEXP_REPLACE	Search a string for a regular expression pattern: regexp_replace('John', '[hn].', '1') --> Jo1	REGEXP_REPLACE	Replace substring(s) matching a POSIX regular expression: regexp_replace('John', '[hn].', '1') --> Jo1	Yes
REGEXP_SUBSTR	Extends the functionality of the SUBSTR function by searching a string for a regular expression pattern: REGEXP_SUBSTR('http://www.aws.com/products', 'http://([[:alnum:]])+\.(?){3,4}/?') --> http://www.aws.com/	REGEXP_MATCHES OR SUBSTRING	Return all captured substrings resulting from matching a POSIX regular expression against the string: REGEXP_MATCHES ('http://www.aws.com/products', 'http://[[:alnum:]])+\.(?){3,4}/?') --> {http://www.aws.com/} OR SUBSTRING ('http://www.aws.com/products', 'http://[[:alnum:]])+\.(?){3,4}/?') --> http://www.aws.com/	No

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
REPLACE	Returns char with every occurrence of search string replaced with a replacement string: replace ('abcdef', 'abc', '123') --> 123def	REPLACE	Returns char with every occurrence of search string replaced with a replacement string: replace ('abcdef', 'abc', '123') --> 123def	Yes
LTRIM / RTRIM	Removes from the left or right end of char all of the characters that appear in set: ltrim ('zzzyaws', 'xyz') --> aws	LTRIM / RTRIM	Remove the longest string containing only characters from characters (a space by default) from the start of string: ltrim('zzzyaws', 'xyz') --> aws	Yes
SUBSTR	Return a portion of char, beginning at character position, substring length characters long: substr('John Smith', 6,1) --> S	SUBSTRING	Extract substring: substring ('John Smith', 6,1) --> S	No
TRIM	Trim leading or trailing characters (or both) from a character string: trim (both 'x' FROM 'xJohnxx') --> John	TRIM	Remove the longest string containing only characters from characters (a space by default) from the start, end, or both ends: trim (both from 'yxJohnxx', 'xyz') --> John	Partly
ASCII	Returns the decimal representation in the database character set of the first character of char: ascii('a') --> 97	ASCII	Returns the decimal representation in the database character set of the first character of char: ascii('a') --> 97	Yes
INSTR	Search string for substring	N/A	Oracle's INSTR function can be simulated using PostgreSQL built-in function.	No
LENGTH	Return the length of char: length ('John S.') --> 7	LENGTH	Return the length of char: length ('John S.') --> 7	Yes
REGEXP_COUNT	Returns the number of times, a pattern occurs in a source string.	N/A	The REGEXP_COUNT function can be used with Amazon Redshift if necessary.	No
REGEXP_INSTR	Search a string position for a regular expression pattern.	N/A	The REGEXP_INSTR function can be used with Amazon Redshift if necessary.	No

DATETIME FUNCTIONS

ADD_MONTHS	Returns the date plus integer months: add_months(sys-date, 1)	N/A	PostgreSQL can implement the same functionality using the '<date>+ interval month' statement: now () + interval '1 month'	No
CURRENT_DATE	Returns the current date in the session time zone: select current_date from dual --> 2017-01-01 13:01:01	CURRENT_DATE	PostgreSQL CURRENT_DATE will return date with no time, use the now() or the current_timestamp function to achieve the same results: select current_timestamp --> 2017-	Partly

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
			01-01 13:01:01	
CURRENT_TIMESTAMP	Returns the current date and time in the session time zone: select current timestamp from dual; --> 2017-01-01 13:01:01	CURRENT_TIMESTAMP	Returns the current date and time in the session time zone: select current timestamp from dual; --> 2017-01-01 13:01:01	Yes
EXTRACT (date part)	Returns the value of a specified datetime field from a datetime or interval expression: EXTRACT (YEAR FROM DATE '2017-03-07') --> 2017	EXTRACT (date part)	Returns the value of a specified datetime field from a datetime or interval expression: EXTRACT (YEAR FROM DATE '2017-03-07') --> 2017	Yes
LAST_DAY	Returns the date of the last day of the month that contains date: LAST_DAY('05-07-2018') -->05-31-2018	N/A	The LAST_DAY function can be used with Amazon Redshift if necessary or can be created using PostgreSQL built-in functions.	No
BETWEEN	Returns the number of months between dates date1 and date2: MONTHS_BETWEEN (sysdate, sysdate-100) --> 3.25	N/A	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: DATE_PART ('month', now()) - DATE_PART('month', now())- interval '100 days'--> 3	No
SYSDATE	Returns the current date and time set for the operating system on which the database server resides: select sysdate from dual --> 2017-01-01 13:01:01	now()	Current date and time including fractional seconds and time zone: select now () --> 2017-01-01 13:01:01.123456+00	No
SYSTIMESTAMP	Returns the system date, including fractional seconds and time zone: Select systimestamp from dual --> 2017-01-01 13:01:01.123456 PM +00:00	NOW()	Current date and time including fractional seconds and time zone: select now () --> 2017-01-01 13:01:01.123456+00	No
LOCALTIMESTAMP	Returns the current date and time in the session time zone in a value of data type TIMESTAMP: select localtimestamp from dual --> 01-JAN-17 10.01.10.123456 PM	LOCALTIMESTAMP	Returns the current date and time in the session time zone in a value of data type TIMESTAMP: select localtimestamp from dual --> 01-JAN-17 10.01.10.123456 PM	Yes
TO_CHAR (datetime)	Converts a datetime or timestamp to data type to a value of VARCHAR2 data type in the format specified by the date format: to_char(sys-	TO_CHAR (datetime)	Convert time stamp to string: TO_CHAR(now(), 'DD-MON-YYYY HH24:MI:SS') --> 01-JAN-2017 01:01:01	Yes

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
	date, 'DD-MON-YYYY HH24:MI:SS') --> 01-JAN-2017 01:01:01			
TRUNC (date)	Returns a date with the time portion of the day truncated to the unit specified by the format model: trunc(systimestamp) --> 2017-01-01 00:00:00	DATE_TRUNC	Truncate to specified precision: date_trunc('day', now()) --> 2017-01-01 00:00:00	No
ENCODING AND DECODING FUNCTIONS				
DECODE	Compares expr to each search value one by one using the functionality of an IF-THEN-ELSE statement	DECODE	PostgreSQL Decode function acts differently from Oracle's, PostgreSQL decode binary data from textual representation in string and does not have the functionality of an IF-THEN-ELSE statement	No
DUMP	Returns a VARCHAR2 value containing the data type code, length in bytes, and internal representation of expr.	N/A	N/A	No
ORA_HASH	Computes a hash value for a given expression.	N/A	N/A	No
NULL FUNCTIONS				
CASE	The CASE statement chooses from a sequence of conditions and runs a corresponding statement: CASE WHEN condition THEN result [WHEN ...] [ELSE result] END	CASE	The PostgreSQL CASE expression is a generic conditional expression, similar to if/else statements in other programming languages: CASE WHEN condition THEN result [WHEN ...] [ELSE result] END	Yes
COALESCE	Returns the first non-null expr in the expression list: coalesce (null, 'a', 'b') --> a	COALESCE	Returns the first of its arguments that is not null: coalesce (null, 'a', 'b') --> a	Yes
NULLIF	Compares expr1 and expr2. If they are equal, the function returns null. If they are not equal, the function returns expr1: NULLIF('a', 'b') --> a	NULLIF	Returns a null value if value1 equals value2 otherwise it returns value1: NULLIF ('a', 'b') --> a	Yes
NVL	Replace null (returned as a blank) with a string in the results of a query: NVL (null, 'a') --> a	COALESCE	Returns the first of its arguments that is not null: coalesce (null, 'a') --> a	No
NVL2	Determine the value returned by a query based on whether a specified expression is null or not null.	N/A	Can use the CASE statement instead.	No

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
ENVIRONMENT AND IDENTIFIER FUNCTIONS				
SYS_GUID	Generates and returns a globally unique identifier (RAW value) made up of 16 bytes: select sys_guid() from dual --> 5A280ABA8C76201EE0530-100007FF691	UUID_GENERATE_V1()	Generates a version 1 UUID: select uuid_generate_v1() --> 90791a6-a359-11e7-a61c-12803bf1597a	No
UID	Returns an integer that uniquely identifies the session user (the user who logged on): select uid from dual --> 84	N/A	Consider using the PostgreSQL current_user function along with other PostgreSQL built-in function to generate a UID.	No
USER	Returns the name of the session user: select user from dual	USER SESSION_USER CURRENT_USER CURRENT_SCHEMA()	User name or schema of current execution context: Select user; or select current_schema();	No
USERENV	Returns information about the current session using parameters: SELECT USERENV ('LANGUAGE') "Language" FROM DUAL	N/A	Please refer to the PostgreSQL documentation for a list of all system functions: https://www.postgresql.org/docs/13/static/functions-info.html	No
CONVERSION FUNCTIONS				
CAST	Converts one built-in data type or collection-typed value into another built-in data type or collection-typed value: cast ('10' as int) + 1 --> 11	CAST	Converting one data type into another: cast ('10' as int) + 1 --> 11	Yes
CONVERT	Converts a character string from a one-character set to another: select convert ('Ã‰ Ì Ø ABCDE', 'US7ASCII', 'WE8ISO8859P1') from dual	N/A	N/A	No
TO_CHAR (string / numeric)	Converts NCHAR, NVARCHAR2, CLOB, or NCLOB data to the database character set: select to_char ('01234') from dual --> 01234	TO_CHAR	Converts the first argument to the second argument: select to_char (01234, '00000') --> 01234	No
TO_DATE	Converts char of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of DATE data type: to_date('01Jan2017', 'DDMonYYYY') --> 01-JAN-	TO_DATE	Convert string to date: to_date ('01Jan2017', 'DDMonYYYY') --> 2017-01-01	Partly

Oracle Function	Function Definition	PostgreSQL Function	Function Definition	Equivalent
	17			
TO_NUMBER	Converts expr to a value of NUMBER data type: to_number('01234') --> 1234 or to_number('01234', '99999') --> 1234	TO_NUMBER	Convert string to numeric: to_number('01234', '99999') --> 1234	Partly

AGGREGATE FUNCTIONS

AVG	AVG returns average value of expr: select avg(salary) from employees	AVG	Average (arithmetic mean) of all input values: select avg(salary) from employees	Yes
COUNT	Returns the number of rows returned by the query: select count(*) from employees	COUNT	The number of input rows: select count(*) from employees	Yes
LISTAGG	Orders data within each group specified in the ORDER BY clause and then concatenates the values of the measure column: select listagg(firstname, ',') within group (order by customerid) from customer	STRING_AGG	Input values concatenated into a string, separated by delimiter: select string_agg(firstname, ',') from customer order by 1;	No
MAX	Returns the maximum value of expr: select max(salary) from employees	MAX	Returns maximum value of expr: select max(salary) from employees	Yes
MIN	Returns the minimum value of expr: select min(salary) from employees	MIN	Returns minimum value of expr: select min(salary) from employees	Yes
SUM	Returns the sum of values of expr: select sum(salary) from employees	SUM	Returns the sum of values of expr: select sum(salary) from employees	Yes

Top-N Query Oracle 12c

FETCH	Retrieves rows of data from the result set of a multi-row query: select * from customer fetch first 10 rows only	FETCH OR LIMIT	Retrieve just a portion of the rows that are generated by the rest of the query: select * from customer fetch first 10 rows only	Yes
-------	--	----------------	--	-----

REGEXP_MATCH is a new pattern matching function that was introduced in PostgreSQL 10:

```
SELECT REGEXP_MATCH('foobarbequebaz', 'bar.*que');
regexp_match
-----
{barbeque}
```

For more details, see:

- <https://www.postgresql.org/docs/13/static/functions.html>
- <https://www.postgresql.org/docs/13/static/functions-math.html>

- <https://www.postgresql.org/docs/13/static/functions-string.html>
- <https://www.postgresql.org/docs/13/static/uuid-ossp.html>

Oracle Create Table as Select (CTAS) vs. PostgreSQL Create Table as Select (CTAS)

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	

Oracle Usage

The Create Table As Select (CTAS) statement creates a new table based on an existing table. It copies the table DDL definitions (column names and column datatypes) and 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.

Examples

Create a table based on an existing table and include data from all columns.

```
CREATE TABLE EMPS
AS
SELECT * FROM EMPLOYEES;
```

Create a table based on an existing table with select columns.

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

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

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

```
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_CREATE TABLE EMPS AS SELECT * FROM EMPLOYEES;
pg_CREATE TABLE EMPS AS
    SELECT EMPLOYEE_ID, FIRST_NAME, SALARY FROM EMPLOYEES ORDER BY 3 DESC;
```

2. PostgreSQL CTAS with no data:

```
pg_CREATE TABLE EMPS AS SELECT * FROM EMPLOYEES WITH NO DATA;
```

For more details, see <https://www.postgresql.org/docs/13/static/sql-createtableas.html>

Oracle Common Table Expression (CTE) vs. PostgreSQL Common Table Expressions (CTE)

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	

Oracle Usage

Common Table Expressions (CTE) provide a way to implement the logic of sequential code or to reuse code. You can define a named sub query and then use it multiple times in different parts of a query statement.

A 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.

Syntax

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

Examples

Create a sub query of the employee count for each department and then use the result set of the CTE in a query.

```
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 Usage

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).

Examples

A PostgreSQL CTE:

```
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:

```
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;

n
--
0
1
2
3
4
5
```

For additional details:

<https://www.postgresql.org/docs/13/static/queries-with.html>

Oracle Identity Columns vs. PostgreSQL SERIAL Type

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Identity and Sequences	Since PostgreSQL 10, there are no differences besides the data types

Oracle Usage

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.

Examples

1. Create a table with an Oracle 12c Identity Column:

```
CREATE TABLE IDENTITY_TST (COL1 NUMBER GENERATED BY DEFAULT AS IDENTITY(START WITH 100
INCREMENT BY 10), COL2 VARCHAR2(30));
```

2. Insert data into the table. The Identity Column automatically generates values for COL1.

```
INSERT INTO IDENTITY_TST(COL2) VALUES ('A');
INSERT INTO IDENTITY_TST(COL1, COL2) VALUES(DEFAULT, 'B');
INSERT INTO IDENTITY_TST(col1, col2) VALUES(NULL, 'C');
SELECT * FROM IDENTITY_TST;
```

COL1	COL2
100	A
110	B

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

PostgreSQL enables you to create a Sequence that is similar to the IDENTITY 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.

Since PostgreSQL 10, there is a new option called identity columns which is similar to SERIAL data type but more SQL standard compliant. The identity columns are highly compatible compare to Oracle's Identity columns.

Examples

Using the PostgreSQL SERIAL pseudo-type (with a Sequence that is created implicitly):

```
CREATE TABLE SERIAL_SEQ_TST(COL1 SERIAL PRIMARY KEY, COL2 VARCHAR(10));

\ds
 Schema |          Name          |   Type    | Owner
-----+-----+-----+-----+
 public | serial_seq_tst_col1_seq | sequence | pg_tst_db

ALTER SEQUENCE SERIAL_SEQ_TST_COL1_SEQ RESTART WITH 100 INCREMENT BY 10;
INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
INSERT INTO SERIAL_SEQ_TST(COL1, COL2) VALUES(DEFAULT, 'B');
SELECT * FROM SERIAL_SEQ_TST;

col1 | col2
-----+-----
100  | A
110  | B
```

To create a table with Identity columns please use the following (this command is Oracle compatible):

```
CREATE TABLE emps (
    emp_id      INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    emp_name    VARCHAR(35) NOT NULL;

INSERT INTO emps (emp_name) VALUES ('Robert');
INSERT INTO emps (emp_id, emp_name) VALUES (DEFAULT, 'Braian');

SELECT * FROM emps;

col1 | col2
-----+-----
1    | Robert
2    | Braian
```

Note: Important to know that in PostgreSQL (both SERIAL and IDENTITY), you can insert any value that you want that won't violate the primary key constraint. If you do that and after that, you will use the identity column sequence value again, the error below might raise:

SQL Error [23505]: ERROR: duplicate key value violates unique constraint "emps_iden_pkey"
Detail: Key (emp_id)=(2) already exists.

For additional details, see:

- <https://www.postgresql.org/docs/13/static/sql-createsequence.html>
- <https://www.postgresql.org/docs/13/static/functions-sequence.html>
- <https://www.postgresql.org/docs/13/static/datatype-numeric.html>
- <https://www.postgresql.org/docs/13/sql-createtable.html>

Oracle Insert From Select vs. PostgreSQL Insert From Select

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		SQL	ERROR LOG and subpartition options are not supported by PostgreSQL

Oracle Usage

You can insert multiple records into a table from another table using the INSERT FROM SELECT statement, which is a derivative of the basic INSERT statement. The column ordering and data types must match between the target and the source tables.

Examples

Simple INSERT FROM SELECT (Explicit):

```
INSERT INTO EMPS (EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID) SELECT EMPLOYEE_ID,
FIRST_NAME, SALARY, DEPARTMENT_ID
FROM EMPLOYEES
WHERE SALARY > 10000;
```

Simple INSERT FROM SELECT (Implicit):

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

This example produces the same result as the preceding example but uses a subquery in the DML_table_expression_clause.

```
INSERT INTO
(SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID FROM EMPS)
VALUES (120, 'Kenny', 10000, 90);
```

Log errors with the Oracle error_logging_clause.

```
ALTER TABLE EMPS ADD CONSTRAINT PK_EMP_ID PRIMARY KEY(employee_id);
EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('EMPS', 'ERRLOG');
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, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/INSERT.html#GUID-903F8043-0254-4EE9-ACC1-CB8AC0AF3423>

PostgreSQL Usage

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.

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
[ OVERRIDING { SYSTEM | USER} VALUE ]
{ 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 [, ...] ) = [ ROW ]( { expression | DEFAULT } [, ...] ) |
    ( column_name [, ...] ) = ( sub-SELECT )
    } [, ...]
[ WHERE condition ]
```

Note: OVERRIDING is a new option since PostgreSQL 10 and relevant for identity columns. SYSTEM VALUE is only for identity column where GENERATE ALWAYS exists; if it's not there and it was specified, then PostgreSQL just ignores it.

Examples

Simple INSERT FROM SELECT (Explicit):

```
INSERT INTO EMPS (EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID)
    SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID
    FROM EMPLOYEES
    WHERE SALARY > 10000;
```

Simple Insert from Select (Implicit):

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

The following example is not compatible with the MySQL.

```
INSERT INTO
    (SELECT EMPLOYEE_ID, FIRST_NAME, SALARY, DEPARTMENT_ID FROM EMPS)
VALUES (120, 'Kenny', 10000, 90);
```

The following example demonstrates using the ON DUPLICATE KEY UPDATE clause to update specific columns when a UNIQUE violation occurs.

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

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;

INSERT 0
```

For more details, see <https://www.postgresql.org/docs/13/static/sql-insert.html>

Oracle Multi-Version Concurrency Control (MVCC) vs. PostgreSQL Multi-Version Concurrency Control (MVCC)

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	PostgreSQL uses auto commit by default (can be changed)

Oracle Usage

Two primary lock types exist in Oracle: 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 users to issue explicit locks on a specific tables using the LOCK TABLE statement.

Lock types can be divided into four categories: DML locks, DDL locks, Explicit (Manual) data locking, and System locks. The following sections describe each category.

DML Locks

DML locks preserve the integrity of data 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:

Statement	Row Locks	Table Lock Mode	RS	RX	S	SRX	X
SELECT ... FROM table...	–	none	Y	Y	Y	Y	Y
INSERT INTO table...	Yes	SX	Y	Y	N	N	N
UPDATE table ...	Yes	SX	Y	Y	N	N	N
MERGE INTO table ...	Yes	SX	Y	Y	N	N	N
DELETE FROM table...	Yes	SX	Y	Y	N	N	N
SELECT ... FROM table FOR UPDATE OF...	Yes	SX	Y	Y	N	N	N
LOCK TABLE table IN...	–						
ROW SHARE MODE		SS	Y	Y	Y	Y	N
ROW EXCLUSIVE MODE		SX	Y	Y	N	N	N
SHARE MODE		S	Y	N	Y	N	N
SHARE ROW EXCLUSIVE MODE		SSX	Y	N	N	N	N
EXCLUSIVE MODE		X	N	N	N	N	N

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

Users have the ability to explicitly create locks 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 performed 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

System locks include Latches, Mutexes, and internal locks.

Examples

Explicitly lock data using the LOCK TABLE command.

```
-- Session 1
LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;

-- Session 2
UPDATE EMPLOYEES
SET SALARY=SALARY+1000
WHERE EMPLOYEE_ID=114;

-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

Explicitly lock data using the SELECT... FOR UPDATE command. Oracle obtains exclusive row-level locks on all the rows identified by the SELECT FOR UPDATE statement.

```
-- Session 1
SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;

-- Session 2
UPDATE EMPLOYEES
SET SALARY=SALARY+1000
WHERE EMPLOYEE_ID=114;
-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Automatic-Locks-in-DDL-Operations.html#GUID-84D392A3-94EC-444D-950F-7829DBCD43EE>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Automatic-Locks-in-DML-Operations.html#GUID-3D57596F-8B73-4C80-8F4D-79A12F781EFD>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Automatic-and-Manual-Locking-Mechanisms-During-SQL-Operations.html#GUID-0304C4AA-BD28-4C2A-B7F5-267532FB9499>

PostgreSQL Usage

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, Postgres **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:

```
\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:

Requested Lock Mode VS current	ACCESS SHARE	ROW SHARE	ROW EXCLUSIV-E	SHARE UPDATE EXCLUSIV-E	SHAR-E	SHARE ROW EXCLUSIV-E	EXCLUSIV-E	ACCESS EXCLUSIV-E
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIV-E					X	X	X	X
SHARE UPDATE EXCLUSIV-E				X	X	X	X	X
SHARE			X	X	X	X	X	X
SHARE ROW EXCLUSIV-E			X	X	X	X	X	X
EXCLUSIV-E		X	X	X	X	X	X	X
ACCESS EXCLUSIV-E	X	X	X	X	X	X	X	X

2. Row-level Locks:

Requested Lock Mode VS current	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X

Requested Lock Mode VS current	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

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
```

- 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
START TRANSACTION;
LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;

-- Session 2
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
START TRANSACTION;
SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;
```

```
-- Session 2
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;

Session 2 - step2:
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;

Session 2 step3:
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;

Session 1 - step4:
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

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

Monitor locks using a SQL query:

```
SELECT
    block.pid                               AS block_pid,
    block_stm.username                      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.username                   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;
```

Generate an explicit lock using the SELECT... FOR UPDATE statement:

```
-- Session 1
START TRANSACTION;
SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;

-- Session 2
UPDATE EMPLOYEES
    SET SALARY=SALARY+1000
    WHERE EMPLOYEE_ID=114;

-- Session 2 waits for session 1 to COMMIT or ROLLBACK
```

Run the SQL query from step #1 monitoring locks while distinguishing between the “blocking” and “waiting” session:

```
-[ RECORD 1 ]-----+
block_pid      | 31743
blocker_user   | aurora_admin
block_mode     | ExclusiveLock
block_locktype | transactionid
block_table    |
block_query    | SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID=114 FOR UPDATE;
block_granted  | t
waiting_locktype | transactionid
waiting_user   | aurora_admin
waiting_table  |
waiting_query  | UPDATE EMPLOYEES
                |     SET SALARY=SALARY+1000
                |     WHERE EMPLOYEE_ID=114;
waiting_mode   | ShareLock
waiting_pid    | 31996
```

Summary

Description	Oracle	PostgreSQL
“Dictionary” tables to obtain information about locks	v\$lock; v\$locked_object; v\$session_blockers;	pg_locks pg_stat_activity
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... KEY SHRE SHARE NO KEY UPDATE UPDATE

For additional details, see:

- <https://www.postgresql.org/docs/13/static/sql-lock.html>
- <https://www.postgresql.org/docs/13/static/explicit-locking.html>

Oracle Merge vs PostgreSQL Merge

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		MERGE	MERGE is not supported by PostgreSQL, workaround available

Oracle Usage

The MERGE statement provides a means to specify single SQL statements that conditionally perform INSERT, UPDATE, or DELETE operations on a 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 UPSERT.

Examples

Use MERGE to insert or update employees who are entitled to a bonus (by year):

```

CREATE TABLE EMP_BONUS(EMPLOYEE_ID NUMERIC,BONUS_YEAR VARCHAR2(4),
SALARY NUMERIC,BONUS NUMERIC, PRIMARY KEY (EMPLOYEE_ID, BONUS_YEAR));

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);

SELECT * FROM EMP_BONUS;

```

EMPLOYEE_ID	BONU	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 more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/MERGE.html#GUID-5692CCB7-24D9-4C0E-81A7-A22436DC968F>

PostgreSQL Usage

PostgreSQL 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.

Examples

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

```

CREATE TABLE EMP_BONUS (
EMPLOYEE_ID NUMERIC,
BONUS_YEAR VARCHAR(4),
SALARY NUMERIC,
BONUS NUMERIC,
PRIMARY KEY (EMPLOYEE_ID, BONUS_YEAR));

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;

SELECT * FROM EMP_BONUS;
employee_id | bonus_year | salary | bonus
-----+-----+-----+-----
103 | 2017 | 9000.00 | 4500.000
104 | 2017 | 6000.00 | 3000.000
105 | 2017 | 4800.00 | 2400.000
106 | 2017 | 4800.00 | 2400.000
107 | 2017 | 4200.00 | 2100.000
109 | 2017 | 9000.00 | 4500.000
110 | 2017 | 8200.00 | 4100.000
111 | 2017 | 7700.00 | 3850.000
112 | 2017 | 7800.00 | 3900.000
113 | 2017 | 6900.00 | 3450.000
115 | 2017 | 3100.00 | 1550.000
116 | 2017 | 2900.00 | 1450.000
117 | 2017 | 2800.00 | 1400.000
118 | 2017 | 2600.00 | 1300.000

```

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 more information, see:

<https://www.postgresql.org/docs/13/static/sql-insert.html>

<https://www.postgresql.org/docs/13/static/unsupported-features-sql-standard.htm>

Oracle OLAP Functions vs. PostgreSQL Window Functions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		OLAP Functions	GREATEST and LEAST functions might get different results in PostgreSQL CONNECT BY is not supported by PostgreSQL, workaround available

Oracle Usage

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 as an alternative to achieving the same result using more complex non-OLAP SQL code.

Common Oracle OLAP Functions:

Function Type	Related Functions
Aggregate	average_rank, avg, count, dense_rank, max, min, rank ,sum
Analytic	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
Hierarchical	hier_ancestor, hier_child_count,, hier_depth, hier_level, hier_order, hier_parent, hier_top
Lag	lag, lag_variance, lag_variance_percent, lead, lead_variance, lead_variance_percent
OLAP DML	olap_dml_expression
Rank	average_rank ,dense_rank, rank, row_number

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/OLAP-Functions.html#GUID-2AE523A7-630C-4907-B91B-89861C141EBD>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Functions.html#GUID-D079EFD3-C683-441F-977E-2C9503089982>

PostgreSQL Usage

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:

Function Type	Related Functions
Aggregate	avg, count, max, min, sum, string_agg
Ranking	row_number, rank, dense_rank, percent_rank, cume_dist, ntile, lag, lead, first_value, last_value, nth_value

Examples

The Oracle rank() function & PostgreSQL rank() function providing the same results

Oracle:

```
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;
```

DEPARTMENT_ID	LAST_NAME	SALARY	COMMISSION_PCT	Rank
80	Russell	14000	.4	1
80	Partners	13500	.3	2
80	Errazuriz	12000	.3	3

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;
```

DEPARTMENT_ID	LAST_NAME	SALARY	COMMISSION_PCT	Rank
80	Russell	14000.00	0.40	1
80	Partners	13500.00	0.30	2
80	Errazuriz	12000.00	0.30	3

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.

Example

PostgreSQL generate_series function:

```
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/13/static/functions-window.html>
<https://www.postgresql.org/docs/13/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.

Function Type	Related Functions
Aggregate	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)
Ranking	DENSE_RANK NTILE PERCENT_RANK RANK ROW_NUMBER

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

Summary

Oracle OLAP Function	Returned Data Type	PostgreSQL Window Function	Returned Data Type	Compatible Syntax
Count	Number	Count	bigint	Yes
Max	Number	Max	numeric, string, date/time, network or enum type	Yes
Min	Number	Min	numeric, string, date/time, network or enum type	Yes
Avg	Number	Avg	numeric, double, otherwise same datatype as the argument	Yes
Sum	Number	Sum	bigint, otherwise same datatype as the argument	Yes
rank()	Number	rank()	bigint	Yes
row_number()	Number	row_number()	bigint	Yes
dense_rank()	Number	dense_rank()	bigint	Yes
percent_rank()	Number	percent_rank()	double	Yes
cume_dist()	Number	cume_dist()	double	Yes
ntile()	Number	ntile()	integer	Yes
lag()	same type as value	lag()	same type as value	Yes
lead()	same type as value	lead()	same type as value	Yes
first_value()	same type as value	first_value()	same type as value	Yes
last_value()	same type as value	last_value()	same type as value	Yes

Oracle Sequences vs PostgreSQL Sequences

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Identity and Sequences	Different syntax for a few options in PostgreSQL

Oracle Usage

Sequences are database objects that serve as unique identity value generators, for example, automatically generating primary key values. Oracle treats sequences as independent objects. The same sequence can generate values for multiple tables.

Sequences can be configured with multiple parameters to 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 18c introduces scalable sequences: a special class of sequences that are optimized for multiple concurrent session usage.

This introduce 3 new options when creating a new sequence:

- SCALE - enable the sequence scalability feature
 - EXTEND - this option will extend in additional 6 digits offset (as default) and the maximum number of digits in the sequence (maxvalue/minvalue)
 - NOEXTEND (default when using the SCALE option) - sequence value will be padded to the max value
- NOSCALE - non-scalable sequence usage

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:** Specifies the maximum limit for values generated by a sequence. It 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 1027 for an ascending sequence.
- **MINVALUE | NOMINVALUE:** Specifies 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 -1026 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:** Specifies the number of sequence values to keep cached in memory for improved performance. CACHE has a minimum value of 2. The NOCACHE parameter causes a sequence to not cache values in memory. Specifying neither CACHE nor NOCACHE will cache 20 values to memory. In the event of a database failure, all unused cached sequence values are lost and gaps in sequence values may occur.
- **SCALE | NOSCALE:** Enable the scalable sequences feature (described above)

Examples

Create a sequence.

```
CREATE SEQUENCE SEQ_EMP
START WITH 100
INCREMENT BY 1
MAXVALUE 999999999999
CACHE 20
NOCYCLE;
```

Drop a sequence.

```
DROP SEQUENCE SEQ_EMP;
```

View sequences created for the current schema / user.

```
SELECT * FROM USER_SEQUENCES;
```

Use a sequence as part of an INSERT INTO statement.

```
CREATE TABLE EMP_SEQ_TST (COL1 NUMBER PRIMARY KEY, COL2 VARCHAR2(30));
INSERT INTO EMP_SEQ_TST VALUES (SEQ_EMP.NEXTVAL, 'A');

COL1          COL2
-----  -----
100           A
```

Query the current value of a sequence.

```
SELECT SEQ_EMP.CURRVAL FROM DUAL;
```

Manually increment the value of a sequence according to the INCREMENT BY specification.

```
SELECT SEQ_EMP.NEXTVAL FROM DUAL;
```

Alter an existing sequence.

```
ALTER SEQUENCE SEQ_EMP MAXVALUE 1000000;
```

Create a scalable sequence.

```
CREATE SEQUENCE scale_seq
MINVALUE 1
MAXVALUE 9999999999
SCALE;

select scale_seq.nextval as scale_seq from dual;

NEXTVAL |
-----|
1010320001|
```

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.

Generate DEFAULT values using sequences.

```
CREATE TABLE SEQ_TST ( COL1 NUMBER DEFAULT SEQ_1.NEXTVAL PRIMARY KEY, COL2 VARCHAR
(30));
INSERT INTO SEQ_TST(COL2) VALUES('A');
SELECT * FROM SEQ_TST;
COL1 COL2
-----
100 A
```

Oracle 12c Session Sequences (Session/Global)

Beginning with Oracle 12c, sequences can be created as session-level or global-level. By adding the SESSION parameter to a CREATE SEQUENCE statement, the sequence is created as a session-level sequence. Optionally, the GLOBAL keyword can be used to create 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.

Create Oracle 12c SESSION and GLOBAL sequences.

```
CREATE SEQUENCE SESSION_SEQ SESSION;
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.

Insert records using an Oracle 12c IDENTITY column (explicitly/implicitly).

```
INSERT INTO IDENTITY_TST(COL2) VALUES('A');
INSERT INTO IDENTITY_TST(COL1, COL2) VALUES(DEFAULT, 'B');
INSERT INTO IDENTITY_TST(col1, col2) VALUES(NULL, 'C');
SELECT * FROM IDENTITY_TST;
```

COL1	COL2
120	A
130	B

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-SEQUENCE.html#GUID-E9C78A8C-615A-4757-B2A8-5E6EFB130571>

PostgreSQL Usage

Sequences in PostgreSQL serve the same purpose as in Oracle; they generate numeric identifiers automatically. The PostgreSQL CREATE SEQUENCE command is mostly compatible with the Oracle CREATE SEQUENCE command. A sequence object is owned by the user that created it.

Oracle 18c introduces scalable sequences, this feature is not always needed but if it and the current PostgreSQL is not scalable enough, other solutions and services can be used to allow high-concurrency data read (to store only sequences data), this option will require more changes in the application layer.

PostgreSQL Sequence Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
[ AS data_type ]
[ 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 PostgreSQL CREATE SEQUENCE parameters are compatible with Oracle. 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.

- **AS**

AS data_type is a new option in PostgreSQL 10. It is for specifying the data type of the sequence. The available options are smallint, integer, and bigint (default). This also determines the maximum and minimum values.

- **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 263 for ascending sequences and -1 for descending sequences.

- **MINVALUE | NO MINVALUE**

Defaults are between 1 for ascending sequences and -263 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.

Examples

Create a sequence:

```
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:

```
DROP SEQUENCE SEQ_1;
```

View sequences created in the current schema and sequence specifications:

```
SELECT * FROM INFORMATION_SCHEMA.SEQUENCES;
OR
\ds
```

Use a PostgreSQL sequence as part of a CREATE TABLE and an INSERT statement:

```
CREATE TABLE SEQ_TST
    (COL1 NUMERIC DEFAULT NEXTVAL('SEQ_1') PRIMARY KEY, COL2 VARCHAR(30));
INSERT INTO SEQ_TST (COL2) VALUES ('A');
SELECT * FROM SEQ_TST;

col1 | col2
-----+-----
 100 | A
```

Use the OWNED BY parameter to associate the sequence with a table:

```
CREATE SEQUENCE SEQ_1 START WITH 100 INCREMENT BY 1 OWNED BY SEQ_TST.COL1;
```

Query the current value of a sequence:

```
SELECT CURRVAL('SEQ_1');
```

Manually increment a sequence value according to the INCREMENT BY value:

```
SELECT NEXTVAL('SEQ_1');
OR
SELECT SETVAL('SEQ_1', 200);
```

Alter an existing sequence:

```
ALTER SEQUENCE SEQ_1 MAXVALUE 1000000;
```

Note: To use the NEXTVAL function, the USAGE and UPDATE permissions on the sequence are needed. To use CURRVAL and LASTVAL functions, the USAGE and SELECT permissions on the sequence are needed.

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 like a regular sequence.

Examples

Using a SERIAL Sequence:

```
CREATE TABLE SERIAL_SEQ_TST(COL1 SERIAL PRIMARY KEY, COL2 VARCHAR(10));

INSERT INTO SERIAL_SEQ_TST(COL2) VALUES('A');
SELECT * FROM SERIAL_SEQ_TST;

col1 | col2
-----+-----
1   | A

\ds

Schema |          Name          |    Type     | Owner
-----+-----+-----+-----+
public | serial_seq_tst_col1_seq | sequence | pg_tst_db
```

Summary

Parameter/Feature	Compatibility with PostgreSQL	Comments
Create sequence syntax	Full, with minor differences	See Exceptions
INCREMENT BY	Full	
START WITH	Full	
MAXVALUE NOMAXVALUE	Full	Use "NO MAXVALUE"
MINVALUE NOMINVALUE	Full	Use "NO MINVALUE"
CYCLE NOCYCLE	Full	USE "NO CYCLE"
CACHE NOCACHE	PostgreSQL does not support the NOCACHE parameter but the default behavior is identical. The CACHE parameter is compatible with Oracle.	
Default values using sequences (Oracle 12c)	Supported by PostgreSQL	CREATE TABLE TBL (COL1 NUMERIC DEFAULT NEXTVAL ('SEQ_1')...)
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/13/static/sql-createsequence.html>
- <https://www.postgresql.org/docs/13/static/functions-sequence.html>
- <https://www.postgresql.org/docs/13/static/datatype-numeric.html>

Oracle Transaction Model vs. PostgreSQL Transactions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Transaction Model	PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions

Oracle Usage

Database transactions are a logical, atomic units of processing containing one or more SQL statements that may run concurrently alongside other transactions. The primary purpose of a transaction is to ensure the ACID model is enforced.

- **Atomicity:** All statements in a transaction are 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 constraints are violated, the entire transaction fails.
- **Isolation:** One transaction is not affected by the behavior of other concurrent 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

The ANSI/ISO SQL standard (SQL92) defines four levels of isolation. Each level provides a different approach for handling concurrent execution of database transactions. Transaction isolation levels manage the visibility of 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. For 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. Oracle supports the following isolation levels:

- **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 transaction only sees data changes that were committed. Uncommitted changes (“dirty reads”) are not possible.
- **Repeatable Read Isolation Level:** A 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 sequentially in the same order.

Isolation levels affect the following 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 level	Dirty read	Non-repeatable read	Phantom read
Read Uncommitted	Permitted	Permitted	Permitted
Read Committed	Not permitted	Permitted	Permitted
Repeatable Read	Not permitted	Not permitted	Permitted
Serializable	Not permitted	Not permitted	Not permitted

Oracle Isolation Levels

Oracle supports the Read Committed and Serializable isolation levels. It also provides a Read-Only isolation level which is not a part of the ANSI/ISO SQL standard (SQL92). Read Committed is the default.

- **Read Committed (default):** Each query executed within a transaction only sees data that was committed before the query itself. The Oracle database never allows 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:** 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 only committed changes are viewable. Oracle relies on the System Change Number (SCN) of the current transaction to obtain a consistent view of the database. Therefore, all database queries only return data committed with respect to the SCN at the time of query execution.

Setting Isolation Levels

Isolation levels can be changed at the transaction and session levels.

Examples

Change the isolation level at the transaction-level.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READ ONLY;
```

Change the isolation-level at a session-level.

```
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED;
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/transactions.html#GUID-B97790CB-DF82-442D-B9D5-50CCE6BF9FBD>

PostgreSQL Usage

The same ANSI/ISO SQL (SQL92) isolation levels apply to PostgreSQL, with several similarities and some differences:

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read Uncommitted	Permitted but not implemented in PostgreSQL	Permitted	Permitted
Read Committed	Not permitted	Permitted	Permitted
Repeatable Read	Not permitted	Not permitted	Permitted but not implemented in PostgreSQL
Serializable	Not permitted	Not permitted	Not permitted

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.

Starting with PostgreSQL 12 option AND CHAIN can be added to COMMIT or ROLLBACK commands to immediately start another transaction with the same parameters as preceding transaction.

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”.

Examples

Configure the isolation level for a specific transaction:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Configure the isolation level for a specific session:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

View the current isolation level:

```
SELECT CURRENT_SETTING('TRANSACTION_ISOLATION'); -- Session
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, see:

http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithParamGroups.html#USER_WorkingWithParamGroups.Modifying

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

Database Feature	Oracle	PostgreSQL
AutoCommit	Off	Depends Autocommit is turned off by default, however, some client tools like psql and more are setting this to ON by default. Check your client tool defaults or run the following command to check current configuration in psql: <pre>\echo :AUTOCOMMIT</pre>
MVCC	Yes	Yes
Default Isolation Level	Read Committed	Read Committed
Supported Isolation Levels	Serializable Read-only	Repeatable Reads Serializable Read-only
Configure Session Isolation Levels	Yes	Yes
Configure Transaction Isolation Levels	Yes	Yes
Nested Transaction Support	Yes	No Consider using SAVEPOINT instead
Support for Transaction SAVEPOINTS	Yes	Yes

Read-Committed Isolation Level

TX1	TX2	Comment
<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100;</pre> <hr/> <pre>employee_id salary -----+----- 100 24000.00</pre>	<pre>select employee_id, salary from EMPLOYEES where employee_id=100;</pre> <hr/> <pre>employee_id salary -----+----- 100 24000.00</pre>	Same results returned from both sessions

TX1	TX2	Comment
<pre>begin; UPDATE employees SET salary=27000 WHERE employee_id=100;</pre>	<pre>begin; set transaction isolation level read committed;</pre>	TX1 starts a transaction; performs an update. TX2 starts a transaction with read-committed isolation level
<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 27000.00</pre>	<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 24000.00</pre>	TX1 will “see” the modified results (27000.00) while TX2 “sees” the original data (24000.00)
	<pre>UPDATE employees SET salary=29000 WHERE employee_id=100;</pre>	Waits as TX2 is blocked by TX1
Commit;		TX1 issues a commit, and the lock is released
	Commit;	TX2 issues a commit
<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 29000.00</pre>	<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 29000.00</pre>	Both queries return the value - 29000.00

Serializable Isolation Level

TX1	TX2	Comment
<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 24000.00</pre>	<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 24000.00</pre>	Same results returned from both sessions
<pre>begin; UPDATE employees SET salary=27000 WHERE employee_id=100;</pre>	<pre>begin; set transaction isolation level serializable;</pre>	TX1 starts a transaction; performs an update. TX2 starts a transaction with isolation level of read committed
<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 27000.00</pre>	<pre>SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 24000.00</pre>	TX1 will “see” the modified results (27000.00) while TX2 “sees” the original data (24000.00)
	update employees set salary-	Waits as TX2 is blocked by TX1

TX1	TX2	Comment
	=29000 where employee_id=100;	
Commit;		TX1 issues a commit, and the lock is released
	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
SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 27000.00	SELECT employee_id, salary FROM EMPLOYEES WHERE employee_id=100; employee_id salary -----+----- 100 27000.00	Both queries will return the data updated according to TX1

For additional details:

<https://www.postgresql.org/docs/13/static/tutorial-transactions.html>

<https://www.postgresql.org/docs/13/static/transaction-iso.html>

<https://www.postgresql.org/docs/13/static/sql-set-transaction.html>

Oracle Anonymous Block vs. PostgreSQL DO

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Stored Procedures PL/SQL	Different syntax may require code rewrite.

Oracle Usage

Oracle PL/SQL is a procedural extension of SQL. The PL/SQL program structure divides the code into blocks 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:

```
SET SERVEROUTPUT ON;
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.

```
SET SERVEROUTPUT ON;
DECLARE
v_sal_chk      NUMBER;
v_emp_work_years NUMBER;
v_sql_cmd      VARCHAR2(2000);
BEGIN
FOR v IN (SELECT EMPLOYEE_ID, FIRST_NAME||' '||LAST_NAME AS
EMP_NAME, HIRE_DATE, SALARY FROM EMPLOYEES)
LOOP
v_emp_work_years:=EXTRACT(YEAR FROM SYSDATE) - EXTRACT (YEAR FROM v.hire_date);
IF v_emp_work_years>=10 and v.salary <= 6000 then
DBMS_OUTPUT.PUT_LINE('Consider a Bonus for: '||v.emp_name);
END IF;
```

```

END LOOP;
EXCEPTION WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('CODE ERR: '||sqlerrm);
END;
/

```

The above example calculates the number of 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 more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/overview.html#GUID-2FBCFBBE-6B42-4DB8-83F3-55B63B75B1EB>

PostgreSQL Usage

PostgreSQL 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:

```

SET CLIENT_MIN_MESSAGES = 'debug';
-- Equivalent To Oracle SET SERVEROUTPUT ON

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

DEBUG: hello world
DO

```

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:

```

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 more information, see: <https://www.postgresql.org/docs/13/static/sql-do.html>

Oracle PL/SQL Cursors vs. PostgreSQL Cursors

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Cursors	<p>TYPE ... IS REF CURSOR is not supported by PostgreSQL</p> <p>Minor different in syntax may require some code rewrite.</p> <p>%ISOPEN, %BULK_EXCEPTIONS, and %BULK_ROWCOUNT are not supported</p>

Oracle Usage

PL/SQL cursors are pointers to data sets on which application logic can iterate. The data sets hold rows returned by SQL statements. You can refer to the active data set in named cursors from within a program.

There are two types of PL/SQL cursors :

- **Implicit Cursors** are session cursors constructed and managed by PL/SQL automatically without being created or defined by a user. PL/SQL opens an implicit cursor each time you run a SELECT or DML statement. Implicit cursors are also called SQL Cursors.
- **Explicit Cursors** are session cursors created, constructed, and managed by a user. Cursors are declared and defined by naming it and associating it with a query. Unlike an implicit cursor, you can reference an explicit cursor using its name. An explicit cursor is called a Named Cursor.

Examples

The following examples demonstrate cursor usage:

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 and v_jobid.
4. The loop uses the %NOTFOUND attribute to terminate when the last row is read from the database.

```

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.

```
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 more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/explicit-cursor-declaration-and-definition.html#GUID-38C5DBA3-9DEC-4AF2-9B5E-7B721D11A77C>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/implicit-cursor-attribute.html#GUID-5A938EE7-E8D2-468C-B60F-81898F110BE1>

PostgreSQL Usage

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.

Examples Declare a Cursor

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

```
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:

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

FOR can be replaced by IS for Oracle compatibility:

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

```
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:

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

Note:

- 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.

Examples Open a Cursor

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

```
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.

```
OPEN c1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tablename) 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**.

```
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:

```
DO $$  
DECLARE  
    var1 integer;  
    c3 CURSOR FOR SELECT * FROM employees where id = var1;  
BEGIN  
    var1 := 1;  
    OPEN c3;  
END$$;
```

Examples 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:

```
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:

```
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:

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

For additional details: <https://www.postgresql.org/docs/13/static/sql-fetch.html>

Example: Close a Cursor

Close a PL/pgSQL cursor using the close command:

```
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):

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):

```
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:

```
MOVE LAST FROM c3;
```

2. Move the Cursor two records back:

```
MOVE RELATIVE -2 FROM c3;
```

3. Move the c3 Cursor two records forward.

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

Summary

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

For additional information on PostgreSQL PL/pgSQL and Cursors:

- <https://www.postgresql.org/docs/13/static/plpgsql-cursors.html>
- <https://www.postgresql.org/docs/13/static/plpgsql-statements.html>

Oracle DBMS_OUTPUT vs. PostgreSQL RAISE

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different paradigm and syntax will require application/drivers rewrite.

Oracle Usage

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

Examples

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.

```
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 more details, see https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/DBMS_OUTPUT.html#GUID-C1400094-18D5-4F36-A2C9-D28B0E12FD8C

PostgreSQL Usage

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

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

Examples

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

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

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 more details, see:

<https://www.postgresql.org/docs/13/static/plpgsql-errors-and-messages.html>

<https://www.postgresql.org/docs/13/static/runtime-config-logging.html#GUC-LOG-MIN-MESSAGES>

Summary

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

For additional details, see: <https://www.postgresql.org/docs/13/static/errcodes-appendix.html>

Oracle DBMS_RANDOM vs. PostgreSQL RANDOM Function

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different syntax may require code rewrite.

Oracle Usage

Oracle's DBMS_RANDOM package provides functionality for generating random numbers or strings as part of an SQL statement or PL/SQL procedure.

The DBMS_RANDOM Package stored procedures include:

- **NORMAL:** Returns random numbers in a standard normal distribution.
- **SEED:** Resets the seed that generates random numbers or strings.
- **STRING** Returns a random string.
- **VALUE:** Returns a number greater than or equal to 0 and less than 1 with 38 digits to the right of the decimal. Alternatively, you could generate a random number greater than or equal to a low parameter and less than a high parameter.

Notes:

- DBMS_RANDOM.RANDOM produces integers in the range [-2^^31, 2^^31].
- DBMS_RANDOM.VALUE produces numbers in the range [0,1] with 38 digits of precision.

Examples

Generate a random number.

```
select dbms_random.value() from dual;

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

select dbms_random.value() from dual;

DBMS_RANDOM.VALUE()
-----
.364792387
```

Generate a random string. The first character determines the returned string type and the number specifies the length.

```
select dbms_random.string('p',10) from dual;
DBMS_RANDOM.STRING('P',10)
```

```
-----
1a'?'z[Q&/2

select dbms_random.string('P',10) from dual;
DBMS_RANDOM.STRING('P',10)
-----
t?!Gf2M60q
```

For more details, see: https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/DBMS_RANDOM.html#GUID-8DC48B0C-3707-4172-A306-C0308DD2EB0F

PostgreSQL Usage

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.

Examples

Generate a random number.

```
select random();
       random
-----
0.866594325285405
(1 row)

select random();
       random
-----
0.524613124784082
(1 row)
```

2. Generate a random string:

```
select md5(random()::text);
       md5
-----
f83e73114eccfed571b43777b99e0795
(1 row)

select md5(random()::text);
       md5
-----
d46de3ce24a99d5761bb34bfb6579848
(1 row)
```

Summary

Description	Oracle	PostgreSQL
Generate a random number	select dbms_random.value() from dual;	select random();
Generate a random number between 1 to 100	select dbms_random.value(1,100) from dual;	select random()*100;
Generate a random string	select dbms_random.string('p',10) from dual;	select md5(random()::text);
Generate a random string in upper case	select dbms_random.string('U',10) from dual;	select upper(md5(random()::text));

For more details, see

<https://www.postgresql.org/docs/13/static/functions-math.html>

<https://www.postgresql.org/docs/13/static/functions-string.html>

Oracle DBMS_SQL Package (11g & 12c) vs. PostgreSQL Dynamic Execution

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different paradigm and syntax will require application/drivers rewrite.

Oracle Usage

The DBMS_SQL package provides an interface to parse and execute dynamic SQL statements, DML commands, and DDL commands (usually from within a PL/SQL package, function, or procedure). DBMS_SQL enables very granular control of SQL cursors and can improve cursor performance in certain cases.

Examples

The following examples demonstrates how to manually open, parse, bind, execute, and fetch data from a cursor using the DBMS_SQL PL/SQL interface.

1. Use DBMS_SQL.OPEN_CURSOR to open a blank cursor and return the cursor handle.
2. Use DBMS_SQLPARSE to parse the statement into the referenced cursor.
3. Use DBMS_SQL.BIND_VARIABLES to attach the value for the bind variable with the cursor.
4. Use DBMS_SQL.EXECUTE to execute the cursor.
5. Use DBMS_SQL.GET_NEXT_RESULT to iterate over the cursor, fetching the next result.
6. Use DBMS_SQL.CLOSE_CURSOR to close the cursor.

```

DECLARE
  c1          INTEGER;
  rc1         SYS_REFCURSOR;
  n           NUMBER;
  first_name  VARCHAR2(50);
  last_name   VARCHAR2(50);
  email       VARCHAR2(50);
  phone_number VARCHAR2(50);
  job_title   VARCHAR2(50);
  start_date  DATE;
  end_date    DATE;
BEGIN
  c1 := DBMS_SQL.OPEN_CURSOR(true);
  DBMS_SQL.PARSE
    (c1, 'BEGIN emp_info(:id); END;', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c1, ':id', 176);
  n := DBMS_SQL.EXECUTE(c1);

  -- Get employee info
  DBMS_SQL.GET_NEXT_RESULT(c1, rc1);
  FETCH rc1 INTO first_name, last_name, email, phone_number;

```

```
-- Get employee job history
DBMS_SQL.GET_NEXT_RESULT(c1, rcl);
LOOP
  FETCH rcl INTO job_title, start_date, end_date;
  EXIT WHEN rcl%NOTFOUND;
END LOOP;
DBMS_SQL.CLOSE_CURSOR(c1);
END;
/
```

There are three other procedures in DBMS_SQL package:

- **RETURN_RESULT:** (New in oracle 12c) Gets a result set and returns it to the client. Because the procedure already returns a result set, the invoker does not have to know the format of the result or the columns it contains (most often used with SQL*Plus).
- **TO_REFCURSOR:** When using DBMS_SQL.OPEN_CURSOR, the numeric cursor ID is returned. If you know the structure of the result of the cursor, you can call the TO_REFCURSOR procedure, stop working with DBMS_SQL, and move to regular commands such as FETCH, WHEN CURSOR%notfound, and others. Before using TO_REFCURSOR, use the procedures OPEN_CURSOR, PARSE and EXECUTE.
- **TO_CURSOR_NUMBER:** Gets a cursor opened in native dynamic SQL. After the cursor is open, it can be converted to a number (cursor id) and then managed using DBMS_SQL procedures.

For more details, see https://docs.oracle.com/en/database/oracle/oracle-database/19/arpis/DBMS_SQL.html#GUID-C96D5BAA-29A9-4AB5-A69E-E31228ECC9E9

PostgreSQL Usage

PostgreSQL doesn't support granular control of programmatic cursors and thus does not have an equivalent for Oracle's DBMS_SQL.

However, you can dynamically parse and execute SQL statements in PostgreSQL, here are two examples:

1. Create dynamic cursor by using FOR with SELECT

```
CREATE OR REPLACE FUNCTION GetErrors ()
RETURNS VARCHAR
AS
$$
DECLARE
  _currow      RECORD;
msg      VARCHAR(200);
TITLE    VARCHAR(10);
CODE_NUM   VARCHAR(10);
BEGIN
  msg := '';

  FOR _currow IN SELECT TITLE, CODE_NUM, count(*) FROM A group by TITLE, CODE_NUM
  LOOP
    TITLE := _currow.TITLE;
    CODE_NUM := _currow.CODE_NUM;
    msg := msg || rpad(TITLE, 20) || rpad(CODE_NUM, 20);
  END LOOP;
END;
```

```

RETURN msg;

END;
$$ LANGUAGE plpgsql;

```

2. Create cursor and then open it for execute with given SQL

```

CREATE OR REPLACE FUNCTION GetErrors () RETURNS VARCHAR AS $$

declare
    refcur refcursor;
    c_id integer;
    title varchar (10);
    code_num varchar (10);
    alert_mesg VARCHAR(1000) := '';
BEGIN
    OPEN refcur FOR execute('select * from Errors');
    loop
        fetch refcur into title, code_num;
        if not found then
            exit;
        end if;
        alert_mesg := alert_mesg||rpad(title,20)||rpad(code_num,20);
    end loop;
    close refcur;
    return alert_mesg;
END;
$$ LANGUAGE plpgsql

```

For additional details, see:

<https://www.postgresql.org/docs/13/static/sql-deallocate.html>

<https://www.postgresql.org/docs/13/static/sql-prepare.html>

For additional details, see: <https://www.postgresql.org/docs/13/static/plpgsql-statements.html#PLPGSQL-STATEMENTS-EXECUTING-DYN>

Oracle EXECUTE IMMEDIATE vs. PostgreSQL EXECUTE and PREPARE

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different syntax may require code rewrite. Remove IMMEDIATE keyword

Oracle Usage

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.

Examples

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 and col_val.

```
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.

```
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.

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

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/EXECUTE-IMMEDIATE-statement.html#GUID-C3245A95-B85B-4280-A01F-12307B108DC8>

PostgreSQL Usage

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.

Examples

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.

```
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:

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

Note:

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

3. Execute a DDL command:

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

For additional details: <https://www.postgresql.org/docs/13/static/functions-string.html>

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.

```
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');
```

Summary

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

For additional details: <https://www.postgresql.org/docs/13/static/plpgsql-statements.html>

Oracle Procedures and Functions vs. PostgreSQL Stored Procedures

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Stored Procedures	Syntax and option differences

Oracle Usage

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, database users must have the CREATE ANY PROCEDURE privilege.
- To execute a procedure or function, database users must have the EXECUTE privilege.

Package and Package Body

In addition to stored procedures and functions, Oracle also provides Packages to 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.

```
EXEC PKG_EMP.CALCULATE_SAL('100');
```

Examples

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 it exists.

```
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
EXEC EMP_SAL_RAISE(200, 1000);

```

Create a function using the CREATE OR REPLACE FUNCTION statement.

```

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;
/
SELECT EMPLOYEE_ID,FIRST_NAME, EMP_PERIOD_OF_SERVICE_YEAR(EMPLOYEE_ID) AS PERIOD_OF_
SERVICE_YEAR FROM EMPLOYEES;
EMPLOYEE_ID FIRST_NAME PERIOD_OF_SERVICE_YEAR
-----
174 Ellen 13
166 Sundar 9
130 Mozhe 12
105 David 12
204 Hermann 15
116 Shelli 12
167 Amit 9
172 Elizabeth 10

```

Create a Package using the CREATE OR REPLACE PACKAGE statement.

```

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;

```

Create a new Package using the CREATE OR REPLACE PACKAGE BODY statement.

```

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;

EXEC PCK_CHINOOK_REPORTS.GET_ARTIST_BY_ALBUM();
EXEC PCK_CHINOOK_REPORTS.CUST_INVOICE_BY_YEAR_ANALYZE;

```

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

For more details, see

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-FUNCTION.html#GUID-156AEDAC-ADD0-4E46-AA56-6D1F7CA63306>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-PROCEDURE.html#GUID-771879D8-BBFD-4D87-8A6C-290102142DA3>

PostgreSQL Usage

PostgreSQL provides support for both stored procedures and stored functions using the CREATE FUNCTION statement. To emphasize, the procedural statements used by PostgreSQL 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 supports additional programming languages, also available in Amazon Aurora PostgreSQL:

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

Use the 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:

```
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:

```
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;

select emp_sal_raise(200, 1000);

```

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

```

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;

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 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

```
EXEC PCK_CHINOOK_REPORTS.GET_ARTIST_BY_ALBUM('');
```

PostgreSQL (converted using Amazon SCT):

- The \$ sign separates the package and the package name.

```
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:

```

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
set client_min_messages = 'debug';
-- Equivalent to Oracle SET SERVEROUTPUT ON
select chinook.pck_chinook_reports$get_artist_by_album(' Fireball');

```

2. Oracle Package - PCK_CHINOOK_REPORTS, Oracle Package Body - CUST_INVOICE_BY_YEAR_ANALYZE:

```

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;
END;
$function$;

-- Executing
SELECT chinook.pck_chinook_reports$cust_invoice_by_year_analyze();

```

New behavior in PostgreSQL version 10 for a set-returning function, used by LATERAL FROM-clause

Previous

```

CREATE TABLE emps (id int, manager int);
INSERT INTO tab VALUES (23, 24), (52, 23), (21, 65);
SELECT x, generate_series(1,5) AS g FROM tab;

```

```

id |g
---|--
23 |1
23 |2
23 |3
23 |4
23 |5
52 |1
52 |2
52 |3
52 |4
52 |5
21 |1
21 |2
21 |3
21 |4
21 |5

```

New

```

SELECT id, g FROM emps, LATERAL generate_series(1,5) AS g;

id |g
---|--
23 |1
23 |2
23 |3
23 |4
23 |5

```

```
52 |1  
52 |2  
52 |3  
52 |4  
52 |5  
21 |1  
21 |2  
21 |3  
21 |4  
21 |5
```

Here the planner could choose to put the set-return function on the outside of the nestloop join, since it has no actual lateral dependency on emps table

For additional details:

<https://www.postgresql.org/docs/13/static/sql-createfunction.html>
<https://www.postgresql.org/docs/13/static/plpgsql.html>
<https://www.postgresql.org/docs/13/static/xplang.html>
<https://www.postgresql.org/docs/13/static/xfunc-sql.html>

Oracle UDFs vs. PostgreSQL User Defined Functions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Procedures	Syntax and option differences

Oracle Usage

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 SQL statements wherever built-in SQL functions can appear.

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

Examples

Create a simple Oracle UDF with arguments for employee HIRE_DATE and SALARY as INPUT parameters and calculate the overall salary over the employee's years of service for the company.

```

CREATE OR REPLACE FUNCTION TOTAL_EMP_SAL_BY_YEARS
(p_hire_date DATE, p_current_sal NUMBER)
RETURN NUMBER
AS
v_years_of_service NUMBER;
v_total_sal_by_years NUMBER;
BEGIN
SELECT EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM to_date(p_hire_date))
INTO v_years_of_service FROM dual;
v_total_sal_by_years:=p_current_sal*v_years_of_service;
RETURN v_total_sal_by_years;
END;
/
-- Verifying
SELECT EMPLOYEE_ID, FIRST_NAME, TOTAL_EMP_SAL_BY_YEARS(HIRE_DATE, SALARY) AS TOTAL_
SALARY
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 more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-FUNCTION.html#GUID-156AEDAC-ADD0-4E46-AA56-6D1F7CA63306>

PostgreSQL Usage

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.

Examples

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

```

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;
$LANGUAGE PLPGSQL;

SELECT EMPLOYEE_ID, FIRST_NAME, TOTAL_EMP_SAL_BY_YEARS(HIRE_DATE, SALARY) AS TOTAL_
SALARY
FROM EMPLOYEES;

```

employee_id	first_name	total_salary
100	Steven	364000.00
101	Neena	204000.00
102	Lex	272000.00
103	Alexander	99000.00
104	Bruce	60000.00

105 David		57600.00
106 Valli		52800.00
107 Diana		42000.00
...		

For more details, see:

- <https://www.postgresql.org/docs/13/static/xfunc.html>
- <https://www.postgresql.org/docs/13/static/sql-createfunction.html>
- <http://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/Welcome.html>

Oracle UTL_FILE

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	PostgreSQL does not have UTL_FILE equivalent

Oracle Usage

Oracle's UTL_FILE PL/SQL package enables you to access files stored outside of the database such as files stored on the operating system, the database server, or a connected storage volume. 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.

Examples

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 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, see: For more details, see https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/UTL_FILE.html#GUID-EBC42A36-EB72-4AA1-B75F-8CF4BC6E29B4

PostgreSQL Usage

Amazon Aurora PostgreSQL does not currently provide a directly comparable alternative for Oracle UTL_FILE package

Oracle UTL_MAIL & UTL_SMTP (11g & 12c) vs. PostgreSQL Scheduled Lamda with SES

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Use Lambda Integration

Oracle UTL MAIL Usage

The Oracle UTL_MAIL package provides functionality for sending email messages. Unlike UTL_SMTP, which is more complex and provided in earlier versions of Oracle, UTL_MAIL supports attachments. For most cases, UTL_Mail is a better choice.

Examples

Install the required mail packages.

```
@{ORACLE_HOME}/rdbms/admin/utlmail.sql
@{ORACLE_HOME}/rdbms/admin/prvtmail.plb
```

Set the smtp_out_server parameter.

```
ALTER SYSTEM SET smtp_out_server = 'smtp.domain.com' SCOPE=BOTH;
```

Send an email message.

```
exec utl_mail.send('Sender@mailserver.com', 'recipient@mailserver.com', NULL, NULL,
'This is the subject', 'This is the message body', NULL, 3, NULL);
```

For more details, see https://docs.oracle.com/database/121/ARPLS/u_mail.htm

Oracle UTL_SMTP Usage

The Oracle UTL_SMTP package provides functionality for sending email messages and is useful for sending alerts about database events. Unlike UTL_MAIL, UTL SMTP is more complex and does not support attachments. For most cases, UTL_Mail is a better choice.

Examples

The following example demonstrates using UTL_SMTP procedures to send email messages:

Install the required scripts.

```
In oracle 12c:
@{ORACLE_HOME}/rdbms/admin/utlsmtip.sql
```

```
In oracle 11g:
```

```
@{ORACLE_HOME}/javavm/install/initjvm.sql
@{ORACLE_HOME}/rdbms/admin/initplsj.sql
```

Create and send an email message:

- UTL_SMTP.OPEN_CONNECTION opens a connection to the smtp server.
- UTL_SMTP.HELO initiates a handshake with the smtp server.
- UTL_SMTP.MAIL Initiates a mail transaction that obtains the senders details.
- UTL_SMTP.RCPT adds a recipient to the mail transaction.
- UTL_SMTP.DATA adds the message content.
- UTL_SMTP.QUIT terminates the SMTP transaction.

```
DECLARE
  smtpconn utl_smtp.connection;
BEGIN
  smtpconn := UTL_SMTP.OPEN_CONNECTION('smtp.mailserver.com', 25);
  UTL_SMTP.HELO(smtpconn, 'smtp.mailserver.com');
  UTL_SMTP.MAIL(smtpconn, 'sender@mailserver.com');
  UTL_SMTP.RCPT(smtpconn, 'recipient@mailserver.com');
  UTL_SMTP.DATA(smtpconn, 'Message body');
  UTL_SMTP.QUIT(smtpconn);
END;
/
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-resources-with-oracle-database-resource-manager.html#GUID-2BEF5482-CF97-4A85-BD90-9195E41E74EF>

PostgreSQL Usage

Aurora PostgreSQL does not provide native support for sending email message from the database. For alerting purposes, use the Event Notification Subscription feature to send email notifications to operators.

The only way to send Email from the database is to use the LAMBDA integration. For more information about Lambda, see <https://aws.amazon.com/lambda>.

Examples

Sending an Email from Aurora PostgreSQL via Lambda Integration

First, configure [AWS SES](#).

In the AWS console, navigate to **SES > SMTP Settings** and click **Create My SMTP Credentials**. Note the SMTP server name; you will use it in the Lambda function.

Using SMTP to Send Email with Amazon SES

You can send email through Amazon SES by using applications or programming languages. To do this, you need to use the SMTP interface, see [Using the Amazon SES SMTP Interface to Send Email](#).

To send email by using the SMTP interface, you need the following information:

Server Name:	email-smtp.eu-central-1.amazonaws.com
Port:	25, 465 or 587
Use Transport Layer Security (TLS):	Yes
Authentication:	Your SMTP credentials. See below for more info

To send email using the Amazon SES SMTP interface, you first have to create a set password that you use to connect to an Amazon SES SMTP endpoint. SMTP credentials are unique to one Region, so you have to generate a unique set of credentials for each Region.

To obtain SMTP credentials for the current Region, choose [Create My SMTP Credentials](#).

Create My SMTP Credentials

Note: Your SMTP credentials are different from your AWS access keys. The two types are [AWS Access Keys](#) and [Credentials With Amazon SES](#).

Enter a name for **IAM User Name (SMTP user)** and click **Create**.

Create User for SMTP

This form lets you create an IAM user for SMTP authentication with Amazon SES. Enter the name of a new IAM user or accept the default and click Create to set up your SMTP credentials.

IAM User Name: Maximum 64 characters

Hide More Information

Amazon SES uses AWS Identity and Access Management (IAM) to manage SMTP credentials. The IAM user name is case sensitive and may contain only alphanumeric characters and the symbols +, -, @, _.

SMTP credentials consist of a username and a password. When you click the Create button below, SMTP credentials will be generated for you.

The new user will be granted the following IAM policy:

```
Statement": [{"Effect": "Allow", "Action": "ses:SendRawEmail", "Resource": "*"}]
```

Create

Note the credentials; you will use them to authenticate with the SMTP server.

Note: After leaving this page, the credentials cannot be retrieved.

Your 1 User(s) have been created successfully.
This is the only time these SMTP security credentials will be available for download. Credentials for SMTP users are only available when creating the user. For your protection, you should never share your SMTP credentials with anyone.

SMTP_USER
SMTP Username: AMAZONSES
SMTP Password: AMAZONSES_PWD

Close **Download Credentials**

Navigate back to the SES page, click **Email Addresses** on the left, and click **Verify a New Email Address**. Before sending email, they must be verified.

SES Home

Identity Management

Domains

Email Addresses

Verify a New Email Address

Send a Test Email Remove

Search email addresses All identities

Email Address Identities

You have not verified any email addresses.
To verify an email address, click the Verify a New Email Address button above.

The next page indicates that the email is pending verification.

After the email is verified, create a table to store messages to be sent by the Lambda function.

```
CREATE TABLE emails (title varchar(600), body varchar(600), recipients varchar(600));
```

To create the Lambda function, navigate to the [Lambda page](#) and click **Create function**.

Lambda > Functions

Actions **Create function**

Filter by tags and attributes or search by keyword

Function name	Description	Runtime	Code size	Last Modified
ExecPg		Python 2.7	822.6 kB	3 hours ago

Select **Author from Scratch**, enter a name for your project, and select **Python 2.7** as the runtime. Be sure to use a role with the correct permissions. Click **Create function**.

Download this [Github project](#).

In your local environment, create two files: main.py and db_util.py. Cut and paste the content below into main.py and db_util.py respectively. Be sure to replace the code highlighted in red with values for your environment.

main.py:

```
#!/usr/bin/python
import sys
import logging
import psycopg2

from db_util import make_conn, fetch_data
def lambda_handler(event, context):
    query_cmd = "select * from mails"
    print query_cmd

    # get a connection, if a connect cannot be made an exception will be raised here
    conn = make_conn()

    result = fetch_data(conn, query_cmd)
    conn.close()

    return result
```

db_util.py:

```
#!/usr/bin/python
import psycopg2
import smtplib
import email.utils
from email.mime.multipart import MIME Multipart
from email.mime.text import MIMEText

db_host = 'YOUR_RDS_HOST'
db_port = 'YOUR_RDS_PORT'
db_name = 'YOUR_RDS_DBNAME'
db_user = 'YOUR_RDS_USER'
db_pass = 'YOUR_RDS_PASSWORD'

def sendEmail(recp, sub, message):
    # Replace sender@example.com with your "From" address.
    # This address must be verified.
    SENDER = 'PUT HERE THE VERIFIED EMAIL'
    SENDERNAME = 'AWS Lambda'

    # Replace recipient@example.com with a "To" address. If your account
    # is still in the sandbox, this address must be verified.
    RECIPIENT = recp

    # Replace smtp_username with your Amazon SES SMTP user name.
    USERNAME_SMTP = "YOUR_SMTP_USERNAME"

    # Replace smtp_password with your Amazon SES SMTP password.
    PASSWORD_SMTP = "YOUR_SMTP_PASSWORD"

    # (Optional) the name of a configuration set to use for this message.
    # If you comment out this line, you also need to remove or comment out
```

```

# the "X-SES-CONFIGURATION-SET:" header below.
CONFIGURATION_SET = "ConfigSet"

# If you're using Amazon SES in an AWS Region other than US West (Oregon),
# replace email-smtp.us-west-2.amazonaws.com with the Amazon SES SMTP
# endpoint in the appropriate region.
HOST = "YOUR_SMTP_SERVERNAME"
PORT = 587

# The subject line of the email.
SUBJECT = sub

# The email body for recipients with non-HTML email clients.
BODY_TEXT = ("Amazon SES Test\r\n"
            "This email was sent through the Amazon SES SMTP "
            "Interface using the Python smtplib package."
            )

# The HTML body of the email.
BODY_HTML = """<html>
<head></head>
<body>
<h1>Amazon SES SMTP Email Test</h1>"""+ message + """</body>
</html>
"""

# Create message container - the correct MIME type is multipart/alternative.
msg = MIMEMultipart('alternative')
msg['Subject'] = SUBJECT
msg['From'] = email.utils.formataddr((SENDERNAME, SENDER))
msg['To'] = RECIPIENT
# Comment or delete the next line if you are not using a configuration set
#msg.add_header('X-SES-CONFIGURATION-SET',CONFIGURATION_SET)

# Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(BODY_TEXT, 'plain')
part2 = MIMEText(BODY_HTML, 'html')

# Attach parts into message container.
# According to RFC 2046, the last part of a multipart message, in this case
# the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)

# Try to send the message.
try:
    server = smtplib.SMTP(HOST, PORT)
    server.ehlo()
    server.starttls()
    #stmpplib docs recommend calling ehlo() before & after starttls()
    server.ehlo()
    server.login(USERNAME_SMTP, PASSWORD_SMTP)
    server.sendmail(SENDER, RECIPIENT, msg.as_string())
    server.close()
# Display an error message if something goes wrong.

```

```

except Exception as e:
    print ("Error: ", e)
else:
    print ("Email sent!")

def make_conn():
    conn = None
    try:
        conn = psycopg2.connect("dbname='%s' user='%s' host='%s' password='%s'" % (db_
name, db_user, db_host, db_pass))
    except:
        print "I am unable to connect to the database"
    return conn

def fetch_data(conn, query):
    result = []
    print "Now executing: %s" % (query)
    cursor = conn.cursor()
    cursor.execute(query)

    print("Number of new mails to be sent: ", cursor.rowcount)

    raw = cursor.fetchall()

    for line in raw:
        print(line[0])
        sendEmail(line[2],line[0],line[1])
        result.append(line)

    cursor.execute('delete from mails')
    cursor.execute('commit')

    return result

```

Note: In the body of db_util.py, Lambda deletes the content of the mails table.

Place the main.py and db_util.py files inside the Github extracted folder and create a new zipfile that includes your two new files.

Return to your Lambda project and change the **Code entry type** to **Upload a .ZIP file**, change the Handler to **mail.lambda_handler**, and upload the file. Click **Save**.

SendingEmail

Throttle Qualifiers Actions Select a test event.. Test Save 4

▼ Designer

Add triggers Click on a trigger from the list below to add it to your function.

API Gateway AWS IoT Alexa Skills Kit Alexa Smart Home CloudFront CloudWatch Events

Add triggers from the list on the left

SendingEmail Unsaved changes

Amazon CloudWatch Logs

Amazon EC2

Resources the function's role has access to will be shown here

Function code Info

1 Code entry type Upload a .ZIP file

Runtime Python 2.7

2 Handler Info main.lambda_handler

3 Function package* Upload pgmail.zip (823.6 kB)
For files larger than 10 MB, consider uploading via S3.

To test the lambda function, click **Test** and enter the **Event name**.

Configure test event

X

A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.

- Create new test event
- Edit saved test events

Event template

Hello World



Event name

Test

```
1 [{}  
2 "key3": "value3",  
3 "key2": "value2",  
4 "key1": "value1"  
5 ]
```

Cancel

Create

Note: The Lambda function can be triggered by multiple options. This walkthrough demonstrates how to schedule it to run every minute. Remember, you are paying for each Lambda execution.

To create a scheduled trigger, use CloudWatch, enter all details, and click **Add**.

SendingEmail

AWS IoT
Alexa Skills Kit
Alexa Smart Home
CloudFront
CloudWatch Events
CloudWatch Logs
CodeCommit

Throttle Qualifiers Actions Test Test Save

SendingEmail Saved

CloudWatch Events Configuration required

Add triggers from the list on the left

Amazon CloudWatch Logs

Amazon EC2

Resources the function's role has access to will be shown here

Configure triggers

Rule
Pick an existing rule, or create a new one.
 Create a new rule

Select or create a new rule

Rule name*
Enter a name to uniquely identify your rule.
CheckForNewEmails

Rule description
Provide an optional description for your rule.
Check for new emails trigger

Rule type
Trigger your target based on an event pattern, or based on an automated schedule.
 Event pattern
 Schedule expression

Schedule expression*
Self-trigger your target on an automated schedule using Cron or rate expressions. Cron expressions are in UTC.
rate(1 minute)

e.g. rate(1 day), cron(0 17 ? * MON-FRI *)

Lambda will add the necessary permissions for Amazon CloudWatch Events to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

Enable trigger
Enable the trigger now, or create it in a disabled state for testing (recommended).

Cancel Add

Note: This example runs every minute, but you can use a different interval. For more information, see <https://docs.aws.amazon.com/lambda/latest/dg/tutorial-scheduled-events-schedule-expressions.html>

Click **Save**.

Tables & Indexes

Case Sensitivity Differences for Oracle and PostgreSQL

Object name case sensitivity is different for Oracle and PostgreSQL. Oracle names are not case sensitive. PostgreSQL names are case sensitive.

By default, AWS SCT uses object name in lower-case for PostgreSQL. In most cases, you'll want to use AWS DMS transformations to change schema, table, and column names to lower case.

To have an upper-case name, you must place the objects names within doubles quotes.

For example, to create a table named EMPLOYEES (upper-case) in PostgreSQL, you should use the following:

```
CREATE TABLE "EMPLOYEES" (
    EMP_ID      NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL,
    AVG_SALARY   NUMERIC NOT NULL);
```

The command below will create a table named employees (lower-case).

```
CREATE TABLE EMPLOYEES (
    EMP_ID      NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL,
    AVG_SALARY   NUMERIC NOT NULL);
```

If doubles quotes weren't used, PostgreSQL will look for object names in their lower-case form, for CREATE commands where doubles quotes weren't used, objects will be created with lower-case names, therefore, to create / query / manipulate an upper-cased (or mixed) object names you must use doubles quotes.

Oracle Common Data Types vs. PostgreSQL Common Data Types

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Data Types	BFILE, ROWID, UROWID are not supported by PostgreSQL

Oracle Usage

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

Oracle Data Types vs. PostgreSQL Data Types

Oracle Data Type Family	Oracle Data Type	Oracle Data Type Characteristic	PostgreSQL Identical Compatibility	PostgreSQL Corresponding Data Type
Character Data Types	CHAR(n)	Maximum size of 2000 bytes	Yes	CHAR(n)
	CHARACTER(n)	Maximum size of 2000 bytes	Yes	CHARACTER(n)
	NCHAR(n)	Maximum size of 2000 bytes	No	CHAR(n)
	VARCHAR(n)	Maximum size of 2000 bytes	Yes	VARCHAR(n)
	NCHAR VARYING(n)	Varying-length UTF-8 string Maximum size of 4000 bytes	No	CHARACTER VARYING(n)
	VARCHAR2(n) 11g	Maximum size of 4000 bytes Maximum size of 32KB in PL/SQL	No	VARCHAR(n)
	VARCHAR2(n) 12g	Maximum size of 32767 bytes MAX_STRING_SIZE=EXTENDED	No	VARCHAR(n)
	NVARCHAR2(n)	Maximum size of 4000 bytes	No	VARCHAR(n)
	LONG	Maximum size of 2GB	No	TEXT
	RAW(n)	Maximum size of 2000 bytes	No	BYTEA
Numeric Data Types	LONG RAW	Maximum size of 2GB	No	BYTEA
	NUMBER	Floating-point number	No	DOUBLE PRECISION
	NUMBER(*)	Floating-point number	No	DOUBLE PRECISION
	NUMBER(p,s)	Precision can range from 1 to 38 Scale can range from -84 to 127	No	DECIMAL(p,s)

Oracle Data Type Family	Oracle Data Type	Oracle Data Type Characteristic	PostgreSQL Identical Compatibility	PostgreSQL Corresponding Data Type
	NUMERIC(p,s)	Precision can range from 1 to 38	Yes	NUMERIC(p,s)
	FLOAT(p)	Floating-point number	No	DOUBLE PRECISION
	DEC(p,s)	Fixed-point number	Yes	DEC(p,s)
	DECIMAL(p,s)	Fixed-point number	Yes	DECIMAL(p,s)
	INT	38 digits integer	Yes	INTEGER / NUMERIC(38,0)
	INTEGER	38 digits integer	Yes	INTEGER / NUMERIC(38,0)
	SMALLINT	38 digits integer	Yes	SMALLINT
	REAL	Floating-point number	No	DOUBLE PRECISION
	DOUBLE PRECISION	Floating-point number	Yes	DOUBLE PRECISION
Date & Time Data Types	DATE	DATE data type stores date and time data (year, month, day, hour, minute and second)	Yes	TIMESTAMP(0)
Date & Time Data Types	TIMESTAMP(p)	Date and time with fraction	Yes	TIMESTAMP(p)
	TIMESTAMP(p) WITH TIME ZONE	Date and time with fraction and time zone	Yes	TIMESTAMP(p) WITH TIME ZONE
	INTERVAL YEAR (p) TO MONTH	Date interval	Yes	INTERVAL YEAR TO MONTH
	INTERVAL DAY(p) TO SECOND(s)	Day and time interval	Yes	INTERVAL DAY TO SECOND(s)
LOB Data Types	BFILE	Pointer to binary file Maximum file size of 4G	No	VARCHAR (255) / CHARACTER VARYING (255)
	BLOB	Binary large object Maximum file size of 4G	No	BYTEA
	CLOB	Character large object Maximum file size of 4G	No	TEXT
	NCLOB	Variable-length Unicode string Maximum file size of 4G	No	TEXT
ROWID Data Types	ROWID	Physical row address	No	CHARACTER (255)
	UROWID(n)	Universal row id Logical row addresses	No	CHARACTER VARYING
XML Data Type	XMLTYPE	XML data	No	XML

Oracle Data Type Family	Oracle Data Type	Oracle Data Type Characteristic	PostgreSQL Identical Compatibility	PostgreSQL Corresponding Data Type
Logical Data Type	BOOLEAN	Values TRUE / FALSE and NULL Cannot be assigned to a database table column	Yes	BOOLEAN
Spatial Types	SDO_GEOMETRY	The geometric description of a spatial object	No	-
	SDO_TOPO_GEOMETRY	Describes a topology geometry	No	-
	SDO_GEOASTER	A raster grid or image object is stored in a single row	No	-
Media Types	ORDDicom	Supports the storage and management of audio data	No	-
	ORDDicom	Supports the storage and management of Digital Imaging and Communications in Medicine (DICOM),	No	-
	ORDDoc	Supports storage and management of any type of media data	No	-
	ORDImage	Supports the storage and management of image data	No	-
	ORDVideo	Supports the storage and management of video data	No	-

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 Character Column Semantics Oracle supports both BYTE and CHAR semantics for column size, which determines the amount of storage allocated for CHAR and VARCHAR columns.

- If you define a field as 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 characters in that field because the physical size of some non-English characters exceeds one byte.
- If you define a field as 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 CHAR), col2 VARCHAR2(10 BYTE));
```

By default, Oracle uses BYTE semantics. When using a multi-byte character set such as UTF8, you must do one of the following:

- Use the CHAR modifier in the VARCHAR2/CHAR column definition.
- Modify the session or system parameter NLS_LENGTH_SEMANTICS to change the default from BYTE to 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 more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Data-Types.html#GUID-A3C0D836-BADB-44E5-A5D4-265BA5968483>

PostgreSQL Usage

PostgreSQL provides multiple data types equivalent to certain Oracle data types. The following table provides the full list of PostgreSQL data types:

PostgreSQL Data Type Family	PostgreSQL Data Type	PostgreSQL Data Type Characteristic
Character Data Types	CHAR	Stores a single character
	CHARACTER	Stores a single character
	CHAR(n)	Stores exactly (n) characters
	VARCHAR(N)	Stores a variable number of characters, up to a maximum of n characters
	TEXT	Specific variant of varchar, which does not require you to specify an upper limit on the number of characters
Numeric Data Types	NUMERIC (P, S)	Exact numeric of selectable precision
	REAL	Single precision floating-point number (4 bytes)
	DOUBLE PRECISION	Double precision floating-point number (8 bytes)
	INT	A signed 4-byte integer that can store -2147483648 to +2147483647
	INTEGER	A signed 4-byte integer that can store -2147483648 to +2147483647
	SMALLINT	A signed 2-byte integer that can store -32768 to +32767
	BIGINT	A signed 8-byte integer, giving approximately 18 digits of precision
	BIT	Stores a single bit, 0 or 1
	BIT VARYING	Stores a string of bits
	MONEY	Equivalent to NUMERIC (9,2), storing 4 bytes of data. Its use is discouraged.
Date & Time Data Types	TIMESTAMP	Stores dates and times from 4713 BC to 1465001 AD, with a resolution of 1 microsecond - 8 bytes

PostgreSQL Data Type Family	PostgreSQL Data Type	PostgreSQL Data Type Characteristic
	INTERVAL	Stores an interval of approximately +/- 178,000,000 years, with a resolution of 1 microsecond - 16 bytes
	DATE	Stores dates from 4713 BC to 32767 AD, with a resolution of 1 day - 4 bytes
	TIME	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
Logical Data Type	BOOLEAN	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.
XML Data Type	XML	XML data
Geometric Data Types	POINT	The column type to specify when you want to use the data models below
	LINE	An (x,y) value
	LSEG	A line (pt1, pt2)
	BOX	A sequence of points, effectively a closed path
Geometric Data Types	PATH	Collection of POINTs
	POLYGON	Collection of LINEs
	CIRCLE	Collection of POLYGONs
PostgreSQL Data Types	JSON	Textual JSON data
	JSONB	Binary JSON data, decomposed
	SERIAL	A numeric column in a table that increases each time a row is added
	OID	An object identifier. Internally, PostgreSQL adds, by default, a hidden oid to each row, and stores a 4-byte integer.
	CIDR	Stores a network address of the form x.x.x.x/y where y is the netmask
	INET	Similar to cidr, except the host part can be 0
	MACADDR	MAC (Media Access Control) address
	MACADDR8	MAC (Media Access Control) address in EUI-64 format (PostgreSQL 10)
	PG_LSN	PostgreSQL Log Sequence Number
	BYTEA	Binary data ("byte array")
	TSQUERY	Text search query
	TSVECTOR	Text search document
	TXID_SNAPSHOT	User-level transaction ID snapshot
	UUID	Universally unique identifier

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 Data Types to PostgreSQL Data Types

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:

```
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.

```
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,
    nlob             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,
    nvarchar          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 data-types that Amazon SCT is unable to automatically convert and where SCT recommended manual actions:

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

Recommended Actions: Either store a named file with the data and create a routine that gets that file from the file system, or store the data blob inside your database.

- **PostgreSQL doesn't have a data type **ROWID**:** ROWIDs are physical row addresses inside Oracle's storage subsystems. The ROWID datatype is primarily used for values returned by the ROWID pseudo-column.

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 data type **UROWID**:** 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/13/static/ddl-system-columns.html>
- <https://www.postgresql.org/docs/13/static/datatype.html>
- <https://aws.amazon.com/documentation/SchemaConversionTool>

Oracle Read-Only Tables and Partitions vs. PostgreSQL Aurora Replicas

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	READ ONLY mode is not supported by PostgreSQL, should use workaround

Oracle Usage

Beginning with Oracle 11g, tables can be marked as “read-only” to prevent 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. Beginning 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 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.

Examples

```

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

INSERT INTO EMP_READ_ONLY VALUES(1, 'John Smith');

1 row created

ALTER TABLE EMP_READ_ONLY READ ONLY;

INSERT INTO EMP_READ_ONLY VALUES(2, 'Steven King');

ORA-12081: update operation not allowed on table "SCT"."TBL_READ_ONLY"

ALTER TABLE EMP_READ_ONLY READ WRITE;

INSERT INTO EMP_READ_ONLY VALUES(2, 'Steven King');

1 row created

```

```

COMMIT;

SELECT * FROM EMP_READ_ONLY;

EMP_ID EMP_FULL_NAME
-----
1 John Smith
2 Steven King

```

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ALTER-TABLE.html>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/vlbg/release-changes.html#GUID-C7A9BAD4-E4C9-4765-88C5-51AC7E97BAF1>

PostgreSQL Usage

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';
CREATE ROLE

ALTER USER aws_READONLY SET DEFAULT_TRANSACTION_READ_ONLY=ON;
ALTER ROLE

GRANT SELECT ON EMP_READ_ONLY TO aws_READONLY;
GRANT
    -- Open a new session with user "aws_READONLY"
SELECT * FROM EMP_READ_ONLY;

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

```

```
INSERT INTO EMP_READ_ONLY VALUES(1, 'John Smith');
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:

```
CREATE DATABASE readonly_db;

ALTER DATABASE readonly_db SET DEFAULT_TRANSACTION_READ_ONLY=ON;

-- Open a new session connected to the "readonly_db" database

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

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:

```
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:

```
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:

```
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/13/static/ddl-priv.html>
- <https://www.postgresql.org/docs/13/static/sql-grant.html>
- <https://www.postgresql.org/docs/13/static/runtime-config-client.html>

Oracle Table Constraints vs. PostgreSQL Table Constraints

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Constraints	REF, ENABLE/DISABLE are not supported by PostgreSQL Constraints on views are not supported by PostgreSQL

Oracle Usage

Oracle provides six types of constraints to enforce data integrity on table columns. Constraints ensure 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

You can create new constraints in two ways:

- **Inline:** Defines a constraint as part of a table column declaration.

```
CREATE TABLE EMPLOYEES (
    EMP_ID NUMBER PRIMARY KEY, ...);
```

- **Out-Of-Line:** Defines a constraint as part of the table DDL during table creation.

```
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.

Privileges

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

PRIMARY KEY Constraints

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 a combination of columns.

Creating a new primary key constraint also implicitly creates a unique index on the primary key column if no 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.

- 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 a primary key.

- Primary keys can be created from multiple columns (composite PK). They are 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

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

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID NUMBER PRIMARY KEY,
    FIRST_NAME  VARCHAR2(20),
    LAST_NAME   VARCHAR2(25),
    EMAIL       VARCHAR2(25));
```

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

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID NUMBER CONSTRAINT PK_EMP_ID PRIMARY KEY,
    FIRST_NAME  VARCHAR2(20),
    LAST_NAME   VARCHAR2(25),
    EMAIL       VARCHAR2(25));
```

Create an out-of-line primary key.

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID NUMBER,
    FIRST_NAME  VARCHAR2(20),
    LAST_NAME   VARCHAR2(25),
    EMAIL       VARCHAR2(25));
CONSTRAINT PK_EMP_ID PRIMARY KEY (EMPLOYEE_ID);
```

Add a primary key to an existing table.

```
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. 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:** Dependent foreign key values in a child table are removed along with the referenced values from the parent table.
- **ON DELETE NULL:** Dependent foreign key values in a child table are updated to NULL.

Examples

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

```
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));
```

Create an Out-Of-Line foreign key with a system-generated constraint name.

```
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));
```

Create a foreign key using the ON DELETE CASCADE clause.

```
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);
```

Add a foreign key to an existing table:

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

UNIQUE Constraints

A unique constraint is similar to a primary key constraint. It 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 a 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.

Examples

Create an inline unique Constraint:

```
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 Constraints

Check constraints are used to validate values in specific columns that 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 be used with functions that are not deterministic (e.g. CURRENT_DATE).
- Check constraints cannot be used with user-defined functions.
- Check constraints cannot be used with pseudo columns such as: CURRVAL, NEXTVAL, LEVEL, or ROWNUM.

Examples

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

```
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 Constraints

A 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.

Examples

```
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 Constraints

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 defining a scope constraint, a rowid constraint, or a referential integrity constraint based on the REF column.

Examples

Create a new Oracle type object.

```
CREATE TYPE DEP_TYPE AS OBJECT (
    DEP_NAME      VARCHAR2(60),
    DEP_ADDRESS   VARCHAR2(300));
```

Create a table based on the previously created type object.

```
CREATE TABLE DEPARTMENTS_OBJ_T OF DEP_TYPE;
```

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

```
CREATE TABLE EMPLOYEES (
    EMP_NAME      VARCHAR2(60),
```

```
EMP_EMAIL  VARCHAR2(60),
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.

Examples

Create a unique constraint with a state of DEFERRABLE.

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

Modify the state of the constraint to ENABLE NOVALIDATE.

```
ALTER TABLE EMPLOYEES
ADD CONSTRAINT CHK_EMP_NAME CHECK(FIRST_NAME LIKE 'a%')
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.

Examples

Create a unique constraint based on an existing index.

```
CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES(EMPLOYEE_ID);

ALTER TABLE EMPLOYEES
    ADD CONSTRAINT PK_CON_UNIQ
    PRIMARY KEY(EMPLOYEE_ID) USING INDEX IDX_EMP_ID;
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

PostgreSQL supports the following types of table constraints:

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

Note: PostgreSQL does not support Oracle's REF constraint.

Similar to constraint declaration in Oracle, PostgreSQL allows creating constraints in-line or out-of-line when specifying table columns.

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

Privileges

You must have privileges on the table in which constraints are created. For foreign key constraints, you must also have the REFERENCES privilege.

Primary Key Constraints

- 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:

```
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:

```
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:

```
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:

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

5. Drop the primary key.

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

Foreign Key Constraints

- 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:

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID      NUMERIC PRIMARY KEY,
    FIRST_NAME       VARCHAR(20),
    LAST_NAME        VARCHAR(25),
    EMAIL            VARCHAR(25),
    DEPARTMENT_ID    NUMERIC 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:

```
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:

```

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:

```

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:

```

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.

```
CREATE TABLE EMPLOYEES (
    EMPLOYEE_ID      NUMERIC PRIMARY KEY,
    FIRST_NAME       VARCHAR(20),
    LAST_NAME        VARCHAR(25),
    EMAIL            VARCHAR(25) CHECK (EMAIL ~ '^[A-Za-z]+@[aws.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

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

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:

```
CREATE UNIQUE INDEX IDX_EMP_ID ON EMPLOYEES(EMPLOYEE_ID);

ALTER TABLE EMPLOYEES
    ADD CONSTRAINT PK_CON_UNIQ PRIMARY KEY USING INDEX IDX_EMP_ID;
```

Summary

Oracle Constraint / Parameter	PostgreSQL Constraint / Parameter
PRIMARY KEY	PRIMARY KEY
FOREIGN KEY	FOREIGN KEY
UNIQUE	UNIQUE
CHECK	CHECK
NOT NULL	NOT NULL
REF	Not Supported
DEFERRABLE	DEFERRABLE
NOT DEFERRABLE	NOT DEFERRABLE

Oracle Constraint / Parameter	PostgreSQL Constraint / Parameter
SET CONSTRAINTS	SET CONSTRAINTS
INITIALLY IMMEDIATE	INITIALLY IMMEDIATE
INITIALLY DEFERRED	INITIALLY DEFERRED
ENABLE	Default, not supported as keyword
DISABLE	Not supported as keyword, NOT VALID can use instead
ENABLE VALIDATE	Default, not supported as keyword
ENABLE NOVALIDATE	NOT VALID
DISABLE VALIDATE	Not supported
DISABLE NOVALIDATE	Not supported
USING_INDEX_CLAUSE	table_constraint_using_index
View Constraints	Not supported
Metadata: DBA_CONSTRAINTS	Metadata: PG_CONSTRAINT

For additional details:

- <https://www.postgresql.org/docs/13/static/ddl-constraints.html>
- <https://www.postgresql.org/docs/13/static/sql-set-constraints.html>
- <https://www.postgresql.org/docs/13/static/sql-altertable.html>

Oracle Temporary Tables vs. PostgreSQL Temporary Tables

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Create tables	<p>GLOBAL temporary table is not supported by PostgreSQL</p> <p>Can't read from multiple sessions in PostgreSQL</p> <p>Table dropped after session ends in PostgreSQL</p>

Oracle Usage

In Oracle, you can create temporary tables for storing data that exists only for the duration of a session or transaction.

Use the CREATE GLOBAL TEMPORARY TABLE statement to create a temporary table. This type of table has a persistent DDL structure, but not persistent data. It 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.
- Storing data required only for the duration of a specific session. When the session ends, the session data is cleared.

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

Oracle 18c introduces private temporary tables which are temporary tables that are only available during session or transaction. After session or transaction ends they are automatically dropped.

Oracle Global Temporary Tables

- Global Temporary Tables store data in the Oracle Temporary Tablespace.
- DDL operations on a temporary table are permitted including ALTER TABLE, DROP TABLE, and 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. They 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: 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, generates 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 more details, see https://docs.oracle.com/en/database/oracle/oracle-database/19/refn/TEMP_UNDO_ENABLED.html#GUID-E2A01A84-2D63-401F-B64E-C96B18C5DCA6

Examples

Create an Oracle Global Temporary Table (with ON COMMIT PRESERVE ROWS).

```

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;

CREATE INDEX IDX_EMP_TEMP_FN ON EMP_TEMP(EMP_FULL_NAME);

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

COMMIT;

SELECT * FROM SCT.EMP_TEMP;

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

```

Create an Oracle Global Temporary Table (with ON COMMIT DELETE ROWS).

```

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;

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

COMMIT;

SELECT * FROM SCT.EMP_TEMP;

```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

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.

Note: Since PostgreSQL 10, partition tables can also be temporary tables

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:

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 and use a Temporary Table, with ON COMMIT PRESERVE ROWS:

```

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;

CREATE INDEX IDX_EMP_TEMP_FN ON EMP_TEMP(EMP_FULL_NAME);

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

COMMIT;

SELECT * FROM SCT.EMP_TEMP;

emp_id | emp_full_name | avg_salary
-----+-----+-----
1 | John Smith | 5000

DROP TABLE EMP_TEMP;

```

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

```

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;

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

COMMIT;

SELECT * FROM SCT.EMP_TEMP;

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

DROP TABLE EMP_TEMP;
DROP TABLE

```

Summary

	Oracle	Aurora PostgreSQL
Semantic	Global Temporary Table	Temporary Table / Temp Table
Create table	CREATE GLOBAL TEMPORARY...	CREATE TEMPORARY... or CREATE TEMP...
Accessible from multiple sessions	Yes	No
Temp table DDL persist after session end / database restart user-managed datafiles	Yes	No (dropped at the end of the session)
Create index support	Yes	Yes
Foreign key support	Yes	Yes
ON COMMIT default	COMMIT DELETE ROWS	ON COMMIT PRESERVE ROWS
ON COMMIT PRESERVE ROWS	Yes	Yes
ON COMMIT DELETE ROWS	Yes	Yes
Alter table support	Yes	Yes
Gather statistics	dbms_stats.gather_table_stats	ANALYZE
Oracle 12c GLOBAL_TEMP_TABLE_STATS	dbms_stats.set_table_prefs	ANALYZE

For more details, see <https://www.postgresql.org/docs/13/sql-createtable.html>

Oracle Triggers vs. PostgreSQL Trigger Procedure

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Triggers	Different paradigm and syntax System triggers are not supported by PostgreSQL

Oracle Usage

A trigger is a procedure that is stored in the database and fired (or executed) when a specified event occurs. The associated event causing 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).

Trigger Types:

- **DML Trigger:** Can be created on Tables or Views and fire when inserting, updating, or deleting data. Triggers can fire before or after DML command execution.
- **INSTEAD OF Trigger:** A special type of DML trigger created on a non-editable view. INSTEAD OF triggers provide an application-transparent method for modifying views that cannot be modified via DML statements.
- **SYSTEM Event Triggers:** Triggers defined at the database or schema level including triggers that fire after specific events:
 - User log-on and log-off.
 - Database events (startup/shutdown), DataGuard events, server errors.

Examples

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.

```

CREATE OR REPLACE TRIGGER PROJECTS_SET_NULL
  AFTER DELETE OR UPDATE OF PROJECTNO ON PROJECTS
  FOR EACH ROW
  BEGIN
    IF UPDATING AND :OLD.PROJECTNO != :NEW.PROJECTNO OR DELETING THEN
      UPDATE EMP SET EMP.PROJECTNO = NULL
      WHERE EMP.PROJECTNO = :OLD.PROJECTNO;
    END IF;
  END;
/

```

Trigger created.

```
DELETE FROM PROJECTS WHERE PROJECTNO=123;

SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;

PROJECTNO
-----
NULL
```

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. It prevents dropping the object and raises an application error.

```
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.

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 more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/CREATE-TRIGGER-statement.html#GUID-AF9E33F1-64D1-4382-A6A4-EC33C36F237B>

PostgreSQL Usage

A trigger is a procedure that is stored in the database and fired (or being executed) when a specified event occurs. 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 CREATE TRIGGER Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ]
}
ON table_name
[ FROM referenced_table_name ]
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
```

```
[ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE function_name ( arguments )
```

where event can be one of:

```
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

Note: REFERENCING is a new option introduced in PostgreSQL 10. It can be used with AFTER trigger to interact with the overall view of the OLD or the NEW TABLE changed rows.

Note that there are some cases that can fire multiple triggers numerous times. This includes triggers that are not planned to run, such as:

- An INSERT with an ON CONFLICT DO UPDATE clause may cause both insert and update operations to fire.
- UPDATE or DELETE caused by foreign-key enforcement can fire triggers. For example, ON UPDATE CASCADE or ON DELETE SET NULL can fire triggers that are supposed to fire on UPDATE or DELETE commands on the table.

PostgreSQL DML Triggers

- PostgreSQL triggers can run BEFORE or AFTER a DML operation.
 - Fire before the operation is attempted on a row.
 - Before constraints are checked and the INSERT, UPDATE, or DELETE is attempted.
 - 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).
 - After the operation was completed, after constraints are checked and the INSERT, UPDATE, or DELETE command completed.
 - If the trigger fires after the event, all changes, including the effects of other triggers, are "visible" to the trigger.
- PostgreSQL triggers can run INSTEAD OF a DML command when created on views.
- 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.

When Fired	Database Event	Row-Level Trigger (FOR EACH ROW)	Statement-Level Trigger (FOR EACH STATEMENT)
BEFORE	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
AFTER	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
INSTEAD OF	INSERT, UPDATE, DELETE	Views	—
	TRUNCATE	—	—

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`.

- `ddl_command_start` - occurs before the execution of a CREATE, ALTER, DROP, SECURITY LABEL, COMMENT, GRANT, REVOKE or SELECT INTO command.
- `ddl_command_end` - occurs after the command completed and before the transaction commits.
- `sql_drop` - fired only for the DROP DDL command. Fires before `ddl_command_end` trigger fire.

Full list of supported PostgreSQL event trigger types: <https://www.postgresql.org/docs/13/static/event-trigger-matrix.html>

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:

```
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;
```

2. Create the trigger:

```
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:

```
DELETE FROM PROJECTS WHERE PROJECTNO=123;
SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;

projectno
```

```
-----  
(0 rows)
```

Example

Create a DDL trigger that is 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):

1. 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:

```
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:

```
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:

```
DROP TABLE EMPLOYEES;

ERROR: The DROP TABLE Command is Disabled
CONTEXT: PL/pgSQL function abort_drop_command() line 3 at RAISE
```

Summary

	Oracle	PostgreSQL
“Before update” trigger, row level	<pre>CREATE OR REPLACE TRIGGER check_update BEFORE UPDATE ON projects FOR EACH ROW BEGIN /*Trigger body*/ END; /</pre>	<pre>CREATE TRIGGER check_update BEFORE UPDATE ON employees FOR EACH ROW EXECUTE PROCEDURE myproc();</pre>
“Before update” trigger, statement level	<pre>CREATE OR REPLACE TRIGGER check_update BEFORE UPDATE ON projects BEGIN /*Trigger body*/ END; /</pre>	<pre>CREATE TRIGGER check_update BEFORE UPDATE ON employees FOR EACH STATEMENT EXECUTE PROCEDURE myproc();</pre>
System / event trigger	<pre>CREATE OR REPLACE TRIGGER drop_trigger BEFORE DROP ON hr.SCHEMA BEGIN RAISE_APPLICATION_ERROR (num => -20000, msg => 'Cannot drop object'); END; /</pre>	<pre>CREATE EVENT TRIGGER trg_drops ON ddl_command_start EXECUTE PROCEDURE trg_drops();</pre>
Referencing :old and :new values in triggers	<p>Use "":NEW" and "":OLD" in trigger body:</p> <pre>CREATE OR REPLACE TRIGGER Upper-NewDeleteOld 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; /</pre>	<p>Use ".NEW" and ".OLD" in trigger Procedure body:</p> <pre>CREATE OR REPLACE FUNCTION log_emp_name_upd() RETURNS trigger LANGUAGE plpgsql AS \$\$ BEGIN IF NEW.last_name <> 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; \$\$</pre> <p>CREATE TRIGGER last_name_change_trg BEFORE UPDATE ON employees FOR EACH ROW EXECUTE PROCEDURE log_last_emp_name_upd();</p>
Database event level trigger	<pre>CREATE TRIGGER register_shutdown</pre>	N/A

	Oracle	PostgreSQL
	<pre>ON DATABASE SHUTDOWN BEGIN Insert into logging values ('DB was shutdown', sysdate); commit; END; /</pre>	
Drop a trigger	DROP TRIGGER last_name_change_trg;	DROP TRIGGER last_name_change_trg on employees;
Modify logic executed by a trigger	<p>Can be used with create or replace</p> <pre>CREATE OR REPLACE TRIGGER UpperNewDeleteOld BEFORE INSERT OR UPDATE OF first_name ON employees FOR EACH ROW BEGIN <<NEW CONTENT>> END; /</pre>	<p>Use CREATE OR REPLACE on the called function in the trigger (trigger stay the same)</p> <pre>CREATE or replace FUNCTION UpperNewDeleteOld() RETURNS trigger AS \$UpperNewDeleteOld\$ BEGIN <<NEW CONTENT>> END; \$UpperNewDeleteOld\$ LANGUAGE plpgsql;</pre>
Enable a trigger	ALTER TRIGGER UpperNewDeleteOld ENABLE;	alter table employees enable trigger Upper-NewDeleteOld;
Disable a trigger	ALTER TRIGGER UpperNewDeleteOld DISABLE;	alter table employees disable trigger Upper-NewDeleteOld;

For additional details:

<https://www.postgresql.org/docs/13/sql-createtrigger.html>

<https://www.postgresql.org/docs/13/static/plpgsql-trigger.html>

Oracle Tablespaces & Data Files vs. PostgreSQL Tablespaces and Data files

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	All supported by PostgreSQL except managing the physical data files

Oracle Usage

The storage structure of an Oracle database contains both physical and logical elements.

- **Tablespaces:** Each Oracle database contains one or more tablespaces, which are logical storage groups used as “containers” for creating new tables and indexes.
- **Data Files:** Each tablespace is made up of one or more data files, which are the physical elements of an Oracle database tablespace. Datafiles can be located on the local file system, located in raw partitions, managed by Oracle ASM, or located on a network file system.

Storage Hierarchy

- **Database:** Each Oracle database is composed of one or more tablespaces.
- **Tablespace:** Each Oracle tablespace is composed of 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 consists of one or more data files.
- **Segments:** Each segment represents a single database object that consumes storage such as tables, indexes, and undo segments.
- **Extent:** Each segment consists of one or more extents. Oracle uses extents to allocate contiguous sets of database blocks on disk.
- **Block:** The smallest unit of I/O for reads and writes. For blocks storing table data, each block can store one or more table rows.

Types of Oracle Database Tablespaces

- **Permanent Tablespaces:** Designated to store persistent schema objects for applications.
- **Undo Tablespace :** A special type of system permanent tablespace used by Oracle to manage UNDO data when running the database in automatic undo management mode.
- **Temporary Tablespace:** Contains schema objects valid for the duration of a session. It is also used for sort operations that cannot fit into memory.

Tablespace Privileges

The following criteria must be met to create a tablespace:

- The database user must have the CREATE TABLESPACE system privilege.
- The database must be in OPEN MODE.

Examples

Create a USERS tablespace comprised of a single data file:

```
CREATE TABLESPACE USERS
  DATAFILE '/u01/app/oracle/oradata/orcl/users01.dbf' SIZE 5242880
  AUTOEXTEND ON NEXT 1310720 MAXSIZE 32767M
  LOGGING ONLINE PERMANENT BLOCKSIZE 8192
  EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT
  NOCOMPRESS SEGMENT SPACE MANAGEMENT AUTO;
```

Drop a tablespace:

```
DROP TABLESPACE USERS;
OR
DROP TABLESPACE USERS INCLUDING CONTENTS AND DATAFILES;
```

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLESPACE.html#GUID-51F07BF5-EFAF-4910-9040-C473B86A8BF9>
- https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/file_specification.html#GUID-580FA726-F712-4410-90CF-783A2DA89688
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/DROP-TABLESPACE.html#GUID-C91F3E94-4503-48DE-9BCA-42E495E6BE11>

PostgreSQL Usage

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.

Note:

- 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:

```
CREATE TABLESPACE TBS_01 LOCATION '/app_data/tbs_01';

\du

Name      | Owner      |          Location
-----+-----+-----+
pg_default | rdsadmin |
pg_global  | rdsadmin |
tbs_01     | rdsadmin | /rdsdbdata/tablespaces/app_data/tbs_01
```

Note: 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:

```
DROP TABLESPACE TBS_01;
```

4. Alter a tablespace:

```
ALTER TABLESPACE TBS_01 RENAME TO IDX_TBS_01;

ALTER TABLESPACE TO IDX_TBS_01 OWNER TO USER1;
```

5. Assign a database with a specific tablespace:

```
CREATE DATABASE DB1 TABLESPACE TBS_01;

SELECT DATNAME, PG_TABLESPACE_LOCATION(DATTABLESPACE) FROM PG_DATABASE
```

```
WHERE DATNAME='db1';

datname | pg_tablespace_location
-----+-----
db1    | /rdsdbdata tablespaces/app_data/tbs_0
```

6. Assign a table with a specific tablespace:

```
CREATE TABLE TBL(COL1 NUMERIC, COL2 VARCHAR(10))
TABLESPACE TBS_01;

SELECT SCHEMANAME, TABLENAME, TABLESPACE FROM PG_TABLES
WHERE TABLENAME='tbl';

schemaname | tablename | tablespace
-----+-----+-----
public    | tbl      | tbs_01
```

7. Assign an index with a specific tablespace:

```
CREATE INDEX IDX_TBL ON TBL(COL1)
    TABLESPACE TBS_01;

SELECT SCHEMANAME, TABLENAME, INDEXNAME, TABLESPACE FROM PG_INDEXES
    WHERE INDEXNAME='idx_tbl';

schemaname | tablename | indexname | tablespace
-----+-----+-----+-----
public    | tbl      | idx_tbl  | tbs_01
```

8. Alter a table to use a different tablespace:

```
ALTER TABLE TBL SET TABLESPACE TBS_02;
```

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:

```
SHOW DEFAULT_TABLESPACE; -- No value
default_tablespace
```

```
-----  
SET DEFAULT_TABLESPACE=TBS_01;  
SHOW DEFAULT_TABLESPACE;  
default_tablespace  
-----  
tbs_01
```

Summary

Feature	Oracle	Aurora PostgreSQL
Tablespace	Exists as a logical object and made from one or more user-specified or system-generated data files.	Logical object that is tied to a specific directory on the disk where datafiles will be created.
Data file	<ol style="list-style-type: none"> 1. Can be explicitly created and resized by the user. Oracle-Managed-Files (OMF) allow for automatically created data files. 2. Each data file can contain one or more tables and/or indexes. 	<p>Behavior is more akin to Oracle Managed Files (OMF).</p> <ol style="list-style-type: none"> 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. <p>Additional files are created:</p> <ol style="list-style-type: none"> 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.
Creates a new tablespace with system-managed data-files	CREATE TABLESPACE sales_tbs DATAFILE SIZE 400M;	create tablespace sales_tbs LOCATION '/postgresql/data';
Create a new tablespace with user-managed data-files	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
Add a datafile to an existing tablespace	ALTER TABLESPACE sales_tbs ADD DATAFILE '/oradata/sales02.dbf' SIZE 10M;	N/A
Per-database tablespace	Supported as part of the Oracle 12c Multi-tablespace	Tablespaces are shared across all data-

Feature	Oracle	Aurora PostgreSQL
	<p>Tenant architecture. Different dedicated tablespaces can be created for different pluggable databases and set as the default tablespace for a PDB:</p> <pre>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;</pre>	<p>bases but a default tablespace can be created and configured for the database:</p> <pre>create tablespace sales_tbs LOCATION '/postgresql/data'; CREATE DATABASE sales OWNER sales_app TABLESPACE sales_tbs;</pre>
Metadata tables	Data Dictionary tables are stored in the SYSTEM tablespace.	System Catalog tables are stored in the pg_global tablespace
Tablespace data encryption	<p>Supported</p> <ol style="list-style-type: none"> 1. Supported using transparent data encryption. 2. Encryption and decryption are handled seamlessly. Users do not have to modify the application to access the data. 	<p>Supported</p> <ol style="list-style-type: none"> 1. Encrypt using keys managed through KMS. 2. Encryption and decryption are handled seamlessly. Users 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 operations. For additional details see the links below. <p>For additional details: AWS Encryption</p>

For additional details:

- <https://www.postgresql.org/docs/13/static/manage-ag-tablespaces.html>
- <https://www.postgresql.org/docs/13/static/sql-createtablespace.html>
- <https://www.postgresql.org/docs/13/static/storage-file-layout.html>
- <https://www.postgresql.org/docs/13/static/storage-fsm.html>
- <https://www.postgresql.org/docs/13/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE>
- <https://www.postgresql.org/docs/13/static/sql-droptablespace.html>
- <https://www.postgresql.org/docs/13/static/sql-altertablespace.html>

Oracle User Defined Types vs. PostgreSQL User Defined Types

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		User Defined Types	<p>FORALL statement and DEFAULT option are not supported by PostgreSQL</p> <p>PostgreSQL doesn't support constructors of the "collection" type</p>

Oracle Usage

Oracle refers to User Defined Types (UDTs) as OBJECT TYPES and are managed using PL/SQL. User Defined Types enable the creation of 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 (a Java class mapped to SQL user defined type)

Examples

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

```

CREATE OR REPLACE TYPE EMP_PHONE_NUM AS OBJECT (
    PHONE_NUM VARCHAR2(11));

CREATE TABLE EMPLOYEES (
    EMP_ID      NUMBER PRIMARY KEY,
    EMP_PHONE   EMP_PHONE_NUM NOT NULL);

INSERT INTO EMPLOYEES VALUES(1, EMP_PHONE_NUM('111-222-333'));
SELECT a.EMP_ID, a.EMP_PHONE.PHONE_NUM FROM EMPLOYEES a;

EMP_ID      EMP_PHONE.P
-----
1           111-222-333

```

Create an Oracle Object Type as a “collection of attributes” for the employees table:

```

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

```

```

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

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

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 more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TYPE.html#GUID-E72E3EE6-DE95-4F58-8941-E2F76D0EAE80>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TYPE-BODY.html#GUID-C4F1591A-6F62-4897-9039-2C3F066F1E9D>

PostgreSQL Usage

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:

```

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.

```

CREATE TYPE float8_range AS RANGE
    (subtype = float8, subtype_diff = float8mi);

```

For more information see: <https://www.postgresql.org/docs/13/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.

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

For additional details:<https://www.postgresql.org/docs/13/static/arrays.html>

PostgreSQL CREATE TYPE Synopsis

```
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:

```

CREATE TYPE EMP_PHONE_NUM AS (
    PHONE_NUM VARCHAR(11));

CREATE TABLE EMPLOYEES (
    EMP_ID      NUMERIC PRIMARY KEY,
    EMP_PHONE   EMP_PHONE_NUM NOT NULL);

INSERT INTO EMPLOYEES VALUES(1, ROW('111-222-333'));

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:

```

CREATE OR REPLACE TYPE EMP_ADDRESS AS OBJECT (
    STATE      VARCHAR(2),
    CITY       VARCHAR(20),
    STREET     VARCHAR(20),
    ZIP_CODE   NUMERIC);

CREATE TABLE EMPLOYEES (
    EMP_ID      NUMERIC PRIMARY KEY,
    EMP_NAME    VARCHAR(10) NOT NULL,
    EMP_ADDRESS EMP_ADDRESS NOT NULL);

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

SELECT a.EMP_NAME,
        (a.EMP_ADDRESS).STATE,
        (a.EMP_ADDRESS).CITY,
        (a.EMP_ADDRESS).STREET,
        (a.EMP_ADDRESS).ZIP_CODE
FROM EMPLOYEES a;

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

```

For additional details:

- <https://www.postgresql.org/docs/13/static/sql-createtype.html>
- <https://www.postgresql.org/docs/13/static/rowtypes.htm>

Oracle Unused Columns vs. PostgreSQL Alter Table

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	PostgreSQL does not support unused columns

Oracle Usage

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.

Examples

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

Display unused columns:

```
SELECT * FROM USER_UNUSED_COL_TABS;

TABLE_NAME          COUNT
-----
EMPLOYEES           3
```

Drop the Column Permanently (physically drop the column):

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

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

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 “drop column” statement:

```
ALTER TABLE EMPLOYEES DROP COLUMN COMMISSION_PCT;
```

2. Verify the operation:

```
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:

```
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:

```
VACUUM FULL VERBOSE EMPLOYEES;
```

For additional details:

<https://www.postgresql.org/docs/13/static/sql-altertable.html>
<https://www.postgresql.org/docs/13/static/sql-vacuum.html>

Oracle Virtual Columns vs. PostgreSQL Views and Functions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	

Oracle Usage

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 either explicitly specify the data type or let the database select the data type 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 are provided for clarity only.

```
COLUMN_NAME [datatype] [GENERATED ALWAYS] AS (expression) [VIRTUAL]
```

- The keyword AS after the column name indicates the column is created as a Virtual Column.
- A Virtual Column does not need to be specified in an INSERT statement.

Examples

Create a table that includes two Virtual Columns:

```
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,
    FINAL_SALARY NUMBER GENERATED ALWAYS AS
```

```
(CASE WHEN SALES_COUNT >= 10 THEN BASE_SALARY + (BASE_SALARY *
(SALES_COUNT * 0.05)) END) VIRTUAL);
```

Insert a new record into the table without specifying values for the Virtual Column:

```
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);
```

Select the email Virtual Column from the table:

```
SELECT email FROM EMPLOYEES;
EMAIL          FINAL_SALARY
-----
jsmith@aws.com      10250
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html#GUID-F9CE0CC3-13AE-4744-A43C-EAC7A71AAAB6>

PostgreSQL Usage

PostgreSQL does not provide a feature that is directly equivalent to a Virtual Column in Oracle before version 12. However, there are workarounds to emulate similar functionality.

Starting with PostgreSQL 12, support for generated columns have been added. Generated columns can be either calculated from other columns values on the fly or calculated and stored. Generated columns are similar to Oracle virtual columns

Alternatives for Virtual Columns for PostgreSQL before version 12:

- **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.

Examples

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:

```
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:

```
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:

```
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:

```
SELECT EMPLOYEE_ID,
       FIRST_NAME,
       LAST_NAME,
       USER_NAME,
       USER_EMAIL(EMPLOYEES)
  FROM EMPLOYEES;

employee_id | first_name | last_name | user_name | user_email
-----+-----+-----+-----+-----+
    1 | John      | Smith     | jsmith    | jsmith@aws.com
    2 | Steven    | King      | sking     | sking@aws.com
```

5. Create a view that incorporates the USER_EMAIL function:

```
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:

```
CREATE INDEX IDX_USER_EMAIL ON EMPLOYEES(USER_EMAIL(EMPLOYEES));
```

7. Verify the Expression Based Index with EXPLAIN:

```
SET enable_seqscan = OFF;

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:

```
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:

```
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:

```
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:

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

SELECT * FROM EMPLOYEES;

employee_id | first_name | last_name | full_name
-----+-----+-----+-----
1 | John | Smith | John Smith
2 | Steven | King | Steven King
```

5. Create an Index based on the “virtual” FULL_NAME column:

```
CREATE INDEX IDX_USER_FULL_NAME ON EMPLOYEES(FULL_NAME);
```

6. Verify the Expression Based Index with EXPLAIN:

```
SET enable_seqscan = OFF;

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/13/sql-createtrigger.html>

Overall Indexes Summary

Usage

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.

Starting with PostgreSQL 12 it is now possible to monitor progress of CREATE INDEX and REINDEX operations by querying system view pg_stat_progress_create_index

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:

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

PostgreSQL CREATE/DROP Index:

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

Oracle ALTER INDEX - RENAME:

```
ALTER INDEX IDX_EMP_ID RENAME TO IDX_EMP_ID_OLD;
```

PostgreSQL ALTER INDEX - RENAME:

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

Oracle ALTER INDEX - TABLESPACE:

```
ALTER INDEX IDX_EMP_ID REBUILD TABLESPACE USER_IDX;
```

PostgreSQL ALTER INDEX - TABLESPACE:

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

Oracle REBUILD INDEX:

```
ALTER INDEX IDX_EMP_ID REBUILD;
```

PostgreSQL REINDEX (REBUILD) INDEX:

```
demo=> REINDEX INDEX IDX_EMP_ID;
```

Oracle REBUILD INDEX ONLINE:

```
ALTER INDEX IDX_EMP_ID REBUILD ONLINE;
```

PostgreSQL REINDEX (REBUILD) INDEX ONLINE:

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

For additional details, see:

- <https://www.postgresql.org/docs/13/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY>
- <https://www.postgresql.org/docs/13/static/sql-alterindex.html>
- <https://www.postgresql.org/docs/13/static/sql-reindex.html>

Summary

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

Oracle Bitmap Indexes vs. PostgreSQL Bitmap

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Index	PostgreSQL does not support BITMAP index - BRIN index can be used in some cases

Oracle Usage

Bitmap indexes are task-specific indexes best suited for providing fast data retrieval for OLAP workloads and are generally very fast for read-mostly scenarios. However, bitmap indexes do not perform well in heavy DML or OLTP workloads.

Unlike B-tree indexes where an index entry points to a specific table row, a bitmap 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.

Examples

Create an Oracle bitmap index.

```
CREATE BITMAP INDEX IDX_BITMAP_EMP_GEN ON EMPLOYEES (GENDER) ;
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-INDEX.html#GUID-1F89BBC0-825F-4215-AF71-7588E31D8BFE>

PostgreSQL Usage

Amazon Aurora PostgreSQL does not currently provide a directly comparable alternative for Oracle Bitmap indexes.

Oracle B-Tree Indexes vs. PostgreSQL B-Tree Indexes

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	

Oracle Usage

B-Tree indexes (B stands for balanced), are the most common index type in a 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 are the default when creating a new index.

Examples

Create a B-Tree index.

```
CREATE INDEX IDX_EVENT_ID ON SYSTEM_LOG(EVENT_ID);
```

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-INDEX.html#GUID-1F89BBC0-825F-4215-AF71-7588E31D8BFE>

PostgreSQL Usage

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.

Examples

Create a PostgreSQL B-Tree Index:

```
CREATE INDEX IDX_EVENT_ID ON SYSTEM_LOG(EVENT_ID);
OR
CREATE INDEX IDX_EVENT_ID1 ON SYSTEM_LOG USING BTREE (EVENT_ID);
```

For more details, see <https://www.postgresql.org/docs/13/static/sql-createindex.html>

Oracle Composite Indexes vs. PostgreSQL Multi-Column Indexes

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		<u>Index</u>	

Oracle Usage

An index created on multiple table columns is known as a multi-column, concatenated, or composite index. The main purpose of composite 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 because the most prevalent columns are accessed first.

Examples

Create a composite index on the HR.EMPLOYEES table.

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

Drop a composite index.

```
DROP INDEX IDX_EMP_COMPI;
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/indexes-and-index-organized-tables.html#GUID-ABE1DE2A-59CC-4ADE-86A5-426B16459464>

PostgreSQL Usage

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.

Examples

Create a multi-column index on the EMPLOYEES table.

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

Drop a multiple-column Index.

```
DROP INDEX IDX_EMP_COMPI;
```

For additional details: <https://www.postgresql.org/docs/13/static/indexes-multicolumn.html>

Oracle Function-Based Indexes vs. PostgreSQL Expression Indexes

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Index	PostgreSQL doesn't support functional indexes that aren't single-column

Oracle Usage

Function-based indexes allow functions to be used in the WHERE clause of queries on indexed columns. Function-based indexes store the output of a function applied on the values of a table column. The Oracle query optimizer only uses a function-based index when the function is used as part of a query.

Oracle updates the index for each DML to ensure that the value that returns from the function is correct.

Examples

Create a function-based index.

```
CREATE TABLE SYSTEM_EVENTS (
    EVENT_ID NUMERIC PRIMARY KEY,
    EVENT_CODE VARCHAR2(10) NOT NULL,
    EVENT_DESCRIPTION VARCHAR2(200),
    EVENT_TIME TIMESTAMPNOT NULL);

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

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/cnct/indexes-and-index-organized-tables.html#GUID-797E49E6-2DCE-4FD4-8E4A-6E761F1383D1>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-INDEX.html#GUID-1F89BBC0-825F-4215-AF71-7588E31D8BFE>

PostgreSQL Usage

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

Examples

1. Creating an Expression Index in PostgreSQL:

```
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 event_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.

```

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;
INSERT 0 1000000

ANALYZE SYSTEM_EVENTS;
ANALYZE

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”:

```

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

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

```

For additional details: <https://www.postgresql.org/docs/13/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY>

Oracle Invisible Indexes vs. PostgreSQL Invisible Indexes

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Index	PostgreSQL does not support Invisible Indexes

Oracle Usage

Oracle's Invisible Index feature gives database administrators the ability to create indexes, or change existing indexes, that are ignored by the optimizer. They are maintained during DML operations and are kept relevant, but are different from usable indexes.

The most common uses for invisible indexes are:

- Testing the effect of a dropped index without actually dropping it.
- Using a specific index for certain operations or modules of an application without affecting the overall application.
- Adding an index to a set of columns on which an index already exists.

Notes:

- Database administrators can force the optimizer to use invisible indexes by changing the OPTIMIZER_USE_INVISIBLE_INDEXES parameter to true.
- Invisible indexes can be used if specified as a HINT.

Examples

Change an index to an invisible index.

```
ALTER INDEX idx_name INVISIBLE;
```

Change an invisible index to a visible index.

```
ALTER INDEX idx_name VISIBLE;
```

Create an invisible index.

```
CREATE INDEX idx_name ON employees(first_name) INVISIBLE;
```

Query all invisible indexes.

```
SELECT TABLE_OWNER, INDEX_NAME FROM DBA_INDEXES
WHERE VISIBILITY = 'INVISIBLE';
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-indexes.html#GUID-3A66938F-73C6-4173-844E-3938A0DBBB54>

PostgreSQL Usage

Currently, Aurora PostgreSQL does not provide a directly comparable alternative for Oracle Invisible Indexes.

Oracle Index-Organized Table (IOT) vs. PostgreSQL “Cluster” Table

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		<u>Index</u>	PostgreSQL does not support Index-Organized Tables, partially workaround available

Oracle Usage

Oracle's Index-organized table (IOT) is a special type of index/table hybrid that physically controls how data is stored at the table and index level. When creating a common database table, or heap-organized table, the data is stored unsorted (as a heap). However, when creating an Index-organized table, the actual table data is stored in a B-tree index structure sorted by the primary key of each row. Each leaf block in the index structure stores both the primary key and non-key columns.

IOTs provide performance improvements when accessing data using the primary key because table records are sorted (clustered) using the primary key and physically co-located alongside the primary key.

Examples

Create an Oracle index-organized table storing “ordered” data based on the primary key.

```
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;

INSERT INTO SYSTEM_EVENTS VALUES(9, 'EVNT-A1-10', 'Critical', '01-JAN-2017');
INSERT INTO SYSTEM_EVENTS VALUES(1, 'EVNT-C1-09', 'Warning', '01-JAN-2017');
INSERT INTO SYSTEM_EVENTS VALUES(7, 'EVNT-E1-14', 'Critical', '01-JAN-2017');
SELECT * FROM SYSTEM_EVENTS;

EVENT_ID      EVENT_CODE      EVENT_DESCRIPTION          EVENT_TIM
-----  -----  -----
1      EVNT-C1-09  Warning          01-JAN-17
7      EVNT-E1-14  Critical         01-JAN-17
9      EVNT-A1-10  Critical         01-JAN-17
```

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

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/indexes-and-index-organized-tables.html#GUID-797E49E6-2DCE-4FD4-8E4A-6E761F1383D1>

PostgreSQL Usage

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.

Examples

```

CREATE TABLE SYSTEM_EVENTS (
    EVENT_ID NUMERIC,
    EVENT_CODE VARCHAR(10) NOT NULL,
    EVENT_DESCIPTION VARCHAR(200),
    EVENT_TIME DATE NOT NULL,
    CONSTRAINT PK_EVENT_ID PRIMARY KEY(EVENT_ID));

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

CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID;
SELECT * FROM SYSTEM_EVENTS;

event_id | event_code | event_desciption | event_time
-----+-----+-----+-----
  1 | EVNT-C1-09 | Warning           | 2017-01-01
  7 | EVNT-E1-14 | Critical          | 2017-01-01
  9 | EVNT-A1-10 | Critical          | 2017-01-01

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

event_id | event_code | event_desciption | event_time
-----+-----+-----+-----
  1 | EVNT-C1-09 | Warning           | 2017-01-01
  7 | EVNT-E1-14 | Critical          | 2017-01-01
  9 | EVNT-A1-10 | Critical          | 2017-01-01
  2 | EVNT-E2-02 | Warning           | 2017-01-01

CLUSTER SYSTEM_EVENTS USING PK_EVENT_ID; -- Run CLUSTER again to re-cluster
SELECT * FROM SYSTEM_EVENTS;

event_id | event_code | event_desciption | event_time
-----+-----+-----+-----
  1 | EVNT-C1-09 | Warning           | 2017-01-01
  2 | EVNT-E2-02 | Warning           | 2017-01-01

```

7		EVNT-E1-14		Critical		2017-01-01
9		EVNT-A1-10		Critical		2017-01-01

For additional details:

- <https://www.postgresql.org/docs/13/static/sql-cluster.html>
- <https://www.postgresql.org/docs/13/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY>

Oracle Local and Global Partitioned Indexes vs. PostgreSQL Partitioned Indexes

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Index	PostgreSQL does not support domain index but behavior is like in Oracle

Oracle Usage

Local and Global Indexes are used for Partitioned Tables in Oracle databases. Each index created on a partitioned table can be specified as either "local" or "global".

- **Local Partitioned Index:** Maintains a one-to-one relationship between the Index partitions and the table partitions. For each table partition, a separate Index Partition is 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 Oracle 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.

Examples

Create a local and global index on a partitioned table.

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

```
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 more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/vldbg/partition-concepts.html#GUID-EA7EF5CB-DD49-43AF-889A-F83AAC0D7D51>

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/vlbg/index-partitioning.html#GUID-569F94D0-E6E5-45BB-9626-5506DE18FF00>

PostgreSQL Usage

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. Also, when using declarative partitions, Global Index is still not supported while creating a "global" index will create an index for each partition, there is a parent index referring to all sub indexes but there is no actual "global" indexes.

- 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, like a Global Indexes in Oracle, has no effect.

While concurrent indexes on partitioned tables build are currently not supported, you may concurrently build the index on each partition individually and then finally create the partitioned index non-concurrently in order to reduce the time where writes to the partitioned table will be locked out. In this case, building the partitioned index is a meta-data only operation.

A CREATE INDEX command invoked on a partitioned table, will RECURSE (default) to all partitions to ensure they all have matching indexes. Each partition is first checked to determine whether an equivalent index already exists, and if so, that index will become attached as a partition index to the index being created, which will become its parent index. If no matching index exists, a new index will be created and automatically attached..

Examples

1. Create the Parent Table:

```
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:

```
CREATE TABLE SYSTEM_LOGS_WARNING (
  CHECK (ERROR_CODE IN('err1', 'err2', 'err3')))

CREATE TABLE SYSTEM_LOGS_CRITICAL (
  CHECK (ERROR_CODE IN('err4', 'err5', 'err6')))
```

3. Create Indexes on each Child Table ("partitions")

```
CREATE INDEX IDX_SYSTEM_LOGS_WARNING ON
  SYSTEM_LOGS_WARNING(ERROR_CODE);

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 more details, see <https://www.postgresql.org/docs/13/static/ddl-partitioning.html>

Oracle Automatic Indexing vs. Self-Managed PostgreSQL

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Index	PostgreSQL does not provide Automatic indexing feature but in self-managed PostgreSQL instances some extensions can be utilized to have automatic index creation

Oracle Usage

Oracle 19 introduces automatic indexing feature. This feature automates the index management tasks by automatically creating, rebuilding, and dropping indexes based on the changes in application workload, thus improving database performance.

Important functionality provided by Automatic Indexing:

- Automatic indexing process runs in the background at a predefined time interval and analyzes application workload. It identifies the tables/columns that are candidates for new indexes and creates new indexes.
- The auto indexes are initially created as invisible indexes. These invisible auto indexes are verified against SQL statements and if the performance is improved, then these indexes are converted as visible indexes.
- Identify and drop any existing under-performing auto indexes or any auto indexes not used for long period.
- Rebuilds the auto indexes that are marked unusable due to DDL operations.
- Provides package DBMS_AUTO_INDEX to configure automatic indexing and for generating reports related to automatic indexing operations.

Note: Notes: Up to date table statistics are very important for the Auto indexing to function efficiently. Tables without statistics or with stale statistics are not considered for auto indexing

Package DBMS_AUTO_INDEX is used to configuring auto indexes and generating reports.

Below are some of the configuration options which can be set by using CONFIGURE procedure of DBMS_AUTO_INDEX package:

- Enabling and disabling automatic indexing in a database
- Specifying schemas and tables that can use auto indexes
- Specifying a retention period for unused auto indexes. By default, the unused auto indexes are deleted after 373 days
- Specifying a retention period for unused non-auto indexes
- Specifying a tablespace and a percentage of tablespace to store auto indexes

Below are some of the reports related to automatic indexing operations which can be generated using REPORT_ACTIVITY and REPORT_LAST_ACTIVITY functions of the DBMS_AUTO_INDEX package.

- Report of automatic indexing operations for a specific period
- Report of the last automatic indexing operation

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-indexes.html#GUID-E4149397-FF37-4367-A12F-675433715904>

PostgreSQL Usage

PostgreSQL does not provide Automatic Indexing feature, but in self-managed PostgreSQL instances, extensions like Dexter (<https://github.com/ankane/dexter>) and HypoPG (<https://hypopg.readthedocs.io/en/latest/>) can be utilized for generating indexes with limitations (Amazon Aurora PostgreSQL does not support these extensions).

The approach taken by these extensions are :

- Identify the queries.
- Update the table statistics if they haven't been analyzed recently.
- Get the initial cost of the queries and create hypothetical indexes on columns that aren't already indexes.
- Get costs again and see if any hypothetical indexes were used. Hypothetical indexes that were used and significantly reduced cost are selected to be indexes.

Examples and how to use guides can be found in the relevant pages mentioned above.

Another applicable option for Aurora for PostgreSQL would be to run a scheduled set of queries to estimate if additional indexes are needed.

The following queries can help determine that.

Find user-tables without primary keys:

```
SELECT c.table_schema, c.table_name, c.table_type
FROM information_schema.tables c
WHERE c.table_schema NOT IN('information_schema', 'pg_catalog') AND c.table_type =
'BASE TABLE'
AND NOT EXISTS(SELECT i.tablename FROM pg_catalog.pg_indexes i
                WHERE i.schemaname = c.table_schema
                AND i.tablename = c.table_name AND indexdef LIKE '%UNIQUE%')
AND NOT EXISTS (SELECT cu.table_name FROM information_schema.key_column_usage cu
                WHERE cu.table_schema = c.table_schema AND
                cu.table_name = c.table_name)
ORDER BY c.table_schema, c.table_name;
```

Query all geometry tables that have no index on the geometry column:

```
SELECT c.table_schema, c.table_name, c.column_name
FROM (SELECT * FROM information_schema.tables WHERE table_type = 'BASE TABLE') As t
INNER JOIN (SELECT * FROM information_schema.columns WHERE udt_name = 'geometry') c
          ON (t.table_name = c.table_name AND t.table_schema = c.table_schema)
LEFT JOIN pg_catalog.pg_indexes i ON
          (i.tablename = c.table_name AND i.schemaname = c.table_schema
          AND indexdef LIKE '%' || c.column_name || '%')
```

```
WHERE i.tablename IS NULL  
ORDER BY c.table_schema, c.table_name;
```

Unused indexes that can probably be dropped:

```
SELECT s.relname, indexrelname, i.indisunique, idx_scan  
FROM pg_catalog.pg_stat_user_indexes s, pg_index i  
WHERE i.indexrelid = s.indexrelid and idx_scan = 0;
```

All of these should not be implemented in a script to decide if indexes should be created or dropped in a production environment. The Oracle Automatic Indexes will first assess if a new index is needed and if so, it will create an invisible index and only after ensuring nothing is harmed, then the index will become visible. This process can't be used in PostgreSQL to avoid any production performance issues as PostgreSQL does not allow for indexes be created have them be invisible.

Special Features & Future Content

Oracle Character Sets vs. PostgreSQL Encoding

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	UTF16 character and NCHAR/NVARCHAR data types are not supported

Oracle Usage

Oracle supports most national and international encoded character set standards including extensive support for Unicode.

Oracle provides two scalar string-specific data types:

- **VARCHAR2:** Stores variable-length character strings with a 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.
- **NVARCHAR2:** Scalar data type used to store Unicode data. Supports AL16UTF16 or UTF8 and is specified during database creation.

Character sets in Oracle are defined at the Instance level (Oracle 11g) or the Pluggable Database level (Oracle 12c R2). In Pre-12cR2 Oracle databases, the character set for the root Container and all Pluggable Databases were required to be identical.

Oracle 18c updates AL32UTF8 and AL16UTF16 character sets to Unicode standard version 9.0.

UTF8 Unicode

Oracle's implementation uses the AL32UTF8 Character Set and provides 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:

- Export/Import from the source Instance/PDB to a new Instance/PDB with a modified character set.
- Use the Database Migration Assistant for Unicode (DMU), which simplifies the migration process to the Unicode character set.

As of 2012, use of the CSALTER utility for character set migrations is deprecated.

Notes:

- Oracle Database 12c Release 1 (12.1.0.1) complies with version 6.1 of the Unicode Standard.
- Oracle Database 12c Release 2 (12.1.0.2) extends the compliance to version 6.2 of the Unicode standard.

- UTF-8 is supported through the AL32UTF8 CS and is valid as both the client and database character sets.
- UTF-16BE is supported through AL16UTF16 and is valid as the national (NCHAR) character set.

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/nlspg/choosing-character-set.html#GUID-BF26E01D-AB92-48FC-855A-69A5B3AF9A92>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/nlspg/appendix-A-locale-data.html#GUID-A9E30C27-FD47-4552-B670-F41A95B11405>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/nlspg/supporting-multilingual-databases-with-unicode.html#GUID-AA09A60E-123E-457C-ACE1-89E4634E492C>

PostgreSQL Usage

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.

RDS ONLY: Starting with PostgreSQL 13, Windows version now support obtaining version information for collations (ordering rules) from OS. This option is relevant for self-managed PostgreSQL installations running on Windows.

When querying the collversion from pg_collation in PostgreSQL running on Windows, prior to version 13 there wasn't any value to reflect the OS collation version, for example version 11 running on Windows:

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');

CREATE COLLATION

select oid, collname, collversion from pg_collation
where collprovider='c' and collname='german';

 oid | collname | collversion
-----+-----+-----
 16394 | german   |
(1 row)

select pg_collation_actual_version (16394);

 pg_collation_actual_version
-----
(1 row)
```

RDS ONLY: Starting PostgreSQL 13 running on Windows:

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');

CREATE COLLATION

select oid, collname, collversion from pg_collation
```

```
where collprovider='c' and collname='german';

oid | collname | collversion
-----+-----+
32769 | german | 1539.5,1539.5
(1 row)

select pg_collation_actual_version (32769);

pg_collation_actual_version
-----
1539.5,1539.5
(1 row)
```

Note:

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 support an NVARHCHAR data type and does not offer support for UTF-16.

Type	Function	Implementation Level
Encoding	Defines the basic rules on how alphanumeric characters are represented in binary format, for example - Unicode Encoding.	Database
Locale	Superset which include LC_COLLATE and LC_CTYPE, among others. LC_COLLATE defines how strings are sorted and needs to be a subset supported by the database Encoding. LC_CTYPE is used to classify if a character is a digit, letter, whitespace, punctuation etc.	Table-Column

Examples

1. Create a database named test01 which uses the Korean EUC_KR Encoding the and the ko_KR locale.

```
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:

```
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.

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/13/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.

Examples

Specify specific collations for individual table columns:

```
CREATE TABLE test1 (col1 text COLLATE "de_DE", col2 text COLLATE "es_ES");
```

Summary

	Oracle	Aurora PostgreSQL
View database character set	SELECT * FROM NLS_DATABASE_PARAMETERS;	select datname, pg_encoding_to_char(encoding), datcollate, datctype from pg_database;
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/13/static/multibyte.html>

Oracle Database Links vs. PostgreSQL DBLink and FDWrapper

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Database Links	Different paradigm and syntax

Oracle Usage

Database Links are schema objects 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.

Examples

Create a database link named `remote_db`. When creating a database link, you have the option to specify the remote database destination using a TNS Entry or to specify the full TNS Connection string.

```
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)))';
```

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 suffix to the table name.

```
SELECT * FROM employees@remote_db;
```

Database links also support DML commands.

```
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 more details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-a-distributed-database.html#GUID-7B0C4627-4473-4313-88D5-FD03CA42D9EA>

PostgreSQL Usage

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.

Examples

1. Load the `dblink` extension into PostgreSQL:

```
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 (`postgres`), port (5432), host (`hostname`), user (`username`) and password (`password`).

```
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).

```
SELECT * from dblink
('myconn', 'SELECT id, name FROM EMPLOYEES') AS p(id int,fullname text);
```

4. Close the connection using the `dblink_disconnect` function.

```
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 host=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/13/static/dblink.html>

Example using the PostgreSQL Foreign Data Wrapper

1. Load the fdw Extension into PostgreSQL.

```
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).

```
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.

```
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.

```
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.

```
SELECT * FROM foreign_emp_tbl;
```

6. You can also “import” an entire schema, or specific tables, without specifying a specific table name:

```
IMPORT FOREIGN SCHEMA hr LIMIT TO (employees)
    FROM SERVER remote_db INTO local_hr;
```

Note:

- Both dblink and FDW store the remote database username and password as plain-text, in two locations:
 - The `pg_user_mapping` view, accessible only to “super users” in the database.
 - When using the `dblink` function, passwords can be stored in your code or procedures inside the database.
- Any changes to PostgreSQL user passwords require changing the FDW/dblink specifications as well.
- 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.

PostgreSQL DBLink vs. FDW

Description	PostgreSQL DBlink	PostgreSQL FDW
Create a permanent reference to a remote table using a database link	Not supported	After creating: - DFW Server definition. - User Mapping. - Run: CREATE FOREIGN TABLE foreign_emp_tbl (id int, name text, address text) SERVER foreign_server OPTIONS (schema_name 'hr', table_name 'employees');
Query remote data	SELECT * FROM dblink('myconn','SELECT * FROM employees') AS p(id int,fullname text, address text);	SELECT * FROM foreign_emp_tbl;
DML on remote data	SELECT * FROM dblink('myconn', \$\$INSERT into employees VALUES (45,'Dan','South side 7432, NY')\$\$) AS t(id int, name text, address text);	INSERT into foreign_emp_tb VALUES (45,'Dan','South side 7432, NY'); (Regular DML)
Run DDL on remote objects	SELECT * FROM dblink ('myconn', \$\$CREATE table my_remote_tbl (a int, b text)\$\$) AS t(a text);	Not Supported

Summary

Description	Oracle	PostgreSQL DBLink
Create a permanent “named” database link	CREATE DATABASE LINK remote CONNECT TO username IDENTIFIED BY password USING 'remote';	Not Supported. You have to manually open the connection to the remote database in your sessions / queries: SELECT dblink_connect('myconn', 'dbname=postgres port=5432 host=hostname user=username password=password');
Query using a database link	SELECT * FROM employees@remote;	SELECT * FROM dblink ('myconn','SELECT * FROM employees') AS p(id int,fullname text, address text);
DML using database link	INSERT INTO employees@remote (employee_id, last_name, email, hire_date, job_id) VALUES (999, 'Claus', 'sclaus@example.com', SYSDATE, 'SH_CLERK');	SELECT * FROM dblink ('myconn', \$\$INSERT into employees VALUES (45,'Dan','South side 7432, NY')\$\$) AS t(id int, name text, address text);
Heterogeneous database link connections, such as Oracle to PostgreSQL or vice-versa	Supported.	create extension oracle_fdw not supported by Amazon RDS.
Run DDL via a database link	Not supported directly, but you can run a procedure or create a job on the remote database and executes the desired DDL commands. dbms_job@remote.submit(l_job, 'execute immediate "create table t (x int)"'); commit;	SELECT * FROM dblink ('myconn', \$\$CREATE table my_remote_tbl (a int, b text)\$\$) AS t(a text);
Delete a database link	drop database link remote;	Not supported. Close the DBLink connection instead. SELECT dblink_disconnect ('myconn');

For additional details, see: <https://www.postgresql.org/docs/13/static/postgres-fdw.html>

Oracle DBMS_SCHEDULER vs. PostgreSQL Scheduled Lambda

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Use Scheduler AWS Lambda

Oracle Usage

The DBMS_SCHEDULER package contains a collection of scheduling functions that can be executed or called from PL/SQL, Create a job and attributes.

When creating a job there are two main objects that should be created too: PROGRAM and SCHEDULE.

A program will define what will be executed when the job calls it.

Scheduler can run database program unit (procedure etc.) or external executable (filesystem sh scripts etc.)

There are three execution methods of jobs:

1. Time Base Scheduling:

Examples of the commands that will create a job with program and schedule:

1. Create a program that will call the procedure UPDATE_HR_SCHEMA_STATS in HR schema.
2. Create a schedule that will set the interval of running the jobs that use it. This schedule will run the job every 1 hour.
3. Create the job itself.

```

BEGIN
DBMS_SCHEDULER.CREATE_PROGRAM(
program_name => 'CALC_STATS',
program_action => 'HR.UPDATE_HR_SCHEMA_STATS',
program_type => 'STORED_PROCEDURE',
enabled => TRUE);
END;
/

BEGIN
DBMS_SCHEDULER.CREATE_SCHEDULE(
schedule_name => 'stats_schedule',
start_date => SYSTIMESTAMP,
repeat_interval => 'FREQ=HOURLY;INTERVAL=1',
comments => 'Every hour');
END;
/

BEGIN
DBMS_SCHEDULER.CREATE_JOB (
job_name          => 'my_new_job3',
program_name      => 'my_saved_program1',
schedule_name     => 'my_saved_schedule1');

```

```
END;
/
```

Create a job without program or schedule:

1. job_type - EXECUTABLE define that our job will run an external script.
2. job_action - Define the location of the external script.
3. start_date - Define since when the job will be enabled.
4. repeat_interval - Define when the job will run, in this example every day at hour 23 (11:00PM).

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB(
job_name=>'HR. BACKUP',
job_type => 'EXECUTABLE',
job_action => '/home/usr/dba/rman/nightly_bck.sh',
start_date=> SYSDATE,
repeat_interval=>'FREQ=DAILY;BYHOUR=23',
comments => 'Nightly backups');
END;
/
```

After the job is created, its attribute can be updated with SET_ATTRIBUTE procedure:

```
BEGIN
DBMS_SCHEDULER.SET_ATTRIBUTE (
name      => 'my_emp_job1',
attribute => 'repeat_interval',
value     => 'FREQ=DAILY');
END;
/
```

2. Event-Based jobs:

Example of creating a schedule that can be used to start a job whenever the Scheduler receives an event indicating that a file arrived on the system before 9AM and then create a job that will use this schedule.

```
BEGIN
DBMS_SCHEDULER.CREATE_EVENT_SCHEDULE (
schedule_name    => 'scott.file_arrival',
start_date       => systimestamp,
event_condition  => 'tab.user_data.object_owner = ''SCOTT''
and tab.user_data.event_name = ''FILE_ARRIVAL''
and extract hour from tab.user_data.event_timestamp < 9',
queue_spec       => 'my_events_q');
END;
/

BEGIN
DBMS_SCHEDULER.CREATE_JOB (
job_name          => my_job,
program_name      => my_program,
start_date        => '15-JUL-04 1.00.00AM US/Pacific',
event_condition   => 'tab.user_data.event_name = ''LOW_INVENTORY''',
queue_spec        => 'my_events_q'
enabled           => TRUE,
```

```

comments          =>  'my event-based job';
END;
/

```

3. Dependency Jobs (Chained):

1. Use DBMS_SCHEDULER.CREATE_CHAIN create chain.
2. Use DBMS_SCHEDULER.DEFINE_CHAIN_STEP to define three steps for this chain. Referenced programs must be enabled.
3. Use DBMS_SCHEDULER.DEFINE_CHAIN_RULE to define corresponding rules for the chain.
4. Use DBMS_SCHEDULER.ENABLE to enable the chain.
5. Use DBMS_SCHEDULER.CREATE_JOB to create a chain job to start the chain daily at 1:00 p.m.

```

BEGIN
DBMS_SCHEDULER.CREATE_CHAIN (
chain_name          =>  'my_chain1',
rule_set_name       =>  NULL,
evaluation_interval =>  NULL,
comments           =>  NULL);
END;
/

BEGIN
DBMS_SCHEDULER.DEFINE_CHAIN_STEP('my_chain1', 'stepA', 'my_program1');
DBMS_SCHEDULER.DEFINE_CHAIN_STEP('my_chain1', 'stepB', 'my_program2');
DBMS_SCHEDULER.DEFINE_CHAIN_STEP('my_chain1', 'stepC', 'my_program3');
END;
/

BEGIN
DBMS_SCHEDULER.DEFINE_CHAIN_RULE ('my_chain1', 'TRUE', 'START stepA');
DBMS_SCHEDULER.DEFINE_CHAIN_RULE (
'my_chain1', 'stepA COMPLETED', 'Start stepB, stepC');
DBMS_SCHEDULER.DEFINE_CHAIN_RULE (
'my_chain1', 'stepB COMPLETED AND stepC COMPLETED', 'END');
END;
/

BEGIN
DBMS_SCHEDULER.ENABLE('my_chain1');
END;
/

BEGIN
DBMS_SCHEDULER.CREATE_JOB (
job_name            =>  'chain_job_1',
job_type            =>  'CHAIN',
job_action          =>  'my_chain1',
repeat_interval    =>  'freq=daily;byhour=13;byminute=0;bysecond=0',
enabled             =>  TRUE);
END;
/

```

A few words on how manage you jobs. There two additional subjects to maintain your jobs:

1. JOB CLASS - when you have a number of jobs that has the same behavior and attributes, maybe you will want to group them together into bigger logical group called “Job Class” and you can give priority between job classes by allocating a high percentage of available resources.
2. WINDOW - when you want to prioritize your jobs based on schedule, you can create a “window” of time that the jobs can run during this window for example non-peak time or at end of the month.

For additional details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/scheduling-jobs-with-oracle-scheduler.html#GUID-D41660D0-D88F-4D9F-8CC8-63D040EDC4E6>

PostgreSQL Usage

Aurora PostgreSQL can be combined with AWS CloudWatch and Lambda to get similar functionality, see [DB Mailing](#).

Oracle External tables vs. PostgreSQL Integration with Amazon S3

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Create Tables	PostgreSQL doesn't support EXTERNAL TABLEs

Oracle Usage

The Oracle External tables feature allows you to create a table in your database that reads data from a source located outside your database (externally).

Beginning with Oracle 12.2, the external table can be partitioned, providing all the benefits of a regular partitioned table.

Oracle 18c adds support for inline external tables, which is a way to get data from external source in a SQL query without having to define and create external table first.

```
SELECT * FROM EXTERNAL ((i NUMBER, d DATE)
TYPE ORACLE_LOADER
DEFAULT DIRECTORY data_dir
ACCESS PARAMETERS (
RECORDS DELIMITED BY NEWLINE
FIELDS TERMINATED BY '|') LOCATION ('test.csv') REJECT LIMIT UNLIMITED) tst_external;
```

Examples

CREATE TABLE with ORGANIZATION EXTERNAL to identify it as an external table. Specify the TYPE to let the database choose the right driver for the data source, the options are:

- ORACLE_LOADER – The data must be sourced from text data files. (default)
- ORACLE_DATAPUMP –The data must be sourced from binary dump files. You can write dump files only as part of creating an external table with the CREATE TABLE AS SELECT statement. Once the dump file is created, it can be read any number of times, but it cannot be modified (that is, no DML operations can be performed).
- ORACLE_HDFS – Extracts data stored in a Hadoop Distributed File System (HDFS).
- ORACLE_HIVE – Extracts data stored in Apache HIVE.
- DEFAULT DIRECTORY – In database definition for the directory path.
- ACCESS PARAMETER – Defines the DELIMITER character and the query fields.
- LOCATION – The file name in the first two data source types or URI in the Hadoop data source (not in use with hive data source).

```
CREATE TABLE emp_load
(id CHAR(5), emp_dob CHAR(20), emp_lname CHAR(30), emp_fname CHAR(30), emp_start_date
DATE) ORGANIZATION EXTERNAL
(TYPE ORACLE_LOADER DEFAULT DIRECTORY data_dir ACCESS PARAMETERS
(RECORDS DELIMITED BY NEWLINE FIELDS
(id CHAR(2),emp_dob CHAR(20), emp_lname CHAR(18), emp_fname CHAR(11), emp_start_date
```

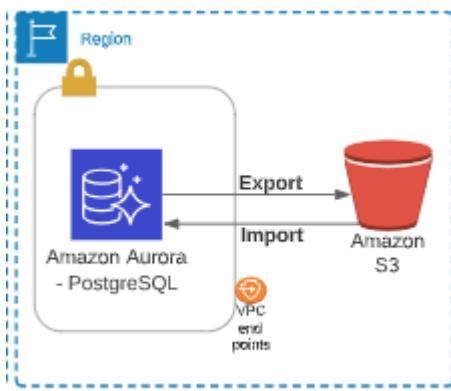
```
CHAR(10) date_format DATE mask "mm/dd/yyyy"))
LOCATION ('info.dat'));
```

For more details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/18/sutil/oracle-external-tables-concepts.html#GUID-44323E01-7D72-45EC-915A-99E596769D9E>

PostgreSQL Usage

Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data.

The following diagram illustrates the solution architecture.



This is the most relevant capability for the Oracle's External Tables in Aurora for PostgreSQL, but requires a significant amount of syntax modifications. The main difference is that there is no open link to files and the data must be transferred from and to PostgreSQL (if all data is needed) .

There are two important operations for Aurora for PostgreSQL and S3 integration :

- Saving data to an S3 file.
- Loading data from an S3 file.

RDS Aurora for PostgreSQL must have permissions to the S3 bucket. For more information, see the links at the end of this section.

In Oracle 18c, the inline external table feature was introduced. this, cannot be achieved in Aurora for PostgreSQL and it depends on the use case but other services can be considered.

For ETLs for example, AWS Glue can be considered.

Saving Data to S3

You can use the `aws_s3.query_export_to_s3` function to query data from an Amazon Aurora PostgreSQL and save it directly into text files stored in an Amazon S3 bucket. Use this functionality to avoid transferring data to the client first, and then copying the data from the client to Amazon S3.

Note: The default file size threshold is six gigabytes (GB). If the data selected by the statement is less than the file size threshold, a single file is created. Otherwise, multiple files are created.

If the execution fails , files already uploaded to Amazon S3 remain in the specified Amazon S3 bucket. You can use another statement to upload the remaining data instead of starting over again.

If the amount of data to be selected is large (more than 25 GB), it is recommended to use multiple executions on different portions of the data to save it to Amazon S3.

Meta-data, such as table schema or file meta-data, is not uploaded by Aurora PostgreSQL to Amazon S3.

Examples

Add S3 extension

```
CREATE EXTENSION IF NOT EXISTS aws_s3 CASCADE;
```

The following statement selects all data in the employees table and saves the data into an Amazon S3 bucket in a different region from the Aurora PostgreSQL instance. The statement returns an error if files that match the sample_employee_data file prefix exist in the specified Amazon S3 bucket.

```
SELECT *
FROM aws_s3.query_export_to_s3(
'SELECT * FROM employees',
aws_commons.create_s3_uri(
'aurora-select-into-s3-pdx',
'sample_employee_data','s3-us-west-2'));
```

The following statement selects all data in the employees table and saves the data into an Amazon S3 bucket in the same region as the Aurora MySQL DB cluster. The statement creates data files in which each field is terminated by a comma (,) character and each row is terminated by a newline (\n) character. It also creates a manifest file. The statement returns an error if files that match the sample_employee_data file prefix exist in the specified Amazon S3 bucket.

```
SELECT *
FROM aws_s3.query_export_to_s3(
'SELECT * FROM employees',
aws_commons.create_s3_uri(
'aurora-select-into-s3-pdx',
'sample_employee_data','us-west-2'), options :='format csv, delimiter $$,$$');
```

query_export_to_s3 summary

Field	Description
query	A required text string containing an SQL query that the PostgreSQL engine runs. The results of this query are copied to an S3 bucket identified in the s3_info parameter.
bucket	A required text string containing the name of the Amazon S3 bucket that contains the file.
file_path	A required text string containing the Amazon S3 file name including the path of the file.
region	An optional text string containing the AWS Region that the bucket is in
options	An optional text string containing arguments for the PostgreSQL COPY command. For more details, see the PostgreSQL COPY documentation

For more details, see <https://aws.amazon.com/blogs/database/export-and-import-data-from-amazon-s3-to-amazon-aurora-postgresql/>

Loading Data from S3

You can use the `table_import_from_s3` function to load data from files stored in an Amazon S3 bucket.

Examples

The following example runs the `table_import_from_s3` function to import gzipped csv from S3 into the `test_gzip` table.

```
CREATE TABLE test_gzip(id int, a text, b text, c text, d text);

SELECT aws_s3.table_import_from_s3(
  'test_gzip', '',
  '(format csv)',
  'myS3Bucket', 'test-data.gz', 'us-east-2');
```

table_import_from_s3 summary

Field	Description
table_name	A required text string containing the name of the PostgreSQL database table to import the data into
column_list	A required text string containing an optional list of the PostgreSQL database table columns in which to copy the data. If the string is empty, all columns of the table are used
options	A required text string containing arguments for the PostgreSQL COPY command. For more details, see the PostgreSQL COPY documentation
s3_info	An <code>aws_commons._s3_uri_1</code> composite type containing the following information about the S3 object: <ul style="list-style-type: none"> • <code>bucket</code> - The name of the Amazon S3 bucket containing the file. • <code>file_path</code> - The Amazon S3 file name including the path of the file. • <code>region</code> - The AWS Region that the file is in. For a listing of AWS Region names and associated values
credentials	The credentials parameter specifies the credentials to access Amazon S3. When you use this parameter, you don't use an IAM role.

For more details, see <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/PostgreSQL.Procedural.Importing.html>

Oracle Inline Views vs. PostgreSQL Inline Views

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
 	 	N/A	

Oracle Usage

Inline views refer to a SELECT statement located in the FROM clause of secondary 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.

Examples

The SQL statement marked in red represents the inline view code. The query returns each employee matched to their salary and department id. In addition, the query returns the average salary for each department using the inline view column SAL_AVG.

```
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;
```

PostgreSQL Usage

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.

Examples

“B” used as an alias.

```
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;
```

Oracle JSON Document Support vs. PostgreSQL JSON Support

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different paradigm and syntax will require application/drivers rewrite

Oracle Usage

JSON documents are based on JavaScript syntax and allow the 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 an Oracle database to support semi-structured data. Oracle JSON support also includes full-text search and several other functions dedicated to querying JSON documents.

Oracle 19 adds new function, `JSON_SERIALIZE`, which allows you to serialize JSON objects to text.

For more information, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/intro-to-json-data-and-oracle-database.html#GUID-17642E43-7D87-4590-8870-06E9FDE9A6E9>.

Examples

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

```
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.

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

FNAME      LNAME      POSTCODE
```

```
-----
John      Doe      90210
-----  
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 more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/intro-to-json-data-and-oracle-database.html#GUID-17642E43-7D87-4590-8870-06E9FDE9A6E9>

PostgreSQL Usage

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 re-parse 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 re-parsing 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.

Starting with PostgreSQL 10, both JSON and JSONB are compatible with full-text search.

For additional details, see: <https://www.postgresql.org/docs/13/static/datatype-json.html>

In order to comply with the full JSON specification, database encoding must be set to UTF8. If the database code page is not set to UTF8, then non-UTF8 characters are allowed and the database encoding will be non-compliant with the full JSON specification.

Examples

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:

```
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:

```
SELECT emp_data FROM employees WHERE emp_data ? 'address';
```

3. Return all JSON items that have an address key or a hobbies key:

```
SELECT * FROM employees WHERE emp_data ?| array['address', 'hobbies'];
```

4. Return all JSON items that have both an address key and a hobbies key:

```
SELECT * FROM employees WHERE emp_data ?& array['a', 'b'];
```

5. Return the value of home key in the phone numbers array:

```
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):

```
SELECT * FROM employees WHERE emp_data->>'address' = '1234 First Street, Capital City';
```

```
SELECT * FROM employees WHERE emp_data->>'address' like '%Capital City%';
```

7. Using operators with JSON values:

Is the left JSON is contain the right JSON

```
select '{"id":132, "name":"John"}'::jsonb @> '{"id":132}'::jsonb;
```

Concatenating two JSON values:

```
select '{"id":132, "fname":"John"}'::jsonb || '{"lname":"Doe"}'::jsonb;
```

Removing keys from JSON:

```
select '{"id":132, "fname":"John", "salary":999999, "bank_account":1234}'::jsonb
      - '{salary,bank_account}'::text[];
```

For additional details:<https://www.postgresql.org/docs/13/static/functions-json.html>

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.

```
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:

```
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 PostgreSQL has to step into each JSON document.

1. Create an index on the address key of emp_data:

```
CREATE idx1_employees ON employees ((emp_data->>'address'));
```

2. Create a GIN index on a specific key or the entire emp_data column:

```
CREATE INDEX idx2_employees ON cards USING gin ((emp_data->'tags'));
CREATE INDEX idx3_employees ON employees USING gin (emp_data);
```

Summary

Feature	Oracle	Aurora PostgreSQL
Return the full JSON document / all JSON documents	emp_data is a column that stores json documents: SELECT emp_data FROM employees;	emp_data is a column that stores json documents: SELECT emp_data FROM employees;
Return a specific element from a JSON document	Return only the address property: SELECT e.emp_data.address FROM employees e;	Return only the address property, for emp_id 1 from the emp_data JSON column in the employees table: SELECT emp_data->>'address' from employees where emp_id = 1;
Return JSON documents matching a pattern in any field	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 '%pattern%';	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%'; SELECT key, value FROM card, lateral jsonb_each_text(data) WHERE value LIKE '%pattern%';
Return JSON documents matching a pattern in specific fields (root level)	SELECT e.emp_data.name FROM employees e WHERE e.data.active = 'true';	Only return results where the "finished" property in the JSON document is true: SELECT * FROM employees WHERE emp_data->>'active' = 'true';

Feature	Oracle	Aurora PostgreSQL
Define a column in a table that supports JSONB documents	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));	1. Create a table with a column defined as JSON: CREATE TABLE json_docs (id integer NOT NULL, data jsonb);

For additional details:

<https://www.postgresql.org/docs/13/static/datatype-json.html>

<https://www.postgresql.org/docs/13/static/functions-json.html>

Oracle Materialized Views vs. PostgreSQL Materialized Views

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Materialized Views	PostgreSQL does not support automatic or incremental REFRESH

Oracle Usage

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 specific queries and can be manually or automatically refreshed based on specific configurations. A Materialized View runs its associated query and stores the results as a table segment.

Oracle Materialized Views are especially useful for:

- Replication of data across multiple databases.
- Data warehouse use cases.
- Increasing performance 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 an MView query can reference tables, views, and other Materialized Views. The source objects that an Mview uses as data sources are also called “master tables” (replication terminology) or “detail tables” (data warehouse terminology).

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 a deferred update where the Materialized View is populated only on the first requested refresh.

Fast and Complete Refresh

- **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.
- **COMPLETE:** The table segment used by the Materialized View is truncated (data is cleared) and repopulated 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, which provides 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 to using Materialized View Logs.

Materialized View Refresh Strategy

- **ON COMMIT:** Refreshes the Materialized View upon any commit made on the underlying associated tables.
- **ON DEMAND:** The refresh is initiated via a scheduled task or manually by the user.

Examples

Create a simple Materialized View named mv1 that executes a simple SELECT statement on the employees table.

```
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM hr.employees;
```

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 allows the Materialized View to be updated.

```
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);
```

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. Also, create a new Materialized View named sales_mv which is 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.

```
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 more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/basic-materialized-views.html#GUID-A7AE8E5D-68A5-4519-81EB-252EAAF0ADFF>

PostgreSQL Usage

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 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 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.

Note: In PostgreSQL 10, the statistics collector is being update properly after a REFRESH MATERIALIZED VIEW execution

Examples

1. Create a materialized view named sales_summary using the sales table as the source for the Materialized View:

```
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:

```
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.

```
CREATE UNIQUE INDEX sales_summary_seller
ON sales_summary (seller_no, sale_date);
```

Summary

Option	ORACLE	PostgreSQL
Create Materialized View	CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;	CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;
Manual refresh of a Materialized View	DBMS_MVIEW.REFRESH('mv1', 'cf'); The--cf parameter configured the refresh method: c is complete and f is fast	REFRESH MATERIALIZED VIEW mv1;
Online refresh of a Materialized 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 refresh of a Materialized View	CREATE MATERIALIZED VIEW LOG ON employees... INCLUDING NEW VALUES; CREATE MATERIALIZED VIEW mv1 REFRESH FAST AS SELECT * FROM employees;	Not Supported
DML on Materialized View data	Supported	Not Supported

For more details, see:

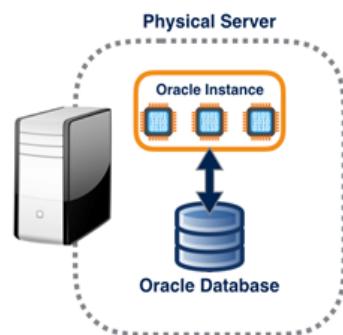
<https://www.postgresql.org/docs/13/static/rules-materializedviews.htm>

Oracle Multitenant vs. PostgreSQL Databases

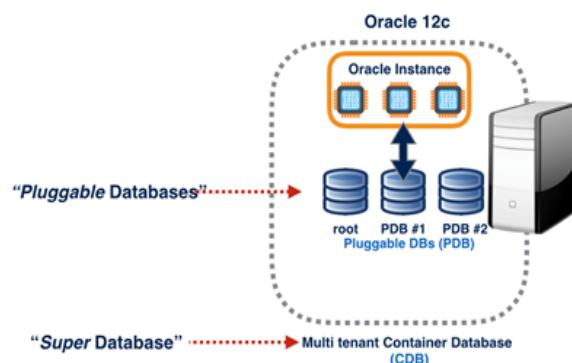
Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Distribute load/applications/ users across multiple instances

Oracle Usage

Oracle 12c introduces a new multitenant architecture that 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 a single Oracle database as shown in the diagram below.



Oracle 12c introduces a new multi-Container Database (CDB) that supports one or more Pluggable Databases (PDB). The CDB can be thought of as a single superset database with multiple pluggable databases. The relationship between an Oracle instance and databases is now 1:N.



Oracle 18c adds following multi-tenant related features:

- DBCA PDB Clone: UI interface which allows cloning multiple pluggable databases (PDB).
- Refreshable PDB Switchover: ability to switch roles between pluggable database clone and its original master.

- CDB Fleet Management: ability to group multiple container databases (CDB) into fleets that can be managed as a single logical database.

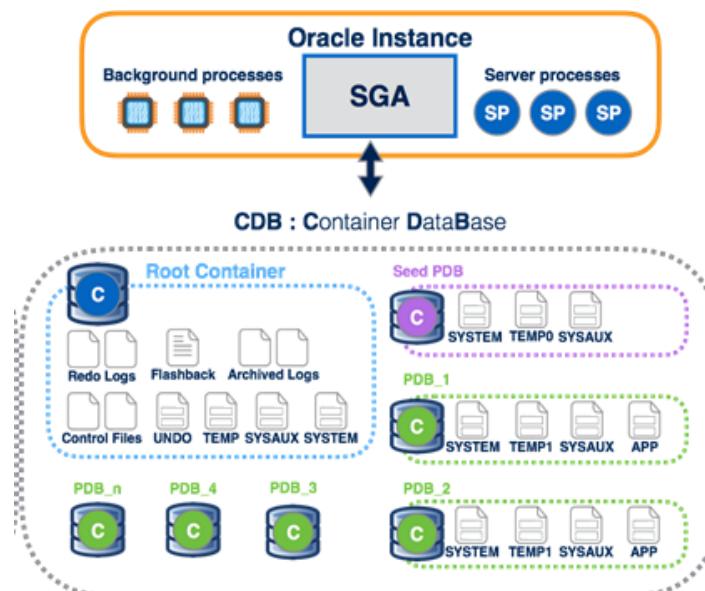
Oracle 19 introduced support to having more than one pluggable database (PDB) in a container database (CDB) in sharded environments.

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 be 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.
- Provides 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 CDB can support one or more PDBs.
- Each PDB contains its own copy of SYSTEM and application tablespaces.
- The PDBs 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 (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 is used as a template for creating new PDBs.



CDB and PDB Semantics

- CDB
 - 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
 - An 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 data files, system data files, and tablespaces along with temporary files to manage objects.

Examples

List existing PDBs created in an Oracle CDB instance.

```
SHOW PDBS;
CON_ID      CON_NAME      OPEN MODE  RESTRICTED
-----      -----
2           PDB$SEED      READ ONLY   NO
3           PDB1          READ WRITE  NO
```

Provision a new PDB from the template seed\$pdb.

```
CREATE PLUGGABLE DATABASE PDB2 admin USER ora_admin IDENTIFIED BY
ora_admin FILE_NAME_CONVERT=('/pdbseed/', '/pdb2/');
```

Alter a specific PDB to READ/WRITE and verify.

```
ALTER PLUGGABLE DATABASE PDB2 OPEN READ WRITE;
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
```

Clone a PDB from an existing PDB.

```
CREATE PLUGGABLE DATABASE PDB3 FROM PDB2 FILE_NAME_CONVERT= ('/pdb2/', '/pdb3/');
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	

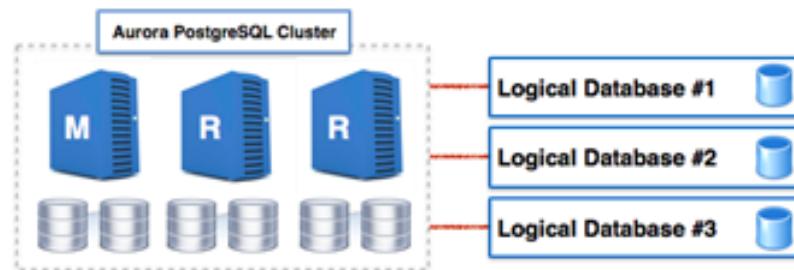
For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/multi/index.html>

PostgreSQL Usage

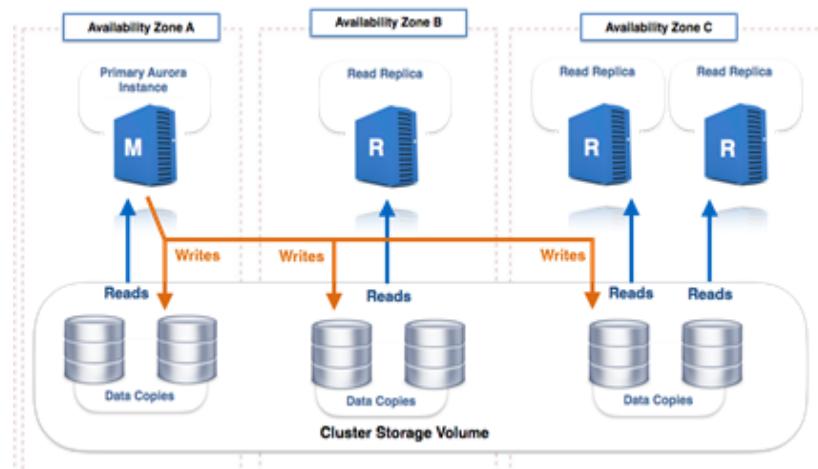
Amazon Aurora PostgreSQL offers a different and simplified architecture to manage and create a multitenant database environment. Using Aurora PostgreSQL, it is possible to provide levels of functionality similar (but not identical) to those offered by Oracle PDBs by creating multiple databases under the same Aurora PostgreSQL cluster and / or using separate Aurora clusters if 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 providing scale-out functionality for application reads and high availability.



An Oracle CDB/Instance is a high-level equivalent to an Amazon Aurora cluster, and an Oracle Pluggable Database (PDB) is equivalent to a PostgreSQL database created inside the Amazon Aurora cluster. Not all features are comparable between Oracle 12c PDBs and Amazon Aurora.

PostgreSQL allows you to copy databases using templates. Database needs to be created or modified to have IS_TEMPLATE flag set to true and then new database could be created with CREATE DATABASE <newdb-name> TEMPLATE <templatedbname>

Starting at Oracle 18c and 19c , a new usages of this feature were introduced like:

- PDB Clone
- Refreshable PDB Switchover
- CDB Fleet Management
- More than one pluggable database (PDB) in a container database (CDB) in sharded environments.

In the AWS cloud, these features can be achieved in many ways and each can be optimized using different services.

Cloning a database inside the PostgreSQL instance is easy, just use the TEMPLATE option with the CREATE DATABASE statement. the following command will copy the emp database to emp_bck

```
CREATE DATABASE emp_bck TEMPLATE emp;
```

To achieve similar functionality like Refreshable PDB Switchover, it depends on the use case but there are multiple options mostly depended on the required granularity:

- Databases in the same instance - "failover" can be done using CREATE DATABASE statement when size and required downtime allow that and use an application failover to point to any of the databases.
- Database links and replication method - database links or AWS DMS can be used to make sure there are two databases in two different instances that are in sync and have application failover to point to the other database when needed.
- PostgreSQL logical replication provides fine-grained control over replicating and synchronizing parts of a database. For example, you can use logical replication to replicate an individual table of a database.

Managing CDB is actually very similar to the AWS orchestration, as you can manage multiple RDS instances there (CDB) and databases inside (PDB), all monitored centrally and can be managed through the AWS console or AWS CLI.

Examples

Create a new database in PostgreSQL using the CREATE DATABASE statement.

```
CREATE DATABASE pg_db1;
CREATE DATABASE pg_db2;
CREATE DATABASE pg_db3;
```

List all databases created under an Amazon Aurora PostgreSQL cluster.

```
\l

      Name   |    Owner    | Encoding | Collate |    Ctype    |
-----+-----+-----+-----+-----+
adminedb | 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

Oracle Resource Manager vs. PostgreSQL Dedicated Amazon Aurora Clusters

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Distribute load/applications/ users across multiple instances

Oracle Usage

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.

Note:

- 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.

Examples

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.

```
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.

```
BEGIN
DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
PLAN          => 'DAYTIME',
GROUP_OR_SUBPLAN => 'OLTP',
COMMENT        => 'OLTP group',
MGMT_P1       => 75);
END;
/
```

For additional details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-resources-with-oracle-database-resource-manager.html#GUID-2BEF5482-CF97-4A85-BD90-9195E41E74EF>

PostgreSQL Usage

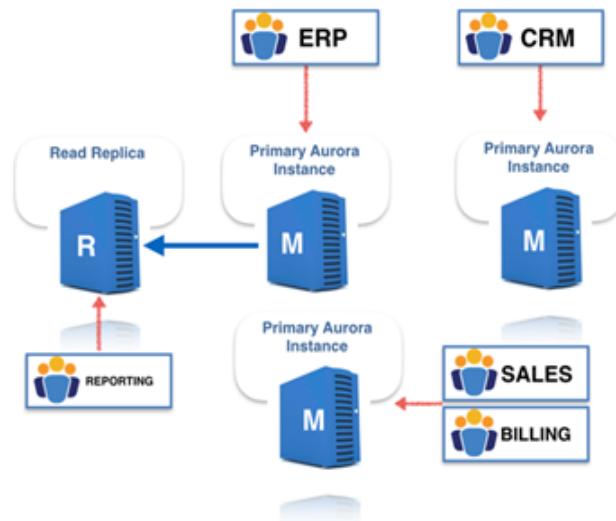
PostgreSQL does not have built-in resource management capabilities that are equivalent to the functionality provided by Oracle Resource Manager. However, due to the elasticity and flexibility provided by “cloud economics”, workarounds could be applicable and such capabilities might not be as of similar 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.

For instance types and resources: <https://aws.amazon.com/ec2/instance-types/>

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.

Examples

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. Go to [instances section under RDS](#)

2. Select the Amazon Aurora cluster that you want to scale-out by adding an additional reader, click on the instance "Actions" button and click on "Add reader".

The screenshot shows the AWS RDS Databases console. In the main table, there are two clusters: 'mysql-aurora-playbook' (Regional, Aurora MySQL, eu-central-1) and 'pg-playbooks' (Regional, Aurora PostgreSQL, eu-central-1). Under 'pg-playbooks', the 'Actions' button is highlighted with a red box labeled '1'. A dropdown menu is open, with the 'Add reader' option highlighted with a red box labeled '2'.

DB identifier	Role	Engine	Region & AZ	Size	Status
mysql-aurora-playbook	Regional	Aurora MySQL	eu-central-1	1 instance	Available
mysql-aurora-playbook-instance-1	Writer	Aurora MySQL	eu-central-1a	db.t3.medium	Available
oraplaybook	Instance	Oracle Standard Edition Two	eu-central-1a	db.t3.medium	Available
pg-playbooks 1	Regional	Aurora PostgreSQL	eu-central-1	1 instance	Available
pg-playbooks-instance-1	Writer	Aurora PostgreSQL	eu-central-1a	db.t3.medium	Available

3. Select the instance class depending on the amount of compute resources your application requires.

The screenshot shows the 'DB instance class' selection dialog. It lists various instance classes and their specifications. The 'db.r5.large' class is selected and highlighted with a blue box. Other visible classes include db.r5.xlarge, db.r5.2xlarge, db.r5.4xlarge, db.r5.8xlarge, db.r5.12xlarge, db.r5.16xlarge, db.r5.24xlarge, and db.r6g.large. To the right, a sidebar shows 'recommended for' and a 'Create' button.

db.r5.large	2 vCPUs	16 GiB RAM	Network: 4,750 Mbps
db.r5.xlarge	4 vCPUs	32 GiB RAM	Network: 4,750 Mbps
db.r5.2xlarge	8 vCPUs	64 GiB RAM	Network: 4,750 Mbps
db.r5.4xlarge	16 vCPUs	128 GiB RAM	Network: 4,750 Mbps
db.r5.8xlarge	32 vCPUs	256 GiB RAM	Network: 6,800 Mbps
db.r5.12xlarge	48 vCPUs	384 GiB RAM	Network: 9,500 Mbps
db.r5.16xlarge	64 vCPUs	512 GiB RAM	Network: 13,600 Mbps
db.r5.24xlarge	96 vCPUs	768 GiB RAM	Network: 19,000 Mbps
db.r6g.large	2 vCPUs	16 GiB RAM	Network: 4,750 Mbps

4. Once completed, click Create Aurora Replica.

Summary

Oracle Resource Manager	Amazon Aurora Instances
Set the maximum CPU usage for a resource group	Create a dedicated Aurora Instance for a specific application
Limit the degree of parallelism for specific queries	<p>SET max_parallel_workers_per_gather TO x;</p> <p><i>Setting the PostgreSQL max_parallel_workers_per_gather parameter</i> should be done as part of your application database connection.</p>
Limit parallel execution	<p>SET max_parallel_workers_per_gather TO x; -- by a single Gather or Gather Merge node OR SET max_parallel_workers TO x; -- for the whole system (since PostgreSQL 10)</p>
Limit the number of active sessions	<p>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.</p> <pre>select pid from pg_stat_activity where username in(select username from pg_stat_activity where state = 'active' group by username having count(*) > 10) and state = 'active' order by query_Start;</pre>
Restrict maximum runtime of queries	<p>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 database procedures or within the application DAL itself.</p> <pre>SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE now()-pg_stat_activity.query_start > interval '5 minutes';</pre>
Limit the maximum idle time for sessions	<p>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.</p> <pre>SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname = 'regress' AND pid <> pg_backend_pid() AND state = 'idle' AND state_change < current_timestamp - INTERVAL '5' MINUTE;</pre>
Limit the time that an idle session holding open locks can block other sessions	<p>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.</p> <pre>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</pre>

Oracle Resource Manager	Amazon Aurora Instances
	AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid AND blocking_locks.pid != blocked_locks.pid JOIN pg_catalog.pg_stat_activity AS blocking_activity ON blocking_activity.pid = blocking_locks.pid WHERE NOT blocked_locks.granted and blocked_activity.state_change < current_timestamp - INTERVAL '5' minute;
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/13/static/runtime-config-resource.html>

Oracle SecureFile LOBs vs. PostgreSQL Large Objects (LOBs)

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	SecureFiles are not supported by PostgreSQL, automation and compatibility refer only to LOBs

Oracle Usage

Large Objects (LOB) is a mechanism for storing binary data in a database. Oracle 11g introduced Secure File LOBS that provide more efficiently storage. They are created using the SECUREFILE keyword as part of the CREATE TABLE statement.

The Primary benefits of using SECUREFILE lobs include:

- Compression:** Uses Oracle advanced compression to analyze SecureFiles LOB data to save disk space.
- De-Duplication:** Automatically detects duplicate LOB data within a LOB column or partition and reduces storage space by removing duplicates of repeating binary data.
- Encryption:** Combined with Transparent Data Encryption (TDE).

Examples

Create a table using a SecureFiles LOB column.

```
CREATE TABLE sf_tab (COL1 NUMBER, COL2_CLOB CLOB)
LOB(COL2_CLOB) STORE AS SECUREFILE;
```

Provide additional options for LOB compression during table creation.

```
CREATE TABLE sf_tab (COL1 NUMBER,COL2_CLOB CLOB)
LOB(COL2_CLOB) STORE AS SECUREFILE COMPRESS_LOB(COMPRESS HIGH);
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/adlob/introduction-to-large-objects.html#GUID-1A2B0023-9EE8-48AF-AA76-171D1FC5C241>

PostgreSQL Usage

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/13/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/13/static/largeobjects.html>

Oracle Views vs. PostgreSQL Views

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		Views	

Oracle Usage

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 a “virtual table” or a “logical table” based on the data from one or more “physical” database tables.

Privileges

- 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).

CREATE (OR REPLACE) VIEW Statements

- **CREATE VIEW:** Creates a new view.
- **CREATE OR REPLACE:** Overwrites an existing view and modifies the view definition without having to manually drop and recreate the original view, and without deleting the previously granted privileges.

Oracle Common View Parameters

Oracle View Parameter	Description
CREATE OR REPLACE	Recreate an existing view (if one exists) or create a new view.
FORCE	Create the view regardless of the existence of the source tables or views and regardless of view privileges.
VISIBLE INVISIBLE	Specify if a column based on the view is visible or invisible.
WITH READ ONLY	Disable DML commands.
WITH CHECK OPTION	Specifies the level of enforcement when performing DML commands on the view.

Examples

Views are classified as either Simple or Complex.

A Simple View is 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). The following example creates and updates a Simple View.

```

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.

```

A Complex View is 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. The following example creates and updates a Complex view.

```

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 more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-VIEW.html#GUID-61D2D2B4-DACC-4C7C-89EB-7E50D9594D30>

PostgreSQL Usage

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.

Views with INSTEAD INSERT triggers can be used with COPY command, with this synopsis:

```
COPY view FROM source;
```

RDS ONLY: Starting with PostgreSQL 13 it is now possible to rename view columns using ALTER VIEW command, this will help the DBA to avoid dropping and recreating the view in order to change a column name.

The following syntax was added to the ALTER VIEW:

```
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_
column_name
```

Prior to PostgreSQL 13 the capability was there but in order to change the view's column name the DBA had to use the ALTER TABLE command.

PostgreSQL View Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ (
  column_name [, ...] )
  [ WITH ( view_option_name [= view_option_value] [, ...] ) ]
  AS query
  [ WITH [ CASCDED | 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/13/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.

```
CREATE [OR REPLACE] VIEW VW_NAME AS SELECT COLUMNS FROM TABLE(s) [WHERE
CONDITIONS];
DROP VIEW [IF EXISTS] VW_NAME;
```

Note: The IF EXISTS parameter is optional

- **WITH [CASCDED | 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.
- CASCDED - 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.

Examples

Creating and updating a view without the CHECK OPTION parameter:

```
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.

UPDATE VW_DEP SET LOCATION_ID=1600;

21 rows updated.
```

Creating and updating a view with the LOCAL CHECK OPTION parameter:

```
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.

UPDATE VW_DEP SET LOCATION_ID=1600;

SQL Error: ERROR: new row violates check option for view "vw_dep"
```

For more information, see:

- <https://www.postgresql.org/docs/13/static/tutorial-views.html>
- <https://www.postgresql.org/docs/13/static/sql-createrule.html>

Oracle XML DB vs. PostgreSQL XML Type & Functions

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Different paradigm and syntax will require application/drivers rewrite.

Oracle Usage

Oracle XML DB is a set of Oracle Database technologies providing XML capabilities for database administrators and developers. It provides native XML support and other features including the native XMLType and XMLIndex.

XMLType represents an XML document in the database that is accessible from SQL. It supports standards such as XML Schema, XPath, XQuery, XSLT, and DOM.

XMLIndex supports all forms of XML data from highly structured to completely unstructured.

XML data can be schema-based or non-schema-based. Schema-based XML adheres to an XSD Schema Definition and must be validated. Non-schema-based XML data does not require validation.

According to Oracle documentation, the aspects you should consider when using XML are:

- The ways that you intend to store your XML data.
- The structure of your XML data.
- The languages used to implement your application.
- The ways you intend to process your XML data.

The most common features are:

- **Storage Model:** Binary XML
- **Indexing:** XML search index, XMLIndex with structured component
- **Database language:** SQL, with SQL/XML functions
- **XML languages:** XQuery and XSLT

Storage Model - Binary XML

Also called “post-parse persistence”, it is the default storage model for Oracle XML DB. It is a post-parse, binary format designed specifically for XML data. Binary XML is XML schema-aware and the storage is very flexible. You can use it for XML schema-based documents or for documents that are not based on an XML schema. You can use it with an XML schema that allows for high data variability or that evolves considerably or unexpectedly. This storage model also provides efficient partial updating and streaming query evaluation.

The other storage option is “Object-relational storage” and is more efficient when using XML as structured data with a minimum amount of changes and different queries. For more information, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/adxdb/xml-db-developers-guide.pdf>.

Indexing - XML search index, XMLIndex with structured component

XML Search Index provides full-text search over XML data. Oracle recommends storing XMLType data as Binary XML and to use XQuery Full Text (XQFT). If you are not using Binary storage and your data is structured XML, you can use the Oracle Text indexes, use the regular string functions such as “contains”, or use XPath “ora[contains”. If you want to use predicates such as “XMLExists” in your WHERE clause, you must create an XML search index.

Examples

The following example creates a SQL directory object, which is a logical name in the database for a physical directory on the host computer. This directory contains XML files. The example inserts XML content from the purOrder.xml file into the orders table.

Create an XMLType table.

```
CREATE TABLE orders OF XMLType;
CREATE DIRECTORY xmldir AS path_to_folderContaining_XML_file;
INSERT INTO orders VALUES
(XMLType(BFILENAME ('XMLEDIR', 'purOrder.xml'),NLS_CHARSET_ID('AL32UTF8')));
```

Create table with an XMLType column.

```
CREATE TABLE xwarehouses (warehouse_id NUMBER, warehouse_spec XMLTYPE);
```

Create an XMLType View.

```
CREATE VIEW warehouse_view AS
SELECT VALUE(p) AS warehouse_xml FROM xwarehouses p;
```

Insert data into an XMLType column.

```
INSERT INTO xwarehouses
VALUES(100, '<?xml version="1.0"?>
<PO pono="1">
<PNAME>Po_1</PNAME>
<CUSTNAME>John</CUSTNAME>
<SHIPADDR>
<STREET>1033, Main Street</STREET>
<CITY>Sunnyvalue</CITY>
<STATE>CA</STATE>
</SHIPADDR></PO>')
```

Create an XML Search Index and query it with XQuery:

1. After the user gets all the privileges needed and set the right parameter in the Oracle Text schema.
2. Create Oracle Text section and preference.
3. Create the XML Search Index (regular index associated with the objects).

```
BEGIN
CTX_DDL.create_section_group('secgroup', 'PATH_SECTION_GROUP');
CTX_DDL.set_sec_grp_attr('secgroup', 'XML_ENABLE', 'T');
```

```

CTX_DDL.create_preference('pref', 'BASIC_STORAGE');
CTX_DDL.set_attribute('pref','D_TABLE_CLAUSE', 'TABLESPACE ts_name LOB(DOC) STORE AS
SECUREFILE(TABLESPACE ts_name COMPRESS MEDIUM CACHE)');
CTX_DDL.set_attribute('pref','I_TABLE_CLAUSE','TABLESPACE ts_name LOB(TOKEN_INFO)
STORE AS SECUREFILE(TABLESPACE ts_name NOCOMPRESS CACHE)');
END;
/

CREATE INDEX po_ctx_idx ON po_binxml(OBJECT_VALUE)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS('storage pref section group secgroup');

```

Query using the above index in XQuery. XQuery is W3C standard for generating, querying and updating XML, Natural query language for XML.

Content, very like SQL in the relational world, advanced form of XPath and XSLT (as read in the beginning of the topic, are for structured data and like regular Oracle Text indexes).

Search in the PATH “/PurchaseOrder/LineItems/LineItem/Description” for values containing “Big” and “Street” and the return their “Title” tag (only in the select).

```

SELECT XMLQuery('for $i in /PurchaseOrder/LineItems/LineItem/Description where $i[.
contains text "Big" ftand "Street"] return <Title>{$i}</Title>' PASSING OBJECT_VALUE
RETURNING CONTENT)
FROM po_binxml
WHERE XMLExists('/PurchaseOrder/LineItems/LineItem/Description [. contains text "Big"
ftand "Street"]'

```

XMLIndex with Structured Component is used for queries that project fixed structured islands of XML content, even if the surrounding data is relatively unstructured. A structured XMLIndex component organizes such islands in a relational format.

You must define the parts of XML data that you search in queries (applies to both XML schema-based and non-schema-based data).

Create an XMLIndex with structured component:

1. Create the base XMLIndex on po_binxml table. OBJECT_VALUE is the XML data stored in the table. All definitions of XML types and Objects are from the XDB schema in the database.
2. Use DBMS_XMLINDEX.registerParameter to add another structure to the index.
3. Create tables (po_idx_tab and po_index_lineitem) to store index data as structured data. Next to each table name there is the root of the PATH in the XML data (/PurchaseOrder and /LineItem). After that, each column is another PATH in this root. Note that in the first table (po_idx_tab) the last column is XMLType. It takes everything under this PATH and saves it in XML datatype.
4. Add the group of structure to the index.

```

CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE)
INDEXTYPE IS XDB.XMLIndex PARAMETERS ('PATH TABLE path_tab');
BEGIN
DBMS_XMLINDEX.registerParameter(
'myparam',
'ADD_GROUP GROUP po_item
XMLTable po_idx_tab ''/PurchaseOrder''
COLUMNS reference VARCHAR2(30) PATH ''Reference'',
requestor VARCHAR2(30) PATH ''Requestor'',
username VARCHAR2(30) PATH ''User''',

```

```

lineitem XMLType      PATH ''LineItems/LineItem'' VIRTUAL
XMLTable po_index_lineitem '/LineItem' PASSING lineitem
COLUMNS itemno      BINARY_DOUBLE PATH ''@ItemNumber'', 
description VARCHAR2(256) PATH ''Description'', 
partno      VARCHAR2(14)  PATH ''Part/@Id'', 
quantity    BINARY_DOUBLE PATH ''Part/@Quantity'', 
unitprice   BINARY_DOUBLE PATH ''Part/@UnitPrice''' );
END;
/
ALTER INDEX po_xmlindex_ix PARAMETERS ('PARAM myparam');

```

For more information, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/adxdb/indexes-for-XMLType-data.html#GUID-FE16BE28-035C-49EE-9683-6ED37876C3FC>

SQL/XML Functions

There are two main SQL/XML groups: SQL/XML publishing functions and SQL/XML query and update functions

SQL/XML Publishing Functions

SQL/XML publishing functions are SQL results generated from XML data (also called “SQL/XML generation functions”).

XMLQuery: Used in SELECT clauses to return the result as XMLType data (See the previous example for creating an XML search index)

XMLTable: Used in FROM clauses to get results using XQuery, and insert the results into a virtual table (can insert into existing database table). For Example:

1. Use XMLTable to generate virtual table from the xml value (OBJECT_VALUE).
2. Generate columns under the root (/PurchaseOrder).
3. One of the columns is XMLType and then another XMLTable call insert deeper into the XML data calling the virtual table (po) and create another virtual table.
4. When using @ the path is looking in the inner tag

```

SELECT po.reference, li.*
FROM po_binaryxml p,
XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
COLUMNS
reference VARCHAR2(30) PATH 'Reference',
lineitem XMLType      PATH 'LineItems/LineItem') po,
XMLTable('/LineItem' PASSING po.lineitem
COLUMNS
itemno      NUMBER(38)  PATH '@ItemNumber',
description VARCHAR2(256) PATH 'Description',
partno      VARCHAR2(14)  PATH 'Part/@Id',
quantity    NUMBER(12, 2) PATH 'Part/@Quantity',
unitprice   NUMBER(8, 4)  PATH 'Part/@UnitPrice') li;

```

XMLEXISTS: Used in WHERE clauses to check if an XQuery expression returns a non-empty query sequence. If it does, it returns TRUE. Otherwise, it returns FALSE. In the following example, the query searches the purchaseorder table for PurchaseOrders that where the “SpecialInstructions” tag is set to “Expedite”.

```
SELECT OBJECT_VALUE FROM purchaseorder
WHERE XMLEXISTS ('/PurchaseOrder[SpecialInstructions="Expedite"]'
PASSED OBJECT_VALUE);
```

XMLCast: Used in SELECT clauses to convert scalar values returned from XQuery to NUMBER, VARCHAR2, CHAR, CLOB, BLOB, REF, or XMLType. For example, after finding the objects that have SpecialInstructions set to Expedite, XMLCast returns the “Reference” in each item as VARCHAR2(100).

```
SELECT XMLCast(XMLQuery('/PurchaseOrder/Reference' PASSING OBJECT_VALUE
RETURNING CONTENT) AS VARCHAR2(100)) "REFERENCE"
FROM purchaseorder
WHERE XMLEXISTS ('/PurchaseOrder[SpecialInstructions="Expedite"]'
PASSED OBJECT_VALUE);
```

For more information, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/XMLELEMENT.html#GUID-DEA75423-00EA-4034-A246-4A774ADC988E>

SQL/XML Query and Update Functions

SQL/XML query and update functions are used to query and update XML content as part of regular SQL operations.

XMLQuery: See the example above.

UPDATE: Where in the SET clause there is XMLType instance, SQL functions or XML constructors that return an XML instance. In the following example, after finding the relevant item with XMLEXISTS in the set clause, the command sets the OBJECT_VALUE to a new file ('NEW-DAUSTIN-20021009123335811PDT.xml') located in the directory ('XMLEDIR').

```
UPDATE purchaseorder po
SET po.OBJECT_VALUE = XMLType(bfilename('XMLEDIR','NEW-DAUSTIN-
20021009123335811PDT.xml'), nls_charset_id('AL32UTF8'))
WHERE XMLEXISTS('$p/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]' PASSING
po.OBJECT_VALUE AS "p");
```

SQL and PL/SQL

SQL and PL/SQL is covered in the SQL and PL/SQL topics

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/XMLQUERY.html#GUID-9E8D3220-2CF5-4C63-BDC2-0526D57B9CDB>

PostgreSQL Usage

The data type xml in PostgreSQL can be used when creating tables, the main advantage to keep the xml data in xml type column and not in regular text column is the xml type check the input to alert if we try to insert wrong data format, in additional, there are support functions to perform type-safe operations on it.

XML can store well-formed “documents” as defined by XML standard or “content” fragments that are defined by the production XMLDecl, this means that content fragments can have more than one top-level element or character node.

IS DOCUMENT can be used to evaluate whether a particular xml value is a full document or only a content fragment.

Note: The xmlelement() and xpath() functions that may not work with non-ASCII data when the server encoding is not UTF-8.

Example

How to create xml data and insert it to the table.

The first insert is Document and the second is just content, the two types of the data can be inserted to the same column.

If insert wrong xml (missing tag etc..) the insert will fail with relevant error.

The query below will retrieve only the DOCUMENT RECORDS.

```
CREATE TABLE test (a xml);

insert into test values (XMLPARSE (DOCUMENT '<?xml version="1.0"?><Series><title>Simpsons</title><chapter>...</chapter></Series>'));

insert into test values (XMLPARSE (CONTENT 'note<tag>value</tag><tag>value</tag>'));

select * from test where a IS DOCUMENT;
```

Converting XML data to rows will be a new feature in PostgreSQL 10, this can be very helpful to read XML data using table equivalent:

```

CREATE TABLE xmldata_sample AS SELECT
xml $$

<ROWS>
<ROW id="1">
<EMP_ID>532</EMP_ID>
<EMP_NAME>John</EMP_NAME>
</ROW>
<ROW id="5">
<EMP_ID>234</EMP_ID>
<EMP_NAME>Carl</EMP_NAME>
<EMP_DEP>6</EMP_DEP>
<SALARY unit="dollars">10000</SALARY>
</ROW>
<ROW id="6">
<EMP_ID>123</EMP_ID>
<EMP_DEP>8</EMP_DEP>
<SALARY unit="dollars">5000</SALARY>
</ROW>
</ROWS>
$$ AS data;

SELECT xmldata.*,
  FROM xmldata_sample,
    XMLTABLE('//ROWS/ROW'
      PASSING data
      COLUMNS id int PATH '@id',
      ordinality FOR ORDINALITY,
      "EMP_NAME" text,
      "EMP_ID" text PATH 'EMP_ID',
      SALARY_USD float PATH 'SALARY[@unit = "dollars"]',
      MANAGER_NAME text PATH 'MANAGER_NAME' DEFAULT 'not specified');

id |ordinality |EMP_NAME |EMP_ID |salary_usd |manager_name
---|-----|-----|-----|-----|-----
1 |1 |John |532 | |not specified
5 |2 |Carl |234 |10000 |not specified
6 |3 | |123 |5000 |not specified

```

Summary

Description	PostgreSQL	Oracle
Create table with XML	CREATE TABLE test (a xml);	CREATE TABLE test OF XMLType; or CREATE TABLE test (doc XMLType);
Insert data into xml column	<pre>INSERT INTO test VALUES (XMLPARSE (DOCUMENT '<?xml version="1.0"?> <PO pono="1"> <PNAME>Po_1</PNAME> <CUSTNAME>John</CUSTNAME> <SHIPADDR> <STREET>1033, Main Street</STREET> <CITY>Sunnyvalue</CITY> <STATE>CA</STATE> </SHIPADDR> </PO>'));</pre>	<pre>INSERT INTO test VALUES ('<?xml version="1.0"?> <PO pono="1"> <PNAME>Po_1</PNAME> <CUSTNAME>John</CUSTNAME> <SHIPADDR> <STREET>1033, Main Street</STREET> <CITY>Sunnyvalue</CITY> <STATE>CA</STATE> </SHIPADDR> </PO>')</pre>
Create Index	<pre>/* It's a bit different index, here we index a specific path so the queries must be the same */ CREATE INDEX test_isbn ON test (((((xpath('/path/tag/text()', a))[1])::text)));</pre>	<pre>CREATE INDEX test_idx ON test (OBJECT_ VALUE) INDEXTYPE IS XDB.XMLIndex PARAMETERS ('PATH TABLE path_tab'); BEGIN0 DBMS_XMLINDEX.registerParameter('myparam', 'ADD_GROUP GROUP a_item XMLTable test_idx_tab "/Path" COLUMNS tag VARCHAR2(30) PATH "tag"); END; / ALTER INDEX test_idx PARAMETERS ('PARAM myparam');</pre>
Create Full-text Index	<pre>/* It's a bit different index, here we index a specific path so the queries must be the same */ CREATE INDEX my_funcidx ON test USING GIN (CAST(xpath('/PNAME/ text()', a) AS TEXT[]));</pre>	<pre>/*After preference and section created in Oracle Text*/ CREATE INDEX test_idx ON test (OBJECT_ VALUE) INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS('storage pref section group sec- group');</pre>
Query using XQuery	Not Supported	<pre>SELECT XMLQuery('for \$i in /PurchaseOrder- /LineItems/LineItem/Description where \$i[. con- tains text "Big"] return <Title>{\$i}</Title>' PASSING OBJECT_VALUE RETURNING CONTENT) FROM xml_tbl;</pre>
Query using XPath	SELECT xpath('//student/firstname/text()', a) FROM test	<pre>select sys.XMLType.extract (doc,'/student/firstname/text()') firstname from test;</pre>

Description	PostgreSQL	Oracle
Function to check if tag exists	SELECT XMLCast(XMLQuery ('/PurchaseOrder/Reference' PASSING OBJECT_VALUE RETURNING CONTENT) AS VARCHAR2(100)) "REFERENCE" FROM purchaseorder WHERE XMLExists('/PurchaseOrder[SpecialInstructions="Expedite"]' PASSING OBJECT_VALUE);	select cast (xpath('//book/title/text()', a) as text[]) as BookTitle from test where xmlexists('//book/title' PASSING by ref a);
Function to cast and return another data type(string)		
Validate schema using XSD	Not out-of-the-box but can create trigger before insert or delete and find tag with XPATH and try to cast the type of the value to know if it's OK, then if something wrong abort the insert or delete command	Supported

For additional information on PostgreSQL XML Types & Functions:

<https://www.postgresql.org/docs/13/static/datatype-xml.html>

<https://www.postgresql.org/docs/13/static/functions-xml.html>

Oracle Log Miner vs. PostgreSQL Logging Options

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	PostgreSQL does not support LogMiner, work-around available

Oracle Usage

Oracle Log Miner is a tool for querying the database Redo Logs and the Archived Redo Logs using an SQL interface. Using Log Miner, you can analyze the content of database “transaction logs” (online and archived redo logs) and gain historical insights on past database activity such as data modification by individual DML statements.

Examples

The following examples demonstrate how to use Log Miner to view DML statements executed on the employees table:

Find the current redo log file.

```
SELECT V$LOG.STATUS, MEMBER
FROM V$LOG, V$LOGFILE
WHERE V$LOG.GROUP# = V$LOGFILE.GROUP#
AND V$LOG.STATUS = 'CURRENT';

STATUS          MEMBER
-----
CURRENT        /u01/app/oracle/oradata/orcl/redo02.log
```

Use the DBMS_LOGMNR.ADD_LOGFILE procedure. Pass the file path as a parameter to the Log Miner API.

```
BEGIN
DBMS_LOGMNR.ADD_LOGFILE ('/u01/app/oracle/oradata/orcl/redo02.log');
END;
/

PL/SQL procedure successfully completed.
```

Start Log Miner using the DBMS_LOGMNR.START_LOGMNR procedure.

```
BEGIN
DBMS_LOGMNR.START_LOGMNR(options=>
dbms_logmnr.dict_from_online_catalog);
END;
/

PL/SQL procedure successfully completed.
```

Run a DML statement.

```
UPDATE HR.EMPLOYEES SET SALARY=SALARY+1000 WHERE EMPLOYEE_ID=116;
COMMIT;
```

Query the V\$LOGMNR_CONTENTS table to view the DML commands captured by the Log Miner.

```

SELECT TO_CHAR(TIMESTAMP, 'mm/dd/yy hh24:mi:ss') TIMESTAMP,
SEG_NAME, OPERATION, SQL_REDO, SQL_UNDO
FROM V$LOGMNR_CONTENTS
WHERE TABLE_NAME = 'EMPLOYEES'
AND OPERATION = 'UPDATE';

----- ----- -----
TIMESTAMP          SEG_NAME    OPERATION
10/09/17 06:43:44 EMPLOYEES   UPDATE

SQL_REDO           SQL_UNDO
----- -----
update "HR"."EMPLOYEES" set
" SALARY" = '3900' where " SALARY"
= '2900' and
ROWID = 'AAAViUAAEAAABVvAAQ';
update "HR"."EMPLOYEES" set
" SALARY" = '2900' where " SALARY"
= '3900' and
ROWID = 'AAAViUAAEAAABVvAAQ';

```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/util/oracle-logminer-utility.html#GUID-3417B738-374C-4EE3-B15C-3A66E01AE2B5>

PostgreSQL Usage

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

1. Navigate to the ["Parameter group" section](#) in the RDS Service of the AWS Console.
2. Click the name of the parameter to edit.
3. Click on the **Parameter group actions** drop-down list.

4. Click the **Edit** button.

Name	DB cluster parameter group	Parameter groups	Description
default.aurora-mysql5.7	aurora-mysql5.7	DB cluster parameter group	Default para
default.aurora-mysql5.7	aurora-mysql5.7	Parameter groups	Default clust
default.aurora-postgresql12	aurora-postgresql12	Parameter groups	Default para
default.aurora-postgresql12	aurora-postgresql12	DB cluster parameter group	Default clust
default.oracle-se2-19	oracle-se2-19	Parameter groups	Default para
non-default-param-group	aurora-postgresql12	Parameter groups	description

5. Set the following parameters:

- shared_preload_libraries = 'pg_stat_statements'
- pg_stat_statements.max = 10000
- pg_stat_statements.track = all

6. Change parameter values and click **Save changes**.

Name	Values	Allowed values
pg_stat_statements.max	10000	100-2147483647

Note: A database reboot may be required for the updated values to take effect.

1. Connect to the and run the following command:

```
CREATE EXTENSION PG_STAT_STATEMENTS;
```

2. Test the PG_STAT_STATEMENTS view to see captured database activity:

```
UPDATE EMPLOYEES
SET SALARY=SALARY+1000
WHERE EMPLOYEE_ID=116;
```

```

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_blk_hit   | 15
shared_blk_read  | 10
shared_blk_dirtied | 0
shared_blk_written | 0
local_blk_hit   | 0
local_blk_read  | 0
local_blk_dirtied | 0
local_blk_written | 0
temp_blk_read   | 0
temp_blk_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.

Examples

1. On the AWS Console, navigate to RDS > Parameter Groups.
2. Set the following parameters:

- log_statement = 'ALL'
- log_min_duration_statement = 1

For example:

The screenshot shows the AWS RDS Parameter Groups interface. At the top, it says 'RDS > Parameter groups > non-default-param-group'. Below that is the title 'non-default-param-group'. Underneath is a table titled 'Parameters' with three columns: 'Name', 'Values', and 'Allowed values'. A search bar at the top has 'log_sta' typed into it, with a red box labeled '1' over it. In the 'Values' column, there is a dropdown menu set to 'all', with a red box labeled '2' over it. The 'Allowed values' column contains 'none, ddl, mod, all'. At the bottom right of the table is a large orange 'Save changes' button, with a red box labeled '3' over it.

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:

```
2017-10-09 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=116
2017-10-09 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%'
```

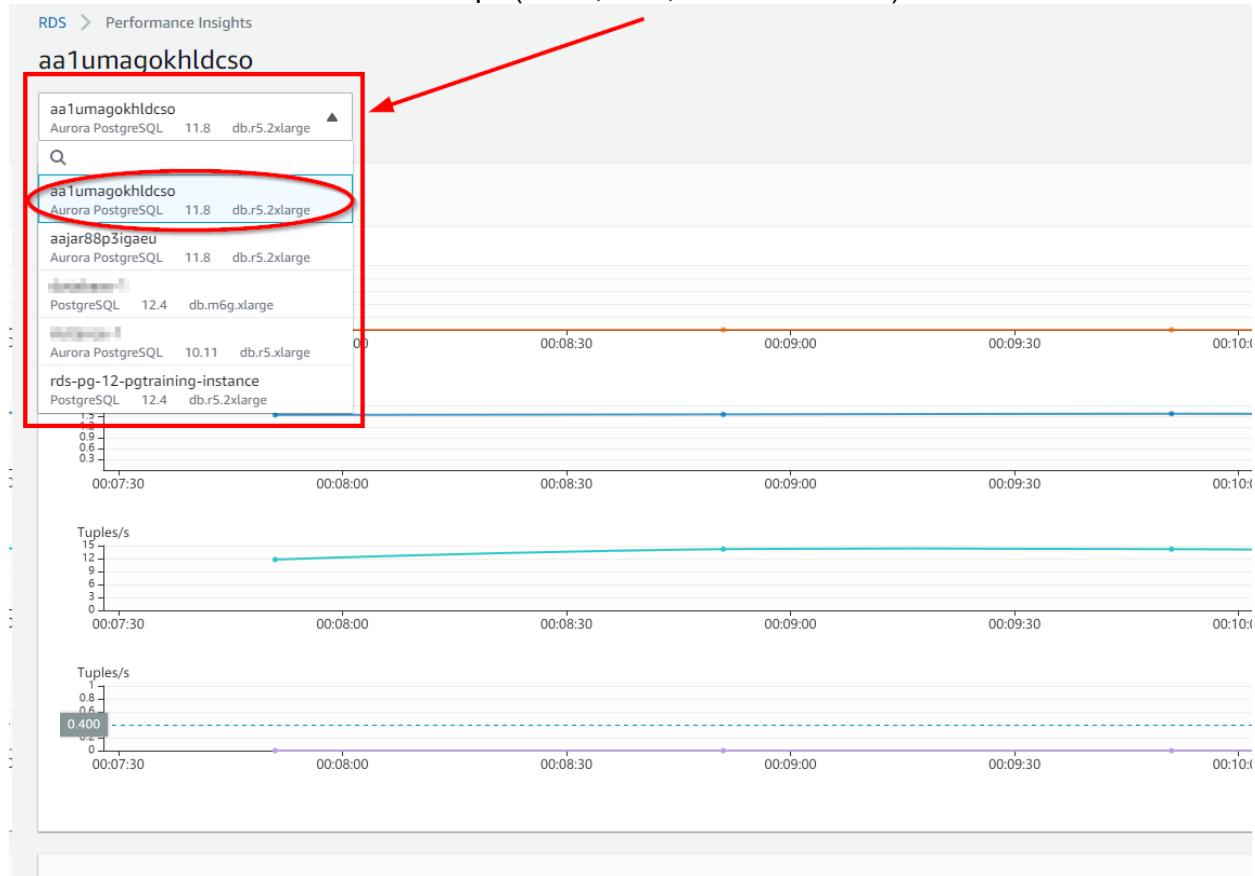
Amazon Aurora Performance Insights

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.

Examples

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.
6. On the AWS Management Console, navigate to RDS > Performance Insights.
7. Select the relevant instance to monitor.

8. Select the timeframe and monitor scope (Waits, SQL, Hosts and Users).



For more details, see:

- <https://www.postgresql.org/docs/13/static/runtime-config-logging.html>
- <https://www.postgresql.org/docs/13/static/pgstatstatements.html>
- http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_LogAccess.Concepts.PostgreSQL.html

High Availability and Disaster Recovery (HADR)

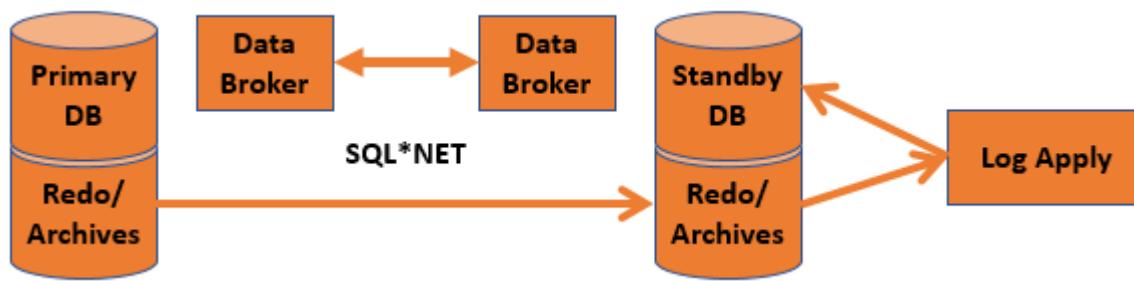
Oracle Active Data Guard vs. PostgreSQL Replicates

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
5 green/red cylinders	N/A	N/A	Distribute load/applications/ users across multiple instances

Oracle Usage

Oracle Active Data Guard (ADG) is a synced database architecture with a primary and standby databases. The difference between Data Guard and ADG is that ADG standby databases allow read access only.

The following diagram illustrates the ADG architecture:



- **Primary DB:** The main database open to read and write operations.
- **Redo/Archive:** The redo files and archives that store the redo entries for recovery operations.
- **Data Broker:** The data guard broker service is responsible for all failover and syncing operations.
- **Standby DB:** The secondary database that allows read operations only. This database remains in recovery mode until it is shut down or becomes the primary (failover or switchover).
- **Log Apply:** Runs all the redo log entries from the redo and archives files on the standby db.
- **Redo/Archive:** Contains the redo files and archives that are synced from the primary log and archive files.
- **Data Broker:** The Data Guard broker service is responsible for all failover and syncing operations.
- All components use SQL*NET protocol.

Special features:

- You can select "asynchronously" for best performance or "synchronously" for best data protection.
- You can temporarily convert a standby database to a snapshot database and allow read/write operations. When you are done running QA, testing, loads, or other operations, it can be switched back to standby.
- A sync gap can be specified between the primary and standby databases to account for human errors (for example creating 12 hours gap of sync).

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sbydb/creating-oracle-data-guard-physical-standby.html#GUID-B511FB6E-E3E7-436D-94B5-071C37550170>

PostgreSQL Usage

Aurora replicas can be used for scaling read operations and increasing availability like Oracle Active Data Guard, but with less configuration and administration. You can easily manage many replicas from the RDS console. Alternatively, you can use the AWS CLI for automation.

When creating Aurora PostgreSQL instances, there are two replication options:

- **Multi-AZ (Availability Zone):** Create a replicating instance in a different region.
- **Instance Read Replicas:** Create a replicating instance in the same region.

In "Instance options" there are two options for this scenario:

- Create Aurora Replica
- Create Cross Region Read Replica

The main differences between the two options are:

- Cross Region creates a new reader cluster in a different region. Use Cross Region for a higher level of Higher Availability and to keep the data closer to the end users.
- Cross Region has more lag between the two instances.
- Additional charges apply for transferring the data between the two regions.

DDL statements executed on the primary instance may interrupt database connections on the associated Aurora Replicas. If an Aurora Replica connection is actively using a database object such as a table, and that object is modified on the primary instance using a DDL statement, the Aurora Replica connection is interrupted.

Rebooting the primary instance of an Amazon Aurora database cluster also automatically reboots the Aurora Replicas for that database cluster.

Before creating a cross region replica, the binlog_format parameter must be enabled.

When using Multi-AZ, the primary database instance switches over automatically to the standby replica if any of the following conditions occur:

- The primary database instance fails.
- An Availability Zone outage.
- The database instance server type is changed.
- The operating system of the database instance is undergoing software patching.
- A manual failover of the database instance was initiated using reboot with fail-over.

Examples

The following walkthrough demonstrates how to create a replica/reader:

1. Navigate to the [RDS instance page](#).
2. Select the instance and click **Instance actions > Add reader**.

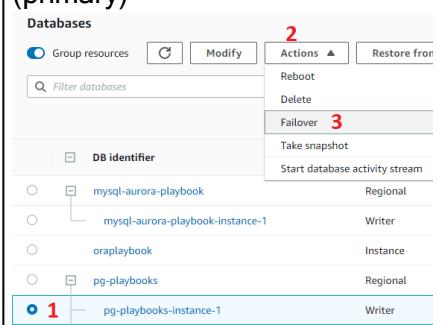
The screenshot shows the AWS RDS Databases page. In the left pane, there's a list of databases. One database, "pg-playbooks" (marked with a red number 1), is selected. In the top right corner, there's a "Actions" button with a dropdown menu. The "Add reader" option in this menu is highlighted with a red box and a red number 3. Other options in the menu include Stop, Start database activity stream, Delete, Upgrade now, Upgrade at next window, Create cross-Region read replica, Create clone, Promote, Restore to point in time, and Add replica auto scaling.

DB identifier	Role	Engine	Region & AZ	Size	Status
mysql-aurora-playbook	Regional	Aurora MySQL	eu-central-1	1 instance	(green)
mysql-aurora-playbook-instance-1	Writer	Aurora MySQL	eu-central-1a	db.t3.medium	(green)
oraplaybook	Instance	Oracle Standard Edition Two	eu-central-1a	db.t3.medium	(green)
pg-playbooks 1	Regional	Aurora PostgreSQL	eu-central-1	1 instance	(green)
pg-playbooks-instance-1	Writer	Aurora PostgreSQL	eu-central-1a	db.t3.medium	(green)

3. On the next page, enter all required details and click **Create**.

After the replica is created, you can execute read and write operations on the primary instance and read-only operations on the replica.

Compare between Oracle Active Data Guard and Aurora PostgreSQL Replicates

Description	Oracle Active Data Guard	Aurora PostgreSQL Replicates
How to switch over	ALTER DATABASE SWITCHOVER TO DBREP VERIFY;	Note that you can't choose to which instance to failover, the instance with the higher priority will become a writer (primary) 
Define automatic failover	EDIT DATABASE db1 SET PROPERTY FASTSTARTFAILOVERTARGET='db1rep'; EDIT DATABASE db1rep SET PROPERTY FASTSTARTFAILOVERTARGET='db1'; ENABLE FAST_START FAILOVER;	Use Multi-AZ on instance creation or by modifying existing instance
asynchronously or synchronously replication	Change to synchronous ===== ALTER SYSTEM SET LOG_ARCHIVE_DEST_2='SERVICE=db1rep AFFIRM SYNC VALID_FOR=(ONLINE_LOGFILES,PRIMARY_ROLE) DB_UNIQUE_NAME=db1rep'; ALTER DATABASE SET STANDBY DATABASE TO MAXIMIZE AVAILABILITY; Change to asynchronous ===== ALTER SYSTEM SET LOG_ARCHIVE_DEST_2='SERVICE=db1rep NOAFFIRM ASYNC VALID_FOR=(ONLINE_LOGFILES,PRIMARY_ROLE) DB_UNIQUE_NAME=db1rep'; ALTER DATABASE SET STANDBY DATABASE TO MAXIMIZE	Not Supported. Only asynchronously replication is in use

Description	Oracle Active Data Guard	Aurora PostgreSQL Replicates
Open standby to read/write and continue syncing afterwards	PERFORMANCE; CONVERT DATABASE db1rep TO SNAPSHOT STANDBY; CONVERT DATABASE db1rep TO PHYSICAL STANDBY;	Not Supported but you can: Restore your database from snapshot (Appendix 1) Run your QA, testing or other operations on the restored instance After you finished dropped the restored instance
Create gaped replication	Create 5 minutes delay ===== ALTER DATABASE RECOVER MANAGED STANDBY DATABASE CANCEL; ALTER DATABASE RECOVER MANAGED STANDBY DATABASE DELAY 5 DISCONNECT FROM SESSION; Returns for no delay ALTER DATABASE RECOVER MANAGED STANDBY DATABASE CANCEL; ALTER DATABASE RECOVER MANAGED STANDBY DATABASE NODELAY DISCONNECT FROM SESSION;	Not Supported

For more details, see:

- <http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Replication.html>
- <http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html>

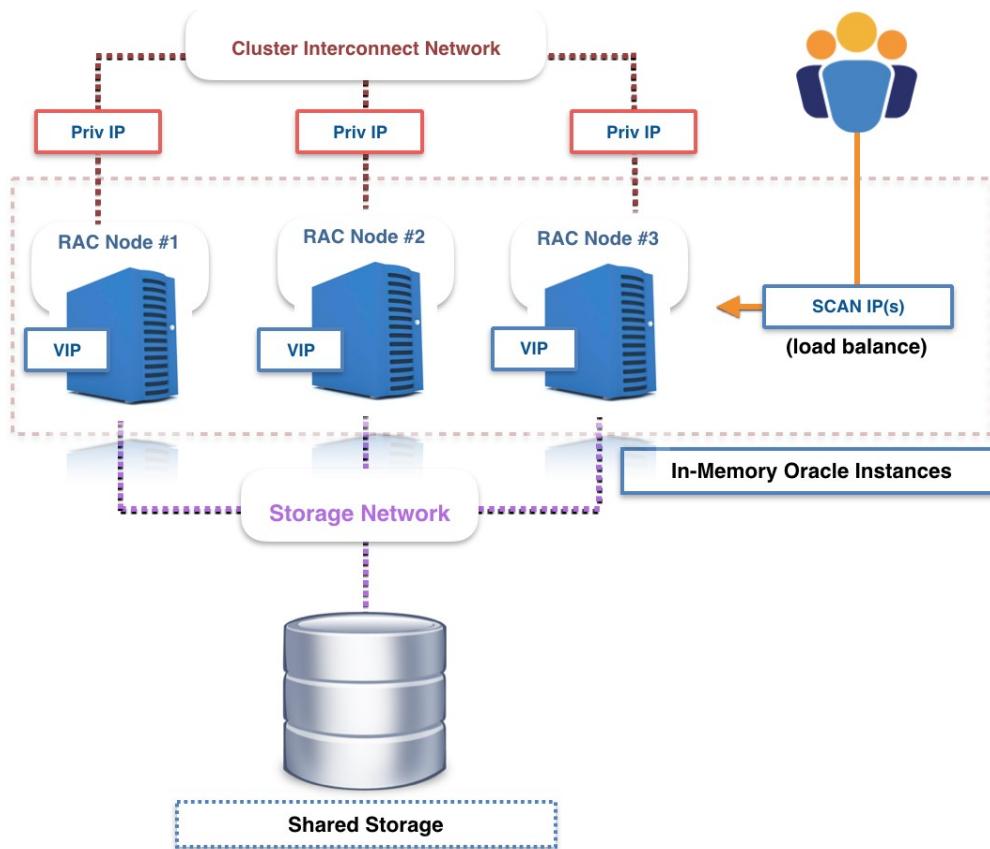
Oracle Real Application Clusters (RAC) vs. PostgreSQL Aurora Architecture

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Distribute load/applications/ users across multiple instances

Oracle Usage

Oracle Real Application Clusters (RAC) is one of the most advanced and capable technologies providing highly available and scalable relational databases. It allows multiple Oracle instances to access a single database. Applications can access the database through the multiple instances in Active-Active mode.

The following diagram illustrates the Oracle RAC architecture:



Oracle RAC requires network configuration of SCAN IPs, VIP IPs, interconnect, and other items. As a best practice, all servers should run the same versions of Oracle software.

Because of the shared nature of the RAC cluster architecture—specifically, having all nodes write to a single set of database data files on disk—the following two special coordination mechanisms ensure Oracle database objects and data maintain ACID compliance:

- **GCS (Global Cache Services):** Tracks the location and status of the database data blocks and helps guarantee data integrity for global access across all cluster nodes.
- **GES (Global Enqueue Services):** Performs concurrency control across all cluster nodes including cache locks and transactions.

These services, which run as background processes on each cluster node, are essential for serializing access to shared data structures in an Oracle database.

Shared storage is another essential component in the Oracle RAC architecture. All cluster nodes read and write data to the same physical database files stored on a disk accessible by all nodes. Most customers rely on high-end storage hardware to provide the shared storage capabilities required for RAC.

In addition, Oracle provides its own software-based storage/disk management mechanism called Automatic Storage Management (ASM). ASM is implemented as a set of special background processes that run on all cluster nodes and allow for easy management of the database storage layer.

Performance and Scale-Out with Oracle RAC

You can add new nodes to an existing RAC cluster without downtime. Adding more nodes increases the level of high availability and enhances performance.

Although you can scale read performance easily by adding more cluster nodes, scaling write performance is more complicated. Technically, Oracle RAC can scale writes and reads together when adding new nodes to the cluster, but attempts from multiple sessions to modify rows that reside in the same physical Oracle block (the lowest level of logical I/O performed by the database) can cause write overhead for the requested block and impact write performance.

Concurrency is another reason why RAC implements a “smart mastering” mechanism that attempts to reduce write-concurrency overhead. The “smart mastering” mechanism enables the database to determine which service causes which rows to be read into the buffer cache and master the data blocks only on those nodes where the service is active. Scaling writes in RAC isn’t as straightforward as scaling reads.

With the limitations for pure write scale-out, many Oracle RAC customers choose to split their RAC clusters into multiple “services,” which are logical groupings of nodes in the same RAC cluster. By using services, you can use Oracle RAC to perform direct writes to specific cluster nodes. This is usually done in one of two ways:

- Splitting writes from different individual “modules” in the application (that is, groups of independent tables) to different nodes in the cluster. This approach is also known as “application partitioning” (not to be confused with database table partitions).
- In extremely non-optimized workloads with high concurrency, directing all writes to a single RAC node and load-balancing only the reads.

In summary, Oracle Real Application Clusters provides two major benefits:

- Multiple database nodes within a single RAC cluster provide increased high availability. No single point of failure exists from the database servers themselves. However, the shared storage requires storage-based high availability or disaster recovery solutions.
- Multiple cluster database nodes enable scaling-out query performance across multiple servers.

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/racad/index.html>

PostgreSQL Usage

Aurora extends the “vanilla” versions of PostgreSQL in two major ways:

- Adds enhancements to the PostgreSQL database kernel itself to improve performance (concurrency, locking, multi-threading, etc.).
- Uses the capabilities of the AWS ecosystem for greater high availability, disaster recovery, and backup/recovery functionality.

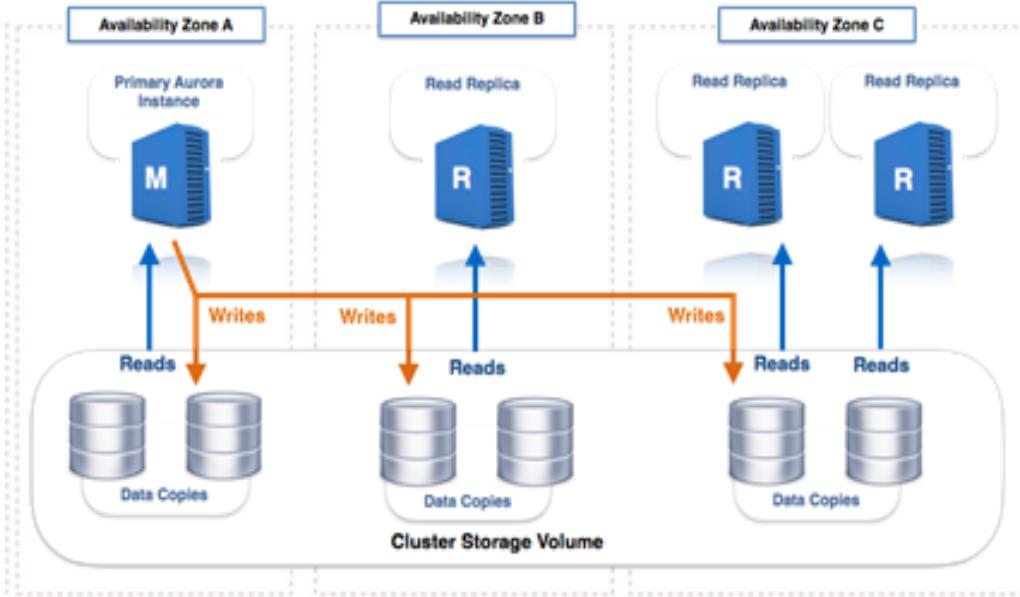
Comparing the Amazon Aurora architecture to Oracle RAC, there are major differences in how Amazon implements scalability and increased high availability. These differences are due mainly to the existing capabilities of PostgreSQL and the strengths the AWS backend provides in terms of networking and storage.

Instead of having multiple read/write cluster nodes access a shared disk, an Aurora cluster has a single primary node (“master”) that is open for reads and writes and a set of replica nodes that are open for reads with automatic promotion to primary (“master”) in case of failures. While Oracle RAC uses a set of background processes to coordinate writes across all cluster nodes, the Amazon Aurora Master writes a constant redo stream to six storage nodes distributed across three Availability Zones within an AWS Region. The only writes that cross the network are redo log records (not pages).

Each Aurora cluster can have one or more instances serving different purposes:

- At any given time, a single instance functions as the primary (“master”) that handles both writes and reads from your applications.
- Up to 15 read replicas can be created in addition to the primary (“master”), which are used for two purposes:
 - **Performance and Read Scalability:** Replicas can be used as read-only nodes for queries and report workloads.
 - **High Availability:** Replicas can be used as failover nodes in the event the master fails. Each read replica can be located in one of the three Availability Zones hosting the Aurora cluster. A single Availability Zone can host more than one read replica.

The following diagram illustrates a high-level Aurora architecture with four cluster nodes: one primary (“master”) and three read replicas. The primary node is located in Availability Zone A, the first read replica in Availability Zone B, and the second and third read replicas in Availability Zone C.



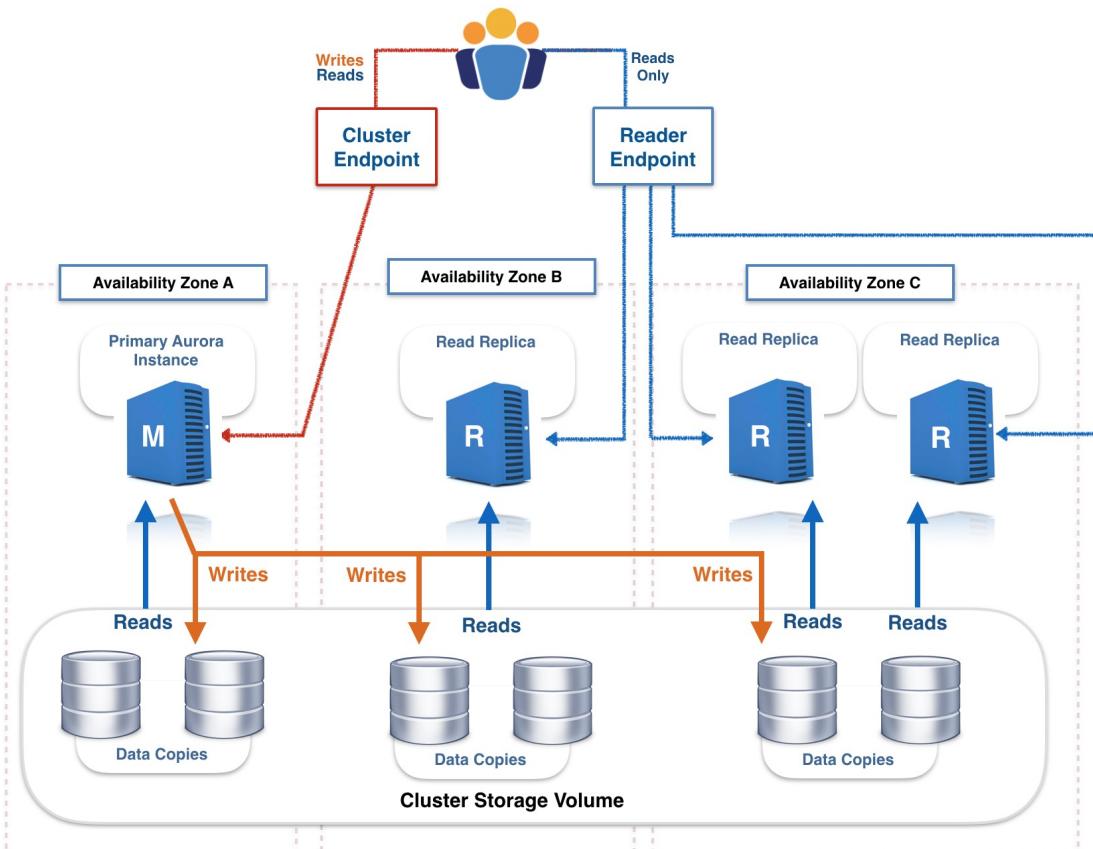
An Aurora Storage volume is made up of 10 GB segments of data with six copies spread across three Availability Zones. Each Amazon Aurora read replica shares the same underlying volume as the master instance. Updates made by the master are visible to all read replicas through a combination of reading from the shared Aurora storage volume and applying log updates in-memory when received from the primary instance after a master failure. Promotion of a read replica to master usually occurs in less than 30 seconds with no data loss.

For a write to be considered durable in Aurora, the primary instance (“master”) sends a redo stream to six storage nodes – two in each availability zone for the storage volume – and waits until four of the six nodes have responded. No database pages are ever written from the database tier to the storage tier. The Aurora Storage volume asynchronously applies redo records to generate database pages in the background or on demand. Aurora hides the underlying complexity.

High availability and Scale-Out in Aurora

Aurora provides two endpoints for cluster access. These endpoints provide both high availability capabilities and scale-out read processing for connecting applications.

- **Cluster Endpoint:** Connects to the current primary instance for the Aurora cluster. You can perform both read and write operations using the cluster endpoint. If the current primary instance fails, Aurora automatically fails over to a new primary instance. During a failover, the database cluster continues to serve connection requests to the cluster endpoint from the new primary instance with minimal interruption of service.
- **Reader Endpoint:** Provides load-balancing capabilities (round-robin) across the replicas allowing applications to scale-out reads across the Aurora cluster. Using the Reader Endpoint provides better use of the resources available in the cluster. The reader endpoint also enhances high availability. If an AWS Availability Zone fails, the application’s use of the reader endpoint continues to send read traffic to the other replicas with minimal disruption.

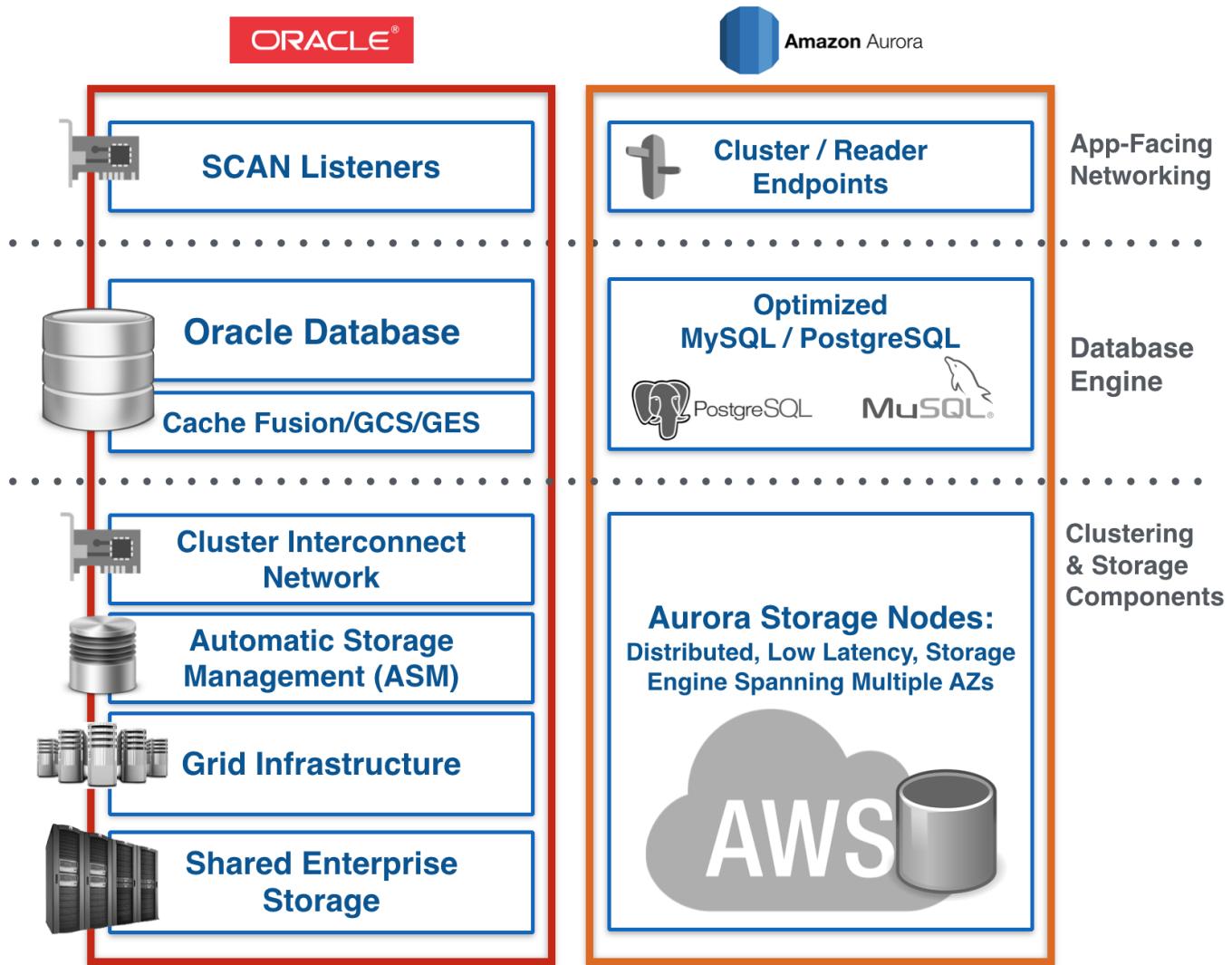


While Amazon Aurora focuses on the scale-out of reads and Oracle RAC can scale-out both reads and writes, most OLTP applications are usually not limited by write scalability. Many Oracle RAC customers use RAC first for high availability and second to scale-out their reads. You can write to any node in an Oracle RAC cluster, but this capability is often a functional benefit for the application versus a method for achieving unlimited scalability for writes.

Summary

- Multiple cluster database nodes provide increased high availability. There is no single point of failure from the database servers. In addition, since an Aurora cluster can be distributed across three availability zones, there is a large benefit for high availability and durability of the database. These types of “stretch” database clusters are usually uncommon with other database architectures.
- AWS managed storage nodes also provide high availability for the storage tier. A zero-data loss architecture is employed in the event a master node fails and a replica node is promoted to the new master. This failover can usually be performed in under 30 seconds.
- Multiple cluster database nodes enable scaling-out query read performance across multiple servers.
- Greatly reduced operational overhead using a cloud solution and reduced total cost of ownership by using AWS and open source database engines.
- Automatic management of storage. No need to pre-provision storage for a database. Storage is automatically added as needed, and you only pay for one copy of your data.
- With Amazon Aurora, you can easily scale-out your reads (and scale-up your writes) which fits perfectly into the workload characteristics of many, if not most, OLTP applications. Scaling out reads usually provides the most tangible performance benefit.

When comparing Oracle RAC and Amazon Aurora side by side, you can see the architectural differences between the two database technologies. Both provide high availability and scalability, but with different architectures.



Overall, Amazon Aurora introduces a simplified solution that can function as an Oracle RAC alternative for many typical OLTP applications that need high performance writes, scalable reads, and very high availability with lower operational overhead.

Feature	Oracle RAC	Amazon Aurora
Storage	Usually enterprise-grade storage + ASM	Aurora Storage Nodes: Distributed, Low Latency, Storage Engine Spanning Multiple AZs
Cluster type	Active/Active · All nodes open for R/W	Active/Active : Primary node open for R/W, Replica nodes open for reads
Cluster virtual IPs	R/W load balancing: SCAN IP	R/W: Cluster endpoint + Read load balancing: Reader endpoint
Internode coordination	Cache-fusion + GCS + GES	---

Feature	Oracle RAC	Amazon Aurora
Internode private network	Interconnect	---
Transaction (write) TTR from node failure	Typically 0-30 seconds	Typically < 30 Seconds
Application (Read) TTR from node failure	Immediate	Immediate
Max number of cluster nodes	Theoretical maximum is 100, but smaller clusters (2 - 10 nodes) are far more common	15
Provides built-in read scaling	Yes	Yes
Provides built-in write scaling	Yes (Under certain scenarios, write performance can be limited and affect scale-out capabilities. For example, when multiple sessions attempt to modify rows contained in the same database block(s).)	No
Data loss in case of node failure	No data loss	No data loss
Replication latency	---	Milliseconds
Operational complexity	Requires database, IT, network, and storage expertise	Provided as a cloud-solution
Scale-up nodes	Difficult with physical hardware, usually requires to replace servers	Easy using the AWS UI/CLI
Scale-out cluster	Provision, deploy, and configure new servers, unless you pre-allocate a pool of idle servers to scale-out on	Easy using the AWS UI/CLI

For more details, see <https://aws.amazon.com/blogs/database/amazon-aurora-as-an-alternative-to-oracle-rac/>

Oracle Traffic Director vs. Amazon RDS Proxy for Amazon Aurora PostgreSQL

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	Some features may be replaced by AWS RDS Proxy

Oracle Usage

Starting with Oracle 18c Oracle Connection Manager can be configured to run in Traffic Director mode. This mode introduces multiple features that help with High Availability, scalability, load balancing, zero downtime and security. Oracle Traffic Director is fast and reliable load-balancing solution. By enabling it for Oracle Connection Manager users can now get following features:

- Increased scalability through usage of transparent connection load-balancing
- Essential high availability feature of zero downtime that includes support for
 - planned database maintenance
 - pluggable database relocation
 - unplanned database outages for read-mostly workloads
- High availability of Connection Manager (CMAN) which avoids single point of failure
- Various security features, such as
 - database proxy
 - firewall
 - tenant isolation in multi-tenant environment
 - DDOS protection
 - Database traffic secure tunneling

Further reading: <https://docs.oracle.com/en/database/oracle/oracle-database/18/netag/configuring-oracle-connection-manager.html#GUID-3917FC5D-4B23-4752-85BA-39A88C4D13F8>

PostgreSQL Usage

Oracle Traffic Director mode for Connection Manager can be potentially replaced by Amazon RDS Proxy for migration to Aurora PostgreSQL

RDS Proxy simplifies connection management for Amazon RDS DB instances and clusters. It handles the network traffic between the client application and the database in an active way first by understanding the database protocol. Then RDS Proxy adjusts its behavior based on the SQL operations from user application and the result sets from the database.

RDS Proxy also reduces the memory and CPU overhead for the database connection management. The database needs less memory and CPU resources when applications open many simultaneous connections.

RDS Proxy also doesn't require applications to close and reopen connections that stay idle for a long time. Similarly, it requires less application logic to reestablish connections in case of a database problem.

The infrastructure for RDS Proxy is highly available and deployed over multiple Availability Zones (AZs). The computation, memory, and storage for RDS Proxy are independent of RDS DB instances and Aurora DB clusters. This separation helps lower overhead on database servers, so that they can devote their resources to serving database workloads. The RDS Proxy compute resources are serverless, automatically scaling based on your database workload.

This topic has been covered in the following topic in this book: [Amazon RDS Proxy](#)

For more information see [Managing Connections with Amazon RDS Proxy](#)

Oracle Data Pump vs. PostgreSQL pg_dump & pg_restore

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Non-compatible tool

Oracle Usage

Oracle Data Pump is a utility for exporting/importing data from/to an Oracle database. It can be used to copy an entire database, entire schemas, or specific objects in a schema. Oracle Data Pump is commonly used as a part of a 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 (without using the `sqlfile` parameter during export), the “dumpfile” generated by Oracle Data Pump is binary (it cannot be opened using a text editor).

Oracle Data Pump Supports:

- **Export 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 with metadata only. Exports can be performed for specific timestamps or Oracle SCNs to ensure cross-object consistency.
- **Import Data to an Oracle database:** The Data Pump IMPDP command imports objects and data from a specific dump file created with the EXPDP command. The IMPDP command can filter on import (for example, 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 file system paths that were pre-configured in the Oracle database as “directories”. During export/import, users must specify the logical “directory” name where the dump file should be created; not the actual file system path.

Examples

Use EXPDP to export the HR schema.

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

* The command contains the credentials to run Data Pump, the logical Oracle directory name for the dump file location (which maps in the database to a physical file system location), the schema name to export, the dump file name, and log file name.

Use IMPDP to import the HR a 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, the logical Oracle directory for where the export dumpfile is located, the dump file name, the schema to export, the name for the dump file, the log file name, and the REMAP_SCHEMA parameter.

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/sutil/oracle-data-pump.html#GUID-501A9908-BCC5-434C-8853-9A6096766B5A>

PostgreSQL Usage

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.

Since PostgreSQL 10, these capabilities were added:

- A schema can be excluded in pg_dump/pg_restore commands
- Can create dumps with no blobs
- Allow to run pg_dumpall by non-superusers, using the --no-role-passwords option
- Create additional integrity option to ensure that the data is stored in disk using fsync() method

Since PostgreSQL 11 following capabilities were added:

- pg_dump/pg_restore now exports / imports relationships between extensions and database objects established with ALTER ... DEPENDS ON EXTENSION, which allows these objects to be dropped when extension is dropped with CASCADE option.

Note:

- 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.
- PostgreSQL dump files can be both plain-text and custom format files. .

Another option to export and import data from PostgreSQL database is to use COPY TO / COPY FROM commands. Starting with PostgreSQL 12 the COPY FROM command, that can be used to load data into DB, has support for filtering incoming rows with WHERE condition

```
CREATE TABLE tst_copy(v TEXT);
COPY tst_copy FROM '/home/postgres/file.csv' WITH (FORMAT CSV) WHERE v LIKE '%apple%';
```

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 "+%Y-%m-%d-%H-%M-%S")
```

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 dbname_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 "+%Y-
%m-%d-%H-%M-%S")
```

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

Download the output file from the 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:

```
CREATE DATABASE mydb_copy TEMPLATE mydb;
```

Summary

Description	Oracle Data Pump	PostgreSQL Dump
Export data to a local file	expdp system/**** schemas=hr dump-file=hr.dmp logfile=hr.log	pg_dump -F c -h host-name.rds.amazonaws.com -U username -d hr -p 5432 > c:\Export\hr.dmp
Export data to a remote file	Create Oracle directory on remote storage mount or NFS directory called EXP_DIR Use export command: expdp system/**** schemas=hr directory=EXP_DIR dumpfile=hr.dmp logfile=hr.log	Export: pg_dump -F c -h host-name.rds.amazonaws.com -U username -d hr -p 5432 > c:\Export\hr.dmp <u>Upload to S3-</u> aws s3 cp c:\Export\hr.dmp s3://my-bucket/backup-\$(date "+%Y-%m-%d-%H-%M-%S")
Import data to a new database with a new name	impdp system/**** schemas=hr dump-file=hr.dmp logfile=hr.log REMAP_SCHEMA=hr:hr_copy TRANSFORM-M=OID:N	pg_restore -h hostname.rds.amazonaws.com -U hr -d hr_restore -p 5432 c:\Export\hr.dmp
Exclude schemas	expdp system/**** FULL=Y directory=EXP_DIR dumpfile=hr.dmp logfile=hr.log exclude=SCHEMA:"HR"	pg_dump -F c -h host-name.rds.amazonaws.com -U username -d hr -p 5432 -N 'log_schema' c:\Export\hr_nolog.dmp

For more details, see:

<https://www.postgresql.org/docs/13/static/backup-dump.html>

<https://www.postgresql.org/docs/13/static/app-pgrestore.html>

Oracle Flashback Database vs. PostgreSQL Amazon Aurora Snapshots

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Storage level backup managed by Amazon RDS

Oracle Usage

Oracle Flashback Database is a special mechanism built into Oracle databases that helps protect against human errors by providing 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 changes applied to a database and to data. Essentially, it stores previous versions of database modifications in the configured database “Fast Recovery Area”.

When using Oracle flashback database, you can choose to restore an entire database to either a user-created restore point, a timestamp value, or to a specific System Change Number (SCN).

Examples

Create a database restore point to which you can flashback a database.

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

Flashback a database to a previously created restore point.

```
shutdown immediate;
startup mount;
flashback database to restore point before_update;
```

Flashback a database to a specific time.

```
shutdown immediate;
startup mount;
FLASHBACK DATABASE TO TIME "TO_DATE('01/01/2017','MM/DD/YY')";
```

For more details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/rcmrf/FLASHBACK-DATABASE.html#GUID-584AC79A-40C5-45CA-8C63-DED3BE3A4511>

PostgreSQL Usage

Snapshots are the primary backup mechanism for Amazon Aurora databases. They are extremely fast and non-intrusive. You can take snapshots using the Amazon RDS Management Console or 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.

Amazon Aurora provides the following types of backups:

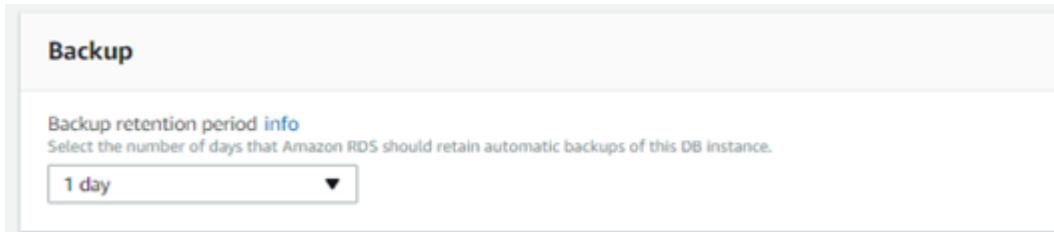
- **Automated Backups:** Always enabled on Amazon Aurora. They do not impact database performance.
- **Manual Backups:** You can create a snapshot at any time. There is no performance impact when taking snapshots of an Aurora database. Restoring data from snapshots requires creation of a new instance. Up to 100 manual snapshots are supported per database.

Examples

Enable Automatic Backups

Follow the steps below to enable Aurora automatic backups and configure the backup retention window as part of the database creation process. This process is equivalent to setting the Oracle RMAN backup retention policy using the “configure retention policy to recovery window of X days” command.

1. Navigate to the [RDS Instance Page](#).
2. If you are launching a new instance or modifying an existing instance, select the backup retention in days.



The following table identifies the default automatic backup time for each region:

Region	Default Backup Window
US West (Oregon)	06:00-14:00 UTC
US West (N. California)	06:00-14:00 UTC
US East (Ohio)	03:00-11:00 UTC
US East (N. Virginia)	03:00-11:00 UTC
Asia Pacific (Mumbai)	16:30-00:30 UTC
Asia Pacific (Seoul)	13:00-21:00 UTC
Asia Pacific (Singapore)	14:00-22:00 UTC
Asia Pacific (Sydney)	12:00-20:00 UTC
Asia Pacific (Tokyo)	13:00-21:00 UTC
Canada (Central)	06:29-14:29 UTC
EU (Frankfurt)	20:00-04:00 UTC
EU (Ireland)	22:00-06:00 UTC
EU (London)	06:00-14:00 UTC
South America (São Paulo)	23:00-07:00 UTC
AWS GovCloud (US)	03:00-11:00 UTC

Perform a Manual Backup

Follow the steps below to perform a manual snapshot backup of an Aurora database. This process is equivalent to creating a full Oracle RMAN backup (“BACKUP DATABASE PLUS ARCHIVELOG”).

1. Navigate to the [RDS Instance Page](#).
2. Select an Aurora PostgreSQL instance, click **Instance actions** and select **Take Snapshot**.

Databases

Group resources C Modify Actions ▲ Restore from S3 Create database

Filter databases

DB identifier

- mysql-aurora-playbook
 - mysql-aurora-playbook-instance-1 Writer Aurora MySQL eu-central-1
- oraplaybook Instance Oracle Standard Edition Two eu-central-1
- pg-playbooks Regional Aurora PostgreSQL eu-central-1
- pg-playbooks-instance-1 Writer Aurora PostgreSQL eu-central-1
- pg-playbooks-reader Reader Aurora PostgreSQL eu-central-1

Reboot Delete Failover Engine Region & AZ

Take snapshot Start database activity stream

Restore a Database From a Snapshot

Follow the steps below to restore an Aurora database from a snapshot. This process is similar to the Oracle RMAN commands RESTORE DATABASE and RECOVER DATABASE. However, instead of running in place, restoring an Aurora database creates a new cluster.

1. Navigate to the [RDS Instance Page](#).
2. Select the snapshot to restore, click **Actions** on the context menu, and select **Restore snapshot**. This action creates a new instance.

RDS > Snapshots

Snapshots (60)

Owned by Me

Actions ▾

1 rds:mysql57test-2018-05-08-04-21

2 rds:mysql57test-2018-05-08-04-21

Snapshot Creation Time Status Progress

Snapshot	Snapshot Creation Time	Status	Progress
rds:mysql57test-2018-05-08-04-21	Wed May 09 07:21:17 GMT+300 2018	available	Completed
rds:mysql57test-2018-05-08-04-21	Tue May 08 07:21:26 GMT+300 2018	available	Completed
rds:mysql57test-2018-05-07-04-21	Mon May 07 07:21:17 GMT+300 2018	available	Completed
rds:mysql57test-2018-05-06-04-21	Sun May 06 07:21:20 GMT+300 2018	available	Completed

Actions ▾

Restore Snapshot

Copy Snapshot

Share Snapshot

Migrate Snapshot

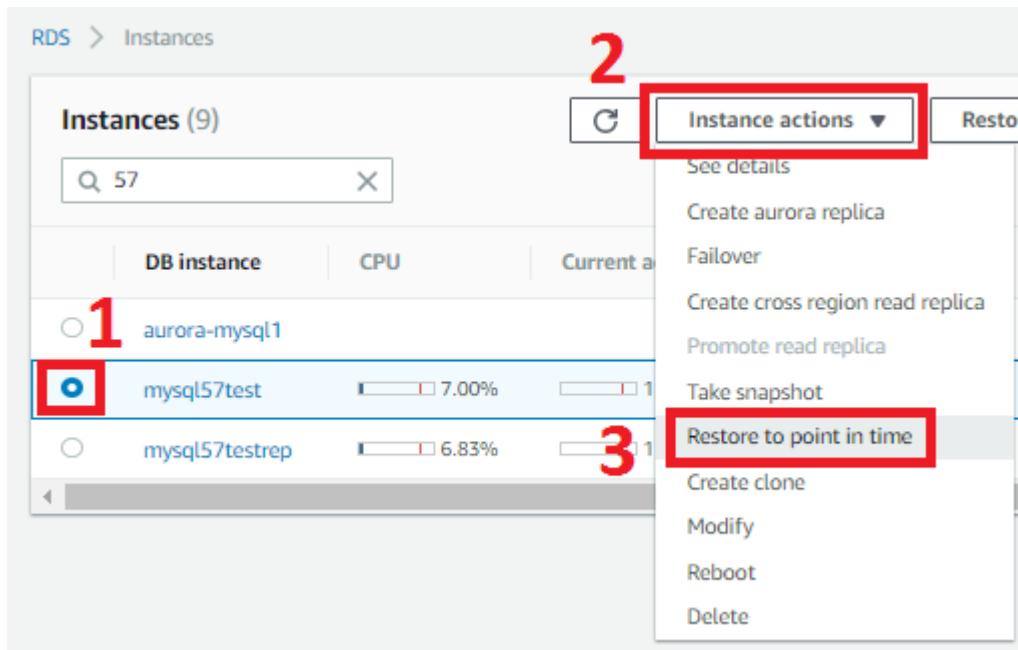
Delete Snapshot

- The web page displays a wizard for creating a new Aurora database instance from the selected snapshot. Enter the required configuration options and click **Restore DB Instance**.

Restore a Database to a Point in Time

Follow the steps below to restore an Aurora PostgreSQL database backup to a specific point in time. This process is similar to running the Oracle RMAN command `SET UNTIL TIME "TO_DATE('XXX')"` before running `RESTORE DATABASE` and `RECOVER DATABASE`.

- Navigate to the [RDS Instance Page](#).
- Select an Aurora instance, click **Instance actions**, and select **Restore to point in time**.



- This process launches 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.

AWS CLI Backup and Restore Operations

In addition to using the AWS web console to backup and restore an Aurora instance snapshot, you can also use the AWS CLI to perform the same actions. The CLI is especially useful for migrating existing automated Oracle RMAN scripts to an AWS environment. The following list highlights some CLI operations:

- Use `describe-db-cluster-snapshots` to view all current Aurora PostgreSQL snapshots.
- Use `create-db-cluster-snapshot` to create a snapshot ("Restore Point").
- Use `restore-db-cluster-from-snapshot` to restore a new cluster from an existing database snapshot.
- 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 new-instance-nodeA --db-instance-class db.r4.large
```

- 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 clusternam	restore --source-db-cluster-identifier clusternam --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 clusternamRestore --db-instance-iden-tifier newinstance-nodeA --db-instance-class db.r4.large
```

Summary

Description	Oracle	Amazon Aurora
Create a “restore point”	CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE;	aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name
Configure flashback “retention period”	ALTER SYSTEM SET db_flashback_retention_target=2880;	Configure the “Backup retention window” setting using the AWS management console or AWS CLI.
Flashback database to a previous “restore point”	shutdown immediate; startup mount; flashback database to restore point before_update;	<p>1. Create new cluster from a snapshot:</p> <pre>aws rds restore-db-cluster-from-snapshot --db-cluster-iden-tifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql</pre> <p>2. Add new instance to the cluster:</p> <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-iden-tifier clusternamRestore --db-instance-identifier new-instance-nodeA --db-instance-class db.r4.large</pre>
Flashback database to a previous point in time	shutdown immediate; startup mount; FLASHBACK DATABASE TO TIME "TO_DATE ('01/01/2017','MM/DD/YY')";	<p>1. Create a new cluster from a snapshot and provide a specific point in time:</p> <pre>aws rds restore-db-cluster-to-point-in-time --db-cluster-iden-tifier clusternamRestore --source-db-cluster-identifier clusternam --restore-to-time 2017-09-19T23:45:00.000Z</pre> <p>2. Add a new instance to the cluster:</p> <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-iden-tifier clusternamRestore --db-instance-identifier new-instance-nodeA --db-instance-class db.r4.large</pre>

For more details, see:

- <http://docs.aws.amazon.com/cli/latest/reference/rds/index.html#cli-aws-rds>
- http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PIT.html
- http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_RestoreFromSnapshot.html

Oracle Flashback Table vs. PostgreSQL Amazon Aurora Snapshots (Flashback Tables)

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Storage level backup managed by Amazon RDS

Oracle Usage

Oracle Flashback Table is a data protection feature used to undo changes to a table and rewind it to a previous state (not from backup). While Flashback table operations are running, the affected tables are locked, but the rest of the database remains available.

Notes:

- If the structure of a table has been changed since the point of restore, the FLASHBACK will fail.
- Row movement must be enabled.
- The data to restore must be found in the undo (dba must manage the size and retention).
- A table can be restored to an System Change Number (SCN), Restore Point, or Timestamp.

Examples

Flashback a table using SCN (query V\$DATABASE to obtain the SCN).

```
SELECT CURRENT_SCN FROM V$DATABASE;
FLASHBACK TABLE employees TO SCN 3254648;
```

Flashback a table using a Restore Point (query V\$RESTORE_POINT to obtain restore points).

```
SELECT NAME, SCN, TIME FROM V$RESTORE_POINT;
FLASHBACK TABLE employees TO RESTORE POINT employees_year_update;
```

Flashback a table using a Timestamp (query V\$PARAMETER to obtain the undo_retention value).

```
SELECT NAME, VALUE/60 MINUTES_RETAINED
FROM V$PARAMETER
WHERE NAME = 'undo_retention';
FLASHBACK TABLE employees TO
TIMESTAMP TO_TIMESTAMP('2017-09-21 09:30:00', 'YYYY-MM-DD HH:MI:SS');
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19;bradv/index.html>

PostgreSQL Usage

Snapshots are the primary backup mechanism for Amazon Aurora databases. They are extremely fast and non-intrusive. You can take snapshots using the Amazon RDS Management Console or 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.

Amazon Aurora provides the following types of backups:

- Automated Backups:** Always enabled on Amazon Aurora. They do not impact database performance.
- Manual Backups:** You can create a snapshot at any time. There is no performance impact when taking snapshots of an Aurora database. Restoring data from snapshots requires creation of a new instance. Up to 100 manual snapshots are supported per database.

Examples

For examples, see [Flashback Database](#).

Summary

Description	Oracle	Amazon Aurora
Create a “restore point”	CREATE RESTORE POINT before_update GUARANTEE FLASHBACK DATABASE;	aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name
Configure flashback “retention period”	ALTER SYSTEM SET db_flashback_retention_target=2880;	Configure the “Backup retention window” setting using the AWS management console or using the AWS CLI.
Flashback table to a previous “restore point”	shutdown immediate; startup mount; flashback database to restore point before_update;	<p>1. Create new cluster from a snapshot:</p> <pre>aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-mysql</pre> <p>2. Add new instance to the cluster:</p> <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-mysql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large</pre> <p>3. Use mysqldump and mysql to copy the table from the restored instance to the original instance</p>
Flashback table to a previous point in time	shutdown immediate; startup mount; FLASHBACK DATABASE TO TIME "TO_DATE ('01/01/2017','MM/DD/YY')";	<p>1. Create a new cluster from a snapshot and provide a specific point in time:</p> <pre>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</pre>

Description	Oracle	Amazon Aurora
		<p>2. Add a new instance to the cluster:</p> <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-mysql --db-cluster-identifier clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large</pre> <p>3. Use mysqlDbExport and mysql to copy the table from the restored instance to the original instance</p>

For more details, see:

- <http://docs.aws.amazon.com/cli/latest/reference/rds/index.html#cli-aws-rds>
- http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PIT.html
- http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_RestoreFromSnapshot.html

Oracle Recovery Manager (RMAN) vs. Amazon RDS Snapshots

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Storage level backup managed by Amazon RDS

Oracle Usage

Oracle Recovery Manager (RMAN) is Oracle's primary backup and recovery tool. It provides its own scripting syntax and can be used to take full or incremental backups of an Oracle database. The following list identifies the types of backups.

- **Full RMAN Backup:** Take a full backup of an entire database or individual Oracle data files. For example, a level 0 full backup.
- **Differential Incremental RMAN Backup:** Performs a backup of all database blocks that have changed from the previous level 0 or 1 backup.
- **Cumulative Incremental RMAN Backup:** Perform a backup all of blocks that have changed from the previous level 0 backup.

Notes:

- RMAN supports online backups of an Oracle database if it has been configured to run in Archived Log Mode.
- RMAN backs up the following files:
 - Database data files.
 - Database control file.
 - Database parameter file.
 - Database Archived Redo Logs.

Examples

Use the RMAN CLI to connect to an Oracle database.

```
export ORACLE_SID=ORCL
rman target=/
```

Perform a full backup of the database and the database archived redo logs.

```
BACKUP DATABASE PLUS ARCHIVELOG;
```

Perform an incremental level 0 or level 1 backup of the database.

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

Restore a database.

```
RUN {
SHUTDOWN IMMEDIATE;
STARTUP MOUNT;
RESTORE DATABASE;
RECOVER DATABASE;
ALTER DATABASE OPEN;
}
```

Restore a specific pluggable database (Oracle 12c).

```
RUN {
ALTER PLUGGABLE DATABASE pdbA, pdbB CLOSE;
RESTORE PLUGGABLE DATABASE pdbA, pdbB;
RECOVER PLUGGABLE DATABASE pdbA, pdbB;
ALTER PLUGGABLE DATABASE pdbA, pdbB OPEN;
}
```

Restore a database to a specific point in time.

```
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;
}
```

List all current database backups created with RMAN.

```
LIST BACKUP OF DATABASE;
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19;bradv/index.html>

PostgreSQL Usage

Snapshots are the primary backup mechanism for Amazon Aurora databases. They are extremely fast and non-intrusive. You can take snapshots using the Amazon RDS Management Console or 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.

Amazon Aurora provides the following types of backups:

- **Automated Backups:** Always enabled on Amazon Aurora. They do not impact database performance.
- **Manual Backups:** You can create a snapshot at any time. There is no performance impact when taking snapshots of an Aurora database. Restoring data from snapshots requires creation of a new instance. Up to 100 manual snapshots are supported per database.

Examples

For examples, see [Flashback Database](#).

Summary

Description	Oracle	Amazon Aurora
Scheduled backups	Create DBMS_SCHEDULER job that will execute your RMAN script on a scheduled basis.	Automatic
Manual full database backups	BACKUP DATABASE PLUS ARCHIVELOG;	Use Amazon RDS dashboard or the AWS CLI command to take a snapshot on the cluster: <pre>aws rds create-db-cluster-snapshot --db-cluster-snapshot-identifier Snapshot_name --db-cluster-identifier Cluster_Name</pre>
Restore database	RUN { SHUTDOWN IMMEDIATE; STARTUP MOUNT; RESTORE DATABASE; RECOVER DATABASE; ALTER DATABASE OPEN; }	Create new cluster from a cluster snapshot: <pre>aws rds restore-db-cluster-from-snapshot --db-cluster-identifier NewCluster --snapshot-identifier SnapshotToRestore --engine aurora-postgresql</pre> Add a new instance to the new/restored cluster: <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier clusternamespace-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large</pre>
Incremental differential	BACKUP INCREMENTAL LEVEL 0 DATABASE; BACKUP INCREMENTAL LEVEL 1 DATABASE;	N/A
Incremental cumulative	BACKUP INCREMENTAL LEVEL 0 CUMULATIVE DATABASE; BACKUP INCREMENTAL LEVEL 1 CUMULATIVE DATABASE;	N/A
Restore database to a specific point-in-time	RUN { SHUTDOWN IMMEDIATE; STARTUP MOUNT; SET UNTIL TIME "TO_DATE('19-SEP-2017 23:45:00','DD-MON-YYYY HH24:MI:SS')"; RESTORE DATABASE; RECOVER DATABASE; ALTER DATABASE OPEN RESETLOGS; }	Create new cluster from a cluster snapshot by given custom time to restore: <pre>aws rds restore-db-cluster-to-point-in-time --db-cluster-identifier clusternamespace-restore --source-db-cluster-identifier clusternamespace --restore-to-time 2017-09-19T23:45:00.000Z</pre> Add a new instance to the new/restored cluster: <pre>aws rds create-db-instance --region us-east-1 --db-subnet-group default --engine aurora-postgresql --db-cluster-identifier</pre>

Description	Oracle	Amazon Aurora
		clustername-restore --db-instance-identifier newinstance-nodeA --db-instance-class db.r4.large
Backup database Archive logs	BACKUP ARCHIVELOG ALL;	N/A
Delete old database Archive logs	CROSSCHECK BACKUP; DELETE EXPIRED BACKUP;	N/A
Restore a single Pluggable database (12c)	RUN { ALTER PLUGGABLE DATABASE pdb1, pdb2 CLOSE; RESTORE PLUGGABLE DATABASE pdb1, pdb2; RECOVER PLUGGABLE DATABASE pdb1, pdb2; ALTER PLUGGABLE DATABASE pdb1, pdb2 OPEN; }	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 host-name.rds.amazonaws.com -U username -d hr -p 5432 > c:\Export\hr.dmp pg_restore -h restored-hostname.rds.amazonaws.com -U hr -d hr_restore -p 5432 c:\Export\hr.dmp Optionally, replace with the old database using ALTER DATABASE RENAME

For more details, see:

<http://docs.aws.amazon.com/cli/latest/reference/rds/index.html#cli-aws-rds>

http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PIT.html

http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_RestoreFromSnapshot.html

Oracle SQL*Loader vs. PostgreSQL pg_dump/pg_restore

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
			Not all functions are supported by PostgreSQL and may require to create manually

Usage

SQL*Loader is powerful utility that imports data from external files into database tables. It has strong parsing engine with few limitations on data formats.

SQL*Loader can be used with or without a control file. A control file enables handling more complicated load environments. For simpler loads, use SQL*Loader without a control file (also referred to as SQL*Loader Express).

The outputs of SQL*Loader include the imported database data, a log file, a bad file (rejected records), and a discard file (if enabled).

Examples

Oracle SQL*Loader is well suited for large databases with a limited number of objects. The process of exporting from a source database and loading to a target database is very specific to the schema. The following example creates sample schema objects, exports from a source, and loads into a target database.

Create a sample source table.

```
CREATE TABLE customer_0 TABLESPACE users
AS SELECT rownum id, o.* FROM all_objects o, all_objects x where rownum <= 1000000;
```

On the target Amazon RDS instance, create a destination table for the loaded data.

```
CREATE TABLE customer_1 TABLESPACE users
AS select 0 as id, owner, object_name, created from all_objects where 1=2;
```

The data is exported from the source database to a flat file with delimiters. This example uses SQL*Plus. For your data, you will likely need to generate a script that does the export for all the objects in the database.

```
alter session set nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
set linesize 800
HEADING OFF FEEDBACK OFF array 5000 pagesize 0
spool customer_0.out
SET MARKUP HTML PREFORMAT ON SET COLSEP ',' SELECT id, owner, object_name, created FROM
customer_0;
spool off
```

Create a control file describing the data. Depending on the data, you may need to build a script that provides this functionality.

```
cat << EOF > sqlldr_1.ctl
LOAD DATA
```

```

INFILE customer_0.out
into table customer_1
APPEND
fields terminated by "," optionally enclosed by ""
(id                  POSITION(01:10)      INTEGER EXTERNAL,
owner               POSITION(12:41)      CHAR,
object_name         POSITION(43:72)      CHAR,
created             POSITION(74:92)      date "YYYY/MM/DD HH24:MI:SS")

```

Import the data using SQL*Loader. Use the appropriate username and password for the target database.

```
sqlldr cust_dba@targetdb control=sqlldr_1.ctl BINDSIZE=10485760 READSIZE=10485760 ROWS-S=1000
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/sutil/oracle-sql-loader.html#GUID-8D037494-07FA-4226-B507-E1B2ED10C144>

PostgreSQL Usage

There are two relevant options when looking for a replacement for the Oracle SQL*Loader utility:

- **PostgreSQL Import:** using an export file similar to a control file.
- **Load from S3 File:** using a table-formatted file on S3 and loading it into a MySQL database.

pg_restore is a good option when it's required to use a tool from another server or a client. The LOAD DATA command can be combined with meta-data tables and EVENT objects to schedule loads.

Another option to export and import data from PostgreSQL database is to use COPY TO/COPY FROM commands. Starting with PostgreSQL 12 COPY FROM command, that can be used to load data into DB, has support for filtering incoming rows with WHERE condition

```

CREATE TABLE tst_copy(v TEXT);

COPY tst_copy FROM '/home/postgres/file.csv' WITH (FORMAT CSV) WHERE v LIKE '%apple%';

```

For more details, see [PostgreSQL pg_dump](#)

Configuration

Oracle Upgrades vs. Aurora for PostgreSQL upgrades

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
N/A	N/A	N/A	

Usage

As a Database Administrator, from time to time a database upgrade is required, it can be either for security fix, but, or a new database feature.

The Oracle upgrades are divided into two different types of upgrades, minor and major.

This topic will outline the differences between the procedure to execute upgrades on your Oracle databases today and how you will run those upgrades post migrating to RDS running Aurora.

The regular presentation of Oracle versions is combined of 4 numbers divided by dots (sometimes you will see a a fifth number)

Either way, major or minor upgrades, the first step to initiate the processes mentioned above would be to install the new Oracle software on the database server, and of course before upgrading a production database to have an extensive amount of testing with the applications using the database to upgrade.

Oracle 18c introduces Zero-Downtime Database Upgrade to automate Database upgrade and potentially eliminate application downtime during the this process.

In order to understand the versions, lets use the following example 11.2.0.4.0

What those digits above means:

- 11 - this is the major database version
- 2 - this is the database maintenance version
- 0 - Application server version
- 4 - Component specific version
- 0 - Platform specific version

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/upgrd/oracle-database-release-numbers.html#GUID-1E2F3945-C0EE-4EB2-A933-8D1862D8ECE2>

Oracle allows the users to set the compatibility level of the database to control the features and some behaviors.

This is being done using the COMPATIBLE parameter, the value for this parameter can be fetched using the following query

```
SELECT NAME, VALUE FROM V$PARAMETER WHERE NAME = 'compatible';
```

Upgrade process

In general, the process to execute major or minor upgrades is the same, minor version upgrade has less steps but overall the process is very similar.

Major upgrade referring to upgrades of the version number in the Oracle version, in the example above "11", the minor upgrade refers to any of the following numbers in the Oracle version, in the example above these will be "2.0.4.0".

Major upgrades are mostly being done in order to gain many new useful features being released between those versions, while minor upgrades are focused on bug and security fixes.

Upgrades can be done using the Oracle upgrade tools or manually.

Oracle tools will perform the following steps and might ask for some inputs or fixes from the user along the process.

The upgrade steps are:

- Upgrade operation type - user will choose either Oracle database upgrade or move database between Oracle software installations
- Database selection - user will select the database to upgrade and the Oracle software to use for this database
- Prerequisite checks - Oracle tools will let the user choose what to do with all issues found and their severity
- Upgrade options - Oracle will let the user to pick his practices to do the upgrade, options like: recompilation and parallelism for those, timezone upgrade, statistics gathering, and more.
- Management options - user will choose to connect / configure Oracle management solutions to the database
- Move database files - user will choose if a data file movement is required to a new devices/path
- Network configuration - Oracle listener configurations
- Recovery options - user to define Oracle backup solutions or using his own
- Summary - a report of all options were selected in previous steps to present before the upgrade
- Progress - monitor and present the upgrade status
- Results - a post upgrade summary

For the manual process, we won't cover all actions in this topic, as there are many steps and commands to execute.

In overall, the eleven steps mentioned above will be divided into many sub-steps and tasks to execute.

For the entire manual option, please see: <https://docs.oracle.com/en/database/oracle/oracle-database/12.2/upgrd/example-manual-upgrade-windows-non-cdb-11203-to-122.html#GUID-30F3DC9C-141A-47DC-9B83-6D0C395E565C>

Aurora for PostgreSQL Usage

After migrating your databases to RDS running Aurora for PostgreSQL, you will still need to upgrade your database instances from time to time, for the same reasons you have done in the past, new features, bugs and

security fixes.

In a managed service like RDS, the upgrade process is much easier and simpler compare to the on-prem Oracle process.

To determine the current Aurora for PostgreSQL version being used, you can use the following aws cli command:

```
aws rds describe-db-engine-versions --engine aurora-postgresql --query '*[].[EngineVersion]' --output text --region your-AWS-Region
```

This can also be queried from the database, using the following queries:

```
SELECT AURORA_VERSION();

aurora_version
-----
4.0.0

SHOW SERVER_VERSION;

server_version
-----
12.4
```

All Aurora and PostgreSQL versions mapping can be found in here: <https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraPostgreSQL.Updates.20180305.html>

AWS doesn't apply major version upgrades on RDS Aurora automatically. Major version upgrades contains new features and functionality which often involves system table and other code changes. These changes may not be backward-compatible with previous versions of the database so application testing are highly recommended.

Applying automatic minor upgrades can be set by configuring the RDS instance to allow it.

You can use the following aws cli command (linux) to determine the current automatic upgrade minor versions.

```
aws rds describe-db-engine-versions --engine aurora-postgresql | grep -A 1 AutoUpgrade | grep -A 2 true | grep PostgreSQL | sort --unique | sed -e 's/"Description":' '//g'
```

Note: If no results are returned, there is no automatic minor version upgrade available and scheduled.

When enabled, the instance will be automatically upgraded during the scheduled maintenance window.

For major upgrades, this is the recommended process:

- Have a version-compatible parameter group ready.

If you are using a custom DB instance or DB cluster parameter group, you have two options:

1. Specify the default DB instance, DB cluster parameter group, or both for the new DB engine version.
2. Create your own custom parameter group for the new DB engine version.

Note: If you associate a new DB instance or DB cluster parameter group as a part of the upgrade request, make sure to reboot the database after the upgrade completes to apply the parameters. If a DB instance needs to be rebooted to apply the parameter group changes, the instance's parameter

group status shows pending-reboot. You can view an instance's parameter group status in the console or by using a CLI command such as describe-db-instances or describe-db-clusters.

- Check for unsupported usage:

1. Commit or roll back all open prepared transactions before attempting an upgrade. You can use the following query to verify that there are no open prepared transactions on your instance.

```
SELECT count(*) FROM pg_catalog.pg_prepared_xacts;
```

2. Remove all uses of the reg* data types before attempting an upgrade. Except for regtype and regclass, you can't upgrade the reg* data types. The pg_upgrade utility can't persist this data type, which is used by Amazon Aurora to do the upgrade.

To verify that there are no uses of unsupported reg* data types, use the following query for each database.

```
SELECT count(*) FROM pg_catalog.pg_class c, pg_catalog.pg_namespace n, pg_catalog.pg_attribute a WHERE c.oid = a.attrelid
AND NOT a.attisdropped
AND a.atttypid IN ('pg_catalog.regproc'::pg_catalog.regtype,
'pg_catalog.regprocedure'::pg_catalog.regtype,
'pg_catalog.regoper'::pg_catalog.regtype,
'pg_catalog.regoperator'::pg_catalog.regtype,
'pg_catalog.regconfig'::pg_catalog.regtype,
'pg_catalog.regdictionary'::pg_catalog.regtype)
AND c.relnamespace = n.oid
AND n.nspname NOT IN ('pg_catalog', 'information_schema');
```

- Perform a backup.

The upgrade process creates a DB cluster snapshot of your DB cluster during upgrading. If you also want to do a manual backup before the upgrade process.

- Upgrade certain extensions to the latest available version before performing the major version upgrade. The extensions to update include the following:

- pgRouting
- postGIS

Run the following command for each extension that you are using.

```
ALTER EXTENSION PostgreSQL-extension UPDATE TO 'new-version'
```

If you are upgrading older versions (older than 12), there are a few more steps, please review here:https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/USER_UpgradeDBInstance.PostgreSQL.html

All mentioned above a prerequisites, the actual upgrade can be done through the console or aws cli.

Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at <https://console.aws.amazon.com/rds/>.
2. In the navigation pane, choose Databases, and then choose the DB cluster that you want to upgrade.
3. Choose Modify. The Modify DB cluster page appears.
4. For DB engine version, choose the new version.

5. Choose Continue and check the summary of modifications.
6. To apply the changes immediately, choose Apply immediately. Choosing this option can cause an outage in some cases. For more information, see Modifying an Amazon Aurora DB cluster.
7. On the confirmation page, review your changes. If they are correct, choose Modify Cluster to save your changes.

Or choose Back to edit your changes or Cancel to cancel your changes.

AWS CLI

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
--db-cluster-identifier mydbcluster \
--engine-version new_version \
--allow-major-version-upgrade \
--no-apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
--db-cluster-identifier mydbcluster ^
--engine-version new_version ^
--allow-major-version-upgrade ^
--no-apply-immediately
```

Summary

Phase	Oracle Step	Aurora for PostgreSQL
Prerequisite	Install new Oracle software	N/A
Prerequisite	Upgrade operation type	N/A
Prerequisite	Database selection	Select right RDS instance
Prerequisite	Prerequisite checks	<ol style="list-style-type: none"> 1. Remove all uses of the reg* data types 2. Upgrade certain extensions 3. Commit or roll back all open prepared transactions SELECT count(*) FROM pg_catalog.pg_prepared_xacts;
Prerequisite	Upgrade options	N/A
Prerequisite	Management options (optional)	N/A
Prerequisite	Move database files (optional)	N/A
Prerequisite	Network configuration (optional)	N/A
Prerequisite	Recovery options	N/A
Prerequisite	Summary	N/A
Prerequisite	Perform a database backup	Run RDS instance backup

Phase	Oracle Step	Aurora for PostgreSQL
Prerequisite	Stop application and connection	Same
Execution	Progress	Status can be reviewed from the console
Post-upgrade	Results	Status can be reviewed from the console
Post-upgrade	Test applications again the new upgraded database	Same
Production deployment	Re-run all steps in a production environment	Same

For additional details, see: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/USER_UpgradeDBIn-stance.PostgreSQL.html

Oracle Alert.log and logs files vs. PostgreSQL Error Log via Amazon RDS Console

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Use Event Notifications Subscription with Amazon Simple Notification Service (SNS)

Oracle Usage

The primary Oracle error log file is the Alert Log. It contains verbose information about database activity including informational messages and errors. Each event includes a timestamp indicating when the event occurred. The Alert Log filename format is alert<sid>.log.

The Alert Log is the first place to look when troubleshooting or investigating errors, failures, and other messages indicating a potential database problem. Common events logged in the Alert Log include:

- Database startup or shutdown.
- Database redo log switch.
- Database errors and warnings, which begin with "ORA-" followed by an Oracle error number.
- Network and connection issues
- Links for a detailed trace files about specific database events.

The Oracle Alert Log can be found inside the database Automatic Diagnostics Repository (ADR), which is a hierarchical file-based repository for diagnostic information: \$ADR_BASE/diag/rdbms/{DB-name}/{SID}/trace.

In addition, several other Oracle server components have unique log files such as the database listener and the Automatic Storage Manager (ASM).

Examples

The following screenshot displays partial contents of the Oracle database Alert Log File.

```

Sun Sep 03 18:27:23 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

```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/monitoring-the-database.html#GUID-E5F89E8E-7FBC-47DD-BA5D-96AFD9CE4BC7>

PostgreSQL Usage

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	PostgreSQL
ORA-00001: unique constraint (string.string) violated	SQLSTATE[23505]: Unique violation: 7 ERROR: duplicate key value violates unique constraint "constraint_name"

For more details, see <https://www.postgresql.org/docs/10/static/errcodes-appendix.html>.

Error Log Types

Log severity levels supported by PostgreSQL:

Log Type	Information Written to Log
DEBUG1...DEBUG5	Provides successively-more-detailed information for use by developers
INFO	Provides information implicitly requested by the user
NOTICE	Provides information that might be helpful to users
WARNING	Provides warnings of likely problems
ERROR	Reports an error that caused the current command to abort
LOG	Reports information of interest to administrators
FATAL	Reports an error that caused the current session to abort
PANIC	Reports an error that caused all database sessions to abort

For more information, see <https://www.postgresql.org/docs/13/static/runtime-config-logging.html>

Examples

Access the PostgreSQL error log using the Amazon RDS/Aurora Management Console:

1. Navigate to [Instances](#) in the RDS Service of the AWS Console.
2. Click the instance name.

3. Click on the "Logs & Events" tab

DB identifier	Role	Engine	Region & AZ
pg-playbooks	Regional	Aurora PostgreSQL	eu-central-1
pg-playbooks-instance-1	Writer	Aurora PostgreSQL	eu-central-1a

Related

Filter databases

Connectivity & security Monitoring **Logs & events** Configuration Maintenance Tags

4. Scroll down to the **Logs** section and select the log to inspect. For example, select the log during the hour the data was experiencing problems. The following screen shot displays partial contents of a PostgreSQL database error log as viewed from the Amazon RDS Management Console.

Name	Last written	Size
error/mysql-error-running.log	Mon Feb 26 11:00:00 GMT+200 2018	10.6 kB
error/mysql-error-running.log.2018-02-09.19	Fri Feb 09 21:30:00 GMT+200 2018	93 kB
error/mysql-error-running.log.2018-02-09.20	Fri Feb 09 21:40:00 GMT+200 2018	4.4 kB
error/mysql-error-running.log.2018-02-09.21	Fri Feb 09 22:30:00 GMT+200 2018	18.1 kB
error/mysql-error-running.log.2018-02-09.22	Fri Feb 09 23:30:00 GMT+200 2018	22.5 kB

5. Check one of the logs and click on

Name	Last written	Logs
error/postgresql.log.2021-02-25-0300	Wed Feb 24 2021 22:00:00 GMT-0500	0 bytes

Viewing Log: error/mysql-error-running.log.2018-02-09.19 (93 kB)

text: background:

```
2018-02-09T19:01:48.847682Z 0 [Warning] 'NO_AUTO_CREATE_USER' sql mode was not set.
2018-02-09T19:01:48.848471Z 0 [Warning] Insecure configuration for --secure-file-priv:
Location is accessible to all OS users. Consider choosing a different directory.
2018-02-09T19:01:48.848497Z 0 [Note] /rdsdbbin/oscar/bin/mysqld (mysqld 5.7.12-log)
starting as process 5478 ...
2018-02-09T19:01:48.881215Z 0 [Warning] InnoDB: Setting innodb_checksums to OFF is
DEPRECATED. This option may be removed in future releases. You should set
innodb_checksum_algorithm=NONE instead.
2018-02-09T19:01:48.881317Z 0 [Note] InnoDB: PUNCH HOLE support not available
2018-02-09T19:01:48.881329Z 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic
builtins
2018-02-09T19:01:48.881333Z 0 [Note] InnoDB: Uses event mutexes
2018-02-09T19:01:48.881336Z 0 [Note] InnoDB: GCC builtin __atomic_thread_fence() is
used for memory barrier
2018-02-09T19:01:48.881340Z 0 [Note] InnoDB: Compressed tables use zlib 1.2.3
2018-02-09T19:01:48.882024Z 0 [Note] InnoDB: == Add SYNC_FAST DDL...
2018-02-09T19:01:48.884857Z 0 [Note] InnoDB: Number of pools: 1
2018-02-09T19:01:48.888837Z 0 [Note] InnoDB: Using CPU crc32 instructions
2018-02-09T19:01:48.890231Z 0 [Note] InnoDB: Initializing buffer pool, total size =
```

PostgreSQL Error Log Configuration

Several parameters control how and where PostgreSQL log and errors files will be placed:

Parameter	Description
<code>log_filename</code>	Sets the file name pattern for log files. Modifiable via an Aurora Database Parameter Group
<code>log_rotation_age</code>	(min) Automatic log file rotation will occur after N minutes. Modifiable via an Aurora Database Parameter Group
<code>log_rotation_size</code>	(kB) Automatic log file rotation will occur after N kilobytes.

Parameter	Description
	Modifiable via an Aurora Database Parameter Group
log_min_messages	Sets the message levels that are logged (DEBUG, ERROR, INFO, etc....). Modifiable via an Aurora Database Parameter Group
log_min_error_statement	Causes all statements generating error at or above this level to be logged (DEBUG, ERROR, INFO, etc....). Modifiable via an Aurora Database Parameter Group
log_min_duration_statement	Sets the minimum execution time above which statements will be logged (ms). Modifiable via an Aurora Database Parameter Group

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

Oracle SGA & PGA Memory Sizing vs. PostgreSQL Memory Buffers

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Different cache names, similar usage

Oracle Usage

An 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. The caches reside in the System Global Area (SGA) and are shared across all Oracle sessions.

In addition to the SGA, each Oracle session is granted an additional area of memory for session-private operations (sorting, private SQL cursors elements, etc.) called the Private Global Area (PGA).

Cache size can be controlled for individual caches or globally, and automatically, by an Oracle database. Setting a unified “memory size” parameter enables Oracle to automatically manage individual cache sizes.

- All Oracle memory parameters are set using the ALTER SYSTEM command.
- Some changes to memory parameters require an instance restart.

Some of the common Oracle parameters that control memory allocations include:

- **db_cache_size:** The size of the cache used for database data.
- **log_buffer:** The cache used to store Oracle redo log buffers until they are written to disk.
- **shared_pool_size:** The cache used to store shared cursors, stored procedures, control structures, and other structures.
- **large_pool_size:** The caches used for parallel queries and RMAN backup/restore operations.
- **Java_pool_size:** The caches used to store Java code and JVM context.

While these parameters can be configured individually, most database administrators choose to let Oracle automatically manage RAM. Database administrators configure the overall size of the SGA, and Oracle sizes individual caches based on workload characteristics.

- **sga_max_size:** Specifies the hard-limit maximum size of the SGA.
- **sga_target:** Sets the required soft-limit for the SGA and the individual caches within it.

Oracle also allows control over how much private memory is dedicated for each session. Database Administrators configure the total size of memory available for all connecting sessions, and Oracle allocates 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:** A hard-limit for the total amount of memory available for all sessions combined (Oracle 12c only).

In addition, instead of manually configuring the SGA and PGA memory areas, you 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 is enabled using the `memory_target` and `memory_max_target` parameters.

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/memory-architecture.html#GUID-913335DF-050A-479A-A653-68A064DCCA41>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/tgdba/database-memory-allocation.html#GUID-E9265077-B296-485A-BC2C-0AF55762D1EC>

PostgreSQL Usage

PostgreSQL provides us with control over how server RAM is allocated. Some of the most important PostgreSQL memory parameters include:

Memory Pool Parameter	Description
<code>shared_buffers</code>	Used to cache database data read from disk. Approximate Oracle Database Buffer Cache equivalent.
<code>wal_buffers</code>	Used to store WAL (Write-Ahead-Log) records before they are written to disk. Approximate Oracle Redo Log Buffer equivalent.
<code>work_mem</code>	Used for parallel queries and SQL sort operations. Approximate Oracle PGA equivalent and/or the Large Pool (for parallel workloads)
<code>maintenance_work_mem</code>	Memory used for certain backend database operations such as: VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY
<code>temp_buffers</code>	Memory buffers used by each database session for reading data from temporary tables.
Total memory available for PostgreSQL Cluster	Controlled by selecting the “DB Instance Class” during instance creation <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Instance specifications <small>Estimate your monthly costs for the DB Instance using the AWS Simple Monthly Calculator.</small> <div style="border: 1px solid #ccc; padding: 2px; margin-top: 2px;"> DB engine Aurora - compatible with MySQL 5.7.12 </div> <div style="border: 1px solid #ccc; padding: 2px; margin-top: 2px;"> DB instance class info </div> <div style="border: 1px solid #ccc; padding: 2px; margin-top: 2px;"> db.t2.small — 1 vCPU, 2 GB RAM db.t2.small — 1 vCPU, 2 GB RAM db.t2.medium — 2 vCPU, 4 GB RAM db.r3.large — 2 vCPU, 15.25 GB RAM db.r3.xlarge — 4 vCPU, 30.5 GB RAM db.r3.2xlarge — 8 vCPU, 61 GB RAM db.r3.4xlarge — 16 vCPU, 122 GB RAM db.r3.8xlarge — 32 vCPU, 244 GB RAM db.r4.large — 2 vCPU, 15.25 GB RAM db.r4.xlarge — 4 vCPU, 30.5 GB RAM db.r4.2xlarge — 8 vCPU, 61 GB RAM db.r4.4xlarge — 16 vCPU, 122 GB RAM db.r4.8xlarge — 32 vCPU, 244 GB RAM db.r4.16xlarge — 64 vCPU, 488 GB RAM </div> </div>

Note: Cluster level parameters, such as `shared_buffers` and `wal_buffers`, are configured using “parameter groups” in the Amazon RDS Management Console.

Examples

View the configured values for database parameters.

```
show shared_buffers
show work_mem
show temp_buffers
```

View the 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';
```

Summary

Note: Use the table below as a general reference only. Functionality may not be identical across Oracle and PostgreSQL.

Description	Oracle	PostgreSQL
Memory for caching table data	db_cache_size	shared_buffers

Description	Oracle	PostgreSQL
Memory for transaction log records	log_buffer	wal_buffers
Memory for parallel queries	large_pool_size	work_mem
Java code and JVM	Java_pool_size	N/A
Maximum amount of physical memory available for the Instance	sga_max_size or memory_max_size	Configured via the Amazon RDS/Aurora Instance class For example: db.r3.large: 15.25GB db.r3.xlarge: 30.5GB Etc.
Total amount of private memory for all sessions	pga_aggregate_target + pga_aggregate_limit	temp_buffers (for reading data from temp tables) work_mem (for sorts)
View values for all database parameters	SELECT * FROM v\$parameter;	Select * from pg_settings;
Configure a session-level parameter	ALTER SESSION SET ...	SET SESSION ...
Configure instance-level parameter	ALTER SYSTEM SET ...	Configured via “Parameter Groups” in the Amazon RDS Management Console.

For additional details:

<https://www.postgresql.org/docs/13/static/runtime-config-wal.html>

<https://www.postgresql.org/docs/13/static/runtime-config-resource.html>

Oracle Instance Parameters vs. AWS RDS Parameter Groups

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Use Cluster and Database/Cluster Parameters

Oracle Usage

Oracle Instance and database-level parameters can be configured using the ALTER SYSTEM command. Certain parameters can be configured dynamically and take immediate effect while other parameters require an instance restart.

- All Oracle instance and database-level parameters are stored in a binary file known as the Server Parameter file (SPFILE).
- The binary SPFILE can be exported to a text file using the following command: CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';

When modifying parameters, you can choose the persistence 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.

Examples

Use the ALTER SYSTEM SET command to configure a value for an Oracle parameter.

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE SCOPE=BOTH;
```

For more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/initialization-parameters-2.html#GUID-FD266F6F-D047-4EBB-8D96-B51B1DCA2D61>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/changing-parameter-values-in-a-parameter-file.html#GUID-4C578B21-DE2B-4210-8EB7-EF28D36CC1CB>

PostgreSQL Usage

When running PostgreSQL databases as Amazon Aurora Clusters, Parameter Groups are used to change to cluster-level and database-level parameters.

Most of the PostgreSQL parameters are configurable in an Amazon Aurora PostgreSQL cluster, but some are disabled and cannot be modified. Since Amazon Aurora clusters restrict access to the underlying operating system, modification to PostgreSQL parameters must be made using Parameter Groups.

Amazon Aurora is a cluster of database 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.

Aurora PostgreSQL Parameter Class	Controlled Via
Cluster-level parameters Single cluster parameter group per Amazon Aurora Cluster	Managed via cluster parameter groups For example: <ul style="list-style-type: none"> The PostgreSQL wal_buffers parameter is controlled via a cluster parameter group. The PostgreSQL autovacuum parameter is controlled via a cluster parameter group. The client_encoding parameter is controlled via a cluster parameter group.
Database Instance-Level parameters Every instance in an Amazon Aurora cluster can be associated with a unique database parameter group	Managed via database parameter groups For example: <ul style="list-style-type: none"> The PostgreSQL shared_buffers memory cache configuration parameter is controlled via a database parameter group with an AWS-optimized default value based on the configured database class: {DBInstanceClassMemory/10922} The PostgreSQL max_connections 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: LEAST({DBInstanceClassMemory/9531392},5000) 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. The PostgreSQL effective_cache_size 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): {DBInstanceClassMemory/10922}

New parameters in PostgreSQL 10:

- enable_gathermerge - enable execution plan Gather Merge
- max_parallel_workers - maximum number of parallel workers process
- max_sync_workers_per_subscription - maximum number of synchronous workers for subscription
- wal_consistency_checking - check consistency of WAL on the standby instance (can't be set in Aurora PostgreSQL)
- max_logical_replication_workers - maximum number of logical replication worker process

6. max_pred_locks_per_relation - Maximum number of records that can be predicate-lock before locking the entire relation (signup)
7. max_pred_locks_per_page - Maximum number of records that can be predicate-lock before locking the entire page
8. min_parallel_table_scan_size - minimum table size to consider parallel table scan
9. min_parallel_index_scan_size - minimum table size to consider parallel index scan

Examples

Create and Configure a New Parameter Group

Follow the steps below to create and configure Amazon Aurora database and cluster parameter groups:

1. Navigate to the "[Parameter group](#)" section in the RDS Service of the AWS Console.
2. Click **Create Parameter Group**.

Note: You cannot edit the default parameter group. You must create a custom parameter group to apply changes to your Amazon Aurora cluster and its database instances.

The screenshot shows the AWS RDS Parameter Groups page. On the left is a sidebar with links like Dashboard, Databases, Query Editor, etc. The main area shows a table titled "Parameter groups (5)" with columns: Name, Family, Type, and Description. The table lists five entries: "default.aurora-mysql5.7" (Family: aurora-mysql5.7, Type: Parameter groups), "default.aurora-mysql5.7" (Family: aurora-mysql5.7, Type: DB cluster parameter group), "default.aurora-postgresql12" (Family: aurora-postgresql12, Type: Parameter groups), "default.aurora-postgresql12" (Family: aurora-postgresql12, Type: DB cluster parameter group), and "default.oracle-se2-19" (Family: oracle-se2-19, Type: Parameter groups). A red box highlights the "Create parameter group" button at the top right of the table area.

3. Select from the **Parameter group family** drop-down list. Select **DB Parameter Group** from the **Type** drop-down list (another option is to select **Cluster Parameter Group** for modifying cluster parameters). Click **Create**.

The screenshot shows the "Create parameter group" wizard. The first step, "Parameter group details", has the following fields:

- Parameter group family:** "aurora-mysql5.7" (highlighted with a red box)
- Type:** "DB Parameter Group" (highlighted with a red box)
- Group name:** An empty input field (highlighted with a red box)
- Description:** An empty input field (highlighted with a red box)

 At the bottom right, there are "Cancel" and "Create" buttons. The "Create" button is highlighted with a red box and has the number "3" above it, indicating the next step in the process.

Modify an Existing Parameter Group

1. Navigate to the ["Parameter group" section](#) in the RDS Service of the AWS Console.
2. Click the name of the parameter to edit.
3. Click on the **Parameter group actions** drop-down list
4. Click the **Edit** button.

The screenshot shows the AWS RDS Parameter groups page. At the top, there is a navigation bar with 'RDS > Parameter groups'. Below it, a table lists six parameter groups. The first group, 'non-default-param-group', has a checkmark next to its name and is highlighted with a blue border. A red number '1' is placed over this row. To the right of the table is a 'Parameter group actions' dropdown menu with three options: 'Edit' (highlighted with a red background and red number '3'), 'Copy', and 'Compare'. A red number '2' is placed above the menu. At the bottom right of the table, there is a 'Create parameter group' button. The table columns are: Name, Description, DB cluster parameter group, and Default cluster parameter group.

Name	Description	DB cluster parameter group	Default cluster parameter group
default.aurora-mysql5.7	aurora-mysql5.7	Parameter groups	Default parameter groups
default.aurora-mysql5.7	aurora-mysql5.7	DB cluster parameter group	Default cluster parameter group
default.aurora-postgresql12	aurora-postgresql12	Parameter groups	Default parameter groups
default.aurora-postgresql12	aurora-postgresql12	DB cluster parameter group	Default cluster parameter group
default.oracle-se2-19	oracle-se2-19	Parameter groups	Default parameter groups
non-default-param-group	aurora-postgresql12	Parameter groups	Default parameter groups

5. Change parameter values and click **Save changes**.

For more details, see <https://www.postgresql.org/docs/13/static/sql-set.html>.

Oracle Session Parameters vs. PostgreSQL Session Parameters

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	SET options are significantly different in PostgreSQL

Oracle Usage

Certain Oracle database parameters and configuration options are modifiable at the session level using the ALTER SESSION command. However, 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, query the v\$parameter view as shown in the following example.

```
SELECT NAME, VALUE FROM V$PARAMETER WHERE ISSES_MODIFIABLE='TRUE';
```

Examples

Change the NLS_LANGUAGE (codepage) parameter of the current session.

```
alter session set nls_language='SPANISH'
Sesi n modificada.

alter session set nls_language='ENGLISH';
Session altered.

alter session set nls_language='FRENCH';
Session modifi e.

alter session set nls_language='GERMAN';
Session wurde ge ndert.
```

Specify the format of date values returned from the database using the NLS_DATE_FORMAT session parameter.

```
select sysdate from dual;
SYSDATE
-----
SEP-09-17
alter session set nls_date_format='DD-MON-RR';
Session altered.
```

```

select sysdate from dual;
SYSDATE
-----
09-SEP-17

alter session set nls_date_format='MM-DD-YYYY';

Session altered.

select sysdate from dual;

SYSDATE
-----
09-09-2017

alter session set nls_date_format='DAY-MON-RR';

Session altered.

```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/changing-parameter-values-in-a-parameter-file.html#GUID-4C578B21-DE2B-4210-8EB7-EF28D36CC1CB>

PostgreSQL Usage

PostgreSQL provides session-modifiable parameters that are configured using the SET SESSION command. Configuration of parameters using SET SESSION will only be applicable in the current session. To view the list of parameters that can be set with SET SESSION , you can query pg_settings:

```
SELECT * FROM pg_settings where context = 'user';
```

Examples of commonly used session parameters:

- **client_encoding** - configures the connected client character set.
- **force_parallel_mode** - forces use of parallel query for the session.
- **lock_timeout** - sets the maximum allowed duration of time to wait for a database lock to release.
- **search_path** - sets the schema search order for object names that are not schema-qualified.
- **transaction_isolation** - sets the current Transaction Isolation Level for the session.

Examples

Change the Time zone of the connected session.

```

set session DateStyle to POSTGRES, DMY;
SET
select now();
      now
-----
Sat 09 Sep 11:03:43.597202 2017 UTC
(1 row)

set session DateStyle to ISO, MDY;

```

```

SET
select now();
      now
-----
2017-09-09 11:04:01.3859+00
(1 row)

```

Summary

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.

Parameter purpose	Oracle	PostgreSQL
Configure time and date format	ALTER SESSION SET nls_date_format = 'dd/mm/yyyy hh24:mi:ss';	SET SESSION datestyle to 'SQL, DMY';
Configure the current default schema/database	ALTER SESSION SET current schema='schema_name'	SET SESSION SEARCH_PATH TO schemaname;
Generate traces for specific errors	ALTER SESSION SET events '10053 trace name context forever';	N/A
Run trace for a SQL statement	ALTER SESSION SET sql_trace=TRUE; ALTER SYSTEM SET EVENTS 'sql_trace [sql:&&sql_id] bind-d=true, wait=true';	N/A
Modify query optimizer cost for index access	ALTER SESSION SET optimizer_index_cost_adj = 50	SET SESSION random_page_cost TO 6;
Modify query optimizer row access strategy	ALTER SESSION SET optimizer_mode=all_rows;	N/A
Memory allocated to sort operations	ALTER SESSION SET sort_area_size=6321;	SET SESSION work_mem TO '6MB';
Memory allocated to hash-joins	ALTER SESSION SET hash_area_size=1048576000;	SET SESSION work_mem TO '6MB';

For more details, see <https://www.postgresql.org/docs/13/static/sql-set.html>

Performance Tuning

Oracle Database Hints vs. PostgreSQL DB Query Planning

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Very limited set of hints in PostgreSQL- Index hints and optimizer hints as comments Syntax differences

Oracle Usage

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 the database optimizer is performed using Database Hints. They can be defined as a directive operation to the optimizer and alter the decisions of how execution plans are generated.

Oracle supports over 60 different database hints, and each database hint can have 0 or more arguments. Database hints are divided into different categories such as optimizer hints, join order hints, and parallel execution hints.

Note: Database hints are embedded directly into the SQL queries immediately following the SELECT keyword using the format /* <DB_HINT> */.

Examples

Force the Query Optimizer to use a specific index for data access.

```

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 more details, see:

- <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Comments.html#GUID-D316D545-89E2-4D54-977F-FC97815CD62E>
- <https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsq/influencing-the-optimizer.html#GUID-8758EF88-1CC6-41BD-8581-246702414D1D>

PostgreSQL Usage

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):

```
SET ENABLE_SEQSCAN=FALSE;
```

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”.

```
SET SEQ_PAGE_COST to 4;
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.

```
SET ENABLE_NESTLOOP to FALSE;
```

For additional details: <https://www.postgresql.org/docs/13/static/runtime-config-query.html>

Oracle Execution Plans vs. PostgreSQL Execution Plans

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Syntax differences Completely different optimizer with different operators and rules in PostgreSQL

Oracle Usage

Execution plans represent the choices made by the query optimizer for accessing database data. The query optimizer generates execution plans for SELECT, INSERT, UPDATE and DELETE statements. Users and database administrators can view execution plans for specific queries and DML operations.

Execution plans are especially useful for performance tuning of queries. For example, determining 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 with the following information:

- Tables access by the SQL statement and the referenced order for each table.
- Access method for each table in the statement (full table scan vs. index access).
- Algorithms used for join operations between tables (hash vs. nested loop joins).
- Operations performed on retrieved data as such as filtering, sorting, and aggregations.
- Information about rows being processed (cardinality) and the cost for each operation.
- Table partitions being accessed.
- Information about parallel executions.

Oracle 19 introduces SQL Quarantine: now queries that consume resources excessively can be automatically quarantined and prevented from being executed. These queries execution plans are also quarantined.

Examples

Review the potential execution plan for a query using the EXPLAIN PLAN statement.

```
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
WHERE LAST_NAME='King' AND FIRST_NAME='Steven';

Execution Plan
-----
Plan hash value: 2077747057
-----
|-----|
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU) | Time
|-----|
```

```
-----
|   0 | SELECT STATEMENT          |           |    1 |    16 |     2 (0) |
00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES  |    1 |    16 |     2 (0) |
00:00:01 |
|*  2 |  INDEX RANGE SCAN         | EMP_NAME_IX |    1 |       |     1 (0) |
00:00:01 |
-----
-----
Predicate Information (identified by operation id):
-----
2 - access("LAST_NAME"='King' AND "FIRST_NAME"='Steven')
```

SET AUTOTRACE TRACEONLY EXPLAIN instructs SQL*PLUS to show the execution plan without actually running the query itself.

The EMPLOYEES table contains indexes for both the LAST_NAME and FIRST_NAME columns. Step 2 of the execution plan indicates the optimizer is performing an INDEX RANGE SCAN in order to retrieve the filtered employee name.

View a different execution plan displaying a FULL TABLE SCAN.

```
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME FROM EMPLOYEES
WHERE SALARY > 10000;
Execution Plan
-----
Plan hash value: 1445457117
-----
| Id  | Operation      | Name        | Rows  | Bytes | Cost (%CPU) | Time      |
|-----|
|   0 | SELECT STATEMENT |             |    72 | 1368 |     3 (0) | 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES  |    72 | 1368 |     3 (0) | 00:00:01 |
-----
Predicate Information (identified by operation id):
-----
1 - filter("SALARY">>10000)
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/generating-and-displaying-execution-plans.html#GUID-60E30B1C-342B-4D71-B154-C26623D6A3B1>

PostgreSQL Usage

The PostgreSQL equivalent to Oracle's EXPLAIN PLAN is the EXPLAIN keyword. The EXPLAIN keyword is used to display the execution plan for a supplied SQL statement. Similar to Oracle, the query planner in PostgreSQL will generate the estimated execution plan for actions such as: SELECT, INSERT, UPDATE and DELETE. It builds a structured tree of plan nodes representing the different actions taken (the sign “->” represents a root line in the PostgreSQL execution plan). In addition, the EXPLAIN statement provides 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 will be an estimation. By using the EXPLAIN ANALYZE command, the statement will execute in addition to displaying the execution plan.

PostgreSQL EXPLAIN Synopsis:

```
EXPLAIN [ ( option value[, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

where option and values can be one of:

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
SUMMARY [ boolean ] (since PostgreSQL 10)
FORMAT { TEXT | XML | JSON | YAML }
```

By default, planning and execution time are displayed when using EXPLAIN ANALYZE, but not in other cases. A new option "SUMMARY" allows explicit control of this information. Use SUMMARY to include planning and execution time metrics in your output.

PostgreSQL provides configurations options that will cancel SQL statements running longer than provided time limit. To use this option, you can set the statement_timeout instance-level parameter.

If value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

Third-party connection pooler solutions like PgBouncer and PgPool build on that and allow more flexibility in controlling how long connection to DB can run, be in idle state etc.

Aurora PostgreSQL Query Plan Management

The Aurora PostgreSQL Query Plan Management (QPM) feature solves the problem of plan instability by allowing database users to maintain stable, yet optimal, performance for a set of managed SQL statements. QPM primarily serves two main objectives:

Plan Stability. QPM prevents plan regression and improves plan stability when any of the above changes occur in the system.

Plan Adaptability. QPM automatically detects new minimum-cost plans and controls when new plans may be used and adapts to the changes.

The quality and consistency of query optimization have a major impact on the performance and stability of any relational database management system (RDBMS). Query optimizers create a query execution plan for a SQL statement at a specific point in time. As conditions change, the optimizer might pick a different plan that makes performance better or worse. In some cases, a number of changes can all cause the query optimizer to choose a different plan and lead to performance regression. These changes include changes in statistics, constraints, environment settings, query parameter bindings, and software upgrades. Regression is a major concern for high-performance applications.

With query plan management, you can control execution plans for a set of statements that you want to manage. You can do the following:

- Improve plan stability by forcing the optimizer to choose from a small number of known, good plans.
- Optimize plans centrally and then distribute the best plans globally.

- Identify indexes that aren't used and assess the impact of creating or dropping an index.
- Automatically detect a new minimum-cost plan discovered by the optimizer.
- Try new optimizer features with less risk, because you can choose to approve only the plan changes that improve performance.

Examples

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

```
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:

```
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)
```

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:

```
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 see: <https://www.postgresql.org/docs/13/static/sql-explain.html>

Oracle Table Statistics vs. PostgreSQL Table Statistics

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Syntax and option differences, similar functionality

Oracle Usage

Table statistics are one of the important aspects affecting SQL query performance. They enable 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 collects table and index statistics during predefined maintenance windows using the database scheduler and automated maintenance tasks. The automatic statistics collection mechanism uses Oracle's data modification monitoring feature that tracks the approximate number of INSERTs, UPDATEs, and DELETEs to determine which table statistics should be collected.

Oracle 19 now allows to gather real-time statistics on tables during regular UPDATE, INSERT and DELETE operations, which ensures that statistics are always up-to-date and are not going stale.

Oracle 19 also introduces High-Frequency Automatic Optimizer Statistics Collection - ability to set up automatic task that will collect statistics for stale objects.

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:

Statistics level	Description
GATHER_INDEX_STATS	Index statistics
GATHER_TABLE_STATS	Table, column, and index statistics
GATHER_SCHEMA_STATS	Statistics for all objects in a schema
GATHER_DICTIONARY_STATS	Statistics for all dictionary objects
GATHER_DATABASE_STATS	Statistics for all objects in a database

Examples

Collect statistics at the table level (schema - HR, table - EMPLOYEES).

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('HR','EMPLOYEES');
END;
/
PL/SQL procedure successfully completed.
```

Collect statistics at a specific column level (schema - HR, table - EMPLOYEES, column - DEPARTMENT_ID).

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('HR','EMPLOYEES',
                               METHOD_OPT=>'FOR COLUMNS department_id');
END;
/
PL/SQL procedure successfully completed.
```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/optimizer-statistics-concepts.html#GUID-C0E74ACE-2706-48A1-97A2-33F52207166A>

PostgreSQL Usage

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.
4. 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. Auto-vacuum is controlled by several parameters.

Individual tables have several storage parameters which can trigger autovacuum process sooner or later. These parameters, like autovacuum_enabled, autovacuum_vacuum_threshold and others can be set or changed using CREATE TABLE or ALTER TABLE statements.

```
ALTER TABLE custom_autovaccum SET (autovacuum_enabled = true, autovacuum_vacuum_cost_delay = 10ms, autovacuum_vacuum_scale_factor = 0.01, autovacuum_analyze_scale_factor = 0.005);
```

The command above will enable autovaccum for the custom_autovaccum table and will specify the autovacuum process to sleep for 10 milliseconds each run.

It will also specify a 1% of the table size to be added to autovacuum_vacuum_threshold and 0.5% of the table size to be added to autovacuum_analyze_threshold when deciding whether to trigger a VACUUM

For additional details: <https://www.postgresql.org/docs/13/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:

```
ANALYZE;
```

2. Gather statistics for a specific table. The VERBOSE keyword displays progress.

```
ANALYZE VERBOSE EMPLOYEES;
```

3. Gather statistics for a specific column:

```
ANALYZE EMPLOYEES (HIRE_DATE);
```

4. Specify the default_statistics_target parameter for an individual table column and reset it back to default:

```
ALTER TABLE EMPLOYEES ALTER COLUMN SALARY SET STATISTICS 150;
```

```
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:

```
SHOW default_statistics_target ;
SET default_statistics_target to 150;
ANALYZE EMPLOYEES ;
```

6. View the last time statistics were collected for a table:

```
SELECT relname, last_analyze FROM pg_stat_all_tables;
```

Summary

Feature	Oracle	PostgreSQL
Analyze a specific database table	BEGIN dbms_stats.gather_table_stats(own-name =>'hr', tabname => 'employees' , ...); END;	ANALYZE EMPLOYEES;
Analyze a database table while only sampling certain rows	Configure via percentage of table rows to sample: BEGIN dbms_stats.gather_table_stats(own-name=>'HR' , ... ESTIMATE_PERCENT=>100); END;	Configure via number of entries for the table: SET default_statistics_target to 150; ANALYZE EMPLOYEES ;
Collect statistics for a schema	BEGIN EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS(ownname => 'HR'); END	ANALYZE;
View last time statistics were collected	select owner,table_name,last_analyzed;	select relname, last_analyze from pg_stat_all_tables;

For additional information on PostgreSQL Collecting Table Statistics:

<https://www.postgresql.org/docs/13/static/sql-analyze.html>

<https://www.postgresql.org/docs/13/static/routine-vacuuming.html#AUTOVACUUM>

Security

Oracle Encryption - TDE vs. PostgreSQL AWS Aurora Encryption

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Use AWS Aurora Encryption

Oracle Usage

Oracle data encryption called Transparent Data Encryption (TDE).

TDE encrypts the data that is saved in the tables or tablespaces and protects data stored on media (also called data at rest) in case this media or data files are stolen.

Oracle uses authentication, authorization, and auditing mechanisms to secure data in the database but TDE is working on the operating system level.

No need to change from the application or client when encrypting data with TDE; the database manages it automatically.

It does not protect data in transit. Use the network encryption solutions discussed.

- The user who wants to configure TDE must have ADMINISTER KEY MANAGEMENT system privilege.
- Data can be encrypted at column level or tablespace level.
- Key of encryption managed in external module is called TDE master encryption.
- There is one master key store per database.

Examples

For storing master encryption key you can configure Oracle's software keystore:

1. Define at sqlnet.ora the ENCRYPTION_WALLET_LOCATION parameter to define where the keystores is, the options are to put to key file in:

- a. Regular filesystem
- b. Multiple DBs shared the same file
- c. ASM filesystem
- d. ASM diskgroup

Register in sqlnet.ora to put key file in ASM diskgroup

```
ENCRYPTION_WALLET_LOCATION=
(SOURCE=
(METHOD=FILE)
(METHOD_DATA=
(DIRECTORY=+ASM_file_path_of_the_diskgroup)))
```

2. Create software keystores, there are three types:

- a. Password-based
- b. Auto-login
- c. Local auto-login

How to create password-based software keystore

Connect to a database with user that have ADMINISTER KEY MANAGEMENT or SYSKM privilege and then create the keystore.

```
sqlplus c##sec_admin as syskm
Enter password: password
Connected.

ADMINISTER KEY MANAGEMENT CREATE KEYSTORE '/etc/ORACLE/WALLETS/orcl' IDENTIFIED BY
password;

keystore altered.
```

3. When password-based is in use, you must open the keystore before any TDE master encryption keys can be created or accessed in the keystore, auto-login and local auto-login are automatically opened (you can close them).

Opening the keystore

```
sqlplus c##sec_admin as syskm
Enter password: password
Connected.

ADMINISTER KEY MANAGEMENT SET KEYSTORE OPEN IDENTIFIED BY password;

keystore altered.
```

4. Set the software master encryption key, the key is stored in the keystore, this key protects the TDE table keys and tablespace encryption keys.

By default the TDE master encryption key is a key that the TDE generates.

To set the software master encryption key:

- Make sure that the database is open in READ WRITE mode
- Connect with the user that have the right privileges and create the master key

```
sqlplus c##sec_admin as syskm
Enter password: password
Connected.

ADMINISTER KEY MANAGEMENT SET KEY IDENTIFIED BY keystore_password WITH BACKUP USING
'emp_key_backup';

keystore altered.
```

5. Encrypt the data

- Encrypt column:
 - Data types that has encrypt support: BINARY_DOUBLE, BINARY_FLOAT, CHAR, DATE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH, NCHAR, NUMBER, NVARCHAR2, RAW (legacy or extended), TIMESTAMP (includes TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE), VARCHAR2 (legacy or extended)
 - Note:** you can't use column encryption with those features
 - Index types other than B-tree
 - Range scan search through an index
 - Synchronous change data capture

4. Transportable tablespaces
5. Columns used in foreign key constraints
- iii. How to create table with encrypted column.

```
CREATE TABLE employee (
    FIRST_NAME VARCHAR2(128),
    LAST_NAME VARCHAR2(128),
    EMP_ID NUMBER,
    SALARY NUMBER(6) ENCRYPT);
```

You can change the algorithm that encrypts the data.
 NO SALT option encrypts without the algorithm.
 USING clause can define the algorithm that is used to encrypt data.

```
CREATE TABLE EMPLOYEE (
    FIRST_NAME VARCHAR2(128),
    LAST_NAME VARCHAR2(128),
    EMP_ID NUMBER ENCRYPT NO SALT,
    SALARY NUMBER(6) ENCRYPT USING '3DES168');
```

The algorithm can be changed.

```
ALTER TABLE EMPLOYEE REKEY USING 'SHA-1';
```

Stop encrypting column.

```
ALTER TABLE employee MODIFY (SALARY DECRYPT);
```

Encrypt tablespace, the TDE encrypts in the SQL layer so all the data types and indexes restrictions aren't applied for tablespace encryption.

- Make sure that COMPATIBLE initialization parameter is set to 11.2.0.0 (minimum).
- Login to the database.
- Create the tablespace, you can't modify existing tablespace, only to create new one. In this example, the first TS created with AES256 algorithm and the second TS created with default algorithm.

```
sqlplus sec_admin@hrpdb
Enter password: password
Connected.

CREATE TABLESPACE encrypt_ts
DATAFILE '$ORACLE_HOME/dbs/encrypt_df.dbf' SIZE 1M
ENCRYPTION USING 'AES256'
DEFAULT STORAGE (ENCRYPT);
CREATE TABLESPACE securitespace_2
DATAFILE '/home/user/oradata/secure01.dbf'
SIZE 150M
ENCRYPTION
DEFAULT STORAGE (ENCRYPT);
```

For more information, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/asoag/introduction-to-trans-parent-data-encryption.html#GUID-62AA9447-FDCD-4A4C-B563-32DE04D55952>

PostgreSQL Usage

Amazon provides the ability to encrypt “data at rest” (data stored in persistent storage).

When enabling data encryption, it will automatically encrypt the database server storage, its automated backups, its read replicas and snapshots by using the AES-256 encryption algorithm.

This encryption will be done by using AWS KMS:

(<http://docs.aws.amazon.com/kms/latest/developerguide/overview.html>).

Once enabled, Amazon will transparently encrypt/decrypt the data without any impact on performances or any user intervention, and there will be no need to set any additional modifications to your clients to support this encryption.

Enabling encryption

As part of the database settings you will be asked to enable encryption and choose a master key.

Encryption

Encryption

Enable Encryption
Select to encrypt the given instance. Master key ids and aliases appear in the list after they have been created using the Key Management Service(KMS) console. [Learn More](#).

Disable Encryption

Master key [info](#)

(default) aws/rds ▾

Master key

This is the master key that will be used to protect the key used to encrypt this database volume. You can select from master keys in your account or type/paste the ARN of a key from a different account. You can create a new master encryption key by going to the Encryption Keys tab of the IAM console.

Note that it is possible to either choose the default key provided for the account or define a specific key based on an IAM KMS ARN from your account or a different account.

Creating an encryption key

To create your own key, browse to the Key Management Service (KMS) and click on "Customer managed keys" and create a new key.

KMS > Customer managed keys

Customer managed keys (0)

Key actions ▾ **Create key**

Filter keys by aliases, key ID, key type, or tag

Aliases Key ID Status Key spec ⓘ Key usage

Empty Resources
No resources to display

Choose relevant options and click on “Next”.

Configure key

Step 1 of 5

Key type Help me choose ↗ **1**

Symmetric
A single encryption key that is used for both encrypt and decrypt operations

Asymmetric
A public and private key pair that can be used for encrypt/decrypt or sign/verify operations

▼ Advanced options

Key material origin Help me choose ↗ **2**

KMS **2**
 External
 Custom key store (CloudHSM)

Cancel **3** Next

Define Alias as the name of the key, add description and click "Next".

Add labels

Step 2 of 5

Create alias and description

Enter an alias and a description for this key. You can change the properties of the key at any time. [Learn more](#)

Alias
db-rds-finance

Description - *optional*
This encryption key should be used to finance data.

Tags - *optional*

You can use tags to categorize and identify your CMKs and help you track your AWS costs. When you add tags to AWS resources, AWS generates a cost allocation report for each tag. [Learn more](#)

This key has no tags.

[Add tag](#)

You can add up to 50 more tags

[Cancel](#) [Previous](#) [Next](#)

You can skip “Define Key Administrative Permissions” (step 3 of 5), click on “Next”.

On the next step, make sure to assign the key to the relevant users who will need to interact with Aurora:

Define key usage permissions

Step 4 of 5

This account
Select the IAM users and roles that can use the CMK in cryptographic operations. [Learn more](#)

<input type="checkbox"/>	Name	Path
<input checked="" type="checkbox"/>	g	am.com

Other AWS accounts

Specify the AWS accounts that can use this key. Administrators of the accounts you specify are responsible for managing the permissions that allow their IAM users and roles to use this key. [Learn more](#)

[Add another AWS account](#)

[Cancel](#) [Previous](#) [Next](#)

On the last step you will be able to see the ARN of the key and its account.

Review and edit key policy

```

1 {
2     "Id": "key-consolepolicy-3",
3     "Version": "2012-10-17",
4     "Statement": [
5         {
6             "Sid": "Enable IAM User Permissions",
7             "Effect": "Allow",
8             "Principal": {
9                 "AWS": "arn:aws:iam:::root"
10            },
11            "Action": "kms:*",
12            "Resource": "*"
13        },
14        {
15            "Sid": "Allow use of the key",

```

Cancel Previous **Finish**

Click on “Finish” and now this key will be listed in under customer managed keys.

Now you will be able to set Master encryption key by using the ARN of the key that you have created or picking it from the list.

Encryption

Enable encryption

Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. [Info](#)

Master key [Info](#)

db-rds-finance	▲
(default) aws/rds	
db-rds-finance	
Enter a key ARN	

KMS key ID

Proceed and finish the instance launch.

SSE-S3 Encryption Feature Overview

Server-Side Encryption with Amazon S3-managed encryption keys (SSE-S3) uses a multi-factor encryption. Amazon S3 encrypts its objects with a unique key and in addition it also encrypts the key itself with a master key that rotates periodically.

SSE-S3 uses AES-256 as its encryption standard.

Once the S3 bucket was enabled with Server-side encryption, the data will be encrypted at rest, meaning that from this stage, any API call will have to include the special header – `x-amz-server-side-encryption`.

Additionally, aws cli command line tool will also need to be added with the switch `--sse`.

For more information please refer to the following pages:

1. SSE-23 with API - <http://docs.aws.amazon.com/AmazonS3/latest/dev/SSEUsingRESTAPI.html>
2. SSE-S3 with AWS CLI S3 - <http://docs.aws.amazon.com/cli/latest/reference/s3/cp.htm>

Enabling SSE-S3

1. Create an [AWS GLUE Job](#).
2. Define the role, bucket, and the script that you would like to use and then open “Script libraries and job parameters (optional)”.
3. Enable “Server-Side Encryption”
4. Submit the job and run it.

From this point, you will notice that the only way to access the files will be by using aws cli s3 along with the --sse switch, or by adding ‘x-amz-server-side-encryption’ to your API calls.

Oracle Roles vs. PostgreSQL Roles

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Syntax and option differences, similar functionality There are no users - only roles in PostgreSQL

Oracle Usage

Oracle roles are groups of privileges 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 operation. 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 and local roles:

- **Common Roles:** 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). Common roles are useful for cross-container operations such as ensuring a common user has a role in every container.
- **Local Roles:** 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. Beginning with Oracle 12.1.0.2, these prefixes can be changed using the COMMON_USER_PREFIX parameter.
- A CONTAINER clause can be added to CREATE ROLE statement to choose the container applicable for the role.

Example

Create a common role.

```
show con_name

CON_NAME
-----
CDB$ROOT

CREATE ROLE c##common_role;

Role created.
```

Create a local role.

```
show con_name
```

```
CON_NAME
-----
ORCLPDB

CREATE ROLE local_role;

Role created.
```

Grant privileges and roles to the local_role database role.

```
GRANT RESOURCE, ALTER SYSTEM, SELECT ANY DICTIONARY TO local_role;
```

Database users to which the local_role role is granted now have all privileges that were granted to the role.

Revoke privileges and roles from the local_role database role.

```
REVOKE RESOURCE, ALTER SYSTEM, SELECT ANY DICTIONARY FROM local_role;
```

For more details, see <https://docs.oracle.com/en/database/oracle/oracle-database/19/dbseg/configuring-privilege-and-role-authorization.html#GUID-89CE989D-C97F-4CFD-941F-18203090A1AC>

PostgreSQL Usage

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.

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.

Oracle	PostgreSQL
Common database user (12c)	Database role with Login
Local database user (12c)	N/A
Database user (11g)	Database role with Login
Database role	Database role without Login
Database users are identical to schema	Database users and schemas are created separately

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

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:

```
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.

Create a role that can login to the database and specify a password:

```
CREATE USER test_user1 WITH PASSWORD 'password';

CREATE ROLE test_user2 WITH LOGIN PASSWORD 'password';
```

Note:

- CREATE USER is identical to CREATE ROLE, except that it implies a login to the database.
- When we provision a new Amazon Aurora cluster, a master user is created as the most powerful user in the database:

The screenshot shows the 'Settings' section of the AWS Aurora configuration interface. It includes fields for 'DB Instance Identifier*' containing 'my_aurora01', 'Master Username*' containing 'super_user', and two fields for 'Master Password*' and 'Confirm Password*' both containing masked values.

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';
```

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;
```

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);
```

Summary

Description	Oracle	PostgreSQL
List all roles	SELECT * FROM dba_roles;	SELECT * FROM pg_roles;
Create a new role	CREATE ROLE c##common_role; Or CREATE ROLE local_role1;	CREATE ROLE test_role;
Grant one role privilege to another database role	GRANT local_role1 TO local_role2;	grant myrole1 to myrole2;
Grant privileges on a database object to a database role	GRANT CREATE TABLE TO local_role;	GRANT create ON DATABASE postgresdb to test_user;
Grant DML permissions on a database object to a role	hr.employees to myrole1;	GRANT INSERT, DELETE ON hr.employees to myrole1;
List all database users	SELECT * FROM dba_users;	SELECT * FROM pg_user;
Create a database user	CREATE USER c##test_user IDENTIFIED BY test_user;	CREATE ROLE test_user WITH LOGIN PASSWORD 'test_user';
Change the password for a database user	ALTER USER c##test_user IDENTIFIED BY test_user;	ALTER ROLE test_user WITH LOGIN PASSWORD 'test_user';
External authentication	Supported via Externally Identified Users	Currently not supported; future support for AWS Identity and Access Management (IAM) users is possible
Tablespace quotas	Alter User c##test_user QUOTA UNLIMITED ON TABLESPACE users;	Not supported
Grant role to user	GRANT my_role TO c##test_user;	GRANT my_role TO test_user;
Lock user	ALTER USER c##test_user ACCOUNT LOCK;	ALTER ROLE test_user WITH NOLOGIN;
Unlock user	ALTER USER c##test_user ACCOUNT UNLOCK;	ALTER ROLE test_user WITH LOGIN;
Grant privileges	GRANT CREATE TABLE TO c##test_user;	GRANT create ON DATABASE postgres to test_user;
Default tablespace	ALTER USER C##test_user default tablespace users;	ALTER ROLE test_user SET default_tablespace = 'pg_global';
Grant select privilege on a table	GRANT SELECT ON hr.employees to c##test_user;	GRANT SELECT ON hr.employees to test_user;
Grant DML privileges on a table	GRANT INSERT,DELETE ON hr.employees to c##test_user;	GRANT INSERT,DELETE ON hr.employees to test_user;
Grant execute	GRANT EXECUTE ON hr.procedure_name to c##test_user;	grant execute on function "newdate()" to test_user; Inside the brackets "()" - specify the arguments types for the function
Limits user connection	CREATE PROFILE app_users LIMIT SESSIONS_PER_USER 5; ALTER	ALTER ROLE test_user WITH CONNECTION LIMIT 5;

Description	Oracle	PostgreSQL
	USER C##TEST_USER PROFILE app_users;	
Create a new database schema	CREATE USER my_app_schema IDENTIFIED BY password;	CREATE SCHEMA my_app_schema;

For more details, see: <https://www.postgresql.org/docs/13/static/sql-createrole.html>

Oracle Database Users vs. PostgreSQL Users

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Syntax and option differences, similar functionality There are no users - only roles in PostgreSQL

Oracle Usage

Database user accounts are used for authenticating connecting sessions and authorizing access for individual users to specific database objects. Database Administrators grant privileges to user accounts, and applications use user accounts to access database objects.

Steps for Providing Database Access to Applications

1. Create a user account in the database. User accounts are typically authenticated using a password. Additional methods of authenticating users also exist.
2. Assign permissions to the database user account enabling access to certain database objects and system permissions.
3. Connecting applications authenticate using the database username and password.

Oracle Database Users Common Properties

- Granting privileges or roles (collection of privileges) to the database user.
- Defining the default database tablespace for the user.
- Assigning tablespace quotas for the user.
- Configuring password policy, password complexity, lock, or unlock the account.

Authentication Mechanisms

- **Username and Password:** (default)
- **External:** Using the operating system or third-party software (such as Kerberos).
- **Global:** Enterprise directory service (such as Active Directory or Oracle Internet Directory).

Oracle Schemas Compared to Users

In an Oracle database, a user equals a schema. This relationship is special because 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 in which to store database objects.
- Newly created schemas are empty, but objects such as tables can be created within them.

Database Users in Oracle 12c

Two types of users exist in the Oracle 12c database:

- **Common Users:** Created in all database containers, root, and Pluggable Databases (PDB). Common users 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

The example below demonstrates the following operations:

- Create a common database user using the default tablespace.
- Grant privileges and roles to the user.
- Assign a profile to the user, unlock the account, and force the user to change the password (PASSWORD EXPIRE).
- 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 more details, see <https://docs.oracle.com/database/121/DBSEG/users.htm>

PostgreSQL Usage

In PostgreSQL there are no users, only roles, role with connect privilege can be considered as a user.

Please refer to the Role topic: [PostgreSQL Role](#)

Physical Storage

Oracle Table Partitioning vs. PostgreSQL Partitions and Table Inheritance

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	<p>Foreign keys referencing to/from partitioned tables are supported on the individual tables in PostgreSQL</p> <p>Some partition types are not supported by PostgreSQL</p>

Oracle Usage

The purpose of database partitioning is to provide support for very large tables and indexes by splitting them into smaller pieces. Each partition has its own name and definitions. They can be managed separately or collectively as one object. From an application perspective, partitions are transparent. Partitioned tables behave the same as non-partitioned tables allowing your applications access using unmodified SQL statements. Table partitioning provides several benefits:

- **Performance Improvements:** Table partitions help improve query performance by accessing a subset of a partition instead of scanning a larger set of data. Additional performance improvements can 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 18c introduces following enhancements to partitioning:

- Online Merging of Partitions and Subpartitions: now it is possible to merge table partitions concurrently with Updates/Deletes and Inserts on a partitioned table.
- Oracle 18c also allows to modify partitioning strategy for the partitioned table: e.g. hash partitioning to range. This can be done both offline and online.

Oracle 19 introduces hybrid partitioned tables: partitions can now be both internal Oracle tables and external tables and sources. It is also possible to integrate both internal and external partitions together in a single partitioned table.

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. The partitions have approximately the same size.

For example, create a hash partitioned table.

```
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.

For example, create a list-partitioned table.

```
CREATE TABLE SYSTEM_LOGS
  (EVENT_NO      NUMBER NOT NULL,
   EVENT_DATE    DATE    NOT NULL,
   EVENT_STR     VARCHAR2(500),
   ERROR_CODE    VARCHAR2(10))
PARTITION BY LIST (ERROR_CODE)
(PARTITION warning VALUES ('err1', 'err2', 'err3') TABLESPACE TB1,
 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, primarily with date values. Range table partitioning can also be implemented with numeric ranges (1-10000, 10001- 20000...).

For example, create a range-partitioned table.

```
CREATE TABLE SYSTEM_LOGS
  (EVENT_NO      NUMBER NOT NULL,
   EVENT_DATE    DATE    NOT NULL,
   EVENT_STR     VARCHAR2(500))
PARTITION BY RANGE (EVENT_DATE)
(PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2015',
                                              'DD/MM/YYYY')) TABLESPACE TB1,
 PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2016',
                                              'DD/MM/YYYY')) TABLESPACE TB2,
```

```
PARTITION EVENT_DATE VALUES LESS THAN (TO_DATE('01/01/2017',
    '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 include:

- Manageability extensions
 - Interval partitioning
 - Partition advisor
- Partitioning key extensions
 - Reference partitioning
 - Virtual column-based partitioning

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.

```
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.

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

Subpartitioning Tables

You can create Subpartitions within partitions to further split the parent partition.

```
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)

```

For more details, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/vlbg/partition-concepts.html#GUID-EA7EF5CB-DD49-43AF-889A-F83AAC0D7D51>

Automatic List Partitioning (Oracle 12c only)

Automatic list partitioning is an enhancement that enables automatic creation of new partitions for new values inserted into a list-partitioned table. An automatic list-partitioned table is created with only one partition. The database creates the additional table partitions automatically.

For example, Create an automatic list-partitioned table.

```

CREATE TABLE SYSTEM_LOGS
  (EVENT_NO      NUMBER NOT NULL,
   EVENT_DATE    DATE    NOT NULL,
   EVENT_STR     VARCHAR2(500),
   ERROR_CODE    VARCHAR2(10))
  PARTITION BY LIST (ERROR_CODE) AUTOMATIC
  (PARTITION warning VALUES ('err1', 'err2', 'err3'))

```

For more details, see

<http://www.oracle.com/technetwork/database/options/partitioning/partitioning-wp-12c-1896137.pdf>

PostgreSQL Usage

Starting in PostgreSQL 10, there is an equivalent option to Oracle Partitions when using RANGE or LIST partitions, as declarative partitions are being supported in PostgreSQL.

Prior to PostgreSQL 10, the table partitioning mechanism in PostgreSQL differed from Oracle. Partitioning in PostgreSQL was implemented using “table inheritance”. Each table partition was represented by a child table which was referenced to a single parent table. The parent table remained empty and was only used to represent the entire table data set (as a meta-data dictionary and as a query source).

In PostgreSQL 10, you still need to create the partition tables manually, but you do not need to create triggers or functions to redirect data to the right partition.

Some of the Partitioning management operations are performed directly on the sub-partitions (sub-tables). Querying can be performed directly on the partitioned table itself.

Starting with PostgreSQL 11 following features were added:

- For partitioned tables, a default partition can now be created that will store data which can't be redirected to any other explicit partitions
- In addition to partitioning by ranges and lists, tables can now be partitioned by a hashed key.
- When UPDATE changes values in a column that's used as partition key in partitioned table, data is moved to proper partitions.
- An index can now be created on a partitioned table. Corresponding indexes will be automatically created on individual partitions.
- Foreign keys can now be created on a partitioned table. Corresponding foreign key constraints will be propagated to individual partitions
- Triggers FOR EACH ROW can now be created on a partitioned table. Corresponding triggers will be automatically created on individual partitions as well.
- When attaching or detaching new partition to a partitioned table with the foreign key, foreign key enforcement triggers are correctly propagated to a new partition.

For more information, see: <https://www.postgresql.org/docs/13/static/ddl-inherit.html>, <https://www.postgresql.org/docs/11/ddl-partitioning.html>

Using The Partition Mechanism

List Partition

```

CREATE TABLE emps (
    emp_id      SERIAL NOT NULL,
    emp_name     VARCHAR(30) NOT NULL
PARTITION BY LIST (left(lower(emp_name), 1));

CREATE TABLE emp_abc
    PARTITION OF emps (
        CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
    ) FOR VALUES IN ('a', 'b', 'c');

CREATE TABLE emp_def
    PARTITION OF emps (
        CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
    ) FOR VALUES IN ('d', 'e', 'f');

INSERT INTO emps VALUES (DEFAULT, 'Andrew');

row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Chris');

row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Frank');

row inserted.

INSERT INTO emps VALUES (DEFAULT, 'Pablo');

SQL Error [23514]: ERROR: no partition of relation "emps" found for row
Detail: Partition key of the failing row contains ("left"(lower(emp_name::text), 1))
= (p).

```

Note: To prevent the above error, ensure that all partitions exist for all possible values in the column that partitions the table. The default partition feature was added in PostgreSQL 11.

Note: Use the MAXVALUE and MINVALUE in your FROM/TO clause. This can help you get all values with RANGE partitions without the risk of creating new partitions.

Range Partition

```

CREATE TABLE sales (
    saledate    DATE NOT NULL,
    item_id     INT,
    price       FLOAT
) PARTITION BY RANGE (saledate);

CREATE TABLE sales_2018q1
    PARTITION OF sales (
        price DEFAULT 0
    ) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');

CREATE TABLE sales_2018q2
    PARTITION OF sales (
        price DEFAULT 0
    ) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_2018q3
    PARTITION OF sales (
        price DEFAULT 0
    ) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');

INSERT INTO sales VALUES ('2018-01-08'),3121121, 100);
row inserted.

INSERT INTO sales VALUES ('2018-04-20'),4378623);
row inserted.

INSERT INTO sales VALUES ('2018-08-13'),3278621, 200);
row inserted.

```

When creating a table with PARTITION OF clause, you can still use the "PARTITION BY" clause with it. Using the "PARTITION BY" clause will create a sub-partition.

A sub-partition can be the same type as the parent partition table or it can be another partition type.

List Combined With Range Partition

In the example below is an example of creating a LIST partition and sub-partitions by RANGE

```

CREATE TABLE salers (
    emp_id          serial not null,
    emp_name        varchar(30) not null,
    sales_in_usd   int not null,
    sale_date       date not null
) PARTITION BY LIST (left(lower(emp_name), 1));

CREATE TABLE emp_abc
    PARTITION OF salers (
        CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
    ) FOR VALUES IN ('a', 'b', 'c') PARTITION BY RANGE (sale_date);

CREATE TABLE emp_def
    PARTITION OF salers (
        CONSTRAINT emp_id_nonzero CHECK (emp_id != 0)
    ) FOR VALUES IN ('d', 'e', 'f') PARTITION BY RANGE (sale_date);

CREATE TABLE sales_abc_2018q1
    PARTITION OF emp_abc (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');

CREATE TABLE sales_abc_2018q2
    PARTITION OF emp_abc (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_abc_2018q3
    PARTITION OF emp_abc (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');

CREATE TABLE sales_def_2018q1
    PARTITION OF emp_def (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-01-01') TO ('2018-03-31');

CREATE TABLE sales_def_2018q2
    PARTITION OF emp_def (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-04-01') TO ('2018-06-30');

CREATE TABLE sales_def_2018q3
    PARTITION OF emp_def (
        sales_in_usd DEFAULT 0
    ) FOR VALUES FROM ('2018-07-01') TO ('2018-09-30');

```

Implementing List “Table Partitioning” with inheritance tables

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 to define allowed values in each table (for example, primary keys or check constraints).
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 ensures that the queries are optimized for working with table partitions.

```
show constraint_exclusion;
constraint_exclusion
-----
partition
```

For additional information on PostgreSQL constraint_exclusion parameter: [https://www.-postgresql.org/docs/13/static/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION](https://www.postgresql.org/docs/13/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 for replacing Oracle’s interval table partitioning include using application-centric methods using PL/pgSQL or other programming languages.

Note:

- 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 or creating these on the individual tables .
- PostgreSQL does not support SPLIT and EXCHANGE of table partitions. For these actions, you will need to plan your data migrations manually (between tables) to re-place the data into the right partition.

Examples

Steps for creating a PostgreSQL “list-partitioned table”:

1. Create the parent table:

```
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:

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

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”):

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

4. Create a function to redirect data inserted into the parent table:

```
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:

```
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:

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

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

```
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...

SELECT * FROM SYSTEM_LOGS_WARNING;
event_no | event_date | event_str | error_code
-----+-----+-----+-----
1 | 2015-05-15 | a... | err1
2 | 2016-06-16 | b... | err3

SELECT * FROM SYSTEM_LOGS_CRITICAL;
event_no | event_date | event_str | error_code
-----+-----+-----+-----
3 | 2017-07-17 | c... | err6
```

Steps for creating a PostgreSQL “range-partitioned table”:

1. Create the parent table:

```
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:

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

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

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”):

```
CREATE INDEX IDX_SYSTEM_LOGS_2015 ON SYSTEM_LOGS_2015(EVENT_DATE);
CREATE INDEX IDX_SYSTEM_LOGS_2016 ON SYSTEM_LOGS_2016(EVENT_DATE);
CREATE INDEX IDX_SYSTEM_LOGS_2017 ON SYSTEM_LOGS_2017(EVENT_DATE);
```

4. Create a function to redirect data inserted into the parent table:

```
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:

```
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:

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

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

```
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...

SELECT * FROM SYSTEM_LOGS_2015;
event_no | event_date | event_str
-----+-----+-----
1 | 2015-05-15 | a...
```

Examples of new partitioning features of PostgreSQL11

Default partitions

```
CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);
CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);
CREATE TABLE tst_part_dflt PARTITION OF tst_part DEFAULT; INSERT INTO tst_part SELECT
generate_series(1,10,1);
SELECT * FROM tst_part1;i
---
1
2
3
4
4 rows)

SELECT * FROM tst_part_dflt;
i
-----
5
6
7
8
9
10
(6 rows)
```

Hash partitioning:

```

CREATE TABLE tst_hash(i INT) PARTITION BY HASH(i);
CREATE TABLE tst_hash_1 PARTITION OF tst_hash FOR VALUES WITH (MODULUS 2, REMAINDER
0);
CREATE TABLE tst_hash_2 PARTITION OF tst_hash FOR VALUES WITH (MODULUS 2, REMAINDER
1);

INSERT INTO tst_hash SELECT generate_series(1,10,1);

SELECT * FROM tst_hash_1;
i
---
1
2
(2 rows)

SELECT * FROM tst_hash_2;
i
---
3
4
5
6
7
8
9
10
(8 rows)

```

UPDATE on partition key:

```

CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);
CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);
CREATE TABLE tst_part_dflt PARTITION OF tst_part DEFAULT;

INSERT INTO tst_part SELECT generate_series(1,10,1);

SELECT * FROM tst_part1;
i
---
1
2
3
4
(4 rows)

SELECT * FROM tst_part_dflt;
i
---
5
6
7

```

```

8
9
10
(6 rows)

UPDATE tst_part SET i=1 WHERE i IN (5,6);

SELECT * FROM tst_part_dfl;
i
-----
7
8
9
10
(4 rows)

SELECT * FROM tst_part1;
i
---
1
2
3
4
1
1
(6 rows)

```

Index propagation on partitioned tables:

```

CREATE TABLE tst_part(i INT) PARTITION BY RANGE(i);

CREATE TABLE tst_part1 PARTITION OF tst_part FOR VALUES FROM (1) TO (5);

CREATE TABLE tst_part2 PARTITION OF tst_part FOR VALUES FROM (5) TO (10);

CREATE INDEX tst_part_ind ON tst_part(i);

\dt+ tst_part
               Partitioned table "public.tst_part"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
---- 
 i      | integer |           |           | plain    |          |
Partition key: RANGE (i)
Indexes:
  "tst_part_ind" btree (i)
Partitions: tst_part1 FOR VALUES FROM (1) TO (5),
            tst_part2 FOR VALUES FROM (5) TO (10)

\dt+ tst_part1
               Table "public.tst_part1"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+

```

```

-----+-----+-----+-----+-----+-----+
---  

 i | integer | | plain |  

Partition of: tst_part FOR VALUES FROM (1) TO (5)  

Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))  

Indexes:  

 "tst_part1_i_idx" btree (i)  

Access method: heap  

\dt+ tst_part2  

Table "public.tst_part2"  

Column | Type | Collation | Nullable | Default | Storage | Stats target | Description  

-----+-----+-----+-----+-----+-----+-----+-----+  

---  

 i | integer | | plain |  

Partition of: tst_part FOR VALUES FROM (5) TO (10)  

Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))  

Indexes:  

 "tst_part2_i_idx" btree (i)  

Access method: heap

```

Foreign keys propagation on partitioned tables:

```

CREATE TABLE tst_ref(i INT PRIMARY KEY);  

ALTER TABLE tst_part ADD CONSTRAINT tst_part_fk FOREIGN KEY (i) REFERENCES tst_ref(i);  

\dt+ tst_part  

Partitioned table "public.tst_part"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description  

-----+-----+-----+-----+-----+-----+-----+-----+  

---  

 i | integer | | plain |  

Partition key: RANGE (i)  

Indexes:  

 "tst_part_ind" btree (i)  

Foreign-key constraints:  

 "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)  

Partitions: tst_part1 FOR VALUES FROM (1) TO (5),  

          tst_part2 FOR VALUES FROM (5) TO (10)  

\dt+ tst_part1  

Table "public.tst_part1"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description  

-----+-----+-----+-----+-----+-----+-----+-----+  

---  

 i | integer | | plain |  

Partition of: tst_part FOR VALUES FROM (1) TO (5)  

Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))  

Indexes:  

 "tst_part1_i_idx" btree (i)

```

```

Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Access method: heap

\dt+ tst_part2
                         Table "public.tst_part2"
 Column | Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
---- i   | integer |           |          | plain   |          |
Partition of: tst_part FOR VALUES FROM (5) TO (10)
Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))
Indexes:
  "tst_part2_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Access method: heap

```

Triggers propagation on partitioned tables:

```

CREATE TRIGGER some_trigger AFTER UPDATE ON tst_part FOR EACH ROW EXECUTE FUNCTION
some_func();

\dt+ tst_part
                         Partitioned table "public.tst_part"
 Column | Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
---- i   | integer |           |          | plain   |          |
Partition key: RANGE (i)
Indexes:
  "tst_part_ind" btree (i)
Foreign-key constraints:
  "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:
  some_trigger AFTER UPDATE ON tst_part FOR EACH ROW EXECUTE FUNCTION some_func()
Partitions: tst_part1 FOR VALUES FROM (1) TO (5),
            tst_part2 FOR VALUES FROM (5) TO (10)

\dt+ tst_part1
                         Table "public.tst_part1"
 Column | Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
---- i   | integer |           |          | plain   |          |
Partition of: tst_part FOR VALUES FROM (1) TO (5)
Partition constraint: ((i IS NOT NULL) AND (i >= 1) AND (i < 5))
Indexes:
  "tst_part1_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:

```

```

some_trigger AFTER UPDATE ON tst_part1 FOR EACH ROW EXECUTE FUNCTION some_func()
Access method: heap

\dt+ tst_part2
Table "public.tst_part2"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
i     | integer |          |          | plain   |          |
Partition of: tst_part FOR VALUES FROM (5) TO (10)
Partition constraint: ((i IS NOT NULL) AND (i >= 5) AND (i < 10))
Indexes:
  "tst_part2_i_idx" btree (i)
Foreign-key constraints:
  TABLE "tst_part" CONSTRAINT "tst_part_fk" FOREIGN KEY (i) REFERENCES tst_ref(i)
Triggers:
  some_trigger AFTER UPDATE ON tst_part2 FOR EACH ROW EXECUTE FUNCTION some_func()
Access method: heap

```

Summary

Oracle Table Partition Type	Build-In PostgreSQL Support
List	Yes
Range	Yes
Hash	Yes
Composite Partitioning (sub Partitioning)	No
Interval Partitioning	No
Partition Advisor	No
Reference Partitioning	No
Virtual Column Based Partitioning	No
Automatic List Partitioning	No
Split / Exchange Partitions	No

For additional details: <https://www.postgresql.org/docs/13/static/ddl-partitioning.html>

Oracle Sharding vs. PostgreSQL Sharding

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
		N/A	PostgreSQL doesn't support sharding

Oracle Usage

Sharding is a method of data architecture where table data is horizontally partitioned across independent databases. These databases are called shards. All of the shards make up a single logical database, which is referred to as a sharded database (SDB). Sharding a table is process of splitting this table between different shards where each shards will have sharded table with the same structure but different subset of rows.

Oracle 18c introduces following sharding enhancements:

- User-defined sharding. Before Oracle 18c, data was redirected across shards by system. With user-defined sharding, users are now able to explicitly redirect sharded table data to specific individual shards.
- Using JSON, BLOB, CLOB and spatial objects functionality in a sharded environment. These objects can now be used in sharded tables.

For additional details, see: <https://docs.oracle.com/en/database/oracle/oracle-database/19/shard/sharding-overview.html#GUID-0F39B1FB-DCF9-4C8A-A2EA-88705B90C5BF>

Monitoring

Oracle V\$ Views and the Data Dictionary vs. PostgreSQL System Catalog & The Statistics Collector

Feature Compatibility	SCT/DMS Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	Table names in queries need to be changed in PostgreSQL

Oracle Usage

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_USERES** - 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, see:

<https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-dictionary-and-dynamic-performance-views.html#GUID-BDF5B748-EB43-4B48-938E-89099069C3BB>

PostgreSQL Usage

PostgreSQL provides three different sets of meta-data 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.

Category	Description
Statistic collection views	Subsystem that collects runtime dynamic information about certain server activities such as statistical performance information. Some of these tables could be thought as comparable to Oracle V\$ views.
System catalog tables	Static metadata regarding the PostgreSQL database and static information about schema objects. Some of these tables could be thought as comparable to Oracle DBA_* Data Dictionary tables.
Information schema tables	Set of views that contain information about the objects defined in the current database. The information schema is specified by the SQL standard and as such, supported by PostgreSQL. Some of these tables could be thought as comparable to Oracle USER_* Data Dictionary tables.
Advance performance monitoring	Use the Performance Insights Console

System Catalog Tables

These are a set of tables used to store dynamic and static meta-data 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

Starting with PostgreSQL 12 it is now possible to monitor progress of CREATE INDEX and REINDEX and CLUSTER and VACUUM FULL operations by querying system views pg_stat_progress_create_index and pg_stat_progress_cluster

RDS ONLY: Starting with PostgreSQL 13 following features have been added:

1. It is now possible to monitor progress of ANALYZE operations by querying system view pg_stat_progress_analyze
2. It is now possible to monitor shared memory usage with system view pg_shmem_allocations

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 name	Purpose
pg_database	Contains information and properties about each database in the PostgreSQL cluster, such as the database encoding settings as well as others.
pg_tables	Information about all tables in the database, such as indexes and the tablespace for each database table.
pg_index	Contains information about all indexes in the database.
pg_cursors	List of currently available/open cursors .

New catalog tables/views in PostgreSQL 10:

- pg_publication - all publications created in the database
- pg_partitioned_table - all partitioned tables in the database
- pg_sequences - all sequences
- pg_statistic_ext - holds table statistics
- pg_subscription - all existing logical replication subscriptions **across the cluster**
- pg_hba_file_rules - summary of the contents of the client authentication configuration file
- pg_publication_tables - mapping between publications and the tables they contain

For additional details:

<https://www.postgresql.org/docs/13/static/catalogs.html>

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.

```
SELECT * FROM pg_stat_activity WHERE STATE = 'active';
```

Common statistics collector views include:

Table name	Purpose
pg_stat_activity	Statistics of currently sessions in the database. Useful for identifying long running queries
pg_stat_all_tables	Performance statistics on all tables in the database, such as identifying table size, write activity, full scans vs. index access, etc.
pg_statio_all_tables	Performance statistics and I/O metrics on all database tables
pg_stat_database	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).
pg_stat_bgwriter	Important performance information on PostgreSQL checkpoints and background writes
pg_stat_all_indexes	Performance and usage statistics on indexes, for example, useful for identifying unused indexes

For additional details:

<https://www.postgresql.org/docs/13/static/monitoring-stats.html#MONITORING-STATS-DYNAMIC-VIEWS-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.

```
select * from information_schema.tables;
```

For additional details: <https://www.postgresql.org/docs/13/static/information-schema.html>

Note: By default, all database users (public) can query both the system catalog tables, the statistics collector views and the information schema.

Summary

Information	Oracle	PostgreSQL
Database properties	V\$DATABASE	PG DATABASE
Database sessions	V\$SESSION	PG_STAT_ACTIVITY
Database users	DBA_USERS	PG_USER
Database tables	DBA_TABLES	PG_TABLES
Database roles	DBA_ROLES	PG_ROLES
Table columns	DBA_TAB_COLS	PG_ATTRIBUTE
Database locks	V\$LOCKED_OBJECT	PG_LOCKS
Currently configured runtime parameters	V\$PARAMETER	PG_SETTINGS
All system statistics	V\$SYSSTAT	PG_STAT_DATABASE
Privileges on tables	DBA_TAB_PRIVS	TABLE_PRIVILEGES
Information about IO operations	V\$SEGSTAT	PG_STATIO_ALL_TABLES

Amazon RDS performance Insights

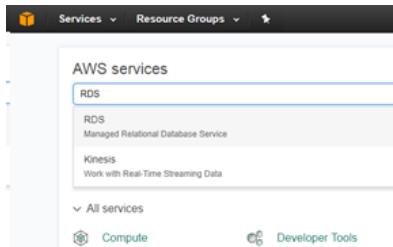
In addition to monitoring database status and activity using queries on metadata tables, Aurora PostgreSQL provides a visual performance monitoring and status information via the “Performance Insights” feature accessible as part of the Amazon RDS Management Console.

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.

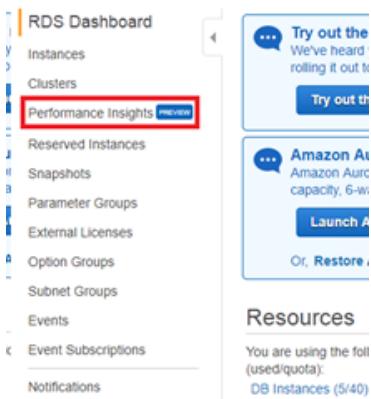
Example

Accessing the Amazon Aurora Performance Insights Console

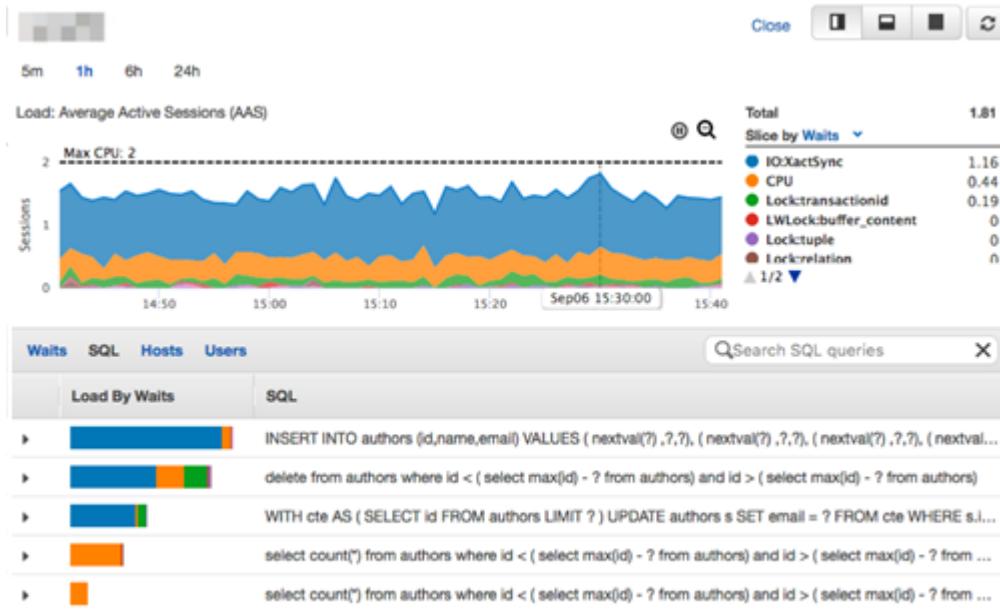
1. Navigate to the RDS section of the AWS Console.



2. Select Performance Insights.



3. 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:

http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PerfInsights.html

Migration Quick Tips

Migration Quick Tips

This section provides migration tips that can help save time as you transition from Oracle to Aurora PostgreSQL. They address many of the challenges faced by administrators new to Aurora PostgreSQL. Some of these tips describe functional differences in similar features between Oracle and Aurora PostgreSQL.

Management

- In Aurora PostgreSQL , the term "Database Snapshot" is equivalent to Oracle RMAN backup
- Partitioning in Aurora PostgreSQL is called "INHERITS" tables and act completely different in terms of management
- Unlike Oracle's statistics, Aurora PostgreSQL does not collect detailed key value distribution; it relies on selectivity only. When troubleshooting execution, be aware that parameter values are insignificant to plan choices.
- Many missing features such as sending emails can be achieved with quick implementations of Amazon's services (like Lambda).
- Parameters and backups are managed by Amazon's RDS,. It is very useful in terms of checking parameter's value against its default and comparing them to another parameter group.
- High Availability can be implemented in few clicks to create Replicas.
- With Database Links, there are two options. The db_link extension is similar to Oracle and the postgres_fdw extension for using Foreign Data Wrapper.

SQL

- Triggers work differently in Aurora PostgreSQL. The syntax for inserted and deleted for each row is NEW and OLD.
- Aurora PostgreSQL does not support many cursors status checks. When declaring cursors in Aurora PostgreSQL , you must create an explicit HANDLER object.
- To execute a stored procedure (functions), use SELECT instead of EXECUTE.
- To execute a string as a query, use Aurora PostgreSQL Prepared Statements instead of EXECUTE (<String>) syntax.
- In Aurora PostgreSQL , IF blocks must be terminated with END IF. WHILE..LOOP loops must be terminated with END LOOP.
- Unlike Oracle, in Aurora PostgreSQL auto commit is ON, make sure to set it to OFF if you want to make the database behavior more similar to Oracle.
- Aurora PostgreSQL does not use special data types for UNICODE data. All string types may use any character set and any relevant collation.
- Collations can be defined at the server, database, and column level, similar to Oracle. They cannot be defined at the table level.
- Oracle's DELETE <Table Name> syntax, which allows omitting the FROM keyword, is invalid in Aurora PostgreSQL . Add the FROM keyword to all delete statements.
- Aurora PostgreSQL SERIAL column property is similar to IDENTITY in Oracle.
- Error handling in Aurora PostgreSQL has less features, but for special requirements, you can log or send alerts by inserting into tables or catching errors.

- Aurora PostgreSQL does not support the MERGE statement. Use the REPLACE statement and the INSERT... ON DUPLICATE KEY UPDATE statement as alternatives.
- You can concatenate strings in Aurora PostgreSQL using the "||" operator, like in Oracle
- Aurora PostgreSQL is much stricter than Oracle in terms of statement terminators. Be sure to always use a semicolons at the end of statements.
- There is no CREATE PROCEDURE syntax; only CREATE FUNCTION. You can create a function that returns void.
- Keep in mind that the window functions GREATEST and LEAST might get different results than the results that might be returned in Oracle from using these functions.
- PostgreSQL doesn't support SAVEPOINT, ROLLBACK TO SAVEPOINT inside of functions
- Aurora PostgreSQL does not support BFILE, ROWID, UROWID data types, try to use other data types.
- Temporary tables are being kept only for the session level and only the session that created the table can query the temporary table.
- PostgreSQL does not support unused or virtual columns, there is no workaround for replacing unused columns, for using similar functionality to the virtual columns, views and functions can be combined
- PostgreSQL does not support automatic or incremental REFRESH for Materialized Views, use triggers instead.
- Explore AWS to locate which features can be replaced with Amazon's services, this can help you with managing your database and decrease costs.
- The architecture in PostgreSQL allows you have multiple databases in a single instance, this can be relevant for consolidation projects.
- Beware of control characters when copying and pasting a script to Aurora PostgreSQL clients. Aurora PostgreSQL is much more sensitive to control characters than Oracle and they result in frustrating syntax errors that are hard to find.

Glossary

ACID

Atomicity, Consistency, Isolation, Durability

AES

Advanced Encryption Standard

ANSI

American National Standards Institute

API

Application Programming Interface

ARN

Amazon Resource Name

AWS

Amazon Web Services

BLOB

Binary Large Object

CDATA

Character Data

CLI

Command Line Interface

CLOB

Character Large Object

CLR

Common Language Runtime

CPU

Central Processing Unit

CRI

Cascading Referential Integrity

CSV

Comma Separated Values

CTE

Common Table Expression

DB

Database

DBCC

Database Console Commands

DDL

Data Definition Language

DEK

Database Encryption Key

DES

Data Encryption Standard

DML

Data Manipulation Language

DQL

Data Query Language

FCI

Failover Cluster Instances

HADR

High Availability and Disaster Recovery

IAM

Identity and Access Management

IP

Internet Protocol

ISO

International Organization for Standardization

JSON

JavaScript Object Notation

KMS

Key Management Service

NUMA

Non-Uniform Memory Access

OLE

Object Linking and Embedding

OLTP

Online Transaction Processing

PaaS

Platform as a Service

PDF

Portable Document Format

QA

Quality Assurance

RDMS

Relational Database Management System

RDS

Amazon Relational Database Service

REGEXP

Regular Expression

SCT

Schema Conversion Tool

SHA

Secure Hash Algorithm

SLA

Service Level Agreement

SMB

Server Message Block

SQL

Structured Query Language

SQL/PSM

SQL/Persistent Stored Modules

SSD

Solid State Disk

SSH

Secure Shell

T-SQL

Transact-SQL

TDE

Transparent Data Encryption

UDF

User Defined Function

UDT

User Defined Type

UTC

Universal Time Coordinated

WMI

Windows Management Instrumentation

WQL

Windows Management Instrumentation Query Language

WSFC

Windows Server Failover Clustering

XML

Extensible Markup Language