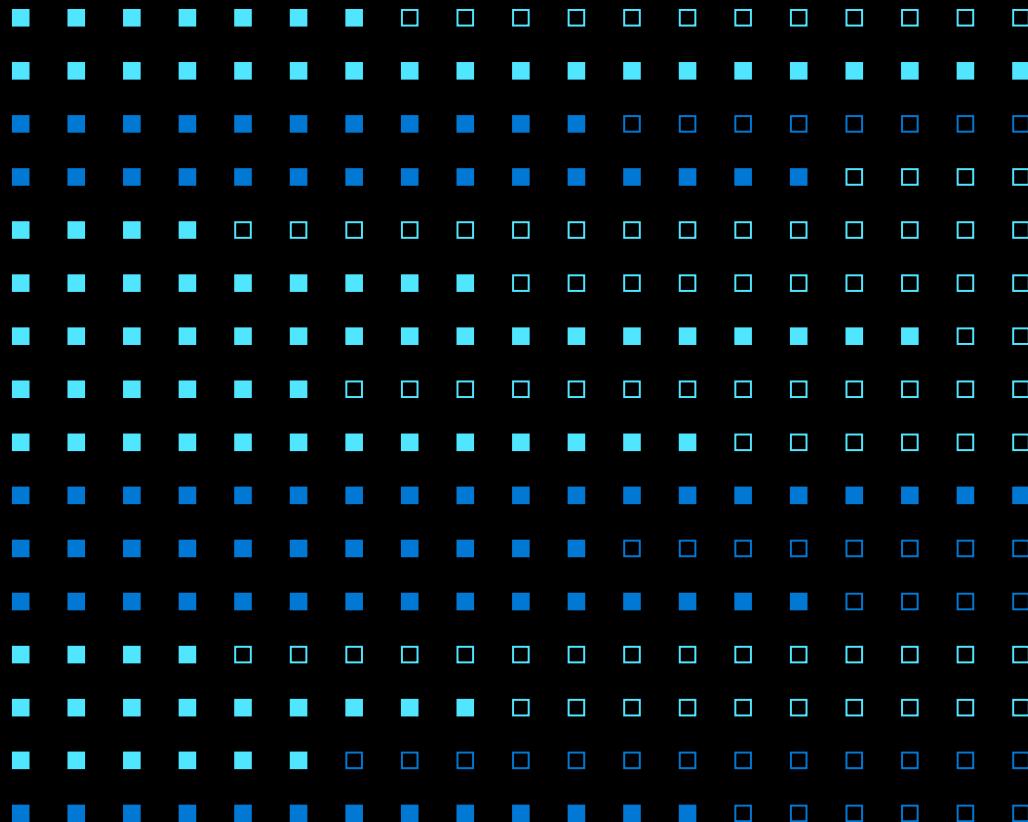


# Oracle to Azure Database for PostgreSQL Migration Guide

[Technical white paper](#)

**Published:** November 2020

**Applies to:** Oracle 10g & above and Azure Database for PostgreSQL





## Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. This content was developed prior to the product or service' release and as such, we cannot guarantee that all details included herein will be exactly as what is found in the shipping product. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. The information represents the product or service at the time this document was shared and should be used for planning purposes only.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Information subject to change at any time without prior notice.

Microsoft, Active Directory, Azure, Bing, Excel, Power BI, SharePoint, Silverlight, SQL Server, Visual Studio, Windows, and Windows Server are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

© 2020 Microsoft Corporation. All rights reserved.



## Contents

Introduction and Concepts.....	15
Migration Guide Summary Table.....	15
Oracle to Azure Database for PostgreSQL: Migration Project Overview.....	22
Main Stages High-Level Overview.....	22
Database Assessment/Discovery.....	23
Identify Success Criteria and Migration Risk.....	23
Azure Database PostgreSQL Target Environment Setup.....	23
Schema Conversion .....	24
Data Migration.....	24
Ora2Pg.....	24
Ora2Pg Installation.....	25
Ora2Pg Post-Installation Actions.....	25
Generating a Migration Project.....	25
Ora2Pg Assessment Report .....	28
Ora2Pg Migration Effort Estimation.....	28
Ora2Pg Oracle Schema Automatic Conversion .....	29
Ora2Pg Data Extraction.....	31
Setting an Azure Database for PostgreSQL as a Migration Target.....	32
Ora2Pg Data Import .....	33
Ora2Pg Import Performance Considerations .....	34
Schema Import and Data Verification .....	34
Azure Database for PostgreSQL Import Process Monitoring.....	41
Data Migration Main Considerations.....	42
High Availability, Backup, and Recovery.....	43
Migrating from Oracle RMAN Backup and Recovery .....	43
Basic RMAN Code Examples .....	43
Oracle to Azure Database for PostgreSQL Compatibility Level .....	44
Migrating from Oracle Datapump Export/Import Utilities and Dump Files .....	45
Oracle Datapump Main Functionalities .....	45
Oracle Datapump Code Example .....	45
Migrating to Azure Database for PostgreSQL Backup and Restore.....	45



Azure Database for PostgreSQL Backup Frequency .....	46
Azure for PostgreSQL Restore Types .....	46
Azure Database for PostgreSQL Backup and Restore Code Examples .....	46
Migrating to Azure Database for PostgreSQL pg_dump and pg_restore .....	47
Oracle Datapump to PostgreSQL pg_dump and pg_restore Conversion Example .....	47
PostgreSQL Versions 12 and 13 Features and Descriptions .....	47
Migrating from Oracle RAC for High Availability Architecture .....	49
Oracle RAC Main Functionalities .....	49
Oracle RAC Architecture .....	49
Migrating from Oracle Data Guard for Disaster / Recovery Architecture .....	50
Oracle Data Guard Main Functionalities .....	50
Oracle Data Guard Architecture Diagram .....	50
Migrating to Azure Database for PostgreSQL High Availability, Scaling, and Read/Write Separation .....	50
Azure Database from PostgreSQL Deployment Types .....	50
Azure Database for PostgreSQL Hyperscale (Citus) Diagram .....	51
Azure Database for PostgreSQL High Availability - Single Server Main Components .....	52
High Availability During Planned Operations and Unplanned Downtime – Single Server .....	52
Adding Read-Replicas for Azure Database for PostgreSQL - Single Server .....	53
Azure CLI Examples .....	53
Azure Database for PostgreSQL High Availability - Hyperscale (Citus) .....	53
Summary .....	54
Migrating from Oracle Flashback Database .....	55
Oracle Flashback Code Examples .....	55
Migrating to Azure Database for PostgreSQL Backup and Restore .....	55
Azure Database for PostgreSQL Backup and Restore Main Features .....	56
Point-in-Time Restore Example Using Azure Cloud Shell .....	56
Point-in-Time Restore Example Using the Azure Console .....	56
Database Architecture, Security, and Miscellaneous Features .....	58
Migrating from Oracle Data Loading/Unload Tools (SQL*Loader) to Azure Database for PostgreSQL COPY Command .....	58
Oracle SQL*Loader Code Example .....	58
Migrating to Azure Database for PostgreSQL COPY Command .....	59
PostgreSQL COPY Command Example (PostgreSQL CLI) .....	59



PostgreSQL COPY Command Example (Client CLI) .....	59
Using the WHERE Clause .....	59
Migrating from Oracle Security Model: Users Roles and Permissions.....	61
Oracle 12c Users.....	61
Oracle Create User and Role Examples.....	61
Migrating to Azure Database for PostgreSQL Roles .....	62
Azure Database for PostgreSQL Single Instance - Adding Roles.....	62
Examples .....	62
Azure Database for PostgreSQL Hyperscale (Citus) - Adding Roles .....	63
Adding Additional Roles for a Hyperscale (Citus) Managed Service .....	64
Enabling Active Directory Authentication with Azure Database for PostgreSQL.....	65
Oracle Essential V\$ Views and Their Target Database Equivalents .....	66
Oracle Dictionary .....	66
Examples of Oracle Dictionary Views.....	66
Oracle V\$ - Dynamic Performance Views .....	66
Examples of Oracle V\$ common views: .....	67
Migrating to Azure Database for PostgreSQL System Dictionary and PostgreSQL Statistic Collector .....	67
Multi-Tenant Architecture: PDBs and CDB databases.....	72
Main Features of Oracle 12c Multi-Tenant Architecture.....	72
Oracle 12c CDB.....	72
Oracle 12c PDB.....	72
Oracle 12c Multi-Tenant Architecture Diagram.....	73
Oracle 12c Multi-Tenant Main Functionalities .....	73
Oracle 12c Code Examples.....	73
Migrating to Azure Database for PostgreSQL Databases Architecture .....	74
Azure Database for PostgreSQL Database - Creating Databases .....	74
Oracle Tablespaces and Data Files.....	76
Oracle Tablespace Overview .....	76
Oracle Tablespace Types .....	76
Oracle Data Files .....	76
Oracle Storage Structure .....	76
Oracle Create Tablespace Code Examples:.....	77
Azure Database for PostgreSQL Tablespaces and Data Files.....	77



Azure Database for PostgreSQL Tablespaces Creation .....	77
Azure Database for PostgreSQL Default Tablespaces.....	77
Migrating from Oracle Resource Manager.....	79
Oracle Resource Manager Main Advantages .....	79
Oracle Create Resource Plan Example.....	80
Migrating to Azure Database for PostgreSQL.....	80
Creating an Azure Database for PostgreSQL via Azure Portal.....	80
Azure Database for PostgreSQL Parameters for Resource Management.....	81
Azure Database for PostgreSQL Server Parameter Example.....	82
Migrating from Oracle SGA & PGA Memory Buffers.....	83
Oracle Automatic Memory Management .....	84
Modifying Oracle Memory Parameters.....	84
Migrating to Azure Database for PostgreSQL Memory Buffers.....	84
Viewing Azure Database for PostgreSQL Parameters - Azure Portal .....	85
Viewing Azure Database for PostgreSQL Parameters - CLI .....	85
Viewing PostgreSQL Parameters Using SQL.....	85
Oracle and PostgreSQL Memory Parameters Comparison Table.....	86
PostgreSQL Versions 12 and 13 New Features.....	87
Migrating from Oracle LogMiner Equivalents .....	88
Oracle LogMiner Example .....	88
Migrating to Azure Database for PostgreSQL Logging Analysis.....	89
PostgreSQL PG_STAT_STATEMENTS View .....	89
Azure Database for PostgreSQL DML Operations Monitoring.....	90
Azure Database for PostgreSQL Query Performance Insight.....	90
Migrating from Oracle Session Parameters Equivalents.....	91
Oracle Code Examples for Altering Session-Level Parameters.....	91
Migrating from Oracle Instance & Database Parameters (SPFILE).....	93
Oracle Configuration Files.....	93
Oracle Altering a Database Parameter Example.....	93
Migrating to Azure Database for PostgreSQL Server Parameters.....	93
Azure Portal - PostgreSQL Server Parameters .....	93
Azure CLI - PostgreSQL Server Parameters.....	94
Altering PostgreSQL Parameters at Session/Transaction Levels.....	95



Migrating to Azure Database for PostgreSQL Session Parameters .....	96
PostgreSQL Commonly Used Session Parameters .....	96
PostgreSQL Session Parameters Examples.....	96
Oracle and PostgreSQL Session Level Parameters Comparison Table (Partial List).....	97
Migrating from Oracle Alert.log (Error Log) .....	98
Migrating to the Azure Database for PostgreSQL Logging.....	99
Azure Database for PostgreSQL Logs.....	100
Azure Monitor Diagnostic Settings .....	101
Database Programming .....	102
Migrating from Oracle Transactional Model and Locking.....	102
Transaction Isolation Levels.....	102
ANSI/ISO SQL Standard (SQL92) Isolation Levels.....	102
Isolation Level Vs. Reads Access (Possible/Not Possible).....	103
Oracle Supported Isolation Levels.....	103
Oracle MVCC (Multi-Version Concurrency Control).....	103
Oracle Isolation Level Code Examples.....	103
Migrating to Azure Database for PostgreSQL Transaction Model and Locking .....	104
PostgreSQL Supported Isolation Levels.....	104
PostgreSQL Isolation Level Vs. Reads Access (Possible/Not Possible).....	104
PostgreSQL MVCC.....	104
Setting Isolation Levels in Azure Database for PostgreSQL .....	104
PostgreSQL Transaction Syntax .....	106
Oracle and PostgreSQL - Transactions Configuration Comparison .....	106
Oracle Procedures and Functions Conversion to PostgreSQL Functions.....	107
Oracle Package and Package Body.....	108
Oracle Procedures and Functions Privileges.....	108
Oracle Stored Procedure and Functions Examples .....	109
PostgreSQL Stored Procedures and Functions .....	113
PostgreSQL Stored Procedure Syntax .....	113
PostgreSQL Function Syntax .....	114
Calling PostgreSQL Stored Procedure and Function Differences.....	114
Oracle PL/SQL Conversion to PostgreSQL PL/pgSQL .....	116
Oracle User-Defined Functions (UDFs).....	120



Oracle User-Defined Function Example.....	120
PostgreSQL UDFs.....	122
PostgreSQL UDF Conversion Example .....	122
Oracle Materialized Views Conversion to PostgreSQL Materialized Views .....	125
Oracle Materialized View Logs .....	125
Oracle Materialized View Refresh Types .....	125
Oracle Materialized View Refresh Method.....	126
Oracle 12c (Release 2) Materialized Views Enhancements .....	126
Oracle Materialized Views Examples.....	126
PostgreSQL Materialized Views .....	128
Oracle and PostgreSQL Materialized Views Core Differences.....	128
PostgreSQL Materialized Views Examples .....	128
Migrating from Oracle EXECUTE IMMEDIATE Statement .....	130
Oracle EXECUTE IMMEDIATE Example .....	130
Migrate to Azure Database for PostgreSQL EXECUTE Statements.....	131
PostgreSQL EXECUTE Example .....	131
PostgreSQL PREPARE Statement .....	132
PostgreSQL PREPARE and EXECUTE Command Examples.....	133
Oracle UTL_FILE to Azure DB PostgreSQL.....	134
Oracle UTL_FILE Usage Example .....	134
Oracle UTL_MAIL and UTL_SMTP (11g & 12c) Conversion to Azure Functions Integration .....	135
Azure Database for PostgreSQL Email Notification .....	143
Azure Functions.....	144
Oracle DBMS_OUTPUT conversion to PostgreSQL RAISE.....	145
Oracle DBMS_OUTPUT Package .....	145
PostgreSQL RAISE Statement.....	146
PostgreSQL and orafce Extension Example.....	148
Oracle DBMS_RANDOM conversion to PostgreSQL RANDOM Function.....	150
Oracle DBMS_RANDOM Examples.....	150
PostgreSQL RANDOM Functionality .....	151
PostgreSQL orafce Supported Random Functions.....	151
PostgreSQL Native and orafce Extension Random Examples .....	152
Oracle DBMS_SQL Package Conversion to PostgreSQL.....	154



PostgreSQL PL/pgSQL Cursors and Dynamic Operations.....	155
Oracle DBMS_SCHEDULER Conversion to Scheduled Azure Functions .....	158
Oracle DBMS_SCHEDULER Examples .....	158
Azure Database for PostgreSQL Job Scheduling Considerations.....	162
Migrating from Oracle Global Temporary Tables .....	165
Oracle Global Temporary Table Main Features and Limitations.....	165
Global Temporary Tables Transaction Support .....	165
Oracle Global Temporary Tables Examples .....	165
Migrate to PostgreSQL Temporary Tables .....	167
PostgreSQL Temporary Table Main Features .....	167
PostgreSQL Temporary Tables ON COMMIT Clause .....	167
PostgreSQL Temporary Tables Examples.....	167
Oracle and PostgreSQL Temporary Tables Summary .....	169
Oracle Virtual Columns conversion to PostgreSQL Views and Functions .....	170
Oracle Virtual Columns Syntax.....	170
Oracle Virtual Columns Examples .....	170
Oracle Virtual Columns Notes/Limitations .....	170
PostgreSQL Virtual Columns Workarounds .....	171
PostgreSQL Views, Functions and Triggers as Alternative Solution Examples.....	171
Migrating from Oracle Index-Organized Tables.....	174
Oracle Index-Organized Table Creation .....	174
Migrate to Azure Database for PostgreSQL Cluster Table.....	175
PostgreSQL CLUSTER Table Example.....	175
Oracle Constraints Conversion to PostgreSQL Constraints.....	178
Oracle Data Integrity Constraint Types.....	178
PRIMARY KEY Constraint .....	178
FOREIGN KEY Constraint.....	179
UNIQUE Constraint .....	181
CHECK Constraint .....	182
NOT NULL Constraint.....	183
REF Constraint.....	183
Oracle Constraints Enhancements .....	184
PostgreSQL Constraints.....	184



PRIMARY KEY Constraint .....	185
FOREIGN KEY Constraint.....	186
CHECK Constraint .....	188
NOT NULL Constraints.....	189
PostgreSQL Constraints Control Options.....	189
Migrating from Oracle Table Partitions.....	192
Table Partitioning Benefits.....	192
Oracle HASH Table Partitioning.....	192
Oracle Range Table Partitioning.....	193
Oracle List Table Partitioning .....	194
Oracle Composite Partitioning.....	194
Oracle Partitioning Extensions.....	195
Split Partitions.....	195
Oracle Exchange Partitions .....	195
Oracle Sub-Partitioning Tables .....	195
Oracle Automatic List Partitioning .....	196
Migrating to Azure Database for PostgreSQL Partitions .....	196
PostgreSQL Declarative Partitioning Mechanism .....	197
PostgreSQL List Partition .....	197
PostgreSQL Range Partition .....	199
PostgreSQL Hash Partition.....	200
PostgreSQL Sub-Partitions.....	202
PostgreSQL Table and Partitions Metadata .....	202
PostgreSQL Partitions Support for Foreign Keys .....	203
PostgreSQL Partitions Index Support .....	204
PostgreSQL Composite or Multi-level Partitioning.....	205
Implementing List Table Partitioning with PostgreSQL Inheritance Tables .....	208
Oracle and PostgreSQL Partitions Comparison Summary .....	212
Migrating from Oracle Local and Global Partition Indexes.....	213
Migrating to Azure Database for PostgreSQL Partitioned Indexes .....	214
PostgreSQL Partitions Features by Release.....	215
Oracle Large Objects (LOBs) Conversion to PostgreSQL LOBs.....	217
Oracle LOB Types.....	217



Oracle LOB Example .....	217
PostgreSQL LOBs .....	218
PostgreSQL LOB Types .....	218
PostgreSQL LOB Examples.....	218
Oracle Database Links Conversion into PostgreSQL DB Link and Foreign-Data Wrapper.....	221
Oracle Database Link Examples .....	221
PostgreSQL DB Link and Foreign-Data Wrapper.....	222
PostgreSQL Remote Access Types.....	222
Migrating from Oracle Triggers.....	227
Oracle Triggers Examples .....	228
Migrating to PostgreSQL Triggers.....	230
PostgreSQL CREATE TRIGGER Syntax.....	230
PostgreSQL Triggers Support for Row-Level and Statement-Level Operations.....	231
PostgreSQL DML Triggers Examples.....	231
PostgreSQL Event Triggers .....	232
Oracle and PostgreSQL Triggers Comparison.....	233
Oracle Common Data Types Conversion to PostgreSQL Common Data Types.....	235
Oracle Date and Time Data Types .....	235
Oracle Numerical Data Types .....	235
Oracle Character Data Types.....	236
Oracle Large Object Data Types .....	237
Oracle Various Data Types.....	237
Oracle Data Types Additional Information.....	238
PostgreSQL Data Types.....	238
Oracle Data Type Conversion Considerations.....	238
Oracle Views Conversion to PostgreSQL Views.....	239
Oracle VIEW Associated Privileges.....	239
Oracle VIEW Syntax.....	239
Oracle Common View Parameters.....	239
Oracle Simple VIEW .....	239
PostgreSQL VIEWS .....	242
PostgreSQL VIEW Syntax .....	242
PostgreSQL VIEWS Associated Privileges .....	242



PostgreSQL VIEWS Associated Parameters .....	242
Executing DML Commands On Views .....	242
PostgreSQL VIEW Conversion Examples .....	243
Oracle Sequences Conversion to PostgreSQL Sequences .....	246
Oracle Basic Sequence Syntax Example .....	246
Oracle Sequence Specifications .....	246
Oracle Sequence Examples .....	247
Oracle 12c Sequence Enhancements .....	248
PostgreSQL Sequences .....	250
PostgreSQL Sequence Syntax .....	250
PostgreSQL Sequence Parameters .....	251
PostgreSQL Sequence Examples .....	252
Generating PostgreSQL Sequence by SERIAL Type .....	253
Oracle Character Sets Conversion to PostgreSQL Character Set .....	256
Oracle Character Sets Control .....	256
Oracle Encoding .....	256
Migration Considerations .....	256
Table Level Collation .....	258
Oracle and PostgreSQL Character Set Summary .....	258
Migrate from Oracle User-Defined Types .....	260
Oracle User-Defined Types Examples .....	260
Migrating to Azure Database for PostgreSQL User Defined Types .....	264
Oracle and PostgreSQL Type Main Differences .....	265
PostgreSQL CREATE TYPE Syntax .....	265
Performance Topics .....	270
Migrating from Oracle Statistics Collection .....	270
Oracle Statistics Gathering Methods .....	270
Oracle Statistics Collection Examples .....	270
Migrating to Azure Database for PostgreSQL Statistics Collection .....	271
PostgreSQL Statistics Collection Methods .....	271
PostgreSQL ANALYZE Statement examples .....	272
Oracle and PostgreSQL Statistics Collection Comparison .....	273
Migrate from Oracle Execution Plans .....	274



Oracle Generate and View Execution Plan Examples .....	274
Migrating to Azure Database for PostgreSQL Execution Plans.....	276
PostgreSQL EXPLAIN and EXPLAIN ANALYZE.....	276
PostgreSQL EXPLAIN and EXPLAIN ANALYZE Examples.....	276
Oracle Database Optimizer Hints Conversion to PostgreSQL Query Planning .....	280
Oracle Optimizer Hints Examples.....	280
PostgreSQL Query Planning .....	282
PostgreSQL Query Planning Examples.....	282
Oracle Indexes conversion to PostgreSQL Indexes .....	288
Oracle Indexes Types and Syntax Examples.....	288
PostgreSQL Indexes .....	291
PostgreSQL Index Syntax.....	291
PostgreSQL Index Types .....	291
PostgreSQL Indexes Creation Examples .....	293
PostgreSQL Indexes - Administration Commands Examples.....	294
Oracle and PostgreSQL Indexes Comparison.....	294
Azure-Specific Topics .....	297
Azure CLI - Common Usages with Azure DB .....	297
Deploying Azure Database for PostgreSQL using Azure CLI .....	297
Oracle to Azure Database for PostgreSQL - Data Migration Paths .....	299
Right-Sizing PostgreSQL and Application VM Instances .....	300
Azure Database for PostgreSQL Sizing Based on Oracle AWR.....	303
AWR Sizing Assessment Key Points .....	303
Gathering AWR General and Sizing Specific Information .....	303



© 2019 Microsoft Corporation. All rights reserved.

## Introduction and Concepts

### Migration Guide Summary Table

The following table summarizes the topics found in the Oracle to PostgreSQL Migration Guide. It shows the associated Oracle Database and Azure Database for PostgreSQL Features and summarizes compatibility, automation level, and summarizes major considerations when migrating.

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
<a href="#">Backup and Recovery – RMAN and Datapump</a>	Azure Database for PostgreSQL Backup and Restore		N/A	Backup and restore are managed by Azure Database for PostgreSQL managed service. In addition, pg_dump and pg_restore can be applied as well.
<a href="#">Oracle RAC for High Availability</a>	Azure Database for PostgreSQL High Availability – Single Server and Hyperscale (Citus)		N/A	Azure Database for PostgreSQL utilizes different architecture. High availability and scalability can be achieved by Azure Database for PostgreSQL services.
<a href="#">Oracle Flashback</a>	Azure Database for PostgreSQL Backup and Restore (Point in Time)		N/A	PostgreSQL does not support Oracle Flashback Database as a direct feature. Azure Database for PostgreSQL point-in-time restore can be used as an alternative.
<a href="#">Oracle SQL*Loader</a>	Azure Database for PostgreSQL COPY Command		N/A	Expect different methods (commands) to achieve the same purpose when converting Oracle SQL*Loader to PostgreSQL COPY command.

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Commented [AKT2]: When Ora2pg does not apply, can we call it NA or N/A?

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
<a href="#">Oracle Security Model: Roles and Permissions</a>	Azure Database for PostgreSQL Roles – Single Server and Hyperscale (Citus)			Expect different terminology between Oracle and PostgreSQL from security and user management perspectives. Migrating from Oracle security to Azure Database for PostgreSQL is supported with a variety of options.
<a href="#">Oracle Dictionary and V\$ Views</a>	Azure Database for PostgreSQL System Catalog and PostgreSQL Statistic Collector		N/A	Oracle V\$ views and internal object names should be altered to the PostgreSQL system catalog objects.
<a href="#">Oracle Multi-Tenant Architecture, CDB and PDB</a>	Azure Database for PostgreSQL Database Architecture			PostgreSQL is built on different architecture, but the same Oracle multitenant database structure can be achieved with Azure Database for PostgreSQL.
<a href="#">Oracle Tablespaces and Data Files</a>	Azure Database for PostgreSQL Tablespaces and Data Files			Expect different architecture regarding tablespaces data files with Azure Database for PostgreSQL.
<a href="#">Oracle Resource Manager</a>	Azure Database for PostgreSQL architecture and parameters		N/A	Although Oracle Resource Manager is not supported, Azure Database for PostgreSQL supports multiple configurations/options for resource

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
				management functionality.
<a href="#">Oracle SGA and PGA Memory Buffers</a>	Azure Database for PostgreSQL memory parameters		N/A	PostgreSQL utilizes different memory buffer architecture, which can be managed by Azure Database for PostgreSQL server parameters.
<a href="#">Oracle LogMiner</a>	Azure Database for PostgreSQL PG_STAT_STATEMENTS, DML Operations Monitoring, Query Performance Insights		N/A	Although PostgreSQL does not support Oracle LogMiner, alternative solutions are available for identifying historical database operations.
<a href="#">Oracle Instance and Database Parameters</a>	Azure Database for PostgreSQL Server Parameters		N/A	Expect different architecture and parameters when using PostgreSQL.  Azure Database for PostgreSQL offers a wide selection of resource types for a specific workload. Database parameters can be configured using the server parameter.
<a href="#">Oracle Session-Level Parameters</a>	Azure Database for PostgreSQL Session Parameters		N/A	PostgreSQL utilizes different syntax for controlling session parameters using the "SET" command.
<a href="#">Oracle Alert.log</a>	Azure Database for PostgreSQL Logs		N/A	Azure Database for PostgreSQL utilizes a different but rich set of logging options.

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
<a href="#">Oracle Transactional Model and Locking</a>	Azure Database for PostgreSQL Transactional Model and Locking	●●●○○	●●○○○	PostgreSQL transaction model, isolation level, and locking behavior are similar to Oracle, but subtransaction (or nested transaction – a transaction within a transaction) is not supported. Save point can be used as an alternative solution.
<a href="#">Oracle Stored Procedures and Functions</a>	Azure Database for PostgreSQL Procedures and Functions	●●●○○	●●●●○	Syntax and features differences.  Creation of stored procedures is supported in PostgreSQL version 11 and later.
<a href="#">Oracle User-Defined Functions (UDFs)</a>	Azure Database for PostgreSQL UDFs	●●●○○	●●●●○	Expect syntax and features differences when performing code conversion.
<a href="#">Oracle Materialized Views</a>	Azure Database for PostgreSQL Materialized Views	●●●○○	●●●●○	PostgreSQL does not support automatic or incremental MViews REFRESH.
<a href="#">Oracle EXECUTE IMMEDIATE</a>	Azure Database for PostgreSQL EXECUTE	●○○○○	●●●●○	Syntax differences may require code rewrite. The IMMEDIATE keyword needs to be converted to EXECUTE and PREPARE commands.
<a href="#">Oracle UTL_FILE</a>	Azure Database for PostgreSQL oracle extension	●●●○○	●○○○○	No UTL_FILE equivalent in PostgreSQL. Requires manual code

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
				conversion and verification.
<a href="#">Oracle UTL_MAIL and UTL_SMTP</a>	Azure Database for PostgreSQL functions		N/A	PostgreSQL provides equivalent functionality. Use Azure Functions integration.
<a href="#">Oracle DBMS_OUTPUT</a>	Azure Database for PostgreSQL RAISE or orafce extension			Requires application/drivers rewrite. Alternatively, use the orafce extension, which has the same syntax and similar functionality.
<a href="#">Oracle DBMS_RANDOM</a>	Azure Database for PostgreSQL RANDOM			Different syntax may require code rewrite.
<a href="#">Oracle Global Temporary Tables</a>	Azure Database for PostgreSQL Temporary Tables			PostgreSQL global temporary tables cannot read from multiple sessions and are dropped after the session ends.
<a href="#">Oracle Virtual Columns</a>	Azure Database for PostgreSQL Views and Functions			Virtual columns are not supported in PostgreSQL version 11 and earlier. Fully supported in version 12 and later.
<a href="#">Oracle Index-Organized Tables</a>	Azure Database for PostgreSQL Cluster Tables			PostgreSQL does not support IOT but offers a partial workaround.
<a href="#">Oracle Constraints</a>	Azure Database for PostgreSQL Constraints			Constraints with REF, ENABLE, and DISABLE options are not supported.
<a href="#">Oracle Table Partitions</a>	Azure Database for PostgreSQL Partitions			Some partition types are not supported in PostgreSQL.

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
<a href="#">Oracle Local and Global Partitioned Indexes</a>	Azure Database for PostgreSQL Partitioned Indexes	●○○○○	●○○○○	PostgreSQL does not support domain indexes.
<a href="#">Oracle Large Objects (LOBs)</a>	Azure Database for PostgreSQL LOBs	●○○○○	●○○●○○	PostgreSQL does not support Oracle Secure File LOBs. LOB type mapping from Oracle to PostgreSQL is different.
<a href="#">Oracle Database Links (DBLinks)</a>	Azure Database for PostgreSQL DB Link and Foreign-Data Wrapper	●○○○○	●○○○○○	PostgreSQL uses different implementation methods and syntax.
<a href="#">Oracle Triggers</a>	Azure Database for PostgreSQL Triggers	●○○○○	●○○●○○	PostgreSQL syntax is different; system triggers are not supported.
<a href="#">Oracle Common Data Types</a>	Azure Database for PostgreSQL Data Types	●○○○○	●○○●○○	PostgreSQL does not support some Oracle data types. Spatial data types might require the PostGIS extension.
<a href="#">Oracle Views</a>	Azure Database for PostgreSQL Views.	●○○○○	●○○●○○	None.
<a href="#">Oracle Sequences</a>	Azure Database for PostgreSQL Sequences	●○○○○	●○○●○○	Some PostgreSQL features and syntax are different.
<a href="#">Oracle Character Sets</a>	Azure Database for PostgreSQL Character Sets	●○○○○	●○○●○○	UTF16 character and NCHAR/NVARCHAR data types are not supported in PostgreSQL.
<a href="#">Oracle User-Defined Types (UDTs)</a>	Azure Database for PostgreSQL UDTs	●○○○○	●○○●○○	PostgreSQL does not support the FORALL statement and DEFAULT option, and does not support

Oracle Database Feature	Azure Database for PostgreSQL Feature	Overall Compatibility / Migration Complexity	Ora2pg Automation Level	Considerations
				collection type constructors.
<a href="#">Migrating from Oracle Statistics Collection</a>	Azure Database for PostgreSQL Statistics Collection	●○●○○○	N/A	Expect different architecture, syntax, and configuration methods.
<a href="#">Oracle Database Optimizer Hints</a>	Azure Database for PostgreSQL Query Planning	●○●○●○○	●○●○●○●○	Optimizer execution plan influencing in PostgreSQL is limited compared to Oracle.
<a href="#">Oracle Indexes</a>	Azure Database for PostgreSQL Indexes	●○●○●○○	●○●○●○●○	PostgreSQL does not support BITMAP index or invisible indexes.

Commented [AKT1]: Need this table legend in every page below so it is easy to follow.



## Oracle to Azure Database for PostgreSQL: Migration Project Overview

Migrating projects from on-premises or data center environments can be difficult and requires multiple steps. The project can involve migrating multiple servers, business applications, and database servers. You need to plan rehosting and refactoring carefully to meet downtime requirements when moving the workloads to the new platform.

However, migrating to the cloud also has numerous benefits, such as providing high availability and lower latency, thereby improving overall business delivery to your customers.

By applying best practices, you can make the process of migrating from an Oracle database to an Azure Database for PostgreSQL predictable. Oracle to PostgreSQL migrations are heterogeneous (refactoring). Code conversion (PL/SQL to PostgreSQL PL/pgSQL) with thorough testing is required throughout the implementation, increasing the overall effort. Alternatively, homogeneous migrations (rehosting), such as PostgreSQL deployed on-premises to PostgreSQL deployed on the cloud, are considered less complex because they require fewer modifications or code conversions.

This migration guide describes the stages of a heterogeneous migration and includes code examples of Oracle proprietary features (based on PL/SQL) coupled with conversion options to Azure Database for PostgreSQL (conversion to PL/pgSQL).

Database migration can be divided into three main stages:

- Pre-migration
- Migration
- Post-migration

### Main Stages High-Level Overview

- Pre-migration
  - Assessing the database (discovery)
  - Creating a migration plan based on discovery findings:
  - Conversion complexities
  - Current database workloads
  - Workaround plans if required
  - Defining success criteria and documenting migration risks
  - Setting up the Azure Database for PostgreSQL environment, based on the size and workload of the source Oracle database
  - Performing schema conversions
- Migration
  - Deploying the converted schema on the Azure Database for PostgreSQL (metadata only - mainly for schema DDL compilation testing).
  - Synchronizing data (data migration) from the source Oracle database to the Azure Database for the PostgreSQL instance, based on the required approach and business requirements of the service level agreement (SLA). Plan the data-migration method in advance to meet SLA requirements.
  - Performing a cutover from the source Oracle database to the Azure Database for PostgreSQL instance.
- Post-migration
  - Workload remediation to ensure high performance in the target platform; for example, changing the application framework to support PostgreSQL instead of Oracle.



- Testing to ensure that the solution adheres to performance and functional requirements and database and application functionality.
- Optimizing and modifying the solution to fix any database performance or functionality issues.

## Database Assessment/Discovery

It is important to assess all applicable workloads for migration from Oracle to PostgreSQL and choose the right workloads to migrate.

The assessment stage (the discovery stage) reveals the current Oracle database specifications, such as:

- Infrastructure
  - High availability/clustering using Oracle Real Application Clusters (RAC).
  - Disaster recovery using Oracle Data Guard (primary and standby instances).
  - Replication and Change Data Capture (CDC) using Oracle GoldenGate.
- Oracle proprietary features and logical objects
  - Number of objects (tables, data types, views, stored procedures, etc.).
  - Usage of Oracle, such as PL/SQL packages, UTL\_FILE, DBMS\_UTILITY.
- Sizing (analyzed when planning the Azure target environment)
  - Database overall storage size (data files, redo logs, archive logs, etc.)
  - Database hardware allocation and configuration (CPU, RAM, storage, network).
- Workload
  - Specific Oracle source database day-to-day workload characteristics to deploy an accurate Azure database for PostgreSQL environment.
- Dependencies
  - A single database can serve many applications, and applications can have dependencies with many databases.
  - Oracle and non-Oracle databases (database links, heterogeneous services).

## Identify Success Criteria and Migration Risk

Measure and monitor the migration effort during all these stages, and define migration success criteria, including:

- Schema conversion: Converted and compile Oracle schemas in PostgreSQL.
- Database performance: Benchmark converted schema objects for performance and functional correctness.
- Database functionality: Make sure the converted business logic returns the same results on the Azure Database for PostgreSQL database server as those on the source Oracle database server.
- Migration risks: In the early stages of the project, identify risks such as:
  - Application support for the new PostgreSQL database platform
  - Data integrity and verification issues.
  - Database performance concerns in PostgreSQL

## Azure Database PostgreSQL Target Environment Setup

The Azure database for PostgreSQL should accommodate the information collected in the discovery stage from infrastructure perspectives (HA, DR, sizing).

Target sizing should consider the following:

- Source database server CPU requirements
- Source database memory usage and allocation
- Source max sessions
- Source max active sessions
- Source detected IOPS



## Schema Conversion

In the overall migration from Oracle to PostgreSQL databases, the schema conversion stage is often challenging and requires a high level of effort.

The Oracle schema/user is the main owner of database objects, including tables, views, triggers, partitions, materialized views, stored procedures, functions, packages, and more. In cases where the core business logic was created on the database layer, the overall migration complexity can be higher because more Oracle PL/SQL code needs to be converted to PostgreSQL PL/pgSQL.

Automation tools such as ora2PG (discussed later in this cookbook) cannot deliver 100% of the Oracle schema conversion, so additional manual code conversions are required. Based on the object identified in the assessment stage, you can measure the schema conversion effort and estimate the time needed.

## Data Migration

This stage should be planned carefully and according to the organization needs and business/technical limits, as well as expected downtimes.

Note that data *migration* is treated separately from data *conversion*. Migration includes only the actions required for the actual migration of data, after the schemas have already been converted, tested, and validated.

When planning data migration from one database platform to another, consider the following:

- Initial data load
  - Data size
  - Data load timing (import)
  - Initial load method
- Application and database downtime
  - Is downtime an option?
  - Downtime duration
- CDC/replication methods
  - Downtime constraints should dictate the data sync method.
  - You can use different tools and approaches for each method.
  - Implement important, prerequisite steps for CDC/replication with the Oracle database as the source platform. Enabling archive logs, enabling supplemental log data, and validating that Primary Keys are on all replicated tables can impact data migration performance.

## Ora2Pg

Although there are several database conversion tools available, [Ora2Pg](#) is an industry standard and powerful open-source tool to perform schema conversion from the Oracle database to the PostgreSQL database. In its core functionality, Ora2Pg connects to the source Oracle database, scans the database structure and schema objects, and generates converted code that can be compiled into a PostgreSQL target platform. In addition, Ora2Pg supports data migration by generating raw data-dump files from Oracle that can be loaded into the PostgreSQL database platform using PostgreSQL COPY (equivalent to the Oracle SQL Loader as a bulk Insert feature).

Ora2Pg minimizes manual conversion required when modifying Oracle schemas into PostgreSQL database objects. It simplifies migrations by creating a migration project with schema-level conversion automation, handling each schema and database object separately. Ora2Pg also offers configuration parameters to support different requirements or capabilities, such as:

- Handling complex database object conversion, including:
  - Tables (data types conversion) and views
  - Partitions management



- Packages and Packages Body (basic PL/SQL code conversion)
- Stored procedures and functions
  - Synonyms
  - Materialized view
  - Triggers
- Indexes and constraints (including handling PKs/FKs dependencies)
  - Providing a parallel option when performing schema and data export/imports.
  - Exporting full data or filtering a subset of data using a WHERE clause.
  - Supporting the Oracle BLOB object as PG BYTEA.

## Ora2Pg Installation

Using best practices, install Ora2Pg on a server with client access to both source and target database platforms (Oracle and Azure Database for PostgreSQL). The installation should be on a server dedicated to running Ora2Pg (for example, Azure Virtual Machine). Ora2Pg can be treated as a middle point between the source Oracle database and the Azure Database for PostgreSQL instances. See [Ora2Pg installations](#) for more information.

- Oracle client
  - For Ora2Pg to access the Oracle database, DBD::Oracle should be installed, which requires the following [packages and libraries](#):
    - Instant-client
    - SDK
    - SQLPLUS
    - libaio1 library
- Ora2Pg libraries
  - Ora2Pg requires Perl libraries with the DBD::Oracle module to establish a connection to the Oracle database.
  - Ora2Pg requires the PostgreSQL client to establish a connection to the Azure Database for the PostgreSQL target environment.
- PostgreSQL client to deploy the converted Oracle schema objects and data transfers. Ora2Pg supports multiple configuration options.

## Ora2Pg Post-Installation Actions

The following sections simulate a basic migration from an Oracle HR sample schema to an Azure Database for PostgreSQL instance, using Ora2Pg.

### Generating a Migration Project

This step creates a Work Tree (project-structured directories/files tree) that serves as a specific migration project, holding the associated configuration file and scripts.

The following example creates:

- The project path (OS level)
- --project\_base – The migration project base directory
- --init\_project – The migration project name (creates the project home directory)

```
$ mkdir -p /mnt/ora_azure_mig  
$ cd /mnt/ora_azure_mig  
$ ora2pg --project_base /mnt/ora_azure_mig --init_project HR
```

### # Process output

```
Creating project HR.  
/mnt/ora_azure_mig/HR/  
    schema/  
        dblinks/  
        directories/  
        functions/  
        grants/  
        mviews/  
        packages/  
        partitions/  
        procedures/  
        sequences/  
        synonyms/  
        tables/  
        tablespaces/  
        triggers/  
        types/  
        views/  
  
    sources/  
        functions/  
        mviews/  
        packages/  
        partitions/  
        procedures/  
        triggers/  
        types/
```



```
views/  
data/  
config/  
reports/
```

The following Ora2Pg scripts are created:

- export\_schema.sh - An automated export script
- import\_all.sh - A prompt-based import script

```
$ ls -l  
total 20  
drwxrwxr-x. 2 miguser miguser 25 Aug 31 11:24 config  
drwxrwxr-x. 2 miguser miguser 6 Aug 31 11:24 data  
-rwx-----. 1 miguser miguser 2010 Aug 31 11:24 export_schema.sh  
-rwx-----. 1 miguser miguser 16061 Aug 31 11:24 import_all.sh  
drwxrwxr-x. 2 miguser miguser 6 Aug 31 11:24 reports  
drwxrwxr-x. 17 miguser miguser 245 Aug 31 11:24 schema  
drwxrwxr-x. 10 miguser miguser 131 Aug 31 11:24 sources
```

Test the connection to the source Oracle database. Modify the Ora2Pg project configuration file according to the Oracle database details and credentials:

```
# Edit the following parameters according to the environment values  
$ vi /mnt/ora_azure_mig/HR/config/ora2pg.conf  
...  
ORACLE_HOME /usr/lib/oracle/19.8/client64  
ORACLE_DSN dbi:Oracle:host=<DNS/IP_ADDR>;sid=<ORACLE_SID>;port=1521  
ORACLE_USER <DBA_USER>  
ORACLE_PWD <Password>  
...  
SCHEMA HR
```



```
...
# Save the ora2pg.conf file and exit
```

Verify Oracle connection from ora2Pg:

```
$ ora2pg -t SHOW_VERSION -c ./config/ora2pg.conf
# Successful output
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0
```

### Ora2Pg Assessment Report

As a pre-conversion step, Ora2Pg creates an assessment report in HTML format, based on the source Oracle database schemas.

The HTML report provides useful information about the migration schema objects, including object count, object status (valid/invalid), level of overall migration complexity (per schema), number of rows for each table, and more.

Generate the Ora2Pg assessment report by running the following:

```
$ ora2pg -t SHOW_REPORT -c config/ora2pg.conf --estimate_cost --dump_as_html > report.html
# launch the report.html file
```

The Ora2Pg assessment report looks like this:

Ora2Pg - Database Migration Report				
Version	Oracle Database 11g Express Edition Release 11.2.0.2.0	Schema	HR	Size
DATABASE LINK	0	0	0	Data bases will be exported as SQL/MED PostgreSQL's Foreign Data Wrapper (FDW) extensions using oracle_fdw.
GLOBAL TEMPORARY TABLE	0	0	0	Global temporary table are not supported by PostgreSQL, and will not be exported. You have to rewrite some application code to match the PostgreSQL temporary table behavior.
INDEX	19	0	3	11 index(es) are concerned by the export, others are automatically generated and will do so on PostgreSQL. Bitmap will be exported as btree_gin index(es) and hash index(es) will be exported as b-tree index(es) if any. Domain index are exported as b-tree but commented to be edited to mainly use FTS. Cluster, bitmap join and IOT indexes will not be exported at all. Reverse indexes are not exported too, you may use a trigrams-based extension to support reverse search. Use 'varchar_pattern_ops' or 'bytea_pattern_ops' operators in your indexes to improve search with the LIKE operator respectively into varchar, text or char columns.
JOB	0	0	0	Job are not exported. You may set external cron job with them.
PROCEDURE	2	0	8	Total size of procedure code: 772 bytes.
SEQUENCE	3	0	1	Sequences are fully supported, but all call to sequence_name.NEXTVAL or sequence_name.CURRVAL will be transformed into NEXTVAL(sequence_name) or CURRVAL(sequence_name).

### Ora2Pg Migration Effort Estimation

A Migration Effort Estimation is included at the bottom of the Ora2Pg assessment report, describing levels of effort and complexity from a migration and a technical standpoint.

**Migration levels:**

- A - Migration that might be run automatically
- B - Migration with code rewrite and a human-days cost up to 5 days
- C - Migration with code rewrite and a human-days cost above 5 days

**Technical levels:**

- 1 = trivial: no stored functions and no triggers
- 2 = easy: no stored functions but with triggers, no manual rewriting
- 3 = simple: stored functions and/or triggers, no manual rewriting
- 4 = manual: no stored functions but with triggers or views with code rewriting
- 5 = difficult: stored functions and/or triggers with code rewriting

**Ora2Pg Oracle Schema Automatic Conversion**

Use the following command to perform Oracle schema conversion:

- The command should be run for each schema separately from the migration project folder.
- The Azure database for PostgreSQL is not affected at this time, because Ora2Pg creates a conversion output file, which is used with the import\_all.sh script.
- This process might take several minutes, depending on the schema objects and size.

**# Export**

```
$ /mnt/ora_azure_mig/HR/export_schema.sh
```

**# Output Example**

```
[=====] 7/7 tables (100.0%) end of scanning.  
[=====] 10/10 objects types (100.0%) end of objects auditing.  
Running: ora2pg -p -t TABLE -o table.sql -b ./schema/tables -c ./config/ora2pg.conf  
[=====] 7/7 tables (100.0%) end of scanning.  
[=====] 7/7 tables (100.0%) end of table export.  
Running: ora2pg -p -t PACKAGE -o package.sql -b ./schema/packages -c ./config/ora2pg.conf  
[=====] 0/0 packages (100.0%) end of output.  
Running: ora2pg -p -t VIEW -o view.sql -b ./schema/views -c ./config/ora2pg.conf  
[=====] 1/1 views (100.0%) end of output.
```



```
Running: ora2pg -p -t GRANT -o grant.sql -b ./schema/grants -c ./config/ora2pg.conf
Running: ora2pg -p -t SEQUENCE -o sequence.sql -b ./schema/sequences -c ./config/ora2pg.conf
[=====>] 3/3 sequences (100.0%) end of output.

Running: ora2pg -p -t TRIGGER -o trigger.sql -b ./schema/triggers -c ./config/ora2pg.conf
[=====>] 1/1 triggers (100.0%) end of output.

Running: ora2pg -p -t FUNCTION -o function.sql -b ./schema/functions -c ./config/ora2pg.conf
[=====>] 0/0 functions (100.0%) end of functions export.

Running: ora2pg -p -t PROCEDURE -o procedure.sql -b ./schema/procedures -c ./config/ora2pg.conf
[=====>] 2/2 procedures (100.0%) end of procedures export.

...
...
```

Once the export process is complete, the following directory structure holds the converted schema DDL and PL/SQL code:

```
$ ls -l schema/
total 0
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:05 dblinks
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:05 directories
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:04 functions
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:04 grants
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:05 mviews
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:04 packages
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:05 partitions
drwxrwxr-x. 2 miguser miguser 96 Aug 31 12:04 procedures
drwxrwxr-x. 2 miguser miguser 26 Aug 31 12:04 sequences
drwxrwxr-x. 2 miguser miguser 25 Aug 31 12:05 synonyms
drwxrwxr-x. 2 miguser miguser 100 Aug 31 12:04 tables
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:04 tablespaces
```



```
drwxrwxr-x. 2 miguser miguser 63 Aug 31 12:04 triggers  
drwxrwxr-x. 2 miguser miguser 6 Aug 31 12:05 types  
drwxrwxr-x. 2 miguser miguser 55 Aug 31 12:04 views
```

## Ora2Pg Data Extraction

Use the following command to extract data from the source Oracle schema.

- The data is stored in files under the migration project data directory and is not loaded into Azure Database for PostgreSQL at this point.
- For actual migrations, this approach requires application downtime.

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

# Process output

```
[=====>] 7/7 tables (100.0%) end of scanning.  
[=====>] 25/25 rows (100.0%) Table COUNTRIES (25 recs/sec)  
[=>      ] 25/215 total rows (11.6%) - (0 sec., avg: 25 recs/sec).  
[=====>] 27/27 rows (100.0%) Table DEPARTMENTS (27 recs/sec)  
[=>      ] 52/215 total rows (24.2%) - (0 sec., avg: 52 recs/sec).  
[=====>] 107/107 rows (100.0%) Table EMPLOYEES (107 recs/sec)  
[=====>      ] 159/215 total rows (74.0%) - (0 sec., avg: 159 recs/sec).  
[=====>] 19/19 rows (100.0%) Table JOBS (19 recs/sec)  
[=====>      ] 178/215 total rows (82.8%) - (0 sec., avg: 178 recs/sec).  
[=====>] 10/10 rows (100.0%) Table JOB_HISTORY (10 recs/sec)  
[=====>      ] 188/215 total rows (87.4%) - (0 sec., avg: 188 recs/sec).  
[=====>] 23/23 rows (100.0%) Table LOCATIONS (23 recs/sec)  
[=====>      ] 211/215 total rows (98.1%) - (0 sec., avg: 211 recs/sec).  
[=====>] 4/4 rows (100.0%) Table REGIONS (4 recs/sec)  
[=====>] 215/215 total rows (100.0%) - (0 sec., avg: 215 recs/sec).  
[=====>] 215/215 rows (100.0%) on total estimated data (1 sec., avg: 215  
recs/sec)
```



```
$ ls -l data/
total 40
-rw-rw-r--. 1 miguser miguser 441 Aug 31 12:29 COUNTRIES_data.sql
-rw-rw-r--. 1 miguser miguser 756 Aug 31 12:29 DEPARTMENTS_data.sql
-rw-rw-r--. 1 miguser miguser 9161 Aug 31 12:29 EMPLOYEES_data.sql
-rw-rw-r--. 1 miguser miguser 798 Aug 31 12:29 JOBS_data.sql
-rw-rw-r--. 1 miguser miguser 672 Aug 31 12:29 JOB_HISTORY_data.sql
-rw-rw-r--. 1 miguser miguser 1326 Aug 31 12:29 LOCATIONS_data.sql
-rw-rw-r--. 1 miguser miguser 129 Aug 31 12:29 REGIONS_data.sql
-rw-rw-r--. 1 miguser miguser 822 Aug 31 12:29 data.sql
```

## Setting an Azure Database for PostgreSQL as a Migration Target

Deploy a single [Azure Database for PostgreSQL](#) instance using this QuickStart [guide](#), used for deploying the Azure Database for PostgreSQL via Azure CLI.

To deploy the converted schema DDLs from the Ora2Pg server to the Azure Database for PostgreSQL instance, run the following command:

```
$ ./import_all.sh -h $AZURE_PG -d hr -o <owner> -U <user>
```

You can use the following parameters for the import script:

- -o - Owner of the database to create
- -U - Username to connect to PostgreSQL
- -s - Import schema only, do not try to import data
- -I - Do not try to load indexes, constraints, and triggers
- -n - Comma-separated list of schemas to create

Run the following to verify object creation over the Azure Database for PostgreSQL target instance:

```
$ psql -h <Azure_Server_Name> -U <AdminUsername@ServerName> -d hr
...
hr=> \dt
      List of relations
Schema |   Name   | Type  | Owner

```



```
-----+-----+-----+-----  
public | countries | table | naya  
public | departments | table | naya  
public | employees | table | naya  
public | job_history | table | naya  
public | jobs | table | naya  
public | locations | table | naya  
public | regions | table | naya  
(7 rows)
```

## Ora2Pg Data Import

Importing the source Oracle raw data (database records) should be performed as a separate step before importing:

- Indexes
- Constraints (PK, Null/Not Null, Check and Unique)
- Foreign keys (separated from the other constraints)
- Triggers

Importing data before the indexes have been created allows faster data load times. Furthermore, importing this “unconstrained” data eliminates potential data integrity conflicts, such as missing foreign key values as a reference to a child table.

Complete the import process in the following order for faster import times and to eliminate possible delays:

- Data
- Indexes
- Constraints
- Foreign keys
- Triggers

Import data example command:

```
$ psql -h <ServerName> -U <AdminUser@ServerName> -d hr < data/data.sql
```

Import indexes, constraints, foreign keys, and triggers command:

```
$ ./import_all.sh -h <AzureServer> -U <AdminUser> -d hr -o <DBOwner> -i
```



### Ora2Pg Import Performance Considerations

Ora2Pg offers several options to run import processes in parallel. Performance benchmarks indicate that running data and other database object imports in parallel reduces import time.

Ora2Pg data import using parallelism command:

```
$ ./import_all.sh -h <AzureServer> -U <AdminUser> -d hr -o <DBOwner> -a -j 4
```

Gathering Oracle database statistics as a pre-step to the migration can improve overall performance:

```
BEGIN  
    DBMS_STATS.GATHER_SCHEMA_STATS  
    DBMS_STATS.GATHER_DATABASE_STATS  
    DBMS_STATS.GATHER_DICTIONARY_STATS  
END;
```

For additional information, refer to [Ora2Pg official documentation](#).

### Schema Import and Data Verification

You can use several methods to validate successful schema object imports from Oracle to PostgreSQL:

- Method 1 - Using SQL statements to verify tables *Source:Target* objects count by schema (owner) and object type by database platform:

#### -- ORACLE Database Objects Details and Count --

```
SELECT OWNER, OBJECT_T  
YPE, COUNT(*)  
FROM DBA_OBJECTS WHERE OWNER IN('HR')  
GROUP BY OWNER, OBJECT_TYPE  
ORDER BY 1, 2;
```

#### -- Tables

```
SELECT owner, TABLE_NAME, TABLESPACE_NAME, STATUS, NUM_ROWS  
FROM dba_tables  
WHERE OWNER IN('HR');
```

**-- Views**

```
SELECT OWNER, VIEW_NAME  
FROM DBA_VIEWS  
WHERE OWNER IN('HR');
```

**-- Stored Procedures and Functions**

```
SELECT OWNER, OBJECT_NAME, OBJECT_TYPE, PROCEDURE_NAME  
FROM DBA_PROCEDURES  
WHERE OWNER IN('HR');
```

**-- Constraints**

```
SELECT OWNER, CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME, R_OWNER,  
R_CONSTRAINT_NAME, DELETE_RULE  
FROM DBA_CONSTRAINTS  
WHERE OWNER IN('HR');
```

**-- Sequences**

```
SELECT SEQUENCE_OWNER, SEQUENCE_NAME, MIN_VALUE, MAX_VALUE, INCREMENT_BY,  
LAST_NUMBER  
FROM DBA_SEQUENCES  
WHERE SEQUENCE_OWNER IN('HR');
```

**-- Indexes**

```
SELECT OWNER, INDEX_NAME, INDEX_TYPE, TABLE_NAME, TABLE_OWNER, VISIBILITY,  
UNIQUENESS, TABLESPACE_NAME  
FROM DBA_INDEXES  
WHERE OWNER IN('HR');
```

**-- Triggers**

```
SELECT OWNER, TRIGGER_NAME, TRIGGER_TYPE, TRIGGERING_EVENT, TABLE_OWNER  
FROM DBA_TRIGGERS  
WHERE OWNER IN('HR');
```

**-- Partitions**

```
SELECT OWNER, TABLE_NAME, PARTITIONING_TYPE, SUBPARTITIONING_TYPE,  
PARTITION_COUNT, STATUS  
FROM DBA_PART_TABLES  
WHERE OWNER IN('HR');
```

**-- Synonyms**

```
SELECT OWNER, SYNONYM_NAME, TABLE_OWNER, TABLE_NAME, DB_LINK  
FROM DBA_SYNONYMS  
WHERE OWNER IN('HR');
```

**-- PostgreSQL Database Objects Details and Count**

```
SELECT TABLE_SCHEMA as "TableSchema", 'TABLE' as "ObjectType", count(*) as "Count" FROM  
INFORMATION_SCHEMA.TABLES  
WHERE TABLE_SCHEMA IN('hr')  
AND TABLE_TYPE='BASE TABLE'  
group by TABLE_SCHEMA;
```

**UNION**

```
SELECT TABLE_SCHEMA, 'VIEW' as "ObjectType", count(*) as "COUNT" FROM  
INFORMATION_SCHEMA.TABLES
```

```
WHERE TABLE_SCHEMA IN('hr')
AND TABLE_TYPE='VIEW'
group by TABLE_SCHEMA;
```

#### UNION

```
SELECT specific_schema, 'PROCEDURE/FUNCTION' as "ObjectType", count(*) as "Count" FROM
INFORMATION_SCHEMA.ROUTINES

WHERE specific_schema IN('hr')
group by specific_schema;
```

#### UNION

```
SELECT SEQUENCE_SCHEMA, 'SEQUENCE' as "ObjectType", count(*) as "Count" FROM
INFORMATION_SCHEMA.SEQUENCES

WHERE SEQUENCE_SCHEMA IN('hr')
group by SEQUENCE_SCHEMA;
```

#### UNION

```
SELECT schemaname, 'INDEX' as "ObjectType", count(*) as "Count" FROM pg_indexes
where schemaname IN('hr')
group by schemaname;
```

#### UNION

```
SELECT trigger_schema, 'TRIGGER' as "ObjectType", count(*) as "Count" from
information_schema.triggers

WHERE trigger_schema IN('hr')
group by trigger_schema;
```

#### UNION

```
select pgd.datname, 'TABLE PARTITION CHILD' as "ObjectType", count(*) as "Count"  
from pg_catalog.pg_class pgc join pg_catalog.pg_inherits pgic  
on pgc.oid=pgic.inherelid  
join pg_catalog.pg_database pgd  
on pgc.relowner = pgd.datdba  
where pgd.datname IN('hr')  
group by pgd.datname  
order by 1, 2;
```

#### -- Tables

```
SELECT table_catalog, table_schema, table_name, table_type  
FROM INFORMATION_SCHEMA.TABLES  
WHERE TABLE_SCHEMA IN('hr')  
AND TABLE_TYPE='BASE TABLE';
```

#### -- Views

```
SELECT table_catalog, table_schema, table_name, table_type  
FROM INFORMATION_SCHEMA.TABLES  
WHERE TABLE_SCHEMA IN('hr')  
AND TABLE_TYPE='VIEW';
```

#### -- Functions

```
SELECT specific_catalog, specific_name, routine_name, routine_type  
FROM INFORMATION_SCHEMA.ROUTINES  
WHERE specific_schema IN('hr');
```

#### -- Constraints

```
SELECT constraint_schema, constraint_name, table_schema, constraint_type  
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS  
WHERE TABLE_SCHEMA IN('hr')  
order by 1 ,2;
```

#### -- Constraints (Count)

```
SELECT constraint_schema, constraint_type, count(*)  
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS  
WHERE TABLE_SCHEMA IN('hr')  
group by constraint_schema, constraint_type  
order by 1, 2;
```

#### -- Sequences

```
SELECT sequence_schema, sequence_name, start_value, minimum_value, maximum_value,  
increment  
FROM INFORMATION_SCHEMA.SEQUENCES  
WHERE SEQUENCE_SCHEMA IN('hr');
```

#### -- Indexes

```
SELECT pg_database.datname, pg_class.relname  
FROM pg_class, pg_index, pg_database  
WHERE pg_class.oid = pg_index.indexrelid and pg_class.relpowner=pg_database.datdba  
and pg_database.datname not in ('cloudsqladmin', 'template0');
```

#### -- Triggers

```
SELECT trigger_schema, trigger_name, event_manipulation, action_orientation, action_timing,  
action_statement
```



```
FROM information_schema.triggers  
WHERE trigger_schema IN('hr');
```

#### -- Partitions

```
select distinct pgc.oid as "TableID", 'Parent' as "TableType", pgc.relname  
from pg_catalog.pg_class pgc join pg_catalog.pg_inherits pgip  
on pgc.oid=pgip.inhparent  
union  
select pgc.oid as "TableID", 'Child' as "TableType", pgc.relname  
from pg_catalog.pg_class pgc join pg_catalog.pg_inherits pgic  
on pgc.oid=pgic.inherelid  
order by 1;
```

- Method 2 - Using the Ora2Pg comparison [tool](#). Ora2Pg provides an easy-to-use schema objects comparison utility that outputs a comparison result of schema object counts. Use this tool to perform row count comparisons between Oracle as the source and PostgreSQL as the target.

```
$ ora2pg -t TEST -c config/ora2pg.conf > migration_diff.txt  
  
# Example output:  
  
[TEST ROWS COUNT]  
  
ORACLEDDB:COUNTRIES:25  
  
POSTGRES:countries:25  
  
ORACLEDDB:CUSTOMERS:6  
  
POSTGRES:customers:6  
  
ORACLEDDB:DEPARTMENTS:27  
  
POSTGRES:departments:27  
  
...
```



## Azure Database for PostgreSQL Import Process Monitoring

The following are performed during the import process to indicate progress:

- Database active sessions - Shows the current running statements (similar to Oracle V\$SESSION view):

```
select datid,
       datname,
       pid,
       username,
       application_name,
       client_addr,
       state,
       query
  from pg_stat_activity
 where datname='hr';
```

- Database schema size - Schema size increases with the import process progression:

```
select schema_name,
       pg_size.pretty(sum(table_size)::bigint) as "disk space",
       round((sum(table_size) /
              pg_database_size(current_database())),5) * 100 as "percent"
  from (select pg_catalog.pg_namespace.nspname as schema_name,
              pg_relation_size(pg_catalog.pg_class.oid) as
              table_size
         from pg_catalog.pg_class join pg_catalog.pg_namespace
           on relnamespace = pg_catalog.pg_namespace.oid) t
 group by schema_name
 order by schema_name;
```



## Data Migration Main Considerations

Consider the following implications and differences between offline and online data migrations:

- When migrating data between different platforms, consider using a set of dedicated tools. For example, Ora2Pg can perform schema conversion and data export/import, but it is not intended to provide an incremental solution for online data modifications. Ora2Pg does not support CDC.
- The initial data load process, designed to move a large set of static data (according to a specific point-in-time) from the Oracle source database platform to the Azure Database for PostgreSQL database platform, can be performed using a CDC solution such as Azure Data Migration Service ([DMS](#)) with no downtime.
- There are several options to ensure data consistency when exporting data from Oracle to PostgreSQL using Ora2Pg as a data migration tool:
  - You can use downtime to eliminate data modification during data export. This is considered the safest option but is not always possible due to business needs or limitations.
  - Ora2Pg uses the '[serializable](#)' isolation level for transaction data consistency, which requires application downtime (no writes) during the export process. Ora2Pg then creates data files (dump) for each source table to import the exported data into Azure DB for PostgreSQL using the [COPY](#) command (similar to Oracle SQL Loader).
- In cases when downtime is not an option, an alternative solution could be to support the initial load process, then implement a data sync from the last position of the initial load process.
- When the initial load process can be performed using Ora2Pg, but the actual migration would be performed later, after both environments are synced, consider using a CDC solution.
- To capture all data modifications from a specific point-in-time, use a streaming/CDC tool to ensure that all modified data from the Oracle source database is synced into the Azure DB for PostgreSQL target environment.

## High Availability, Backup, and Recovery

### Migrating from Oracle RMAN Backup and Recovery

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**

N/A

#### Differences Summary:

- Oracle backup and recovery do need to be converted as it managed by Azure Database for PostgreSQL backup and restore

Oracle provides Recovery Manager (RMAN) for performing full or incremental backups to mitigate human mistakes, storage failures, and block-level corruption. Although Oracle also supports additional methods, RMAN is a standard and robust tool for managing backup and restore for all Oracle files (data files, archive logs, etc.). Oracle RMAN uses an internal management language (RMAN Scripts) that supports and can only be implemented for Oracle databases.

RMAN creates a physical backup. Logical backups are performed by the Oracle EXPDP/IMPDP utility. Backups are similar for Azure Database for PostgreSQL, where physical backups are done by Azure at the service level and logical backups are created by pg\_dump to export the data to another location.

RMAN supports the following main backup methods:

- Full backup
  - Generates a standalone copy of all data, control files, redo log files, and archive logs of the source Oracle database.
- Incremental backup
  - Backs up only changes to the latest full backup, or from the latest backup performed on an Oracle database. This allows database restore from a full backup to establish a data baseline, while implementing one or more incremental backups to apply additional, more recent changes to the database.
  - Oracle incremental backup types include:
    - Cumulative: Captures all data modifications performed in the database since the last full backup.
    - Differential: Captures all data modifications performed in the database since the last backup. These can be either full or incremental.
- RMAN duplicate
  - Duplicates an entire Oracle database for creating additional and identical environments (QA, DEV, etc.), or for applying the "[Lift and Shift](#)" migration approach. This can be used with Oracle Data Guard once the RMAN duplication is complete to establish primary/standby architecture.

For more information, see [Introduction to Oracle Data Guard](#).

#### Basic RMAN Code Examples

RMAN full backup, including the archive logs:



```
BACKUP DATABASE PLUS ARCHIVELOG;
```

List all available database backups created by RMAN:

```
LIST BACKUP OF DATABASE;
```

RMAN restore database command:

```
RUN {
  SHUTDOWN IMMEDIATE;
  STARTUP MOUNT;
  RESTORE DATABASE;
  RECOVER DATABASE;
  ALTER DATABASE OPEN;
}
```

RMAN (Oracle 12c) restore a specific pluggable database (PDB):

```
RUN {
  ALTER PLUGGABLE DATABASE pdb1, pdb2 CLOSE;
  RESTORE PLUGGABLE DATABASE pdb1, pdb2;
  RECOVER PLUGGABLE DATABASE pdb1, pdb2;
  ALTER PLUGGABLE DATABASE pdb1, pdb2 OPEN;
}
```

### Oracle to Azure Database for PostgreSQL Compatibility Level

None, but not required.

Oracle RMAN scripts are not required to be converted as Azure Database for PostgreSQL. They can be backed up by the service level automatically.

For some use cases, the scripts can be written using the Azure CLI to restore or change backup retention.

See the following for additional information:

- [Oracle RMAN](#)
- [az postgres server](#)



## Migrating from Oracle Datapump Export/Import Utilities and Dump Files

[back to summary](#)

Oracle Datapump is the primary tool for creating logical backups (user-created data). You can use Datapump to:

- Create schema-level object backup for tables, views, indexes, stored procedures, etc.
- Create metadata-level backups and restores.
- Generate dump files that are consistent with the database system change number (SCN) (using the FLASHBACK\_SCN parameter) or the Oracle instance timestamp (using the FLASHBACK\_TIME parameter).
- Restore specific objects or full schemas.
- Migrate small-medium size databases or schema along with CDC/replication products, using consistent export and replicating from a known SCN value to continue CDC until achieving full synchronization.
- Generate a readable text SQL file from a binary dump backup file.

You can use Datapump locally or remotely (export and import). It uses Oracle Directory to interact with the operating system directories and files.

### Oracle Datapump Main Functionalities

- Datapump Export - The EXPDP utility generates a binary dump file that holds the exported data which can be implemented at schema or object level.
- Datapump Import - The IMPDP utility performs the import operations (restore) based on the dump file generated using the EXPDP utility.

### Oracle Datapump Code Example

Use Datapump EXPSP to create a dump file for a schema named 'scott':

```
$ expdp system/**** directory=expdp_dir schemas=scott
dumpfile=expdp_scott.dmp logfile= expdp_scott.log
```

Use Datapump IMPDP to import a schema named 'scott' to a new schema name 'scott\_dev':

```
$ impdp system/**** directory=expdp_dir schemas=scott
dumpfile=expdp_scott.dmp logfile=impdp_scott.log
REMAP_SCHEMA=scott:scott_dev
```

You can use pg\_dump to export data, and it covers most EXPDP options.

See the following for more information:

- [Oracle Datapump](#)
- [Migrating to Azure Database for PostgreSQL pg\\_dump and pg\\_restore](#)

## Migrating to Azure Database for PostgreSQL Backup and Restore

To implement an Oracle RMAN backup-and-restore policy, you need to understand Azure Database for PostgreSQL backups and restores. Azure Database for PostgreSQL generates automatic backups (on locally redundant or geo-redundant storage), which can be used to perform restores to any point in time.

Azure Database for PostgreSQL Backup and Restore has the following capabilities and characteristics:



- Creates automatic backups for data files and transaction logs.
- Depending on the storage size, backups can be full and differential for databases up to 4 TB of storage (some regions support 4 TB), or snapshot backups for databases up to 16 TB.
- Restores can be performed to a point-in-time, according to the configured backup retention period.
- The default retention backup period is seven days, and the maximum retention period can be configured up to 35 days.
- All backups are encrypted using AES 256-bit encryption.
- Backup settings can be set when creating the database server or modified for an existing database server.
- For more information see - [Backup and restore in Azure Database for PostgreSQL - Single Server](#)

### Azure Database for PostgreSQL Backup Frequency

- Database servers with up to 4 TB of storage:
  - Full backups occur once every week.
  - Differential backups occur twice a day.
  - Transaction log backups occur every five minutes.
- Database servers with up to 16 TB of storage:
  - Backups are snapshot-based as physical backups.
  - The first full snapshot backup is performed immediately after a server is provisioned, and that snapshot is retained as the database server base backup. Subsequent snapshot backups are differential backups only.
  - Differential snapshot backups do not occur on a fixed schedule.
  - Three differential snapshot backups are performed daily.
  - Transaction log backups occur every five minutes.

### Azure for PostgreSQL Restore Types

- Point-in-time restore
  - This restore can be performed to any point-in-time within the configured backup retention period.
  - A new server is created in the same Azure region as the original server.
  - The restored database server uses the same (original) configuration for the pricing tier, compute generation, number of vCores, storage size, backup retention period, and backup redundancy options.
  - This type of restore can incorporate user mistakes at the logical level, which may require that data be restored to a previous state.
- Geo-restore
  - This option allows Azure Database for PostgreSQL server restores to another Azure region where the service is available and if the database is configured for geo-redundant backups.
  - This is the default recovery option when the database server is unavailable in the region where it is being hosted.

### Azure Database for PostgreSQL Backup and Restore Code Examples

These examples are performed using the [Azure CLI](#), and can also be done through the console., and can also be done through the console.

- Alter backup retention for an existing database server:

```
az postgres server update --name pgdb1 --resource-group resgrp1 --backup-retention 15
```



- Perform a point-in-time restore according to a specific time:

```
az postgres server restore --resource-group resgrp1 --name pgdb1-restored --restore-point-in-time 2020-06-10T10:41:00Z --source-server pgdb1
```

For more information, see [Azure for PostgreSQL restores](#).

### Migrating to Azure Database for PostgreSQL pg\_dump and pg\_restore

To migrate from the legacy Oracle Datapump utilities (EXPDP and IMPDP) for logical backups and restores, use the Azure Database for PostgreSQL pg\_dump and pg\_restore commands:

- Oracle Datapump EXPDP - Use pg\_dump to extract data from a PostgreSQL database into a dump file.
- Oracle Datapump IMPDP - Use pg\_restore to restore a PostgreSQL database from a dump file created using pg\_dump.

You can run Oracle Datapump EXPDP and IMPDP from a CLI or from PL/SQL code using the DBMS\_DATAPUMP package. Both can be converted into the PostgreSQL native export and import tools: pg\_dump and pg\_restore, which can be run by using a PostgreSQL client holding the pg\_dump and pg\_restore utilities, from a remote server or by using the Azure CLI.

### Oracle Datapump to PostgreSQL pg\_dump and pg\_restore Conversion Example

Oracle Datapump export:

```
expdp system directory=expdp_dir schemas=scott dumpfile=exp_scott.dmp logfile=expdp_scott.log
```

Oracle Datapump import:

```
impdp system directory=expdp_dir schemas=scott dumpfile=expdp_scott.dmp logfile=impdp_scott.log
```

Azure Database for PostgreSQL export conversion:

```
pg_dump -h Azure_PG_Host -U user_name -d db_name -f scott_dump.sql
```

Azure Database for PostgreSQL import conversion:

```
pg_restore -h Azure_PG_Host -U user_name -d db_name -f scott_dump.sql
```

### PostgreSQL Versions 12 and 13 Features and Descriptions

Parameter	Description	PostgreSQL 12	PostgreSQL 13
-----------	-------------	---------------	---------------

pg_dump			
--schema --exclude-schema	<b>Existing parameter</b>  Except patterns.	Supported	Supported
--table --exclude-table	<b>Existing parameter</b>  Except patterns.	Supported	Supported
--extra-float-digits	<b>New parameter</b>  Use a specified value of extra_float_digits when dumping floating-point data.	Supported	Supported
--on-conflict-do-nothing	<b>New Parameter</b>  Add ON CONFLICT DO NOTHING to INSERT commands.	Supported	Supported
PG_COLOR	<b>New Parameter</b>  Specifies whether to use color in diagnostic messages	Supported	Supported
--include-foreign-data=foreignserver	<b>New Parameter</b>  Dump the data for any foreign table with a foreign server matching foreign server pattern.	N/A	Supported
pg_restore			
PG_COLOR	<b>New Parameter</b>  Specifies whether to use color in diagnostic messages.	Supported	Supported

See the following for additional information:

[PostgreSQL pg\\_dump and pg\\_restore](#)

[pg\\_dump and pg\\_restore on Azure](#)

## Migrating from Oracle RAC for High Availability Architecture

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- Azure Database for PostgreSQL utilize different architecture while high-availability and scalability can be achieved by Azure Database for PostgreSQL services.

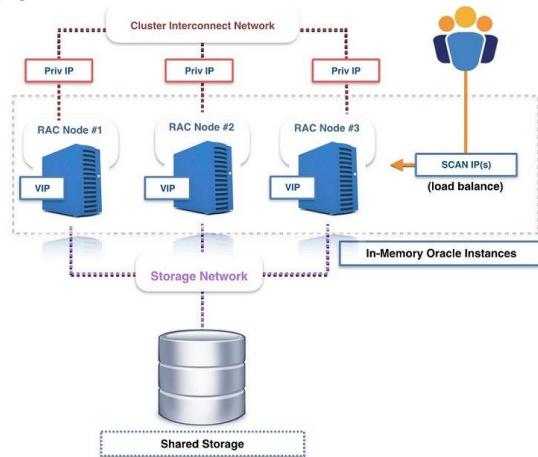
Oracle Real Application Cluster (RAC) is an advanced Oracle database solution for enabling high availability, scalability, high performance, and processing distribution. It includes several Oracle systems, including Grid Infrastructure and Automatic Storage Management (ASM), and is constructed using multiple Oracle database nodes that share the same storage layer.

Shared storage is an essential component in the Oracle RAC architecture; all the cluster nodes can read and write data to the same physical storage (database files), which are stored on disks and can be accessed by all nodes. In addition, Oracle provides a software-based storage and disk management mechanism called Automatic Storage Management (ASM), which is implemented as a set of background processes that run on all cluster nodes and simplifies management of the database storage layer.

### Oracle RAC Main Functionalities

- High Availability: Single point of failure (SPOF) is the shared storage.
- Workload Distribution: Read/write or logical by database role separation.
- Scalability: Adding more database nodes to the cluster improves workload distribution, leading to better overall database performance.

### Oracle RAC Architecture



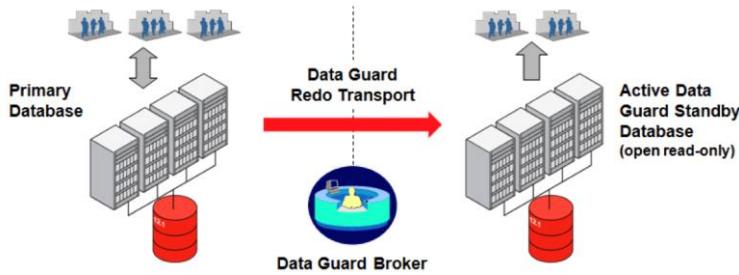
### Migrating from Oracle Data Guard for Disaster / Recovery Architecture

Oracle Data Guard deploys a primary database that accepts both read and write operations, with an additional standby Oracle database for failover. The standby database is constantly synchronized with the primary database by applying each committed transaction performed in the primary database. By default, the standby database is not at OPEN state, and cannot be accessed for read or write operations. In addition, Oracle provides Active Data Guard (ADG), which allows access for real-time read operation.

### Oracle Data Guard Main Functionalities

- In case of failure on the primary database, you can configure automatic failover using the data broker, which handles all failover and synchronization operations.
- You can configure a synchronization delay between the primary database and the standby database to account for human errors, such as deleting records by mistake.
- You can configure both Synchronous and Asynchronous relationships to improve performance and increase data protection between the primary and standby databases.
- The standby database can be temporarily converted to and from a snapshot database (performing data operations), allowing read/write operations.

### Oracle Data Guard Architecture Diagram



**Oracle Data Guard to Azure Database for PostgreSQL Compatibility Level:** None

For more information, see [Oracle RAC](#).

### Migrating to Azure Database for PostgreSQL High Availability, Scaling, and Read/Write Separation

Although Oracle RAC is a proprietary feature that technically cannot be converted, the built-in, high availability support in Azure Database for PostgreSQL can achieve the same core functionality. Azure Database for PostgreSQL, as a single instance architecture, guarantees high availability, with a Service Level Agreement (SLA) of 99.99% uptime. In addition, Azure Database for PostgreSQL allows the database to remain running during planned maintenance operations, wherein most other circumstances downtime is inevitable.

### Azure Database from PostgreSQL Deployment Types

- Single Server
  - A managed, PostgreSQL database service that provides the following:
    - High Availability
    - Dynamic scaling
    - Read/write separation by adding read replicas
    - Security support (encryption for data at rest and in transit)

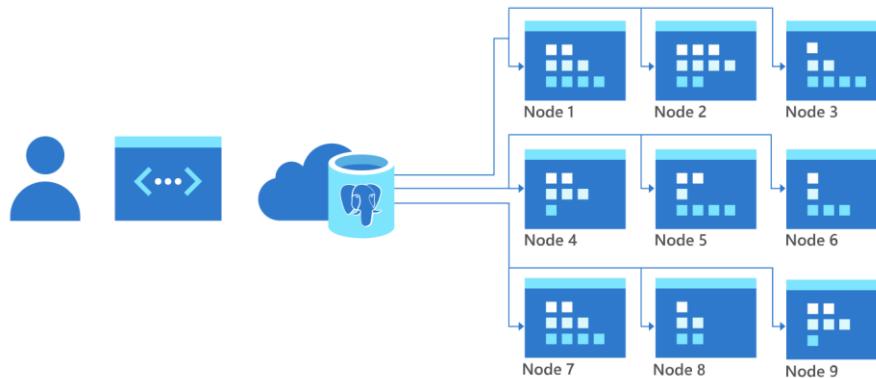
- Automatic backups and point-in-time restore capabilities
  - Metrics-based monitoring and alerting, [Azure PostgreSQL Concepts and Monitoring](#).
- Consider using Single Server when you have the following specific workload types and requirements (infrastructure deployment differences):
  - Basic (suited for getting started, QA or Dev environments)
  - General purpose
  - Memory optimized
- Hyperscale (Citus)
  - Offers horizontal scaling of SQL queries across multiple Azure Databases for PostgreSQL nodes using sharding.
  - The nodes in a server group collectively can hold more data and use more CPU core processing than would be possible on a single server.
  - The Hyperscale (Citus) architecture allows the database to scale by adding more nodes to the server group to support dynamic workload scenarios.
  - The Hyperscale (Citus) architecture consists of two main components:
    - Coordinator
    - Workers

Every Hyperscale (Citus) server group has a coordinator node and multiple workers. The application sends SQL queries to the coordinator node, which addresses it to the relevant workers and accumulates the results.

Applications cannot connect directly to workers.

One of the most significant features of the Hyperscale (Citus) solution is the ability to distribute tables (sharding), meaning that it is possible to store different rows on different worker nodes, thereby providing high performance.

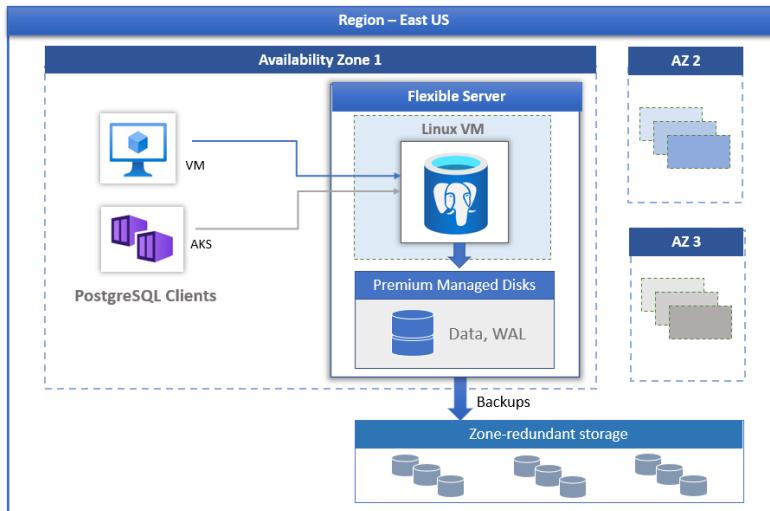
#### Azure Database for PostgreSQL Hyperscale (Citus) Diagram



For more information, see [Azure Database from PostgreSQL Single Server Hyperscale \(Citus\)](#) or the [quick start guide](#).

- Flexible server (Preview)

Flexible server is a fully managed database service that provides more granular control and flexibility of database management functions and configuration settings, based on user requirements. With flexible server, you can collocate the database engine with the client-tier for lower latency and choose high availability within a single availability zone and across multiple availability zones. Flexible servers also provide better cost optimization controls, with the ability to stop and start your server as well as burstable compute tier that is ideal for workloads that do not need full compute capacity continuously.



#### Azure Database for PostgreSQL High Availability - Single Server Main Components

- PostgreSQL Database Server
  - Azure Database for PostgreSQL provides the infrastructure to support mission critical databases such as: security, high availability, scaling-by-demand, and fast recovery from multiple, catastrophic scenarios.
  - The Azure Database for PostgreSQL software and Azure storage are separated in order to support many of these modern requirements - PostgreSQL data modifications are stored synchronously in the form of Write Ahead Logs (WAL) on Azure Storage, which is attached to the related database server. During the database checkpoint process, data pages from the database server memory are also flushed to the storage.
- Azure Storage
  - All PostgreSQL physical data files and WAL files are stored on Azure Storage, which is architected to store three copies of data within a region to ensure redundancy, availability, and reliability.
  - The Azure Database for PostgreSQL storage layer is autonomous from the database server; it can be detached from a failed database server and reattached to a new database server.
  - Azure Storage continuously monitors for any storage failures (such as block corruption), and when found, automatically fixes the issue by replacing the faulty data blocks.
- Gateway - Acts as a database proxy, routing all client connections to the database server.

#### High Availability During Planned Operations and Unplanned Downtime – Single Server

- Planned Operations



- Maintenance operations (such as increasing the database server storage) can be performed without downtime by using remote storage for fast detach/re-attach after the failover.
- Allows for scaling operations (up/down) without downtime.
- Unplanned Operations
  - During failures such as hardware malfunctions or networking issues leading to unplanned downtime, a new database server is automatically provisioned in seconds, including failover to the new database server, ensuring business continuity.
  - Remote storage is automatically attached to the new database server while the Azure Database for PostgreSQL engine performs recovery operations using WAL and database files, allowing clients to easily connect to the database server.

For more information, see [Azure Database for PostgreSQL High Availability](#).

### Adding Read-Replicas for Azure Database for PostgreSQL - Single Server

The Azure Database for PostgreSQL can support up to five read-replica instances.

The read-replica database server is replicated from the master database server, while acting as read-only database servers. The replication method is asynchronous, using the PostgreSQL engine native replication system.

Using Azure Database for PostgreSQL read-replicas allows workload distribution for read/write operations. While the master database server can accept both reads and writes, read replicas can serve as a read-only database server, accepting some of the read workload (such as reports, BI analytics, etc.).

Read-replica database servers can be deployed in a different region from the master server region, which improves data center disaster-recovery scenarios, and also enables allocating certain read-replica database servers closer to the user location.

#### Azure CLI Examples

Add a read-replica database server:

```
az postgres server replica create --name pgrr1 -replica --source-server pgdb1 --resource-group pgrsgrp
```

Add a read-replica database server on a different region:

```
az postgres server replica create --name pgrr2 -replica --source-server pgdb1 --resource-group pgrsgrp --location southcentralus
```

List available read-replica database servers:

```
az postgres server replica list --server-name pgdb1 --resource-group pgrsgrp
```

For more information, see [Azure Database for PostgreSQL read-replicas](#).

### Azure Database for PostgreSQL High Availability - Hyperscale (Citus)

To avoid database server downtime, Azure Database for PostgreSQL - Hyperscale (Citus) provides high availability by assigning every node in a server group a standby (resilient) node.



If the main node (Coordinator) becomes unavailable or unhealthy, its standby instance is promoted to replace it. The standby instance accepts any incoming connections previously directed to the failed instance.

You can configure high availability during creation of the Azure Database for PostgreSQL Hyperscale (Citus) server group, or post-deployment, using the "Configure" tab for the server group on the Azure portal.

Hyperscale (Citus) runs proactive health checks on every node in the server group, and after four failed checks it determines that a specific node is down. Hyperscale then promotes a standby instance as a primary (failover) and provisions a new standby for the new primary instance.

For more information, see [Hyperscale \(Citus\) High Availability](#).

## Summary

Whether migrating from Oracle RAC as a high availability, scaling, and work distribution solution, or from Oracle Data Guard as a disaster/recovery solution with its primary and standby architecture, Azure Database for PostgreSQL offers an enterprise-ready, minimally managed solution for mission-critical database requirements. It supports high availability, scaling, and read/write separation with automatic failovers and add read replicas feature. The solution is idea in scenarios where:

- Real-time analytics require sub-second response times across billions of records.
- Multi-tenant database is supported.
- There are dynamically growing SaaS applications with storage size needs being scaled up constantly.

## Migrating from Oracle Flashback Database

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- PostgreSQL does not support Oracle Flashback Database as a direct feature but utilizing Azure Database for PostgreSQL point-in-time restore can be used as an alternative solution.

Oracle Flashback supports returning an entire database to a previous state or point-in-time, or restoring a specific table after a significant logical error, such as accidentally deleted data or dropped objects. Oracle Flashback captures all data modifications applied to a database, essentially storing previous versions of database modifications in the configured database OS location known as the Fast Recovery Area.

### Oracle Flashback Code Examples

Create an Oracle Flashback database restore point:

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

Use Oracle Flashback database to restore to a restore point:

```
FLASHBACK DATABASE TO RESTORE POINT release;
```

Revert to a previous point in time:

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

For more information, see [Oracle RMAN](#).

### Migrating to Azure Database for PostgreSQL Backup and Restore

Azure Database for PostgreSQL offers point-in-time restores, which you can use much like you would Oracle Flashback to return to a previous database state. You can perform a point-in-time restore within the defined backup retention period, creating a new database server in the same Azure region based on the original database server configuration (hardware, backup retention period, and backup redundancy option).

Use the point-in-time restore utility to restore to a clean prior state. Like Oracle Flashback, the Azure Database for PostgreSQL point-in-time restore is useful for compromised data scenarios. For example, database users could perform incorrect DML operations, causing data integrity issues or dropping database objects such as tables or even compete databases. The point-in-time restore feature supports only database-level restores and not object-level restores like Oracle Flashback.

## Azure Database for PostgreSQL Backup and Restore Main Features

- Creates automatic backups for data files and transaction logs.
- Depending on the supported maximum storage size, we either take full and differential backups (4-TB max storage servers) or snapshot backups (up to 16-TB max storage servers).
- These backups allow you to restore a server to any point-in-time within your configured backup retention period.
- The default retention backup period is seven days, and the maximum retention period can be configured up to 35 days.
- Restores can be performed to a point-in-time, according to the configured backup retention period.
- All backups are encrypted using AES 256-bit encryption.
- Backup settings can be set when creating the database server or modified for an existing database server.

## Point-in-Time Restore Example Using Azure Cloud Shell

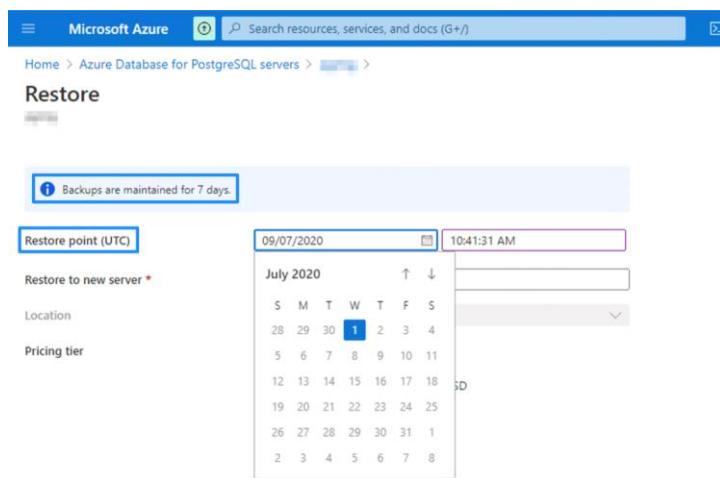
Perform a point-in-time restore:

```
az postgres server restore --resource-group resgrp1 --name pgdb1-restored --  
restore-point-in-time 2020-06-10T10:41:00Z --source-server pgdb1
```

- Resource group: The resource group for the source PostgreSQL database server.
- Name: The name of the new database server to be created based on the source database server backup.
- Restore point-in-time: The selected time to which the PostgreSQL database server needs to be restored; must be within the configured backup time period. Specify the local time zone, such as 2020-06-10T10:41:00-08:00, or use the UTC Zulu format, for example 2020-06-10T10:41:00Z.
- Source server: The Azure Database for PostgreSQL name or ID of the source database server to restore from.

## Point-in-Time Restore Example Using the Azure Console

Select the restore to point-in-time required date, time, and restore target server:





For more information, see [Azure Database for PostgreSQL point-in-time recovery](#).

## Database Architecture, Security, and Miscellaneous Features

### Migrating from Oracle Data Loading/Unload Tools (SQL\*Loader) to Azure Database for PostgreSQL COPY Command

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

#### Differences Summary:

- Expect different methods (commands) for achieving the same purpose when converting Oracle SQL\*Loader to PostgreSQL COPY command

Oracle provides the SQL\*Loader utility for loading data from external files, such as CSV files. You can configure SQL\*Loader (SQLLDR) using the SQL\*Loader control file, which contains metadata and process configurations, such as defining the source external file and the target Oracle table. SQL\*Loader supports both fixed and variable source files.

#### Oracle SQL\*Loader Code Example

Control file:

```
$ cat employees.csv
load data
infile '/home/data/employees.csv'
into table hr.employees
fields terminated by ","
(id, emp_name, dept, salary)
```

Parameters:

- "infile": The operation system path for the source data file.
- "into table": Oracle target table in which the data will be loaded.
- "fields terminated by": Delimiter used in the data file.
- Column names: Oracle table columns to be associated with the load process.

Target table:

```
create table employee (
id integer,
Emp_name varchar2(10),
```



```
dept varchar2(15),
salary integer
);
```

Apply SQL\*Loader data load:

```
$ sqlldr user/password control=<path to the control file>
```

### Migrating to Azure Database for PostgreSQL COPY Command

Although Oracle SQL\*Loader is not supported by PostgreSQL, its core functionality can be simulated by using the PostgreSQL COPY command. Use COPY to transfer data from local standard files (stored at the OS file-system level) to the Azure Database for PostgreSQL tables. You can run COPY from the PostgreSQL command-line interface (CLI), and it supports any single-character delimiter (comma, tab, etc.) or fixed-sized columns in the input files. The source file must reside on the same machine from which the COPY command is run, and cannot be accessed directly from [Azure Blob storage](#).

You can use COPY in the following scenarios:

- PostgreSQL-to-PostgreSQL migration
- CSV load; can use exported CSV from another database type
- Copy data from the web (by URL)

Unlike pg\_dump, COPY requires no space on the client side (data copied through the pipe) and in most cases is faster. However, COPY is useful mostly for data and not for other schema objects like views, functions, and sequences.

Additionally, you can use Azure Data Factory to copy data into and out of Azure Database for PostgreSQL tables. For more information, see [Azure Data Factory connector to Azure Database for PostgreSQL](#).

When loading data from a standard file into a PostgreSQL table, the file structure and the table schema should be identical to eliminate compatibility errors.

### PostgreSQL COPY Command Example (PostgreSQL CLI)

Use '\COPY' to eliminate the "ERROR: must be superuser to COPY to or from a file" error notification.

```
\COPY <tableName> FROM '/data/employees.csv' WITH csv;
```

### PostgreSQL COPY Command Example (Client CLI)

```
$ psql -U <userName> -h <Azure-Server> -d <DB_Name> -c "\copy employees from
'/data/employees.csv' WITH csv;" -W
```

### Using the WHERE Clause

In PostgreSQL 12, the optional WHERE clause was added and can be used with the COPY command to filter rows that are processed.



Sub-queries are not allowed in WHERE used with the COPY command.

See the following for additional information:

- [Oracle SQL\\*Loader](#)
- [PostgreSQL COPY Syntax](#)
- [Microsoft Moving data with PostgreSQL COPY and \COPY commands](#)

## Migrating from Oracle Security Model: Users Roles and Permissions

[back to summary](#)

**General compatibility level:**



**Ora2pg automation capability:**



### Differences Summary:

- Expect different terminology between Oracle and PostgreSQL from security and user management perspectives, nonetheless, migrating from Oracle security to Azure Database for PostgreSQL is supported with a wide selection of options.

An Oracle user or a schema is an entity that owns database objects such as tables, views, and functions, and can act according to specific permissions granted by the database administrator. (In Oracle terminology, an Oracle user is identical to a database schema.) Oracle database users (individuals or applications) can authenticate at the database level with the username and database password, and then run database operations. In addition, Oracle users can be limited in resources; for example, limiting a user by usable database storage size (setting quota usage on the user allocated Tablespace).

Oracle roles serve as a collection of specific permissions that can be granted according to specific business functionality, such as analytics department, DBA administrators, specific applications, etc.

Oracle users can be authenticated using several methods:

- At the database level by assigning database user and password.
- At the OS level by external tools such as Kerberos or third-party software.
- At the organization level by using Active Directory/LDAP or Oracle Internet Directory.

### Oracle 12c Users

To address the Oracle 12c multitenant architecture, Oracle introduced two new types of users:

- Local Users
  - Can be created only in a specific pluggable database (PDB).
  - Multiple database users can have the same username, created on multiple Pluggable Databases (PDBs); not supported before Oracle 12C.
- Common Users
  - Created in all database containers: root (CDB), and (PDB).
  - Must use the 'C##' prefix in the username (for example, c##test\_user).

### Oracle Create User and Role Examples

Create new user syntax:

```
CREATE USER username  
  IDENTIFIED BY password  
  [DEFAULT TABLESPACE tablespace]  
  [QUOTA {size | UNLIMITED} ON tablespace]  
  [PROFILE profile]
```



```
[PASSWORD EXPIRE]  
[ACCOUNT {LOCK | UNLOCK}];
```

Create a new user (simple) with permission grant:

```
CREATE USER <userName> IDENTIFIED BY <password>;  
GRANT CONNECT TO <userName>;
```

Create a new role:

```
CREATE ROLE dba_admin;  
GRANT CONNECT, ALTER SYSTEM, ALTER DATABASE TO dba_admin;  
REVOKE CONNECT, ALTER SYSTEM, ALTER DATABASE FROM dba_admin;
```

For more information, see [Oracle Users](#).

### Migrating to Azure Database for PostgreSQL Roles

User and roles in PostgreSQL have different meanings than they do in Oracle. User and roles in PostgreSQL share the same functionality as a database entity with the permission to connect, own database objects, and perform database operations. A role can be considered a user, a group, or both, depending on how it is used. PostgreSQL roles that are not configured with login permissions can be considered identical to Oracle database roles.

PostgreSQL roles and Oracle 12c roles are similar, as they are global in database scope (applies for all databases in a specific Oracle instance).

Azure Database for PostgreSQL users/roles should be distinguished between the following:

- Single Server or Flexible Server
- Hyperscale (Citus)

### Azure Database for PostgreSQL [Single Instance](#) - Adding Roles

The database admin user and password must be set during the database initial configuration via the Azure console or Azure CLI. This connection should be used for creating any additional roles other than the built-in PostgreSQL roles which are:

- azure\_pg\_admin
- azure\_superuser
- User-defined admin user

### Examples

Use the following SQL metadata statement to view existing roles:

```
SELECT rolname FROM pg_roles;
```

Note that the user defined admin user is a member of the azure\_pg\_admin role with the following privileges:



- LOGIN
- NOSUPERUSER
- INHERIT
- CREATEDB
- CREATEROLE
- NOREPLICATION

In addition, note that `azure_superuser` is a reserved role for managing the Azure Database for PostgreSQL as a service and cannot be accessed by the user-defined admin user.

Add an additional admin user to an Azure Database for PostgreSQL database:

```
CREATE ROLE <new_user> WITH LOGIN NOSUPERUSER INHERIT CREATEDB CREATEROLE  
NOREPLICATION PASSWORD '<Password!>';  
  
GRANT azure_pg_admin TO <new_user>;
```

Adding additional database users to an Azure Database for PostgreSQL database:

```
CREATE DATABASE <newdb>;  
  
CREATE ROLE <db_user> WITH LOGIN NOSUPERUSER INHERIT CREATEDB NOCREATEROLE  
NOREPLICATION PASSWORD '<Password!>';  
  
GRANT CONNECT ON DATABASE <newdb> TO <db_user>;
```

Grant additional privileges to an existing role using the admin user:

```
GRANT ALL PRIVILEGES ON DATABASE <newdb> TO <db_user>;
```

Test the connection using a newly created user:

```
$ psql --host=azureserver.postgres.database.azure.com --port=5432 --  
username=db_user@mydemoserver --dbname=newdb
```

### Azure Database for PostgreSQL [Hyperscale \(Citus\)](#) - Adding Roles

Applying PostgreSQL roles for Azure Database for PostgreSQL Hyperscale (Citus) requires an additional role to manage Hyperscale as a managed PaaS service.

Access using PostgreSQL superuser role is allowed only by Microsoft, so Hyperscale introduced the "citus" role, which has the following permissions:

- Read all configuration variables, even variables normally visible only to superusers.
- Read all `pg_stat_*` views and use various statistics-related extensions -- even views or extensions normally visible only to superusers.
- Execute monitoring functions that may take ACCESS SHARE locks on tables, potentially for a long time.
- [Create PostgreSQL extensions](#) (because the role is a member of `azure_pg_admin`).

The "citus" role **cannot**:

- Create roles
- Creating databases

### Adding Additional Roles for a Hyperscale (Citus) Managed Service

Add additional roles for a Hyperscale Managed service via the Azure Portal. The "citus" role cannot add additional roles using the Azure CLI (Home > Hyperscale > Roles > + Add).

The screenshot shows the Azure portal interface for managing roles in a PostgreSQL server group named 'pghyper1'. On the left, there's a sidebar with options like Overview, Activity log, Tags, Compute + storage, Networking, Connection strings, and Roles. The 'Roles' option is highlighted. In the main content area, there's a 'Server group roles' section with a note about new user roles being created on all nodes. A modal window titled 'Add role' is open, prompting for 'Role name' (with 'citus' entered), 'Password' (with 'New password' entered), and 'Confirm password' (with 'Re-enter new password' entered). There are 'Save' and 'Discard' buttons at the bottom of the modal.

The user is created on the coordinator node of the server group and distributed to all Hhyperscale (Citus) worker nodes.

Roles created through the Azure portal are assigned with the LOGIN permission--users who can sign into the database.

Once the role is created, you can test the connection from a remote PostgreSQL client:

```
$ psql "host=<Name>.postgres.database.azure.com port=5432 dbname=citus
user=<RoleName> sslmode=require"

Password for user <RoleName>:

psql (12.3, server 11.9)

SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits:
256, compression: off)

Type "help" for help.

citus=>
```



Add Hyperscale (Citus) database permissions to a new role, applying all privileges for a specific table:

```
GRANT ALL PRIVILEGES ON DATABASE <DBName> TO <RoleName>;
```

Hyperscale (Citus) propagates single-table GRANT statements through the entire cluster, applying them on all worker nodes. It also propagates system-wide GRANTS (for example, for all tables in a schema) to the coordinator node and distributes it to all of the worker nodes:

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO <RoleName>;
```

Deleting an existing Hyperscale (Citus) role or changing a password must be done from Azure Portal because the "Citus" role is not allowed to perform this operation:

The screenshot shows the Azure portal interface for managing roles in a PostgreSQL server group. The left sidebar shows navigation options like Overview, Activity log, Tags, Settings, Compute + storage, Networking, Connection strings, and Roles. The Roles section is currently selected. The main content area displays a table of roles with two entries: 'citus' and 'pgadmin'. The 'pgadmin' entry has a context menu open, with the 'Delete' option highlighted. Other visible options in the menu include 'Reset the password' and three dots (...).

For more information, see [PostgreSQL role](#).

### Enabling Active Directory Authentication with Azure Database for PostgreSQL

Microsoft Azure Active Directory (Azure AD) authentication layer can be integrated with Azure Database for PostgreSQL to establish a connection using identities defined in Azure AD.

With Azure AD authentication, you can manage database user identities and other Microsoft services in a central location, which simplifies permission management.

For more information, see [Active Directory Authentication with Azure Database for PostgreSQL](#).

## Oracle Essential V\$ views and Their Target Database Equivalents

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- Oracle V\$ views and internal objects names will be required to be altered to PostgreSQL system catalog objects.

### Oracle Dictionary

Oracle dictionary (database metadata) is a collection of internal tables and views that provides information about the instance itself. For example, information about database objects (tables, views, stored procedures, etc.), users, permissions, constraints, database logical and physical layout (tablespaces and data files), and more, can be described at a very granular level.

Use the following statement to query Oracle dictionary table and views information:

```
SELECT * FROM DICTIONARY;
```

Oracle distinguishes dictionary views by three main permission levels:

- DBA\_<TableName> - Can be accessed only by an Oracle user with the DBA role.
- ALL\_<TableName> - Can be accessed by all Oracle users (for general dictionary data).
- USER\_<TableName> - Can be accessed only by the user itself, by a user with the DBA role, or by users that have the appropriate permissions.

### Examples of Oracle Dictionary Views

Table/View Name	Description
DBA_USERS	Provides metadata information about all users in the database.
DBA_TABLES	Provides metadata information about all tables in the database.
DBA_VIEWS	Provides metadata information about all views in the database.
DBA_DATA_FILES	Provides information about all physical datafiles in the database.
USER_TABLES	Provides metadata information about all tables for the connected user.
USER_TAB_COLS	Provides metadata information about user tables columns.

For more information, see [Oracle Dictionary](#).

### Oracle V\$ - Dynamic Performance Views

The Oracle V\$ dynamic performance views are a collection of internal sources of real-time statistical information about the Oracle instance state and configuration. The V\$ views provide run-time operation data at a very wide



range of layers; for example, users sessions, resource usage, wait events, locking events, SQL statement execution analysis, and more. The V\$ views are continuously updated as the Oracle database is running. Access to the Oracle V\$ views is allowed by default to the SYS and SYSTEM users and can be granted to additional users if required (for example, monitoring users).

#### Examples of Oracle V\$ common views:

Table/View Name	Description
V\$DATABASE	Displays information about the database from the control file.
V\$SESSION	Lists session information for each current session.
V\$INSTANCE	Displays the state of the current instance.
V\$LOCKED_OBJECT	Displays information about all objects with active locks.
V\$SESSION_LONG_OPS	Displays the status of various operations that run for longer than 6 seconds (in absolute time).
V\$MEMORY_TARGET_ADVICE	Provides information about how the MEMORY_TARGET parameter should be sized based on current sizing and satisfaction metrics.

For more information, see [Oracle V\\$](#).

#### Migrating to Azure Database for PostgreSQL System Dictionary and PostgreSQL Statistic Collector

As in Oracle, PostgreSQL also provides database dictionary (metadata) and real-time statistical information in the form of tables and views. PostgreSQL dictionary information and database statistical information about operations occurring in real time can be found and queried using the following tables/views or tools:

- **PostgreSQL information schema**
  - Consists of a set of views that contain information about the objects defined in the current database.
  - Information schema views do not contain information about PostgreSQL specific features. To inquire about specific features, it must query the system catalogs or other PostgreSQL-specific views.

#### Examples of PostgreSQL information schema views:

View Name	Description
key_column_usage	Identifies all columns in the current database that are restricted by some unique primary key or foreign key constraint
parameters	Contains information about the parameters (arguments) of all functions in the current database.
routines	Contains all functions and procedures in the current database, only those functions and procedures that the current user has access to are shown.



sequences	Contains all sequences defined in the current database; only those sequences that the current user has access to are shown.
tables	Contains all tables and views defined in the current database; only those tables and views that the current user has access to are shown.

For more information, see PostgreSQL [Information Schema](#).

- PostgreSQL system catalogs
  - Hold regular tables that store the database schemas metadata, information about tables and columns, and internal database management information.
  - Updated by the PostgreSQL management system when applying on-going operations such as CREATE DATABASE or CREATE TABLE.

#### Examples of PostgreSQL system catalog views

View Name	Description
pg_database	Stores information about the available databases
pg_index	Contains information about all indexes in the database
pg_policy	Stores row level security policies for tables. A policy includes the kind of command that it applies to (possibly all commands), the roles that it applies to, the expression to be added as a security-barrier qualification to queries that include the table, and the expression to be added as a WITH CHECK option for queries that attempt to add new records to the table.
pg_tables	Show information about all tables in the database, such as indexes and the tablespace for each database table
pg_cursors	List of currently available/open cursors

For more information, see PostgreSQL [System Catalog](#).

- PostgreSQL statistics collector
  - A PostgreSQL subsystem that supports statistical internal data collection and reporting of information about PostgreSQL database server activity. For example, PostgreSQL statistic collector can count accesses to tables and indexes in both disk-block and individual-row terms.
  - Can track total number of rows and information about vacuum and analyze actions for each table. It can also count calls to user-defined functions (UDFs) and the total time spent in each function.

#### Examples of PostgreSQL statistical collector views

View Name	Description
pg_stat_activity	Show information related to the activity of a process



pg_stat_all_tables	Show performance statistics on all tables in the database, such as identifying table size, write activity, full scans vs. index access and more
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_archiver	One row only, showing statistics about the WAL archiver process's activity
pg_stat_all_indexes	One row for each index in the current database, showing statistics about accesses to that specific index

For more information, see PostgreSQL [Statistical Collector](#).

#### Oracle and PostgreSQL comparison table for dictionary and statistical data

Property	Oracle Object	PostgreSQL Object
<b>Database Users</b>	DBA_USERS	PG_USER
<b>Database Tables</b>	DBA_TABLES	PG_TABLES
<b>Table Privileges</b>	DBA_TAB_PRIVS	TABLE_PRIVILEGES
<b>Database Roles</b>	DBA_ROLES	PG_ROLES
<b>Table Columns</b>	DBA_TAB_COLS	PG_ATTRIBUTE
<b>Database Locks</b>	V\$LOCKED_OBJECT	PG_LOCKS
<b>Database Information</b>	V\$DATABASE	PG_DATABASE
<b>Database Sessions</b>	V\$SESSION	PG_STAT_ACTIVITY
<b>Runtime Parameters</b>	V\$PARAMETER	PG_SETTINGS
<b>IO Operations</b>	V\$SEGSTAT	PG_STATIO_ALL_TABLES

- Azure Database for PostgreSQL Monitoring
  - Azure Database for PostgreSQL provides multiple metrics that can provide observation, insights, and the ability to identify specific events occurring at the database level. These include real-time activities that can impact database performance.
  - Each metric is emitted at a one-minute frequency and has up to 93 days of history.
  - Metrics include (partial list):
    - cpu\_percent
    - memory\_percent
    - storage\_percent
    - active\_connections
    - connections\_failed
- Query Store
  - Provides a way to track query workloads over time.



- Simplifies performance troubleshooting by helping to quickly find the longest running and most resource-intensive queries.
- Automatically captures a history of queries and runtime statistics and retains them for further analysis.
- By default, the Query Store is disabled. To enable it on the server, perform the following actions in the Azure Portal:
  - Sign into the Azure portal and select your Azure Database for PostgreSQL server.
  - Select Server Parameters in the Settings section of the menu.
  - Search for the pg\_qs.query\_capture\_mode parameter.
  - Set the value to TOP and Save.
- Accessing Query Store Information:
  - Query Store data is stored in the azure\_sys database on the Azure Database for PostgreSQL server.
  - The following query returns information about queries in Query Store:

```
SELECT * FROM query_store.qs_view;
```

- The following query returns information about wait events stats:

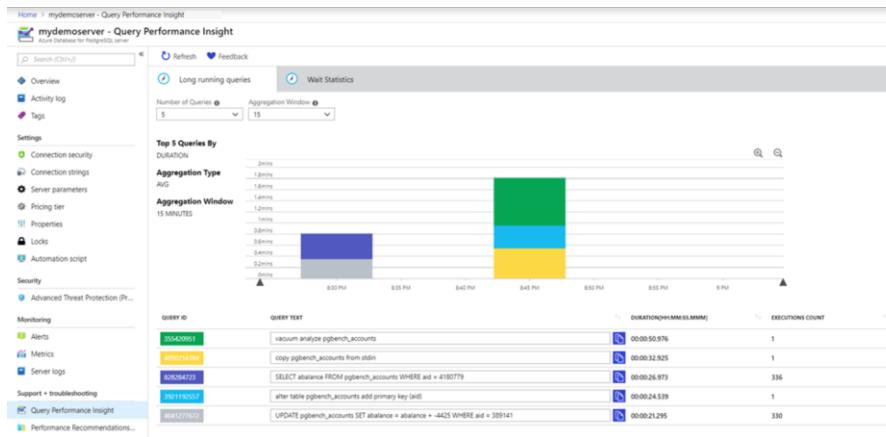
```
SELECT * FROM ;
```

- Return a complete list of all Query Store tables:

```
SELECT * FROM pg_catalog.pg_tables
WHERE schemaname != 'pg_catalog'
AND schemaname != 'information_schema';
```

For more information, see [Query Store](#).

- Query Performance Insight
  - You can view and analyze database operational and performance metrics using Azure Portal Query Performance Insight, which can help identify specific workloads that have an impact on database performance.
  - Query Performance Insight works in conjunction with Query Store to provide visualizations accessible from the Azure portal.
  - To view the Query Performance Insight visualization charts:
    - Sign into the Azure portal and select your Azure Database for PostgreSQL server.
    - Select Azure Database for PostgreSQL database server.
    - Select Query Performance Insight under the Intelligent Performance section of the menu bar.



For more information, see [Query Performance Insights](#).

- **Server Logs**

By enabling logging on your server Azure Database for PostgreSQL, resource logs can be sent to Azure Monitor logs, Event Hubs, and a Storage Account for further analysis.

For more information, see [Server Logs](#).

## PostgreSQL versions 12 and 13 New Features

Feature/parameter	Description	PostgreSQL 12	PostgreSQL 13
Statistics Collector	pg_stat_gssapi	Supported	Supported
	pg_stat_progress_create_index	Supported	Supported
	pg_stat_progress_cluster	Supported	Supported
New views	pg_stat_slru	N/A	Supported
	pg_stat_progress_basebackup	N/A	Supported
	pg_stat_progress_analyze	N/A	Supported
System Catalogs	pg_statistic_ext_data	Supported	Supported
New views	pg_stats_ext	Supported	Supported

## Multi-Tenant Architecture: PDBs and CDB databases

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- PostgreSQL is built on different architecture, but the same Oracle multi-tenant database structure can be achieved with Azure Database for PostgreSQL.

Oracle database version 12c introduced *multi-tenant architecture* that differs significantly from previous Oracle database releases. Previous versions of Oracle database architecture included the option to deploy only one database per instance. Additional databases could be added only with a new instance, and each instance required that you define a dedicated resources configuration.

Starting with Oracle 12c, one Oracle instance can hold multiple databases in the form of Pluggable Databases (PDBs) under a root database named the Container Database (CDB). Additional versions of Oracle databases, such as Oracle 18c/19c, are based on the Oracle 12c architecture.

### Main Features of Oracle 12c Multi-Tenant Architecture

- An Oracle 12c CDB can support one or more PDBs.
- A single Oracle 12c PDB shares resources such as instance memory and background processes.
- PDBs enable consolidation of many databases and applications into individual containers under the same Oracle instance hosting multiple databases under a single Oracle software installation.
- Each PDB has its own set of SYSTEM and user-defined tablespaces.
- A single CDB (or CDB\$ROOT) exists in a CDB and contains Oracle Instance Redo Logs, undo tablespace (unless Oracle 12.2 local undo mode is enabled), and control files.
- To generate additional PDBs, the seed PDB can be used as the template. (The seed PDB is in the CDB.)

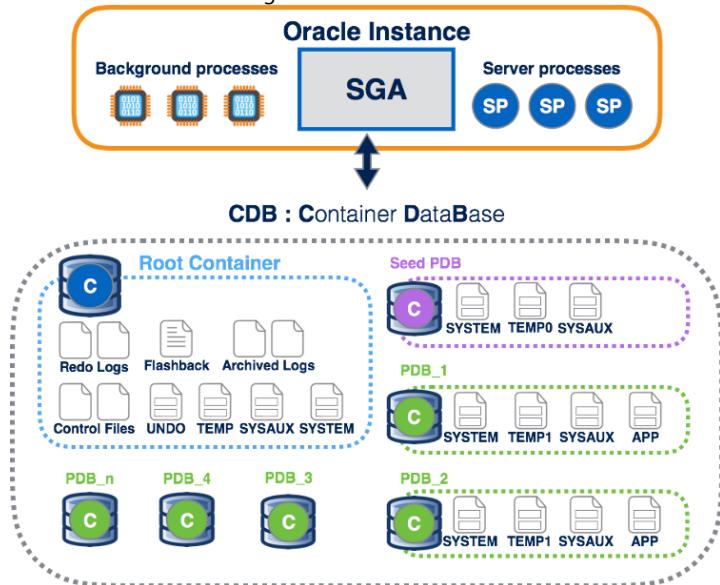
### Oracle 12c CDB

- Main database that holds the root container (cdb\$root, one per Oracle instance).
- Hold the data dictionary for the root container and for all PDBs.
- Is generated as part of the Oracle 12c software installation.
- Includes the Oracle control files, its internal set of system tablespaces, the instance undo tablespaces (unless Oracle 12.2 local undo mode is enabled), and the instance redo logs.

### Oracle 12c PDB

- An independent database that exists under the Oracle CDB.
- Platform that hosts the user/application data, while the CDB is for database administration only (dictionary/metadata).
- Can be created from the pdb\$seed (template database) or as a clone of an existing PDB.
- Has its own set of application and system data files and tablespaces, along with temporary files to manage objects and metadata information.

### Oracle 12c Multi-Tenant Architecture Diagram



### Oracle 12c Multi-Tenant Main Functionalities

- Oracle 12c PDBs can be cloned and transported to different CDBs/Oracle instances.
- Oracle 12c PDBs can be used as a portable collection of schemas.
- Oracle 12c PDBs can be used to separate applications from one another on a single Oracle deployment (and not by schema-level separation).
- The multi-tenant architecture enables database consolidation, which is based on many individual applications sharing a single Oracle server.
- Backups are supported at both the multi-tenant CDB level as well as on the individual PDB level for both physical and logical backups (RMAN and Datapump).
- The multi-tenant architecture provides an easier way to patch and upgrade individual clients and applications using PDBs.

### Oracle 12c Code Examples

Create a new PDB from the pdb\$seed template:

```
CREATE PLUGGABLE DATABASE PDB1 admin USER <AdminUser> IDENTIFIED BY
<password> FILE_NAME_CONVERT('/pdbseed/','/pdb1/');
```

List available PDBs:

```
SHOW PDBS;
```



CON_ID	CON_NAME	OPEN MODE	RESTRICTED
-----	-----	-----	-----
2	PDB\$SEED	READ ONLY	NO
3	PDB1	READ WRITE	NO

Create a new PDB by cloning from an existing PDB:

```
CREATE PLUGGABLE DATABASE PDB2 FROM PDB1 FILE_NAME_CONVERT=
('/pdb1/','/pdb2/');
```

For more information, see [Oracle 12c Multi-Tenant Architecture](#).

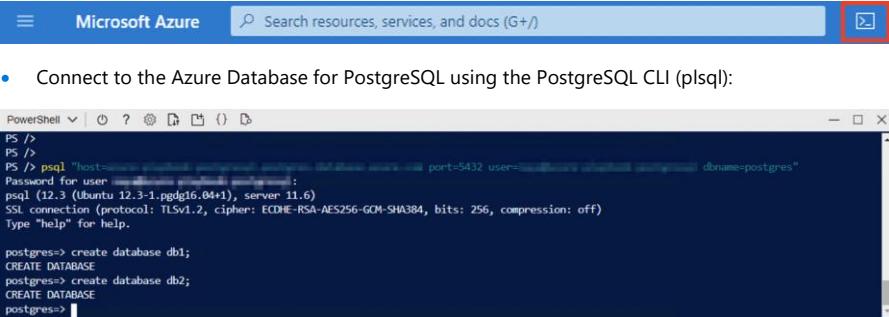
### Migrating to Azure Database for PostgreSQL Databases Architecture

Azure Database for PostgreSQL can host many databases, in the same way as the native PostgreSQL database; PostgreSQL is a multi-tenant platform . By creating additional databases, you can emulate the same functionalities of Oracle 12c PDBs. For example, you can separate business logic into many different databases (Sales, Marketing, HR, etc.), or migrate Oracle schemas (users) to a specific PostgreSQL database.

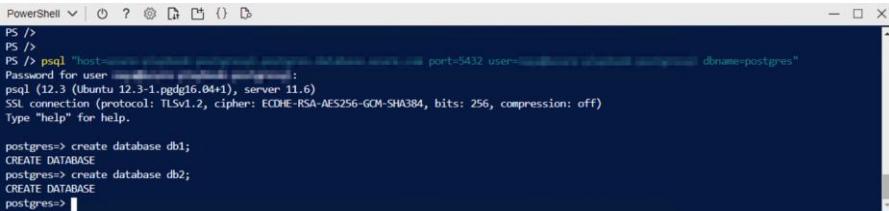
### Azure Database for PostgreSQL Database - Creating Databases

Use the Azure Cloud Shell to create multiple databases:

- Create an Azure Database for PostgreSQL resource (Single Server/Hyperscale (Citus))
- Launch the Azure Database for PostgreSQL using Cloud Shell:



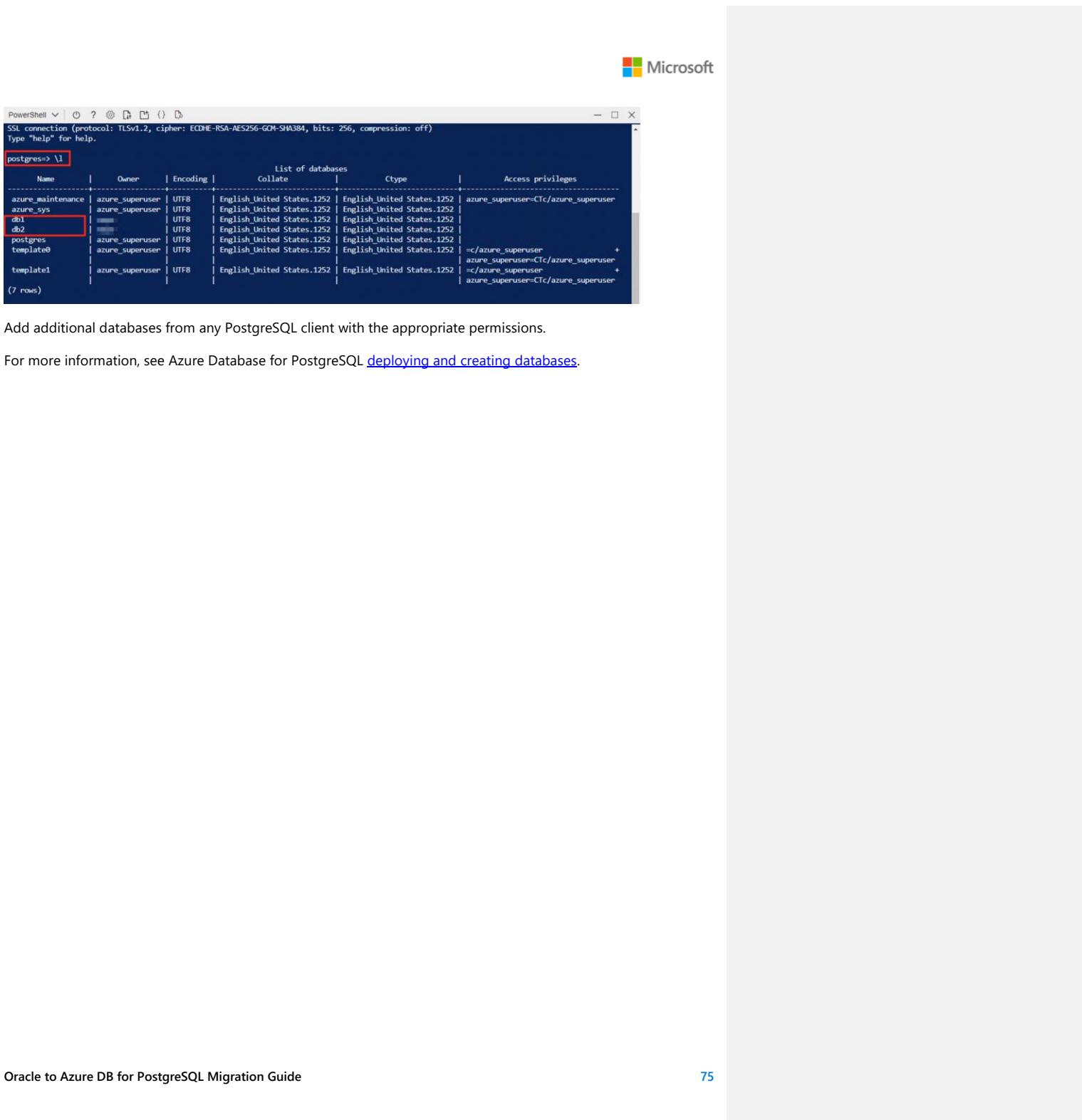
- Connect to the Azure Database for PostgreSQL using the PostgreSQL CLI (psql):



```
PS />
PS />
PS /> psql "host=... port=5432 user=... dbname=postgres"
Password for user ...:
psql (12.3 (Ubuntu 12.3-1.pgdg16.04+1), server 11.6)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=> create database db1;
CREATE DATABASE
postgres=> create database db2;
CREATE DATABASE
postgres=>
```

- You can create additional databases under the same Azure Database for PostgreSQL database server:



## Oracle Tablespaces and Data Files

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Expect different architecture regarding tablespaces data files with Azure Database for PostgreSQL.

Oracle tablespaces and data files are the two main components of Oracle storage architecture that interact with the OS file system.

### Oracle Tablespace Overview

Tablespaces are the Oracle logical layer for creating and defining database data files. Oracle tablespaces by themselves do not store any data, but instead hold the data file configuration, such as initial and maximum storage size, operation system data file path, data file extend size, etc.

### Oracle Tablespace Types

- Permanent tablespace: User-defined tablespace for storing persistent schema objects for users and applications.
- Undo tablespace: A unique system tablespace type used by Oracle to manage UNDO data (store data to perform UNDO operations) when running the database in automatic undo management mode.
- Temporary tablespace: Contains schema objects that are valid for the duration of a session. Also used to perform ORDER BY operations that cannot utilize the current memory resources.

### Oracle Data Files

An Oracle tablespace can hold one or more data files. Data files are the physical elements that store the database data. Data files can be mapped on the local file system as raw partitions managed by Oracle ASM or located on the network file system. The common data files extension is .dbf.

### Oracle Storage Structure

- Database - An Oracle database has one or more tablespaces.
- Tablespace - An Oracle tablespace has one or more datafiles.
- Data files - Each Oracle tablespace is made up of one or more data files.
- Segments - Each Oracle segment represents a single database object, such as a table, view, or index, that consumes storage.
- Extent - Each Oracle segment is made up of one or more extents.
- Block - The smallest unit of I/O that can be used by a database for reads and writes. For blocks that store table data, each block can store one or more table rows. The default size is 8192 bytes.

```
show parameters db_block_size
```

NAME	TYPE	VALUE
<hr/>		
db_block_size	integer	8192



### Oracle Create Tablespace Code Examples:

Create a new tablespace with a single data file named sales\_01.dbf:

```
CREATE TABLESPACE TS_SALES
DATAFILE '/u01/app/oracle/data/sales/sales_01.dbf' SIZE 10M
AUTOEXTEND ON NEXT 10M MAXSIZE 32767M
LOGGING ONLINE PERMANENT BLOCKSIZE 8192
EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT
NOCOMPRESS SEGMENT SPACE MANAGEMENT AUTO;
```

Drop an existing tablespace:

```
DROP TABLESPACE TS_SALES;
```

For more information, see [Oracle storage structure](#).

### Azure Database for PostgreSQL Tablespaces and Data Files

PostgreSQL shares some Oracle storage structure concepts in that PostgreSQL tablespaces are the logical layer where data files are stored at the operating system level, and the data files store the actual database objects and data.

- PostgreSQL Tablespace
  - Stores the operating system path (directory) where the data files are stored.
  - Does not have user-configured segmentation into multiple and separate data files. When a PostgreSQL tablespace is created, PostgreSQL automatically creates the files required to store the data.
- PostgreSQL Data Files
  - Allocated to a specific PostgreSQL tablespace and used to store database objects such as tables, views, and stored procedures.
  - Are automatically created by PostgreSQL. Each table and index are stored in a separate operating system file and named after the table or database object filenode number.

### Azure Database for PostgreSQL Tablespaces Creation

Because Azure Database for PostgreSQL is a fully managed service, permission to interact with the operating system file system is not needed.

The CREATE TABLESPACE command requires PostgreSQL superuser permission, which is not provided in Azure Database for PostgreSQL managed services. This reduces the complexity of managing the PostgreSQL database storage layer, which is handled by the Azure Databases for PostgreSQL managed service.

### Azure Database for PostgreSQL Default Tablespaces

There are two default Azure Database for PostgreSQL tablespaces that are created with the database deployment by default:

- pg\_global tablespace
  - Used for the shared system catalogs.



- Stores objects that are visible to all Cluster databases
- pg\_default tablespace
  - Serves as the main default tablespace for all created database objects.
  - The default tablespace of the template1 and template0 system databases.

## Migrating from Oracle Resource Manager

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- Although Oracle Resource Manager is not support, Azure Database for PostgreSQL support multiple configurations/options for achieving similar resource management functionality.

The Oracle Database Resource Manager is an Oracle proprietary infrastructure feature that provides granular control of database resources that can be allocated to applications, users, and database services. The Oracle Database Resource Manager (RM) enables management of multiple workloads contending for system and database resources by assigning priorities to specific workloads and sessions.

Oracle provides a set of PL/SQL APIs (using the DBMS\_RESOURCE\_MANAGER package) to control the resources that can be allocated to users or sessions according to a defined and managed resource plan. The plan specifies how the resources are distributed among resource consumer groups, which are user sessions grouped by resource requirements. A resource plan directive associates a resource consumer group with a plan and specifies how resources are to be allocated to the group.

### Oracle Resource Manager Main Advantages

- Provides ongoing database system resource monitoring.
- Guarantees a specified amount of minimum CPU per session, regardless of the load on the system and the number of users.
- Distributes available CPU resources by assigning defined CPU percentages to different users and applications.
- Supports parallel degree operations limitation performed by members of a group of users or limits the number of parallel execution servers that a group of users can use. This ensures that all available parallel execution servers are not allocated to one group of users.
- Provides an active session pool, specifying the maximum number of user sessions allowed to be concurrently active in a group of users.
- Prevents execution of operations that the optimizer estimates will run for longer than the specified limit.
- Limits the amount of time that a session can be idle. This can be further defined to mean only sessions that are blocking other sessions.

### Oracle Resource Manager Main Components:

RM Component	Description
Resource plan	A container for directives that specify how resources are allocated to resource consumer groups. You specify how the database allocates resources by activating a specific resource plan.
Resource consumer group	A group of sessions that are grouped together based on resource requirements. The Resource Manager allocates resources to resource consumer groups, not to individual sessions.



Resource plan directive	Associates a resource consumer group with a particular plan and specifies how resources are to be allocated to that resource consumer group.
-------------------------	--

### Oracle Create Resource Plan Example

The following example utilizes a PL/SQL block to create a directive for the OLTP group (OLTP\_GRP) that switches any session in that group to the LOW\_GROUP consumer group if a call in the session exceeds 5 seconds of CPU time.

```
BEGIN  
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (  
        PLAN          => 'DAYTIME',  
        GROUP_OR_SUBPLAN => 'OLTP_GRP',  
        COMMENT       => 'OLTP GRP',  
        MGMT_P1      => 85,  
        SWITCH_GROUP  => 'LOW_GRP',  
        SWITCH_TIME   => 5);  
  
END;  
/  
/
```

For more information, see [Oracle Resource Manager](#).

### Migrating to Azure Database for PostgreSQL

Although PostgreSQL does not provide a feature equivalent to Oracle Resource Manager, much of its functionality can be emulated in the Azure Database for PostgreSQL architecture and PostgreSQL parameters. In the past, using the Oracle database architecture and license method made sense for deploying multiple databases on the same server. Utilizing Oracle Resource Manager is an important part from this architecture; for example, deploying and managing CRM, HR, Sales, and Marketing databases on the same Oracle server while applying resource management.

When using Azure Database for PostgreSQL, you can create a dedicated database for each application while allocating specific resources for each of the systems (deploying multiple Azure Database for PostgreSQL databases). You can also adjust Azure Database for PostgreSQL parameters to achieve more granular resource management; for example, you can control the degree of parallelism for specific queries.

### Creating an Azure Database for PostgreSQL via Azure Portal

From Azure Portal, add a new Azure Database for PostgreSQL database and select "Configure server" to view more options for database server resources.



**Single server**

Microsoft

Server details

Enter required settings for this server, including picking a location and configuring the compute and storage resources.

Server name \*

Data source \*  None  Backup

Location \*

Version \*

Compute + storage

**Configure**

**Basic** Up to 2 vCores with Variable IO performance (1-2 vCore)

**General Purpose** Up to 84 vCores with predictable IO performance (2-84 vCore)

**Memory Optimized** Up to 32 memory optimized vCores with predictable IO performance (2-32 vCore)

Please note that changing to and from the Basic compute tier or changing the backup redundancy options after server creation is not supported.

Compute Generation - Learn more about compute generation  Gen 5

vCore - What is a vCore?  4 vCores

Storage cannot be scaled down

**PRICE SUMMARY**

Gen 5 Compute generation

Cost per vCore	54.98
vCores selected	x 4

+ General Purpose Storage

Cost per GB / month	0.10
Storage selected (in GB)	x 100

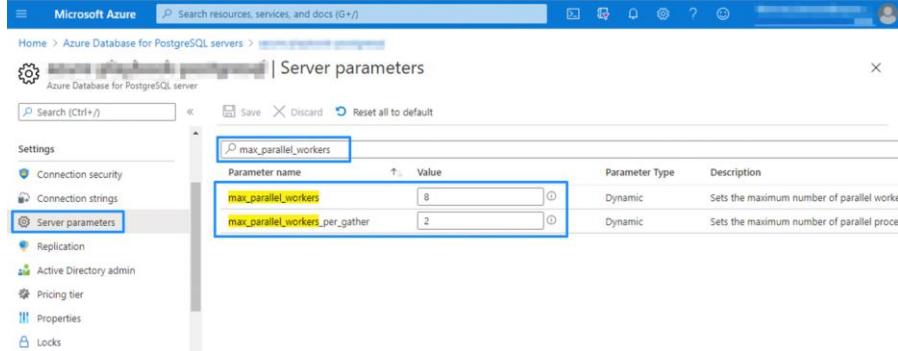
### Azure Database for PostgreSQL Parameters for Resource Management

You can change PostgreSQL parameters on the Azure Portal [Server Parameters](#) page.

Resource Management Feature	Description
Limit parallel execution	Modify the <code>max_parallel_workers</code> parameter.
Limit queries parallelism degree	Modify the <code>max_parallel_workers_per_gather</code> parameter.
Identify the number of active sessions	<pre>select pid from pg_stat_activity where username in(select username from pg_stat_activity where state = 'active' group by username having count(*) &gt; 10)</pre>

	<pre>and state = 'active' order by query_Start;</pre>
Stop queries that pass a defined maximum runtime (for example, 5 min)	<pre>SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE now()-pg_stat_activity. query_start &gt; interval '5 minutes';</pre>
Stop queries that pass a defined idle time	<pre>SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname = 'regress' AND pid &lt;&gt; pg_backend_pid() AND state = 'idle' AND state_change &lt; current_timestamp - INTERVAL '5' MINUTE;</pre>

### Azure Database for PostgreSQL Server Parameter Example



The screenshot shows the Azure portal interface for managing a PostgreSQL server. The left sidebar lists various settings like Connection security, Connection strings, Server parameters (which is selected), Replication, Active Directory admin, Pricing tier, Properties, and Locks. The main content area is titled 'Server parameters' and shows a table with two rows:

Parameter name	Value	Parameter Type	Description
max_parallel_workers	8	Dynamic	Sets the maximum number of parallel workers.
max_parallel_workers_per_gather	2	Dynamic	Sets the maximum number of parallel processes per gather.

For more information, see [PostgreSQL resource consumption](#).

## Migrating from Oracle SGA & PGA Memory Buffers

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**

N/A

**Differences Summary:**

- PostgreSQL utilizes different memory buffer architecture which can be managed by Azure Database for PostgreSQL server parameters.

An Oracle database provides the ability to manage and configure the database memory buffers (RAM) to tune the overall database performance according to specific workloads. Oracle caches reside in the System Global Area (SGA) and are shared across all Oracle sessions. Oracle memory architecture and additional areas of memory using session-private operations, such as sorting and private SQL cursors elements, use the Private Global Area (PGA). You can control the Oracle cache size for specific memory areas in the architecture, or globally and automatically. Setting the memory size parameters enables Oracle to automatically manage individual memory components.

Oracle memory buffers include:

- Database Buffer Cache
- Redo Log Buffer
- Java Pool
- Shared Pool
  - Library Cache
  - Data Dictionary Cache
- Large Pool

Most frequently modified Oracle memory parameters include:

- `sga_max_size` - Hard-limit maximum size of the SGA.
- `sga_target` - Required soft limit for the SGA and the individual caches within it.
- `pga_aggregate_target` - Soft limit controlling the total amount of memory available for all sessions combined.
- `pga_aggregate_limit` - Hard limit for the total amount of memory available for all sessions combined (Oracle 12c only).

To view the SGA Oracle memory parameters:

show parameters sga;		
NAME	TYPE	VALUE
<hr/>		
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	576M



```
sga_target          big integer 576M
```

## Oracle Automatic Memory Management

Oracle AMM provides additional parameters for managing and tuning the instance memory:

- `memory_target`
- `memory_max_target`

```
show parameters memory;

NAME          TYPE      VALUE
-----
memory_max_target  big integer 0
memory_target    big integer 0
```

## Modifying Oracle Memory Parameters

You can change Oracle memory parameters using the `ALTER SYSTEM` command. Some parameters will require a database reboot to take effect.

```
alter system set sga_target=300M scope=both;
```

For more information, see [Oracle Memory Management](#).

## Migrating to Azure Database for PostgreSQL Memory Buffers

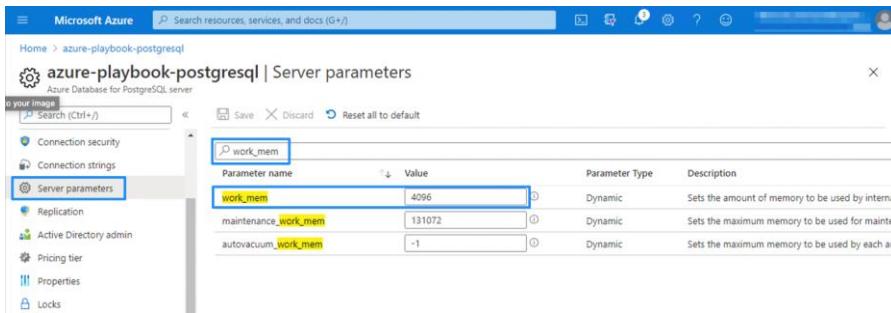
Like Oracle, PostgreSQL memory cache parameters (RAM) can be used to adjust or tune the database performance for specific workload requirements. You can modify Azure Database for PostgreSQL parameters on the Azure Portal "Server Parameters" page or using by Azure CLI commands. Following are the most commonly modified memory parameters.

PostgreSQL Memory Parameter	Overview
<code>work_mem</code>	Specifies the amount of memory used by internal sort operations and hash tables before writing to temporary disk files.
<code>wal_buffers</code>	Used to store WAL (Write-Ahead Log) records before they are written to disk. Can be used as an Oracle Redo Log Buffer equivalent.

<code>maintenance_work_mem</code>	Specifies the maximum amount of memory used by maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY.
<code>temp_buffers</code>	Sets the maximum number of temporary buffers used by each database session.
<code>shared_buffers</code>	Sets the amount of memory the database server uses for shared memory buffers.  This parameter cannot be modified in Azure Database for PostgreSQL. Its value is set automatically according to the PostgreSQL instance type.

### Viewing Azure Database for PostgreSQL Parameters - Azure Portal

The parameters can be viewed and modified; change the value, and then Save.



Parameter name	Value	Parameter Type	Description
<code>work_mem</code>	4096	Dynamic	Sets the amount of memory to be used by internal
<code>maintenance_work_mem</code>	131072	Dynamic	Sets the maximum memory to be used for mainten
<code>autovacuum_work_mem</code>	-1	Dynamic	Sets the maximum memory to be used by each aut

### Viewing Azure Database for PostgreSQL Parameters - CLI

```
show work_mem;

work_mem
-----
6MB
(1 row)
```

### Viewing PostgreSQL Parameters Using SQL

```
select * from pg_settings;
```

The SET SESSION command can modify the value of parameters that support session-specific settings. Changing the value using the SET SESSION command for one session has no effect on other sessions.



```
SET SESSION work_mem='6144MB';
```

The SET LOCAL command can modify the current value of memory parameters that can be set locally to a single transaction. Changing the value using the SET LOCAL command for one transaction has 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='6144MB';
```

Resetting the parameter value on a run-time level:

```
RESET work_mem;
```

Altering Azure Database for PostgreSQL Memory Parameters Using [Azure CLI](#):

```
az postgres server configuration set --name work_mem --resource-group myresourcegroup --server mydemoserver --value 4096
```

Viewing the Azure Database for PostgreSQL memory parameter value using Azure CLI:

```
az postgres server configuration show --name work_mem --resource-group myresourcegroup --server mydemoserver
```

#### Oracle and PostgreSQL Memory Parameters Comparison Table

Memory Parameter	Oracle	PostgreSQL
Physical memory available for the instance	sga_max_size memory_max_size	Configured according to the Azure Database for PostgreSQL instance type
Memory for transaction log records	log_buffer	wal_buffers
Memory for parallel queries	large_pool_size	work_mem
Total amount of private memory for all sessions	pga_aggregate_target pga_aggregate_limit	temp_buffers (reads operations) work_mem (sorts operations)
Database parameters	SELECT * FROM v\$parameter;	SELECT * FROM pg_settings;
Alter a session level parameter	ALTER SESSION SET...	SET SESSION...



Alter instance level parameter	ALTER SYSTEM SET...	Azure Database for PostgreSQL "Server Parameters" page or Azure CLI
--------------------------------	---------------------	---

#### PostgreSQL Versions 12 and 13 New Features

Feature/parameter	Description	PostgreSQL 12	PostgreSQL 13
<b>shared_memory_type</b> (enum)	Specifies the shared memory implementation that the server should use for the main shared memory region that holds PostgreSQL shared buffers and other shared data.	Supported	Supported
<b>logical_decoding_work_mem</b>	Specifies the maximum amount of memory to be used by logical decoding before some of the decoded changes are written to local disk.	N/A	Supported
<b>maintenance_io_concurrency</b> (integer)	Similar to <b>effective_io_concurrency</b> but used for maintenance work done on behalf of many client sessions.	N/A	Supported

For more information, see [PostgreSQL Memory Parameters](#).

## Migrating from Oracle LogMiner Equivalents

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- Although PostgreSQL does not support Oracle LogMiner, alternative solutions are available for identifying database operation performed historically.

Oracle LogMiner enables you to view historical operations written into Oracle REDO logs and Archive logs using the Oracle LogMiner API SQL interface; see [Using LogMiner to Analyze Redo Log Files](#) for more information. You can analyze historical data, without needing point-in-time recovery, to gain better insights on performed DML operations.

### Oracle LogMiner Example

Run the DBMS\_LOGMNR.ADD\_LOGFILE procedure to one or more REDO logs or archive logs to be analyzed.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
    LOGFILENAME => 'LogName', -
    OPTIONS => DBMS_LOGMNR.NEW);
```

Start LogMiner using the DBMS\_LOGMNR.START\_LOGMNR procedure.

```
BEGIN
    DBMS_LOGMNR.START_LOGMNR(options=>
        dbms_logmnr.dict_from_online_catalog);
END;
/
```

Run the DBMS\_LOGMNR.ADD\_LOGFILE procedure then add one or more REDO logs or Archive logs to be analyzed.

```
SELECT username AS USR,
       (XIDUSN || '.' || XIDS LT || '.' || XIDSQN) AS XID,
       operation,
       SQL_REDO,
       SQL_UNDO
```



```
FROM V$LOGMNR_CONTENTS  
WHERE username IN ('');
```

For more information, see [Oracle LogMiner](#).

### Migrating to Azure Database for PostgreSQL Logging Analysis

Although PostgreSQL does not have an equivalent to Oracle LogMiner, there are several options you can use to get a historical view of the performed DML operations at the database level. Azure Database for PostgreSQL also offers logging that can provide useful insight on current and historical database operations.

### PostgreSQL PG\_STAT\_STATEMENTS View

The PostgreSQL [pg\\_stat\\_statements](#) module provides a method for tracking statistics of all SQL statements executed on a PostgreSQL database server. This feature is enabled by default.

The screenshot shows the Azure portal interface with the URL 'https://portal.azure.com/#blade/Microsoft\_Azure\_DatabaseForPostgreSQL/Overview/ResourceName'. The left sidebar has 'Server parameters' selected. In the main content area, under 'pg\_stat\_statements', the 'pg\_stat\_statements.max' parameter is set to 5000, 'pg\_stat\_statements.save' is set to ON, 'pg\_stat\_statements.track' is set to TOP, and 'pg\_stat\_statements.track\_utility' is set to ON. A tooltip for 'pg\_stat\_statements.track' indicates it controls which statements are counted by pg\_stat.

### Querying the PG\_STAT\_STATEMENTS View Example

```
create table employees (empid numeric primary key, empName varchar(100),  
empSal int);  
  
CREATE TABLE  
  
insert into employees values(1, 'John Smith', 10000);  
INSERT 0 1  
  
SELECT userid, dbid, query, calls, total_time  
  FROM PG_STAT_STATEMENTS  
 WHERE LOWER(QUERY) LIKE '%insert%';  
  
userid | dbid | query | calls | total_time
```



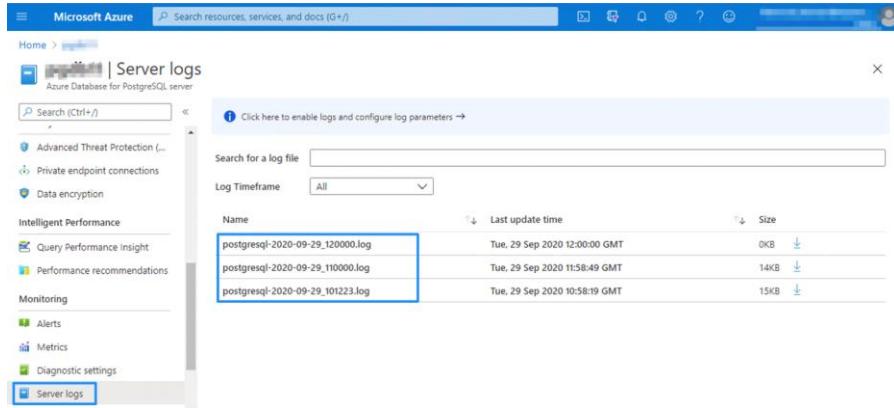
```

-----+
14419 | 14417 | insert into tbl values ($1)      | 1 | 0.520997010376442
14419 | 14417 | insert into employees values ($1, $2, $3)| 1 | 1.87676221373878
14419 | 14417 | insert into employees values ($1, $2, $3)| 1 | 1.82104583551841
(3 rows)

```

## Azure Database for PostgreSQL DML Operations Monitoring

You can view DML operations on the Azure Database for PostgreSQL "Server logs" page. You can also download the PostgreSQL server log for analysis.



Name	Last update time	Size
postgresql-2020-09-29_120000.log	Tue, 29 Sep 2020 12:00:00 GMT	0KB
postgresql-2020-09-29_110000.log	Tue, 29 Sep 2020 11:58:49 GMT	14KB
monitoring-2020-09-29_101223.log	Tue, 29 Sep 2020 10:58:19 GMT	15KB

## Azure Database for PostgreSQL Query Performance Insight

Azure Database for PostgreSQL provides a graphical view of current and historical SQL statements, including DML and DDL operations.

To use Azure Query Performance Insights:

- The [Query Store](#) must be enabled.
- To view Azure Database for PostgreSQL Query Performance Insights, select the PostgreSQL database resource from Azure Portal, and from the left side, select "Azure Query Performance Insights" from the "Intelligent Performance" menu.

Microsoft

For more information, see [Azure PostgreSQL Query Performance Insight](#).

### Migrating from Oracle Session Parameters Equivalents

You can modify certain session parameters in Oracle, such as date and time formats and languages settings. To view all Oracle parameters that can be modified at a session level, issue the following SQL statement:

```
SELECT NAME, VALUE FROM V$PARAMETER WHERE ISSES_MODIFIABLE='TRUE';
```

### Oracle Code Examples for Altering Session-Level Parameters

Alter the session date and time formats:

```
select sysdate from dual;

SYSDATE
-----
29-SEP-20

alter session set nls_date_format='DD-MON-YYYY hh24:mi:ss';

Session altered.

select sysdate from dual;
```



```
SYSDATE
```

```
-----  
01-SEP-2020 13:24:53
```

Alter the session language parameter (NLS\_LANGUAGE):

```
ALTER SESSION SET NLS_LANGUAGE= 'AMERICAN' NLS_TERRITORY= 'AMERICA'  
NLS_CURRENCY= '$'  
ALTER SESSION SET nls_language='SPANISH';
```

For more information, see [Oracle Session Parameters](#).

## Migrating from Oracle Instance & Database Parameters (SPFILE)

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**

N/A

### Differences Summary:

- Expect different architecture and parameters when using PostgreSQL.
- Azure Database for PostgreSQL offers a wide selection of resource types for specific workloads and in addition database parameters can be configured using the server parameter.

You can use database parameters to apply a granular level of configuration to your Oracle database. Some parameters can be modified at the run-time level (session level or instance as level effecting all sessions). You can configure Oracle instance and database-level parameters using the ALTER SYSTEM command. Some parameters can be configured dynamically and take immediate effect, and other static parameters require a database reboot.

### Oracle Configuration Files

The latest Oracle database versions use a binary file called the SPFILE. Earlier versions a text version of the database parameter called the PFILE.

You can export the binary SPFILE to a text file using the following command:

```
CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';
```

You can use the SCOPE parameter to determine when and how changes take effect.

- SCOPE = SPFILE – Changes are stored in the SPFILE only and take effect on the next database reboot.
- SCOPE = MEMORY - Changes happen immediately but do not persist after a database reboot.
- SCOPE = BOTH - Changes happen immediately and are persistent.

### Oracle Altering a Database Parameter Example

```
ALTER SYSTEM SET sga_target=300M SCOPE=BOTH;
```

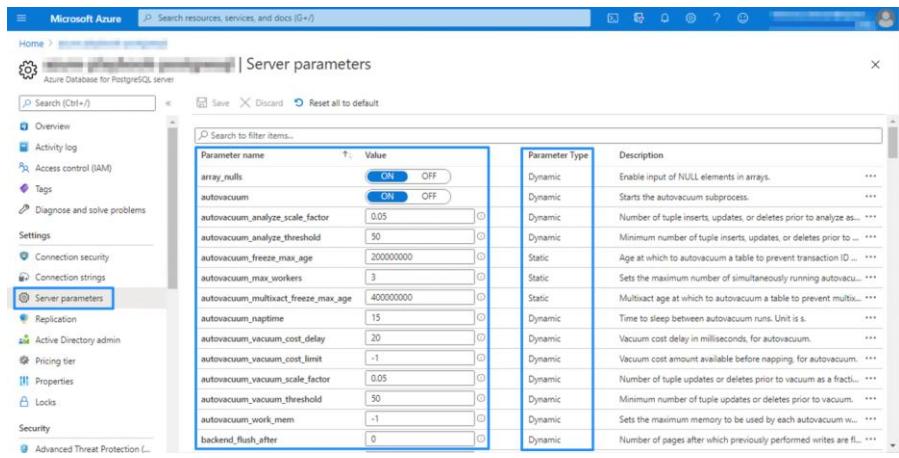
For more information, see [Oracle Parameters](#).

## Migrating to Azure Database for PostgreSQL Server Parameters

Azure Database for PostgreSQL utilizes the Server Parameters to alter the PostgreSQL database server parameters. Changing PostgreSQL parameters can be viewed and modified by both Azure Portal and Azure CLI for instance level parameters while session/transaction parameters can be set at run-time from the session itself.

### Azure Portal - PostgreSQL Server Parameters

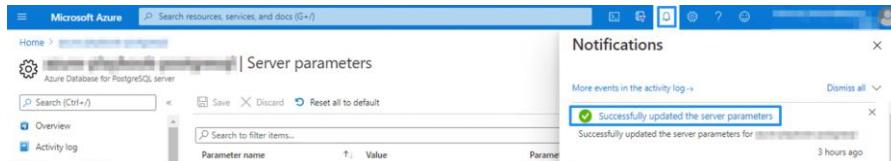
Select the Azure Database for PostgreSQL resource, and from the left side select "[Server Parameters](#)".



Parameter name	Value	Parameter Type	Description
array_nulls	ON / OFF	Dynamic	Enable input of NULL elements in arrays.
autovacuum	ON / OFF	Dynamic	Starts the autovacuum subprocess.
autovacuum_analyze_scale_factor	0.05	Dynamic	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of the total number of tuples in the table.
autovacuum_analyze_threshold	50	Dynamic	Minimum number of tuple inserts, updates, or deletes prior to analyze as a count.
autovacuum_freeze_max_age	20000000	Static	Age at which to autovacuum a table to prevent transaction ID wraparound.
autovacuum_max_workers	3	Static	Sets the maximum number of simultaneously running autovacuum workers.
autovacuum_multixact_freeze_max_age	40000000	Dynamic	Multixact age at which to autovacuum a table to prevent multixact wraparound.
autovacuum_naptime	15	Dynamic	Time to sleep between autovacuum runs. Unit is s.
autovacuum_vacuum_cost_delay	20	Dynamic	Vacuum cost delay in milliseconds, for autovacuum.
autovacuum_vacuum_cost_limit	-1	Dynamic	Vacuum cost amount available before napping, for autovacuum.
autovacuum_vacuum_scale_factor	0.05	Dynamic	Number of tuple updates or deletes prior to vacuum as a fraction.
autovacuum_vacuum_threshold	50	Dynamic	Minimum number of tuple updates or deletes prior to vacuum.
autovacuum_work_mem	-1	Dynamic	Sets the maximum memory to be used by each autovacuum worker.
backend_flush_after	0	Dynamic	Number of pages after which previously performed writes are flushed.

The PostgreSQL parameter type indicates if the parameter can be altered dynamically or requires a reboot to take effect.

Modify the parameter and then save it for it to take effect. You can monitor success or failure in the notification area (bell icon on the upper right of the Azure Portal).



## Azure CLI - PostgreSQL Server Parameters

You can also alter Azure Database for PostgreSQL parameters using the Azure CLI.

Alter Azure Database for PostgreSQL memory parameters:

```
az postgres server configuration set --name autovacuum --resource-group myresourcegroup --server mydemoserver --value ON
```

View Azure Database for PostgreSQL memory parameter values using the Azure CLI:

```
az postgres server configuration show --name autovacuum --resource-group myresourcegroup --server mydemoserver
{
  "allowedValues": "on,off",
  "dataType": "Boolean",
```



```
"defaultValue": "on",
"description": "Starts the autovacuum subprocess.",
"id": "/subscriptions/<PATH>",
"name": "autovacuum",
"resourceGroup": "GRP",
"source": "system-default",
"type": "Microsoft.DBforPostgreSQL/servers/configurations",
"value": "on"
}
```

View all PostgreSQL parameters using a SQL statement:

```
select * from pg_settings;
```

#### Altering PostgreSQL Parameters at Session/Transaction Levels

- Session Level - Use 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 has no effect on other sessions.

```
SET SESSION work_mem='6144MB';
```

- Transaction Level – Use the `SET LOCAL` command to modify the current value of memory parameters that can be set locally to a single transaction. Changing the value using the `SET LOCAL` command for one transaction has no effect on other transactions from the same session. After issuing a `COMMIT` or `ROLLBACK`, the session-level settings take effect.

```
SET LOCAL work_mem='6144MB';
```

- Reset the parameter value on a run-time level:

```
RESET work_mem;
```

For more information, see [PostgreSQL parameters](#).

## Migrating to Azure Database for PostgreSQL Session Parameters

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- PostgreSQL utilizes different syntax for controlling session parameters using the "SET" command.

Like Oracle, PostgreSQL provides session-level parameters that you can configure using the PostgreSQL SET SESSION command. PostgreSQL session parameter changes (using SET SESSION) have effect only on the current session.

To view the list of PostgreSQL session parameters that can be set with SET SESSION , run the following:

```
SELECT * FROM pg_settings where context = '<'UserName'>;
```

### PostgreSQL Commonly Used Session Parameters

Parameter Name	Description
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.
client_encoding	Sets the connected client character set.
force_parallel_mode	Forces use of parallel query for the session.
lock_timeout	Sets the maximum time to wait for a database lock to release.

### PostgreSQL Session Parameters Examples

Set the search path:

```
SET search_path TO <RoleName>, public;
```

Set the session date and time format (dmy - day, month, year):

```
select now();  
  
now  
-----  
2020-09-29 13:46:05.903486+00
```

(1 row)

SET datestyle TO postgres, dmy;

SET

select now();

now

-----  
Tue 29 Sep 13:46:23.309659 2020 UTC

(1 row)

Set session time zone by code:

SET TIME ZONE 'PST8PDT';

Set session time zone by name:

SET TIME ZONE 'Europe/Rome';

**Oracle and PostgreSQL Session Level Parameters Comparison Table (Partial List)**

Parameter Properties	Oracle	PostgreSQL
Set the current default user/database	ALTER SESSION SET current schema='SchemaName'	SET SESSION SEARCH_PATH TO <RoleName>;
Configure time and date format	ALTER SESSION SET nls_date_format = 'dd/mm/yyyy hh24:mi:ss';	SET SESSION datestyle to postgres, DMY;

For more information, see [PostgreSQL Session Parameters](#).

## Migrating from Oracle Alert.log (Error Log)

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

### Differences Summary:

- Azure Database for PostgreSQL utilizes different but rich set of logging options.

Oracle database logging architecture includes a key log file named Alert.log, which is a chronological and textual readable file that holds verbose information about database activity operation such as general notes (for example, instance startup/shutdown), networking issues, and minor and critical error messages.

Oracle Alert.log is often the first log to investigate for insights during and after errors and when troubleshooting database-level failures. The actual alert log name includes the key word "alert" and the Oracle instance name; for example, "alert\_XE.log" where XE is the instance name. Oracle known error messages have the "ORA-" prefix plus the error code with additional description about the error.

Oracle alert log also indicates system-generated trace files (OS full path and file name) that have more verbose messages and descriptions about specific database events, mainly errors and failures.

### Oracle finding the Alert.log operation system path example:

Use Oracle SHOW PARAMETER <ParamName> or query the V\$PARAMETER view:

```
show parameter background_dump_dest;
NAME          TYPE    VALUE
-----
background_dump_dest string  /u01/app/oracle/diag/rdbms/xe/XE/trace
```

### Oracle Alert log start-up notification example:

```
Starting up:
Oracle Database 11g Express Edition Release 11.2.0.4.0 - 64bit Production.
System parameters with non-default
values: processes      = 500
sessions           = 772
sga_target         = 600M
control_files      = "/u01/app/oracle/oradata/XE/control.dbf"
```



```
db_recovery_file_dest      = "/u01/app/oracle/fast_recovery_area"  
db_recovery_file_dest_size = 10G  
transactions              = 610  
undo_management           = "AUTO"  
undo_tablespace            = "UNDOTBS1"  
remote_login_passwordfile. = "EXCLUSIVE"  
...  
...
```

#### Oracle Alert Log error message example:

```
Tue May 05 14:56:05 2020  
Errors in file /u01/app/oracle/diag/rdbms/xe/XE/trace/XE_j000_221.trc:  
ORA-12012: error on auto execute of job "SYS"."BSLN_MAINTAIN_STATS_JOB"  
ORA-06550: line 1, column 807:  
PLS-00201: identifier 'DBSNMP.BSLN_INTERNAL' must be declared  
ORA-06550: line 1, column 807:PL/SQL: Statement ignored  
Tue May 05 14:56:06 2020  
...
```

For more information, see [Oracle Alert log](#).

#### Migrating to the Azure Database for PostgreSQL Logging

Azure Database for PostgreSQL provides full access to the PostgreSQL database server logs. Like Oracle, PostgreSQL standard logs provide information about database events such as errors, query information, connections, and general and performance issues, as well as troubleshooting and repair action such as changing the database server configuration.

You can configure the Azure Database for PostgreSQL logging level to include additional events Using Azure Portal, navigate to Azure Database for PostgreSQL resource and go to the Server Parameters page (left side). Filter logging parameters by specifying "log\_":

Microsoft

Some logging parameters, like `log_checkpoints` and `log_connections`, are enabled by default. Others must be enabled explicitly.

### Azure Database for PostgreSQL Logs

Azure Database for PostgreSQL provides a short-term storage location for log files:

- Configure the retention period for short-term log storage using the `log_retention_period` parameter.
- The default value is 3 days, and the maximum is 7 days.
- Short-term storage can hold up to 1 GB of log files. After 1 GB, the oldest files, regardless of retention period, are deleted to make room for new logs.
- Azure Database for PostgreSQL creates a new file every 1 hour or 100 MB, whichever comes first. Logs are appended to the current file as they are emitted from the PostgreSQL database server.
- You can download the logs for extended retention.
- To access the PostgreSQL log, navigate to the Azure Database for PostgreSQL resource, and on the left side, choose "Server Logs".



Example log line from the PostgreSQL log file:

```
2020-09-30 17:00:03 UTC-5d773cc3.3c-LOG: connection received: host=101.0.0.6  
port=34331 pid=16216
```

Like Oracle, PostgreSQL provides a prefix and description for error events. For more information, see [PostgreSQL error codes](#).

### Azure Monitor Diagnostic Settings

Azure Database for PostgreSQL is integrated with [Azure Monitor diagnostic settings](#).

Diagnostic settings allow you to send the Postgres logs in JSON format to Azure Monitor Logs for analytics and alerting, Event Hubs for streaming, and Azure Storage for archiving.

For more information, see [Azure Database for PostgreSQL logging](#).

## Database Programming

### Migrating from Oracle Transactional Model and Locking

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

#### Differences Summary:

- PostgreSQL transaction model, isolation level and locking behavior is similar to Oracle but sub-transaction (or nested transaction – a transaction within a transaction) is not supported, save point can be used as an alternative solution.

Transaction model and locking refers to the fundamentals of an RDBMS transaction management, under the ACID compatibility which allows SQL DDL/DML operations to run as an atomic processing unit while external/other transactions can run concurrently and perform data modification as well.

The ACID model ensures the following elements within a transaction:

- Atomicity - Ensures that all processing units (SQL statements) inside a single transaction act as a complete unit. If an error occurs in one of the processing units, the transaction fails (is rolled back) under the principle of "all or nothing".
- Consistency - Ensures that all processed data in a transaction is consistent. All data is validated according to all defined logical rules, constraints, cascades, and triggers that have been applied on the database.
- Isolation - Ensures that all transactions occur in isolation with no effect from other transactions running and modifying data at the same run time. So, a transaction cannot read data from any other transaction that has not yet committed its modified data.
- Durability – Ensures that once a transaction is committed, it remains in the system even in the event of system failure that require a restore. All data modification made within the transaction must be stored permanently.

#### Transaction Isolation Levels

An RDBMS isolation level defines modified data disclosure between transactions occurring at the same time. The ANSI/ISO SQL standard (SQL92) offers four levels of isolation, and each isolation level has different approaches to handle concurrent execution of database transactions. The main objective of an RDBMS transaction isolation level is to manage the visibility and accessibility of the modified data to other running transactions. When accessing the same data with several concurrent transactions, the selected level of transaction isolation influences the way transactions interact. Defining the isolation level can control the data versioning or data consistency in the following terms:

- Dirty reads - Data in a transaction can access modified data from another transaction that was not committed yet.
- Phantom reads - The same query in a transaction can return different results when the same query runs multiple times after data modification operations (such as DML operations).
- Nonrepeatable reads - Can happen during a transaction; a row is retrieved twice and the values in the row differ between reads.

#### ANSI/ISO SQL Standard (SQL92) Isolation Levels

- Read Uncommitted - The lowest restriction isolation level, allowing dirty reads to occur.

- Read Committed – The transaction only sees data modifications that were committed; dirty reads are not allowed.
- Repeatable Read - A running transaction can see data modification only after the different transactions commit their data and only when the running transaction has committed its data modifications. This is a stricter approach than the previously described isolation levels.
- Serializable Isolation - The highest isolation level. Prevents any read or write operations while a transaction is performing data modification; it applies a table level lock.

#### **Isolation Level Vs. Reads Access (Possible/Not Possible)**

Isolation Level \ Reads Access	Dirty Reads	Phantoms	Nonrepeatable Reads
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

#### **Oracle Supported Isolation Levels**

Oracle database supports the Read Committed (by default) and Serializable isolation levels. In addition, Oracle has a proprietary isolation level called Read-Only, which is not part of the ANSI/ISO SQL standard. This isolation level **Transaction Model** allows no DML operations during the transaction and only sees data committed at the time the transaction began.

#### **Oracle MVCC (Multi-Version Concurrency Control)**

Oracle MVCC provides automatic read consistency across the entire database for all database sessions. Using MVCC, database sessions "see" data based on a single point in time, ensuring that only committed changes are accessible. Oracle uses the System Change Number (SCN) of the current transaction to obtain a consistent view of the database. Every database query returns only data that was committed with respect to the SCN taken at the time of query execution.

#### **Oracle Isolation Level Code Examples**

Session-level isolation level setting:

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

Transaction-level isolation level setting:

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



For more information, see [Oracle Isolation Levels](#).

### Migrating to Azure Database for PostgreSQL Transaction Model and Locking

PostgreSQL supports the ANSI/ISO SQL (SQL92) isolation levels, with some differences.

#### PostgreSQL Supported Isolation Levels

- Read-Committed - The PostgreSQL default isolation level. A SELECT statement (without a FOR UPDATE/SHARE clause) in a transaction sees only committed data that was issued before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions.
- Repeatable Read - Ensures that only data that was committed before the transaction began can be seen; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. However, the query does see the effects of previous updates executed in its own transaction, even though they are not yet committed.
- Serializable - The strictest transaction isolation, this level emulates serial transaction execution for all committed transactions, as if transactions had been executed one after another, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must retry transactions due to serialization failures.

#### PostgreSQL Isolation Level Vs. Reads Access (Possible/Not Possible)

Isolation Level \ Reads Access	Dirty reads	Phantom Reads	Non-Repeatable Reads	Serialization
Read Uncommitted	Allowed but not in PG	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Allowed but not in PG	Possible
Serializable	Not Possible	Not Possible	Not Possible	Not Possible

#### PostgreSQL MVCC

Like Oracle, PostgreSQL utilizes Multi-Version Concurrency Control (MVCC), which allows transactions to work with a consistent snapshot of data, ignoring changes made by other transactions that have not yet been committed or rolled back. Each transaction can see only a snapshot of accessed data accurate to its execution start time, regardless of what other transactions are doing concurrently.

#### Setting Isolation Levels in Azure Database for PostgreSQL

- Azure Portal – To change the Azure Database for PostgreSQL isolation level at the instance level on the Azure Portal with the database [server parameters](#):
  - Go to Azure Database for PostgreSQL resource > Server parameters.
  - In the search box, type “default\_transaction\_isolation”.

Microsoft

Home > [REDACTED] | Server parameters

Microsoft Azure | Search resources, services, and docs (G+)

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Settings Connection security Connection strings Server parameters Replication Active Directory admin

default\_transaction\_isolation

Parameter name	Value	Parameter Type	Description
default_transaction_isolation	READ COMMITTED	Dynamic	Sets the transaction isolation level of each new session.
	SERIALIZABLE		
	REPEATABLE READ		
	READ COMMITTED		
	READ UNCOMMITTED		

Set the PostgreSQL isolation level at a session level:

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

Set the PostgreSQL isolation level at the transaction level:

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

Verify the PostgreSQL current isolation level by SQL statement (session):

- SELECT CURRENT\_SETTING('TRANSACTION\_ISOLATION');
- current\_setting
- -----
- repeatable read
- (1 row)

Verify the PostgreSQL current isolation level by SQL statement (instance):

- SHOW DEFAULT\_TRANSACTION\_ISOLATION;
- default\_transaction\_isolation
- -----
- repeatable read
- (1 row)

For more information, see [PostgreSQL Transaction Models](#).

### PostgreSQL Transaction Syntax

```
START TRANSACTION [ transaction_mode [, ...] ]
```

where transaction\_mode is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
    COMMITTED | READ UNCOMMITTED }

READ WRITE | READ ONLY

[ NOT ] DEFERRABLE
```

### Oracle and PostgreSQL - Transactions Configuration Comparison

Feature/parameter	Oracle	PostgreSQL	Notes
Auto Commit	Default: Off	Default: On	PostgreSQL can be set to Off as Oracle
MVCC	Supported	Supported	
Isolation Level	Default: Read Committed	Default: Read Committed	
Supported Isolation Levels	<ul style="list-style-type: none"> <li>• Read-only</li> <li>• Serializable</li> </ul>	<ul style="list-style-type: none"> <li>• Repeatable Reads</li> <li>• Serializable</li> </ul>	
Configure Session Isolation Levels	Yes	Yes	
Configure Transaction Isolation Levels	Yes	Yes	
Nested Transaction Support	Yes	No	PostgreSQL workaround: transaction savepoints (more information <a href="#">here</a> )
Support for Transaction Savepoints	Yes	Yes	

For more information, see [PostgreSQL Transactions](#).

## Oracle Procedures and Functions Conversion to PostgreSQL Functions

[back to summary](#)

**General compatibility level:**



**Ora2pg automation capability:**



### Differences Summary:

- Expect syntax and features differences when performing code conversion.
- PostgreSQL supports the creation of stored procedures since PostgreSQL version 11, until PostgreSQL version 11 only functions were supported (PostgreSQL function can be used to return values or perform database operations).

PL/SQL database programming language is Oracle's internal (or expansion language to SQL), to support complex business logic stored and run from within the database layer. Oracle's PL/SQL enable to users to store stored procedures and functions (in the form of stored procedure, functions, or packages) and execute these code objects as reusable elements. As both PL/SQL and SQL share the same database processes (running from different engines), this would have optimal efficiency supporting both DML operation, data querying while implementing code elements such as LOOPS, Cursors, Logical IF/ELSE, error handling, variables, input and output parameters and more, within the same code.

Oracle uses the following syntax for creating [Stored Procedures](#) and [Functions](#).

- Oracle stored procedure syntax:

```
CREATE [OR REPLACE] <Procedure Name>(arg1 data_type, ...)  
AS  
BEGIN  
....  
END <Procedure Name>;
```

- Oracle function syntax

```
CREATE [OR REPLACE] FUNCTION <Function Name> (arg1 data_type, ...)  
RETURN data_type  
BEGIN  
  RETURN <variable/other>  
END;
```

Stored Procedures are used to perform database actions with PL/SQL (for example, DML/DDL operations), and to perform a calculation and return a result.

### Oracle Package and Package Body

Oracle packages are schema objects that are related to specific database business logic holding variables, constants, cursors, exceptions, procedures, functions, and subprograms under one logical reference. Typically, a package has a definition layer (the package itself) and logical layer known as the package body. A package specification is mandatory while the package body can be required or optional, depending on the package specification.

Oracle Packages are considered code containers for related stored procedures, stored functions, and additional database program objects:

- Package Specification - Declares and describes all related PL/SQL elements (no code involved).

```
CREATE [OR REPLACE] PACKAGE [schema_name.]<package_name>
IS | AS
  declarations;
END [<package_name>];
```

- Package Body Specification
  - The package body is related directly to the package specification and has the same name along with the body keyword.
  - Contains the actual executable PL/SQL and SQL code as stored procedures, functions, error-handling, and more.

```
CREATE [OR REPLACE] PACKAGE BODY [schema_name.]<package_name>
IS | AS
  declarations
  implementations;
[BEGIN
[EXCEPTION]
END [<package_name>];
```

To execute a code object from a specific package, both the package name and the package body object (stored procedure or function) must be specified:

```
EXEC <PackageName>.<PackageBody-ObjectName>;
```

### Oracle Procedures and Functions Privileges

- CREATE PROCEDURE - System privilege required to create schema-native procedures and functions.
- CREATE ANY PROCEDURE - System privilege required to create functions or procedures in other schemas.
- EXECUTE - System privilege required to run stored procedures or functions.



## Oracle Stored Procedure and Functions Examples

Function to get the total amount of invoice by customer ID and year:

```
CREATE OR REPLACE FUNCTION invoice_cust_year
  (p_cust_id INT, p_invoice_year INT)
  RETURN NUMBER
AS
  v_invoice_sum NUMBER:= NULL;
BEGIN
  SELECT SUM(total) INTO v_invoice_sum
  FROM invoice
  WHERE customerid = p_cust_id
  AND EXTRACT(YEAR FROM invoicedate) = p_invoice_year;

  IF v_invoice_sum IS NULL THEN
    RETURN 0;
  ELSE
    RETURN v_invoice_sum;
  END IF;
END;
/
SELECT invoice_cust_year (39, 2013) FROM dual;

INVOICE_CUST_YEAR(39,2013)
-----
22.77
```

Stored procedure: apply customer credits by a condition of invoice amount per year:

```
CREATE OR REPLACE PROCEDURE cust_year_discount_analysis
  (p_invoice_year INT, p_credits NUMBER)
```



```
AS
BEGIN
    FOR V IN
        (SELECT customerid, SUM(total) AS total
         FROM invoice
         WHERE EXTRACT(YEAR FROM invoicedate) = p_invoice_year
         GROUP BY customerid)
    LOOP
        BEGIN
            IF V.total > 10 THEN
                INSERT INTO credits
                VALUES (creditid.NEXTVAL, sysdate, V.customerid,
                        p_credits, 'Open');
            ELSE
                DBMS_OUTPUT.PUT_LINE('Customer: '|| V.customerid||'
                                     didn''t reached discount amount
                                     on:'||p_invoice_year);
            END IF;
        EXCEPTION
            WHEN OTHERS THEN
                RAISE_APPLICATION_ERROR
                (-20002,'An error has occurred inserting an order.');
        END;
    END LOOP;
END;
/
EXEC cust_year_discount_analysis(2013);
```

Package example based on the previous function and stored procedure:



```
CREATE OR REPLACE PACKAGE pkg_cust_invoices
AS
FUNCTION invoice_cust_year (p_cust_id INT, p_invoice_year INT)
RETURN NUMBER;
PROCEDURE cust_year_discount_analysis
(p_invoice_year INT, p_credits NUMBER);
END cust_invoices;
/
```

Package body example based on the previous function and stored procedure:

```
CREATE OR REPLACE PACKAGE BODY pkg_cust_invoices
AS
FUNCTION invoice_cust_year
(p_cust_id INT, p_invoice_year INT)
RETURN NUMBER
AS
v_invoice_sum NUMBER:= NULL;
BEGIN
SELECT SUM(total) INTO v_invoice_sum
FROM invoice
WHERE customerid = p_cust_id
AND EXTRACT(YEAR FROM invoicedate) = p_invoice_year;

IF v_invoice_sum IS NULL THEN
RETURN 0;
ELSE
RETURN v_invoice_sum;
END IF;
END;
```



```
PROCEDURE cust_year_discount_analysis
  (p_invoice_year INT, p_credits NUMBER)
AS
BEGIN
  FOR V IN
    (SELECT customerid, SUM(total) AS total
     FROM invoice
     WHERE EXTRACT(YEAR FROM invoicedate) = p_invoice_year
     GROUP BY customerid)
  LOOP
    BEGIN
      IF V.total > 10 THEN
        INSERT INTO credits
        VALUES (creditid.NEXTVAL, sysdate, V.customerid,
                p_credits, 'Open');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Customer: ' ||
                             V.customerid||' didn''t reached discount
                             amount on: '||p_invoice_year);
      END IF;

      EXCEPTION
      WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR
        (-20002,'An error has occurred inserting an
order.');
    END;
  END LOOP;
END;
```

```
/  
  
-- Calling a function from the package and package body  
SELECT pkg_cust_invoices.invoice_cust_year(30, 2013) FROM dual;  
  
CUST_INVOICES.INVOICE_CUST_YEAR(30,2013)  
-----  
8.91  
  
-- Execution the stored procedure from the package  
EXEC pkg_cust_invoices.cust_year_discount_analysis(2013, 1.99);
```

### PostgreSQL Stored Procedures and Functions

The PostgreSQL extension language is PL/pgSQL, which has many syntax and function similarities to Oracle PL/SQL (for example, stored procedures and functions), but also has many differences (for example, lack of support for Oracle packages and package body functionality). PostgreSQL stored procedures are supported only from PostgreSQL version 11, while functions were the main objects to create, develop, and manage database layer business logic.

A PostgreSQL function can return a value and perform database operations (DML/DDL), but PostgreSQL stored procedures cannot. The primary advantage of PostgreSQL is the ability to open and manage transactions. (The PostgreSQL stored procedure call method also differs from PostgreSQL functions.)

### PostgreSQL Stored Procedure Syntax

```
CREATE [ OR REPLACE ] PROCEDURE  
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [,  
... ] ] )  
    { LANGUAGE Lang_name  
     | TRANSFORM { FOR TYPE type_name } [, ... ]  
     | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
     | SET configuration_parameter { TO value | = value | FROM CURRENT }  
     | AS 'definition'  
     | AS 'obj_file', 'Link_symbol'  
    } ...
```



### PostgreSQL Function Syntax

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
    | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE Lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
} ...
```

### Calling PostgreSQL Stored Procedure and Function Differences

PostgreSQL stored procedures:

- Using the COMMIT clause.
- Executing the stored procedure with the CALL(<SP\_NAME>) clause.

```
-- Create a temporary table
CREATE TEMPORARY TABLE tmp (col1 INT);

-- Create PostgreSQL stored procedure
CREATE OR REPLACE PROCEDURE sp_test(INT, INT)
  LANGUAGE plpgsql
  AS $$
```



```
BEGIN
    INSERT INTO tmp VALUES ($1 + $2);
    COMMIT;
END;
$$;

-- Call the stored procedure
CALL sp_test(1, 2);

-- Verify data
SELECT * FROM tmp;

col1
-----
 3
(1 row)
```

PostgreSQL functions:

- Cannot apply the COMMIT cause
- Call functions using SELECT(<FunctionName>)

```
-- Create a temporary table
CREATE TEMPORARY TABLE tmp (col1 INT);

CREATE OR REPLACE FUNCTION func_test(INT, INT)
RETURNS VOID AS $$$
DECLAREv_results INT;
BEGIN
    INSERT INTO tmp VALUES ($1 + $2);
END;
$$ LANGUAGE plpgsql;
```

```
-- Call the function
SELECT func_test(1, 2);

-- Verify data
SELECT * FROM tmp;

col1
-----
 3
(1 row)
```

Additional notes:

- Use “\$\$” characters to avoid single-quoted string escape elements; you do not need to use escape characters in the code when using single quotation marks.
- The LANGUAGE PLPGSQL parameter specifies the created function language.
- The INPUT (P\_NUM) parameter behaves like Oracle PL/SQL INPUT.
- To create a PostgreSQL stored procedure/function, the USAGE privilege is required.

### Oracle PL/SQL Conversion to PostgreSQL PL/pgSQL

PostgreSQL PL/pgSQL is the ideal language for migrating from Oracle’s PL/SQL code because many of the Oracle PL/SQL syntax elements are also supported by PostgreSQL PL/pgSQL (for example, the CREATE OR REPLACE syntax).

Although PostgreSQL does not support Oracle packages functionality, you can use workarounds.

The following conversion examples were used with Ora2Pg to convert Oracle PL/SQL Package (holding a stored procedure and a function) into PostgreSQL PL/pgSQL.

Create a schema named as the package name to support Oracle package naming convention:

```
CREATE SCHEMA pkg_cust_invoices;
```

Convert Oracle’s package function using PostgreSQL PL/pgSQL:

```
CREATE OR REPLACE FUNCTION pkg_cust_invoices.invoice_cust_year
  (p_cust_id integer, p_invoice_year integer)
RETURNS numeric AS $body$
```



```
DECLARE v_invoice_sum numeric:= NULL;  
BEGIN  
    SELECT SUM("Total") INTO v_invoice_sum  
    FROM "Invoice"  
    WHERE "CustomerId" = p_cust_id  
    AND EXTRACT(YEAR FROM "InvoiceDate") = p_invoice_year;  
  
    IF v_invoice_sum IS NULL THEN  
        RETURN 0;  
    ELSE  
        RETURN v_invoice_sum;  
    END IF;  
END;  
$body$  
LANGUAGE PLPGSQL  
SECURITY DEFINER;
```

Call the function to return the same results as Oracle:

```
SELECT pkg_cust_invoices.invoice_cust_year(30, 2013);  
  
invoice_cust_year  
-----  
8.91  
(1 row)
```

Convert Oracle's package stored procedure using PostgreSQL PL/pgSQL. Note that Ora2Pg converts the Oracle stored procedure into a PostgreSQL function by default; minor modifications are required in to convert it as PostgreSQL procedure.

```
CREATE OR REPLACE PROCEDURE  
    pkg_cust_invoices.cust_year_discount_analysis
```

```
(p_invoice_year integer, p_credits numeric)
LANGUAGE plpgsql
AS $$

DECLARE v record;
BEGIN

FOR v IN
(SELECT "CustomerId", SUM("Total") AS total
FROM "Invoice"
WHERE EXTRACT(YEAR FROM "InvoiceDate") = p_invoice_year
GROUP BY "CustomerId")

LOOP

BEGIN

IF v.total > 10 THEN

    INSERT INTO credits (creditid, credit_date,
                         customerid, credit_amount, credit_status)
    VALUES (nextval('creditid'), LOCALTIMESTAMP,
            v."CustomerId", p_credits, 'Open');

ELSE

    RAISE NOTICE 'Customer: % didn''t reached discount
                  amount on: %', v."CustomerId", p_invoice_year;

END IF;

EXCEPTION
WHEN OTHERS THEN

    RAISE EXCEPTION 'An error has occurred';

END;

END LOOP;

END;
$$;
```



Execute the stored procedure to achieve the same results as Oracle:

```
CALL pkg_cust_invoices.cust_year_discount_analysis(2013, 1.99);
```

For more information, see the following:

- [Oracle CREATE PROCEDURE](#)
- [Oracle CREATE FUNCTION](#)
- [Oracle CREATE PACKAGE](#)
- [PostgreSQL SQL Procedural Language](#)
- [PostgreSQL Procedure](#)
- [PostgreSQL Function](#)

## Oracle User-Defined Functions (UDFs)

[back to summary](#)

**General compatibility level:**



**Ora2pg automation capability:**



### Differences Summary:

- Expect syntax and features differences when performing code conversion.

Oracle user-defined functions (UDFs), or stored functions, enable you to extend the built-in Oracle database functions answer to a specific requirement or address specific business needs. Oracle UDFs use Oracle PL/SQL extension language to include both SQL and PL/SQL engines to return the function result.

Oracle UDFs main usage patterns:

- As a scalar function - Oracle UDFs can return a single value from a SELECT statement.
- Simultaneous operations - Oracle UDFs can be used in parallel with DML operations.
- Clause alignment - UDFs can be used in WHERE, GROUP BY, ORDER BY, HAVING, CONNECT BY, and START WITH statements.

### Oracle User-Defined Function Example

The following function reruns the employee number of active days, months, or years according to the employee hire date and employee ID value. It uses several Oracle functions and %Type, which is later converted to PostgreSQL PI/pgSQL.

```
CREATE OR REPLACE FUNCTION emp_active_period
  (p_emp_id INT, p_period_type CHAR)
  RETURN VARCHAR2
  AS
    v_emp_hire_date employee.hiredate%TYPE;
    v_num_results NUMERIC;
    v_output VARCHAR2(20);

  BEGIN
    SELECT hiredate INTO v_emp_hire_date
      FROM employee WHERE employeeid = p_emp_id;

    IF p_period_type IS NULL OR p_period_type = 'y' THEN
      v_num_results:=EXTRACT(YEAR FROM sysdate) -
                    EXTRACT(YEAR FROM v_emp_hire_date);
```



```
v_output:='Years: '||v_num_results;

ELSIF p_period_type = 'm' THEN
    v_num_results:=MONTHS_BETWEEN(sysdate, v_emp_hire_date);
    v_output:='Months: '||v_num_results;

ELSIF p_period_type = 'd' THEN
    v_num_results:=ROUND(sysdate - v_emp_hire_date);
    v_output:='Days: '||v_num_results;
END IF;

RETURN v_output;
END;
```

Verify the function results:

```
SELECT EMP_ACTIVE_PERIOD(4, 'y') FROM DUAL;
EMP_ACTIVE_PERIOD(4,'Y')
-----
Years: 17

SELECT EMP_ACTIVE_PERIOD(4, 'm') FROM DUAL;
EMP_ACTIVE_PERIOD(4,'M')
-----
Months: 209

SELECT EMP_ACTIVE_PERIOD(4, 'd') FROM DUAL;
EMP_ACTIVE_PERIOD(4,'D')
-----
Days: 6372
```

## PostgreSQL UDFs

PostgreSQL UDFs serve the same purpose as Oracle UDFs, providing additional capabilities for the existing database function and serving a wide variety of business use cases. PostgreSQL functions are created using the PL/pgSQL extension language when converting Oracle PL/SQL UDFs.

### PostgreSQL UDF Conversion Example

The following example was converted using Ora2Pg with minor code modifications:

- The Oracle %TYPE data type definition is supported.
- [orafce](#) (PostgreSQL extension for Oracle functions) is used to support Oracle's MONTH\_BETWEEN function, and Oracle SYSDATE is converted as oracle.sysdate():

```
CREATE OR REPLACE FUNCTION emp_active_period
  (p_emp_id INTEGER, p_period_type CHAR)
RETURNS VARCHAR AS $body$  
DECLARE  
  v_emp_hire_date "Employee"."HireDate"%TYPE;  
  v_num_results NUMERIC;  
  v_output VARCHAR(30);  
  
BEGIN  
  SELECT "HireDate" INTO v_emp_hire_date  
  FROM "Employee" WHERE "EmployeeId" = p_emp_id;  
  
  IF p_period_type IS NULL OR p_period_type = 'y' THEN  
    v_num_results:=EXTRACT(YEAR FROM localtimestamp) -  
      EXTRACT(YEAR FROM v_emp_hire_date);  
    v_output:='Years: '||v_num_results;  
  
  ELSIF p_period_type = 'm' THEN  
    v_num_results:=(oracle.months_between(NOW(),  
      v_emp_hire_date));  
    v_output:='Months: '||round(v_num_results);  
  
  ELSIF p_period_type = 'd' THEN
```



```
v_num_results:=(date_part('day', ORACLE.sysdate() -  
v_emp_hire_date));  
v_output:='Days: '||v_num_results;  
END IF;  
RETURN v_output;  
END;  
$body$  
LANGUAGE plpgsql  
SECURITY DEFINER;
```

Verify that the function results are identical to Oracle's results:

```
SELECT EMP_ACTIVE_PERIOD(4, 'y') FROM DUAL;  
emp_active_period  
-----  
Years: 17  
(1 row)  
  
SELECT EMP_ACTIVE_PERIOD(4, 'm') FROM DUAL;  
emp_active_period  
-----  
Months: 209  
(1 row)  
  
SELECT EMP_ACTIVE_PERIOD(4, 'd') FROM DUAL;  
emp_active_period  
-----  
Days: 6372  
(1 row)
```

For more information, see the following:

- Oracle [User-Defined Functions](#)
- PostgreSQL [User-defined Functions](#)
- PostgreSQL [CREATE FUNCTION](#)

**PostgreSQL 11 updates:**

- Added support for stored procedures
- Remove WITH clause in CREATE FUNCTION

**PostgreSQL 12 updates:**

None

**PostgreSQL 13 updates:**

None

## Oracle Materialized Views Conversion to PostgreSQL Materialized Views

[back to summary](#)

**General compatibility level:**



**Ora2pg automation capability:**



**Differences Summary:**

- PostgreSQL does not support automatic or incremental MVviews REFRESH

Oracle Materialized Views (MViews, known as Snapshots in previous Oracle database versions) are database objects that can be based on the results of SQL queries from one or more tables/views, while the data can be altered or flattened (changing the original data structure by adding JOINs or scalar/group functions) to serve table replication from one site to another. The materialized view is considered as a replica from a source/master table based on data from a specific single point in time (snapshot). The materialized views (or MViews) stored defined queries that can be configured for data refresh by the following methods:

- Manually refresh (on demand)
- Automatic refresh (by event/schedule)

From a performance perspective, additional indexes that address specific requirements can be created on MViews. These may be different from the source tables/views schema. Similar to regular views, materialized views are based on an SQL statement calling the required columns from defined tables and applying any required filtering and scalar functions or aggregation. The main difference is that the materialized views data is physically stored in the database data files with the associated indexes.

Oracle materialized views common use cases include:

- Multi-database replication
- Data warehousing operations
- Enhancing performance and efficiency by storing repeatable, complex queries.

### Oracle Materialized View Logs

To support any DML operation on source tables, Oracle database stores additional records to describe those changes in the materialized view log and then uses the log to refresh the materialized views based on the source table. Without a materialized view log, the refresh process must be executed with a complete refresh, which can cause longer refresh times and impact database performance.

### Oracle Materialized View Refresh Types

- Initial data population method
  - Oracle can update contents immediately or on a deferred timeline.
  - The BUILD IMMEDIATE option instructs Oracle to update the materialized view contents immediately (run the underlying query).
  - The BUILD DEFERRED option populates the materialized view only on the first requested refresh.
- Fast refresh and complete refresh - Oracle can perform incidental and comprehensive updates:
  - REFRESH FAST
    - Updates the data incrementally, refreshing only those rows that have changed since the last refresh of the materialized view.
    - Can be mimicked by [rollup tables](#).



- Materialized view logs are required for fast refresh.
- COMPLETE
  - Fully repopulates the entire table used by the materialized view after clearing (truncating) materialized view data.

### Oracle Materialized View Refresh Method

Oracle enables ad hoc or time-specific refreshes using the ON COMMIT and ON DEMAND commands.

- ON COMMIT - Refreshes the materialized view with each commit to the underlying associated tables. This is the default when not specifying a refresh method.
- ON DEMAND - Refreshes the materialized view through a scheduled task or by the user.

### Oracle 12c (Release 2) Materialized Views Enhancements

Oracle 12.2 introduced the concept of real-time materialized views, which provides fresh data to user queries even when the materialized view is not in sync with its base tables because of data changes. This feature is not currently supported by PostgreSQL.

For more information, see [Oracle Real-time Materialized Views](#).

### Oracle Materialized Views Examples

Basic materialized view creation syntax specifying the REFRESH ON DEMAND method:

```
CREATE MATERIALIZED VIEW MV_INVOICE
  USING INDEX
  REFRESH ON DEMAND
  AS
  SELECT invoiceid,
         customerid,
         Invoicedate,
         SUM(Total)
    FROM invoice
   GROUP BY invoiceid, customerid, Invoicedate;
```

You can use materialized views to retrieve data from remote database tables by specifying an Oracle database link. Use the FOR UPDATE clause to allow updates performed on the materialized view directly:

```
CREATE MATERIALIZED VIEW MV_INVOICE
  FOR UPDATE
  USING INDEX
  REFRESH ON DEMAND
  AS
```



```
SELECT invoiceid,
       customerid,
       Invoicedate,
       SUM(Total)
  FROM invoice@DB_LINK
 WHERE customerid IN(SELECT customerid FROM CUSTOMER
                      WHERE country = 'Australia')
 GROUP BY invoiceid, customerid, Invoicedate;

-- Refresh manually
EXECUTE DBMS_MVIEW.REFRESH('mv_cust_invoice','C');
```

The following example shows a materialized view constructed from two tables with the REFRESH FAST ON COMMIT refresh method, which includes the creation of the materialized view logs for each table to support fast refresh:

```
CREATE MATERIALIZED VIEW LOG ON customer
  WITH ROWID
  INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON invoice
  WITH ROWID
  INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW mv_cust_invoice
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT
  AS
    SELECT C.company,
           EXTRACT(YEAR FROM invoicedate) AS invoice_year,
           SUM(I.total)
      FROM customer C JOIN invoice I
```



```
ON C.customerid = I.customerid  
GROUP BY C.company, EXTRACT(YEAR FROM invoicedate);
```

## PostgreSQL Materialized Views

Similar to Oracle, PostgreSQL supports materialized views associated with defined SQL queries and storing the results in the database data files for faster and consistent analysis. Indexes are created with PostgreSQL materialized views, and statistics are collected by PostgreSQL Statistics Collector. PostgreSQL materialized views data can be refreshed on demand, using the REFRESH MATERIALIZED VIEW command, to repopulate the materialized view data.

### Oracle and PostgreSQL Materialized Views Core Differences

- Refresh method
  - PostgreSQL materialized views can only be manually refreshed or by setting an automated procedure to run the REFRESH MATERIALIZED VIEW command.
  - PostgreSQL materialized views do not by default perform an autorefresh after updates to the underlying tables. You need to create a table trigger to implement data changes on the materialized view.
  - When performing a materialized view refresh, the CONCURRENTLY parameter can be used to eliminate concurrent selects on the materialized view from being locked. Without this option, a refresh that affects a large number of rows tends to use fewer resources and complete more quickly but could also block other connections that are trying to read from the materialized view. This option is only allowed if there is at least one UNIQUE index on the materialized view.
- Refresh scope
  - PostgreSQL materialized views only support complete (full) refresh.
  - The PostgreSQL REFRESH MATERIALIZED VIEW command completely replaces the contents of a materialized view with new data taken from the source tables/views.
- DML operations - PostgreSQL materialized views do not support DML operations.
- Materialized views logs - Materialized views logs do not exist in PostgreSQL.
- Materialized views syntax - Oracle materialized views BUILD and REFRESH methods are not supported by PostgreSQL materialized views syntax. In addition, USING INDEX and TABLESPACE are not supported.

## PostgreSQL Materialized Views Examples

Basic materialized views CREATE syntax with a REFRESH command applied:

```
CREATE MATERIALIZED VIEW MV_INVOCIE  
AS  
SELECT "InvoiceId",  
       "CustomerId",  
       "InvoiceDate",  
       SUM("Total")  
FROM "Invoice"  
GROUP BY "InvoiceId", "CustomerId", "InvoiceDate";
```



```
-- Refresh manually  
REFRESH MATERIALIZED VIEW MV_INVOICE;
```

A materialized views refresh specifying the CONCURRENTLY parameter:

```
CREATE UNIQUE INDEX ON MV_INVOICE("InvoiceId")  
REFRESH MATERIALIZED VIEW CONCURRENTLY MV_INVOICE;
```

See the following for more information:

- [Oracle Basic Materialized Views](#)
- [Oracle Materialized View Concepts and Architecture](#)
- [PostgreSQL Materialized Views](#)
- [PostgreSQL Refresh Materialized View](#)

#### PostgreSQL 11 updates

- Allow parallel when creating materialized views

#### PostgreSQL 12 updates

None

#### PostgreSQL 13 updates

None

## Migrating from Oracle EXECUTE IMMEDIATE Statement

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

### Differences Summary:

- Syntactical differences may require code rewrite.
- Remove IMMEDIATE keyword and convert to EXECUTE and PREPARE PostgreSQL commands.

The Oracle EXECUTE IMMEDIATE command applies dynamic SQL statements issued from an Oracle stored procedures/packages body using variables embedded in PL/SQL and SQL DML/DDL operations.

### Oracle EXECUTE IMMEDIATE Example

The following example shows a basic EXECUTE IMMEDIATE statement for creating a new Oracle user dynamically, using a stored procedure that accepts two input parameters (username and password):

- Create a PL/SQL stored procedure that accepts two input parameters.
- Define a CREATE USER statement using the input parameters as dynamic values received from the user issuing the stored procedure.
- EXECUTE IMMEDIATE is used to run the statement using dynamic values (username and password).

```
CREATE OR REPLACE PROCEDURE USER_DYNAMIC_CREATION
  (P_USER_NAME VARCHAR2, P_USER_PASSWD VARCHAR2)
AS
  V_USER_CHECK INT;
BEGIN
  -- VERIFY IF USER ALREADY EXISTS
  SELECT COUNT(*) INTO V_USER_CHECK
  FROM DBA_USERS
  WHERE USERNAME = P_USER_NAME;

  DBMS_OUTPUT.PUT_LINE(V_USER_CHECK);

  -- CREATE A NEW USER IF USER DOES NOT EXISTS
  IF V_USER_CHECK = 0 THEN
    EXECUTE IMMEDIATE 'CREATE USER '''||P_USER_NAME||' IDENTIFIED BY '
                      ||P_USER_PASSWD;
```



```
ELSE
    DBMS_OUTPUT.PUT_LINE('ERROR - USER EXISTS');
END IF;
END;
/

-- Execute the stored procedure
EXEC USER_DYNAMIC_CREATION('SALES', 'PASSWORD');
```

## Migrate to Azure Database for PostgreSQL EXECUTE Statements

Although Oracle EXECUTE IMMEDIATE syntax is not supported by PostgreSQL, you can implement dynamic SQL statements using the PostgreSQL EXECUTE command.

The PostgreSQL EXECUTE command prepares and executes dynamic commands, retrieves data per SQL commands and runs DML/DDL statements. The PostgreSQL EXECUTE command can be used with bind variables same as Oracle.

### PostgreSQL EXECUTE Example

The following example converts the Oracle stored procedure to create a new user (schema) dynamically by accepting two input parameters (username and password):

- The Oracle stored procedure is converted to a PostgreSQL function.
- PL/SQL EXECUTE IMMEDIATE is converted to the PostgreSQL EXECUTE command.
- The stored procedure outcome remains the core functionality and output.
- The conversion was done by Ora2Pg with manual adjustments.

```
CREATE OR REPLACE FUNCTION user_dynamic_creation
(P_USER_NAME text, P_USER_PASSWD text)
RETURNS VOID AS $body$
DECLARE
    V_USER_CHECK integer;
BEGIN
    -- VERIFY IF USER ALREADY EXISTS
    SELECT COUNT(*) INTO V_USER_CHECK
    FROM pg_roles
```



```
WHERE rolname = P_USER_NAME;
RAISE NOTICE '%', V_USER_CHECK;

-- CREATE A NEW USER IF USER DOES NOT EXISTS
IF V_USER_CHECK = 0 THEN
    EXECUTE 'CREATE ROLE'||P_USER_NAME||' WITH LOGIN ENCRYPTED PASSWORD
'||P_USER_PASSWD||''';
ELSE
    RAISE NOTICE 'ERROR - USER EXISTS';
END IF;
END;
$body$
LANGUAGE PLPGSQL
SECURITY DEFINER;

-- Run the Function
SELECT user_dynamic_creation('new_role', 'password');

-- PostgreSQL EXECUTE DDL command Example:

DO $$DECLARE
BEGIN
    EXECUTE 'CREATE TABLE events (event_id integer)';
END$$;
;
```

### PostgreSQL PREPARE Statement

You can use the PostgreSQL PREPARE statement when repeatable SQL statements are used by the application/users multiple times, passing different values. A prepared statement is a server-side object, linked to a database object ID, designed to enhance system performance. When executed, the specified prepared statement



is parsed and analyzed and then rewritten according to the specified parameters. Prepared statements are used to avoid parsing the same SQL statements repeatedly. This feature is not available in Oracle but is similar to Oracle bind variables.

### PostgreSQL PREPARE and EXECUTE Command Examples

The SQL command is prepared with a user-specified qualifying name and executed several times, without the need for reparsing.

```
-- Defining the PREPARE Statement
PREPARE events_insert (numeric, text, bool)
AS
  INSERT INTO EVENTS ($1, $2, $3);

-- Executing the PREPARE statement using PostgreSQL EXECUTE
EXECUTE events_insert (10, 'event-10', 'f');
EXECUTE events_insert (20, 'event-20', 't');
EXECUTE events_insert (30, 'event-30', 't');
```

See the following for more information:

- Oracle - [EXECUTE IMMEDIATE Statement](#)
- PostgreSQL - [PREPARE Syntax](#)
- PostgreSQL - [EXECUTE Syntax](#)

## Oracle UTL\_FILE to Azure DB PostgreSQL

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- PostgreSQL does not have a UTL\_FILE equivalent as a built-in feature.
- As Azure Database for PostgreSQL is a managed service, the interaction with files on the database server level is not possible
- Note that the PostgreSQL extension [orafce](#), which is [supported](#) by Azure Database for PostgreSQL (see [supported extensions](#)), is an alternative solution for Oracle's UTL\_FILE functionality where possible (using the PostgreSQL orafce extension will require manual PL/SQL to PL/pgSQL code conversion and functionality verification).

Oracle's UTL\_FILE package is a built-in proprietary feature that enables access for files stored externally to the database (for example, operating system storage, database server, or a connected storage volume such as NFS). Package procedures include the following subprograms to handle, open, read, and write external files (see [Oracle UTL\\_FILE sub-programs](#) for a complete list):

- UTL\_FILE.FOPEN
- UTL\_FILE.FCLOSE
- UTL\_FILE.FCLOSE\_ALL
- UTL\_FILE.FFLUSH
- UTL\_FILE.FILE\_TYPE
- UTL\_FILE.GET\_LINE
- UTL\_FILE.PUT\_LINE
- UTL\_FILE.FGETATTR
- UTL\_FILE.FGETPOS
- UTL\_FILE.FOPEN\_NCHAR
- UTL\_FILE.FREMOVE
- UTL\_FILE.FRENAME
- UTL\_FILE.FSEEK

### Oracle UTL\_FILE Usage Example

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

- UTL\_FILE.FILE\_TYPE creates and handles the file.
- UTL\_FILE.FOPEN opens access to the file and specifies:
  - The logical Oracle directory object pointing to the O/S folder where the file resides.
  - The file name and the file access mode: 'A'=append mode, 'W'=write mode
- UTL\_FILE.PUT\_LINE writes a single line to the output file.
- UTL\_FILE.GET\_LINE reads a line from the input file into a variable.
- UTL\_FILE.FCLOSE closes the file edit mode.

```
DECLARE
    FHANDLE UTL_FILE.FILE_TYPE;
```

```
BEGIN  
  
    FHANDLE := UTL_FILE.FOPEN(  
        'TMP_DIR',  
        'file.txt',  
        'a');  
  
    UTL_FILE.PUT(FHANDLE, 'Hello world!');  
    UTL_FILE.FCLOSE(FHANDLE);  
  
EXCEPTION  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('ERROR: ' || SQLCODE || ' - ' ||  
            SQLERRM);  
        RAISE;  
    END;  
/  
/
```

For more information, see [Oracle UTL\\_FILE](#).

## Oracle UTL\_MAIL and UTL\_SMTP (11g & 12c) Conversion to Azure Functions Integration

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**

N/A

**Differences Summary:**

- PostgreSQL supports Oracle's UTL\_MAIL package or includes an equivalent functionality.
- [Azure Functions](#) integration should be used for Oracle's UTL\_MAIL conversion purposes.

### Oracle UTL\_MAIL Package

The Oracle UTL\_MAIL package enables sending email messages from within the Oracle database itself according to user-defined conditions.

The solution uses [SendGrid on Microsoft Azure](#) to provide reliable email services, and Azure Functions provide the code required to invoke the SendGrid API.

**Commented [SS3]:** Reminder: Should we update this section with the information provided in the 'Sending emails from Azure Database for PostgreSQL and Azure SQL Database with Azure Functions and SendGrid on Azure.docx' file?

**Commented [SS4R3]:** Add this link as well - <https://github.com/microsoft/DataMigrationTeam/tree/master/PostgreSQLtoAzure/Scripts/AF%20SendMail>



The cloud infrastructure provides up-to-date servers to your application without provisioning Virtual Machines or any other solution and with reduced costs.

The solution can scale vertically or horizontally by adding more resources or executing multiple Azure Functions in parallel. The frequency for pooling the database for new messages also can be adjusted. All source code is provided for custom expansion.

The integration with the databases is based on a table queue where you insert one row for each message you want to send. You can have as many database threads writing to the queue tables as well as multiple Azure Functions sending email concurrently from the same queues.

The component that delivers email runs outside the database engine in a separate process. The database continues to queue e-mail messages even if the external process stops or fails.

## Solution Components



### Function App



### SendGrid

## Setup

### 1. Create a SendGrid Account

- Sign into the Azure portal.
- Create a SendGrid resource.

The screenshot shows the Azure portal interface. At the top, there are navigation links for 'SendGrid' and 'Twilio SendGrid'. Below this, a search bar contains the text 'SendGrid'. A 'Create' button is visible. The main content area displays a summary of the SendGrid service, mentioning it's the world's largest cloud-based service for delivering email that matters. It highlights SendGrid's proven platform successfully delivers over 188 transactional and marketing related emails each month for internet and mobile-based customers like Airbnb, Pandora, Hubspot, Spotify, Uber and FourSquare as well as more traditional enterprises like Wal-Mart, Intel and Costco. It notes that Azure customers receive up to 25,000 emails per month for free with paid packages starting at only \$9.99 per month. For customers requiring the ability to send larger email volumes, SendGrid also offers Silver, Gold, Platinum and Premier packages, which include a dedicated IP as well as additional IP and user management features.

- Create a SendGrid account and obtain an API KEY.

After you follow the online instructions to create your SendGrid resource, click "Manage" to manage the send grid and continue the process to obtain an API KEY, including the required Sender Identity confirmation.

## 2. Create a queue table:

- You can create one queue table on each database you want to send messages or create a table on a central database and have other databases write to this table.
- Processed rows are deleted from the queue table after the email is sent. You can create triggers on the queue table if you want to save history.
- The message is deleted from the queue table only after its guaranteed delivery to the SendGrid API. Hence in case of API failure no messages will be lost.
- You can run as many Azure Functions reading from the same queue tables. There is no conflict, and no email is lost or sent more than once.

### Create Queue Table Script for Azure Database for PostgreSQL

```
CREATE TABLE SendGridQueue (
    Sender          varchar(200),
    Tos_comma_separated  varchar(2000),
    Subject         varchar(200),
    PlainTextContent  varchar(4000),
    HtmlContent      varchar(4000),
    id              int GENERATED ALWAYS AS IDENTITY,
    PRIMARY KEY (id)
);
```

### Create Queue Table Script for Azure SQL Database

```
CREATE TABLE SendGridQueue (
    Sender          varchar(200),
    Tos_comma_separated  varchar(2000),
    Subject         varchar(200),
    PlainTextContent  varchar(4000),
    HtmlContent      varchar(4000),
    id              int PRIMARY KEY IDENTITY
);
```

Insert an email message into the Queue Table for sending:

```
INSERT INTO sendgridqueue (sender, tos_comma_separated, subject, plaintextcontent, htmlcontent)
VALUES ('test@example.com', 'test@example.com, test@live.com', 'Test Email Subject', 'Test Email PlainText Body', '<strong>Test Email HTML Body</strong>');
```

**Fields:**

Sender	The "from:" email you are sending
tos_comma_separated	List of "to:" emails we are sending separate by commas
Subject	The email subject
Plaintextcontent	Plain text email body
Htmlcontent	Html email body

## 3. Create and Publish your Azure Function with Visual Studio 2019.

- Open project  AFSendMail.sln in Visual Studio 2019.
- Solution is a C# project of project type Azure Functions and the Target Framework is .NET CORE 3.1.
- Required nugget references:



If you want to run the solution locally, configure for local testing your API Key and Connection String in the solution project file local.settings.json.



Your connection string determines if you are using ADO.NET for Azure SQL Database or Azure Database for PostgreSQL.

Sample connections strings:

```
Server=jmnetopg.postgres.database.azure.com;Database={your_database};Port=5432;User Id=joseneto@jmnetopg;Password={your_password};Ssl Mode=Require;
```

Or

```
Server=tcp:jmnetodb.database.windows.net,1433;Initial Catalog=testdb;Persist Security Info=False;User ID=joseneto;Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
```



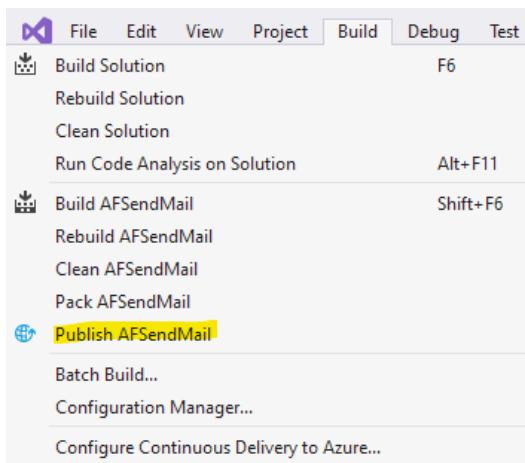
Note: Only one type of database connection is allowed per Azure Function. If you have multiple databases servers, you must use one Azure Function for each.

This is defined in this fragment of the code:

```
// Gather the connection string
string connString = Environment.GetEnvironmentVariable("CONNECTION_STRING");
if (connString == null)
    throw new Exception("CONNECTION_STRING is undefined ");

// Based on connection string hint determine if we are connecting to Azure SQL DB or Azure DB for PostgreSQL
if (connString.ToLower().Contains("postgres.database.azure.com"))
{
    ds = PostgreSQldbHelper.RetrieveMessages(connString);
}
else if (connString.ToLower().Contains("database.windows.net"))
{
    ds = SQldbHelper.RetrieveMessages(connString);
}
else
    throw new Exception("CONNECTION_STRING is invalid, must connect to AzureSQL Database for PostgreSQL or Azure SQL Database");
```

- d. In Visual Studio 2019, select the option to Publish to Azure:



- e. Before Publishing for the first time, you have to configure your Azure Environment to run Azure Functions, including selecting a Resource Group, Azure Function Plan, and Storage Account.

For more information, see the following:

[Quickstart: Create your first function in Azure using Visual Studio](#)  
[Develop Azure Functions using Visual Studio](#)  
[Deployment technologies in Azure Functions](#)

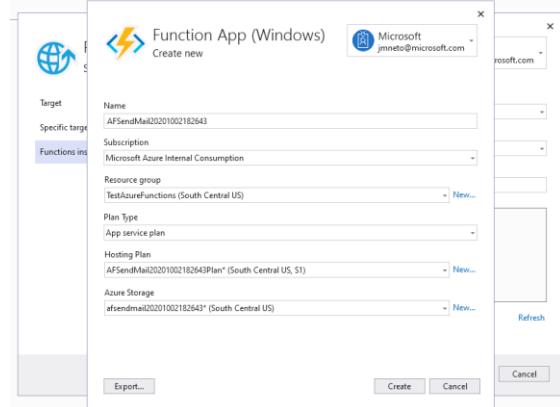
**Note:** "Always On" option is required. To make sure your Azure Function continues running constantly, you must select **App Service Plan** when creating the host for the Azure Function. On an



App Service plan, the functions runtime goes idle after a few minutes of inactivity, so only HTTP triggers "wake up" your functions. Always on is available only on an App Service plan.

[Enable Always On when running on dedicated App Service Plan](#)

Make sure you select App Service Plan to make sure your Function stays on and running.



f. After publishing, refresh the hosting app:

The screenshot shows the Azure portal interface for the 'AFSendMail20201002182643' function app. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events (preview), Functions, App keys, App files, Proxies, Deployment, Deployment slots, Deployment Center, Settings, Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains, TLS/SSL settings, and Networking. The main content area displays the app's details: Resource group (TestAzureFunctions), Status (Running), Location (South Central US), Subscription (Microsoft Azure Internal Consumption), Subscription ID (5e38e20e-0d1e-4ff0-9cc0-e88d0bf9fd8c), and Tags (none). Below this, there are two metrics charts: 'Memory working set' and 'Function Execution Count'. The 'Memory working set' chart shows usage fluctuating between 20MB and 160MB. The 'Function Execution Count' chart shows a single data point at 0.

4. Configure SENDGRID\_API\_KEY and CONNECTION\_STRING in Azure Portal.



- a. The solution requires two Application Settings to be configured:

**CONNECTION\_STRING**

**SENDGRID\_API\_KEY**

- b. Open your function configuration screen and add Application Settings SENDGRID\_API\_KEY and CONNECTION\_STRING:

The screenshot shows the Azure Function App Configuration page for 'AFSendMail20201002182643'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events (preview), Functions (selected), App keys, App files, Proxies, Deployment (Deployment slots, Deployment Center), Settings (Configuration selected), and Authentication / Authorization.

The main area displays the 'Application settings' tab. It shows a table of application settings with columns for Name and Value. The settings listed are:

Name	Value
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click to show value
APPLICATIONINSIGHTS_CONNECTION_STRING	Hidden value. Click to show value
AzureWebJobsDashboard	Hidden value. Click to show value
AzureWebJobsStorage	Hidden value. Click to show value
<b>CONNECTION_STRING</b>	Hidden value. Click to show value
ConnectionStrings__AzureWebJobsStorage	Hidden value. Click to show value
FUNCTIONS_EXTENSION_VERSION	Hidden value. Click to show value
FUNCTIONS_WORKER_RUNTIME	Hidden value. Click to show value
<b>SENDGRID_API_KEY</b>	Hidden value. Click to show value
WEBSITE_RUN_FROM_PACKAGE	Hidden value. Click to show value

- c. Configure the interval for SendGrid Azure Function to pool the Queue table and send emails.

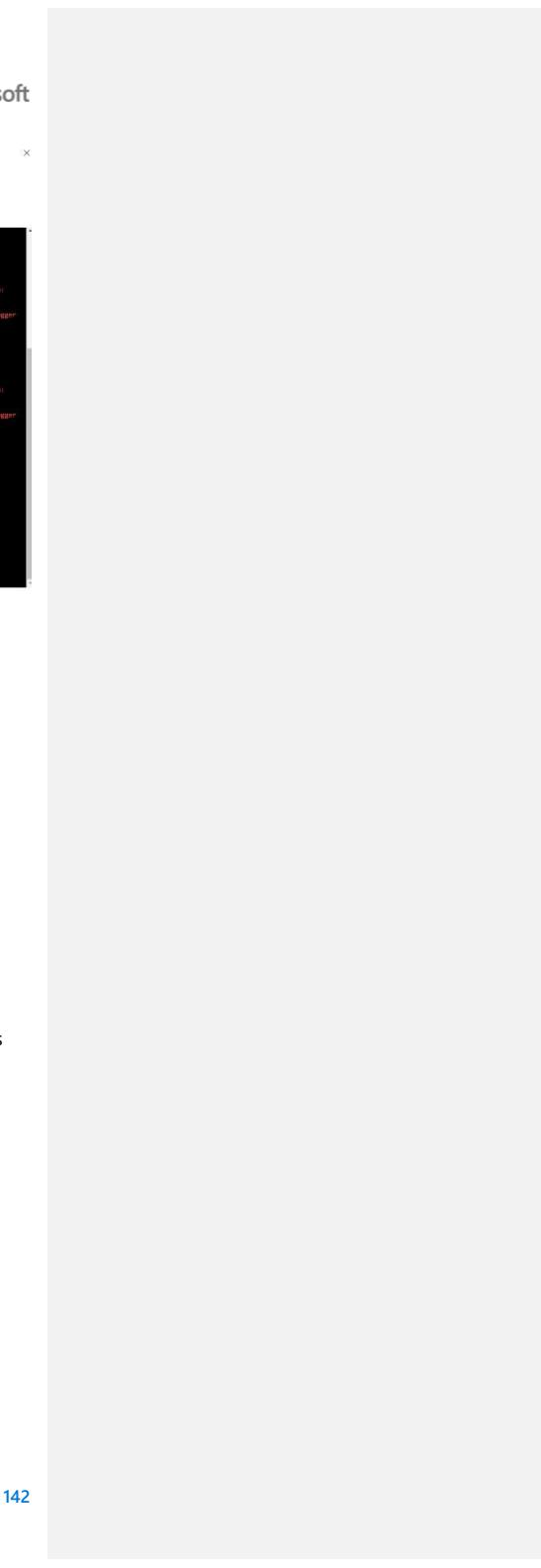
The timer trigger is defined in this portion of the code and is defined as a NCRONTAB expression.

```
namespace AFSendMail
{
    public static class SendMail
    {
        [FunctionName("SendMail")]
        public static void Run([TimerTrigger("0 */1 * * *")] TimerInfo myTimer, ILogger log)
        {
            try
            {

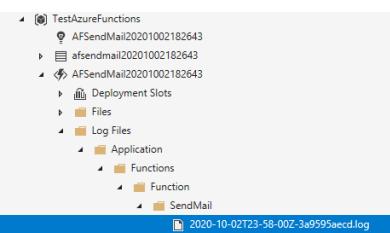
```

## Monitoring

You can see the Azure Function console function to follow operation. (Application insights must be activated.)



Azure Function log can also be seen in Visual Studio Cloud Explorer.



For more information, see [AF\\_SendMail](#).

## Oracle UTL\_SMTP Package

The Oracle UTL\_SMTP package is Oracle's earlier implementation of database and email integration. It enables sending email messages and alerts about database events. UTL\_SMTP is more complex than UTL\_MAIL and does not support attachments. For most cases, UTL\_MAIL is more suitable.

```
-- OS level packages installation
$ #{@{ORACLE_HOME}}/rdbms/admin/utlmail.sql
$ #{@{ORACLE_HOME}}/rdbms/admin/prvtmail.plb

-- Configuring the SMTP server
ALTER SYSTEM SET smtp_out_server = 'smtp.domain.com' SCOPE=BOTH;
```



```

EXEC utl_mail.send('SenderEmail@domain.com',
                   'recipientEmail@domain.com',
                   NULL,
                   NULL,
                   'Subject field',
                   'Message body',
                   NULL, 3, NULL);
/

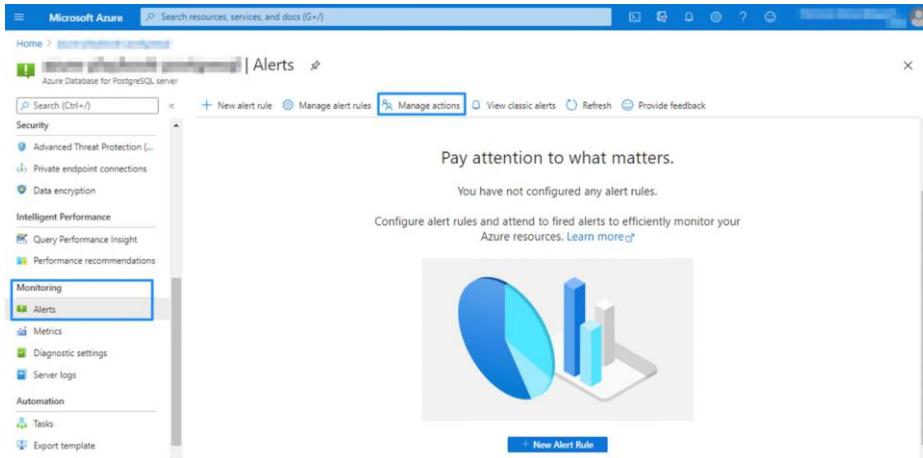
```

### Azure Database for PostgreSQL Email Notification

Although PostgreSQL does not support sending email messages from the database itself, Azure provides several alternative solutions to support sending email based on specific database events, such as alerts using Azure Database for PostgreSQL monitoring rules. For user-defined events, you can use Azure to send emails based on a specific database event.

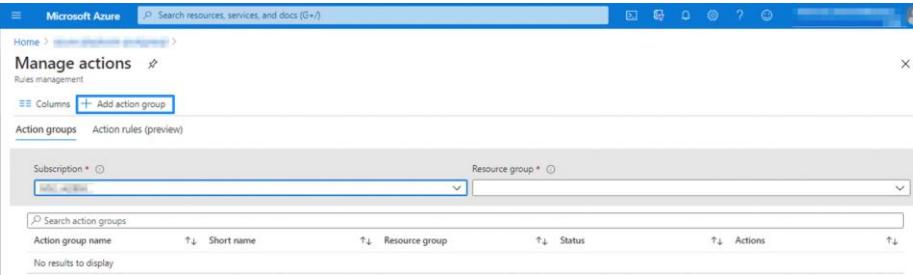
To configure alerts for sending email in Azure Database for PostgreSQL:

1. On Azure Portal, navigate to the Azure Database for PostgreSQL resource.
2. On the left side, select "Alerts" under "Monitoring" and click "Manage actions".

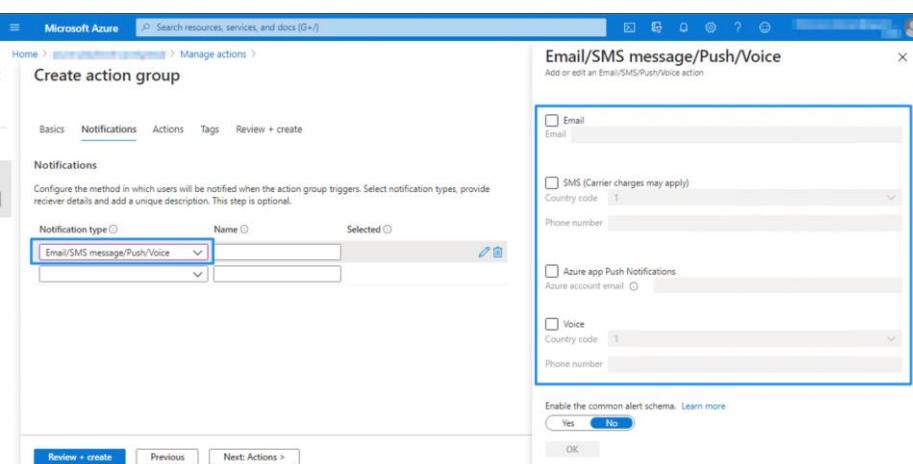


3. Click "+ Add action group"





4. Under "Instance details", type the action group name and click "Next:Notifications".  
 5. Under the "Notification type" select "Email/SMS message/Push/Voice".  
 6. On the right side, email configuration is available.



For more information, see [Azure Portal - Set alerts for Azure Database for PostgreSQL](#).

### Azure Functions

As an alternative to Oracle UTL\_MAIL, consider using Azure Functions to implement complex functionality using code. Azure Functions is a serverless application that can be launched by a specific type of event. [Supported triggers](#) include responding to changes in data, responding to messages, running on a schedule, or an HTTP request.

See the following for additional information:

- [Oracle UTL\\_MAIL](#)
- [Azure Functions](#)
- [Create a function in Azure that is triggered by a timer](#)
- [Azure Function & Azure Database for PostgreSQL example](#)

## Oracle DBMS\_OUTPUT conversion to PostgreSQL RAISE

[back to summary](#)

### General compatibility Level:

### Ora2pg automation capability:

### Differences Summary:

- Different paradigm and syntax require application/drivers rewrite because PostgreSQL does not have an equivalent to Oracle DBMS\_OUTPUT.
- PostgreSQL uses the [RAISE](#) statement to print user output.
- As an alternative solution, the PostgreSQL extension [orafce](#), which is [supported](#) by Azure Database for PostgreSQL (see [supported extensions](#)), can be used with the same syntax and has similar functionality to Oracle DBMS\_OUTPUT.

### Oracle DBMS\_OUTPUT Package

The Oracle DBMS\_OUTPUT package is typically used for displaying output messages from a PL/SQL store procedure and can be used as a debugging tool for development and notifications/errors output.

In the following example, DBMS\_OUTPUT.PUT\_LINE is used with a combination of bind variables to dynamically display values from a table running in a FOR LOOP. With the for-loop iteration, DBMS\_OUTPUT.PUT\_LINE prints the columns output. To display notifications on to the screen, you must configure the session with SET SERVEROUTPUT ON.

```
-- Set the Oracle session to display the DBMS_OUTPUT.PUT_LINE output
set serveroutput on;

CREATE OR REPLACE PROCEDURE emp_name
AS
BEGIN
    FOR v IN(SELECT FIRSTNAME, LASTNAME FROM EMPLOYEE) LOOP
        DBMS_OUTPUT.PUT_LINE(
            'Employee Full Name: '||v.FIRSTNAME||' '||v.LASTNAME);
    END LOOP;
END;
/
exec emp_name;
```

```
-- Procedure output  
  
Employee Full Name: Andrew Adams  
Employee Full Name: Nancy Edwards  
Employee Full Name: Jane Peacock  
Employee Full Name: Margaret Park  
Employee Full Name: Steve Johnson  
Employee Full Name: Michael Mitchell  
Employee Full Name: Robert King  
Employee Full Name: Laura Callahan
```

In addition to the output of information on the screen, the PUT and PUT\_LINE procedures in the DBMS\_OUTPUT package enable to buffer information 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.

### **PostgreSQL RAISE Statement**

You can use the PostgreSQL RAISE statement as an alternative to Oracle's DBMS\_OUTPUT package as the PostgreSQL native print output solution. Use the PostgreSQL [orafe](#) extension to reduce the syntax conversion/replacement requirements when performing Oracle to PostgreSQL schema conversion. Both methods are demonstrated in the following examples.

PostgreSQL RAISE can be used with several levels of severity:

- 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, such as checkpoint activity.
- FATAL - Reports an error that caused the current session to abort.
- PANIC - Reports an error that caused all database sessions to abort.

### **PostgreSQL RAISE Statement Example**

RAISE DEBUG (where DEBUG is the configurable severity level) for functionality similar to Oracle DBMS\_OUTPUT.PUT\_LINE.

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



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

-- Code output

```
DEBUG: hello world
DO
```

Convert from Oracle DBMS\_OUTPUT.PUT\_LINE to PostgreSQL RAISE statement using Ora2Pg.

```
CREATE OR REPLACE FUNCTION proc1 ()
RETURNS VOID AS $body$
DECLARE
    v record;
BEGIN
    FOR v in(SELECT "FirstName", "LastName" from "Employee")
    LOOP
        RAISE NOTICE 'Employee Full Name: % %',
                      v."FirstName" , v."LastName";
    END LOOP;
END;
$body$
LANGUAGE PLPGSQL
SECURITY definer;

-- Code output
NOTICE: Employee Full Name: Andrew Adams
NOTICE: Employee Full Name: Nancy Edwards
NOTICE: Employee Full Name: Jane Peacock
NOTICE: Employee Full Name: Margaret Park
```



```
NOTICE: Employee Full Name: Steve Johnson  
NOTICE: Employee Full Name: Michael Mitchell  
NOTICE: Employee Full Name: Laura Callahan  
NOTICE: Employee Full Name: Robert King
```

### PostgreSQL and orafce Extension Example

Convert from Oracle DBMS\_OUTPUT.PUT\_LINE to PostgreSQL orafce extension using the same syntax as DBMS\_OUTPUT.PUT\_LINE:

```
-- Add the orafce extension  
  
CREATE EXTENSION orafce;  
  
CREATE EXTENSION  
  
-- Set session DBMS_OUT.SERVEROUTPUT parameter to ON  
-- Equivalent To Oracle SET SERVEROUTPUT ON statement  
  
SELECT dbms_output.serveroutput('on');  
  
-- Create Oracle function holding the DBMS_OUTPUT.PUT_LINE function  
  
CREATE OR REPLACE FUNCTION emp_name()  
RETURNS void  
AS $body$  
DECLARE  
    empName record;  
BEGIN  
    FOR empName IN(  
        SELECT "FirstName", "LastName" from "Employee")  
    LOOP  
        PERFORM  
            DBMS_OUTPUT.PUT_LINE(empName."FirstName" || ' ' ||  
                empName."LastName") from dual;  
    END LOOP;  
END;
```



```
$body$  
LANGUAGE PLPGSQL  
SECURITY definer;  
  
-- Execute the Function  
SELECT emp_name();  
  
Employee Full Name: Andrew Adams  
Employee Full Name: Nancy Edwards  
Employee Full Name: Jane Peacock  
Employee Full Name: Margaret Park  
Employee Full Name: Steve Johnson  
Employee Full Name: Michael Mitchell  
Employee Full Name: Laura Callahan  
Employee Full Name: Robert King
```

Azure Database for PostgreSQL List of supported extensions by SQL statement:

```
SELECT * FROM pg_available_extensions;
```

See the following for more information:

- [Oracle DBMS\\_OUT Package](#)
- [PostgreSQL Errors and Messages](#)
- [PostgreSQL log\\_min\\_messages](#)
- [PostgreSQL Error Codes](#)

## Oracle DBMS\_RANDOM conversion to PostgreSQL RANDOM Function

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Different syntax may require code rewrite.
- As an alternative solution, the PostgreSQL extension [orafce](#), which is [supported](#) by Azure Database for PostgreSQL (see [supported extensions](#)), can be used with the same syntax and has similar functionality to Oracle DBMS\_RANDOM.

Oracle DBMS\_RANDOM generates random numbers or strings as part of an SQL statement or PL/SQL procedure and include the following elements:

- 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 of precision. Alternatively, you could generate a random number greater than or equal to a low parameter and less than a high parameter.

Oracle DBMS\_RANDOM sub-programs include:

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

### Oracle DBMS\_RANDOM Examples

Generate a random numeric value:

```
SELECT dbms_random.VALUE() FROM dual;
```

```
DBMS_RANDOM.VALUE()
```

```
-----  
.37544896
```

```
SELECT dbms_random.VALUE() FROM dual;
```

```
DBMS_RANDOM.VALUE()
```

```
-----  
.235945549
```

Generate a random string:

```
SELECT dbms_random.STRING('P',6) FROM dual;
```

```
DBMS_RANDOM.STRING('P',6)
```

```
-----  
|GjF=q
```

```
SELECT dbms_random.string('P',6) FROM dual;
```

```
DBMS_RANDOM.STRING('P',6)
```

```
-----  
o`V`k'
```

### PostgreSQL RANDOM Functionality

PostgreSQL does not provide a dedicated package equivalent to Oracle DBMS\_RANDOM, but the orafce extension can provide an alternative solution using identical Oracle syntax and similar functionality for the DBMS\_RANDOM package. You can use PostgreSQL native functions as workarounds, such as generating random numbers using the `random()` function. To generate random strings, use the value returned from the `random()` function coupled with an `md5()` function.

### PostgreSQL [orafce](#) Supported Random Functions

Package DBMS_RANDOM Sub-Programs	Description
<code>dbms_random.initialize(int)</code>	Initialize package with a seed value
<code>dbms_random.normal()</code>	Returns random numbers in a standard normal distribution
<code>dbms_random.random()</code>	Returns a random number from -2^31 to 2^31
<code>dbms_random.seed(int)</code>	Generate seed value
<code>dbms_random.seed(text)</code>	Reset seed value
<code>dbms_random.string(opt text(1), len int)</code>	Create random string
<code>dbms_random.terminate()</code>	Terminate package (do nothing in Pg)



dbms_random.terminate()	Returns a random number from 0.0 to 1.0
dbms_random.value(low double precision, high double precision)	Returns a random number from <low> to <high>

## PostgreSQL Native and Oracle Extension Random Examples

Generate a random numeric value (PostgreSQL native):

```
SELECT random();
```

```
random
```

```
-----
```

```
0.19321764446795
```

```
(1 row)
```

```
SELECT random();
```

```
random
```

```
-----
```

```
0.707136384677142
```

```
(1 row)
```

Generate a random string value (PostgreSQL native):

```
SELECT md5(random()::text);
```

```
md5
```

```
-----
```

```
4e233580e1a3f95c635d73b8e2292f11
```

```
SELECT md5(random()::text);
```

```
md5
```

```
-----  
f31bb576cf49b09a572b2211cea2167  
(1 row)
```

Use the orafce extension to generate random values with the same syntax as Oracle:

```
SELECT dbms_random.VALUE() FROM dual;  
  
      value  
-----  
0.19329833984375  
(1 row)  
  
SELECT dbms_random.VALUE() FROM dual;  
  
      value  
-----  
0.4798583984375  
(1 row)  
  
SELECT dbms_random.STRING('p',6) FROM dual;  
string  
-----  
kCJ|r:  
(1 row)  
  
SELECT dbms_random.STRING('p',6) FROM dual;  
string  
-----  
`'$*_+  
(1 row)
```

See the following for more information:

- [Oracle DBMS RANDOM](#)
- [PostgreSQL Mathematical Functions and Operators](#)
- [PostgreSQL String Functions and Operators](#)

### Oracle DBMS\_SQL Package Conversion to PostgreSQL

**General compatibility Level:** 

**Ora2pg automation capability:** 

#### Differences Summary:

- Different paradigm and syntax require application rewrite.

Oracle DBMS\_SQL package provides rich APIs to parse and execute dynamic SQL statements, DML commands, and DDL commands, typically from within a PL/SQL package, function, or procedure. Oracle's DBMS\_SQL enables very granular control of SQL cursors and can improve cursor performance in certain cases while providing a managed and controlled approach to opening, parsing, binding, executing, and fetching data from a cursor using the DBMS\_SQL PL/SQL interface.

Following are the main DBMS\_SQL subprograms:

- DBMS\_SQL.OPEN\_CURSOR - Open a blank cursor and return the cursor handle.
- DBMS\_SQLPARSE - Parse the statement into the referenced cursor.
- DBMS\_SQL.BIND\_VARIABLES - Attach the value for the bind variable with the cursor.
- DBMS\_SQL.EXECUTE - Execute the cursor.
- DBMS\_SQL.GET\_NEXT\_RESULT - Iterate over the cursor, fetching the next result.
- DBMS\_SQL.CLOSE\_CURSOR - Close the cursor.

The Oracle DBMS\_SQL package provides additional functionality:

- RETURN\_RESULT
  - Introduced in Oracle 12c, this sub-program 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.

### Oracle DBMS\_SQL Package Example



```
CREATE OR REPLACE PROCEDURE ora_object_drop
(p_obj_name IN VARCHAR2, p_obj_type IN VARCHAR2)
IS
-- The static cursor retrieving all matching objects
CURSOR obj_cur IS
  SELECT object_name, object_type FROM user_objects
  WHERE object_name LIKE UPPER (p_obj_name)
  AND   object_type LIKE UPPER (p_obj_type);

cur PLS_INTEGER := DBMS_SQL.OPEN_CURSOR;
fdbk PLS_INTEGER;
BEGIN
-- Search for matching objects
FOR obj_rec IN obj_cur LOOP
-- Initiating the cursor, parse and execute the drop statement
DBMS_SQLPARSE (
  cur, 'DROP ' || obj_rec.object_type || ' ' ||
  obj_rec.object_name, DBMS_SQL.NATIVE);
fdbk := DBMS_SQL.EXECUTE (cur);
END LOOP;
DBMS_SQL.CLOSE_CURSOR(cur);
END;
/
```

### PostgreSQL PL/pgSQL Cursors and Dynamic Operations

PostgreSQL does not support granular control of programmatic cursors and therefore does not have an equivalent for Oracle DBMS\_SQL. However, you can dynamically parse and execute SQL statements using PostgreSQL PL/pgSQL.

Create a dynamic cursor by using the FOR loop with SELECT statement:

```
CREATE OR REPLACE FUNCTION fetch_messages()
RETURNS VARCHAR
```



```
AS
$$
DECLARE
    cur      RECORD;
    msg      VARCHAR(200);
    text     VARCHAR(10);
    msg_num  VARCHAR(10);

BEGIN
    msg := '';
    FOR cur IN
        SELECT text, msg_num, count(*)
        FROM msg_table
        GROUP BY text, msg_num
    LOOP
        text:= cur.text;
        msg_num := cur.msg_num;
        msg := msg||rpad(text,20)||rpad(msg_num,20);
    END LOOP;
    RETURN msg;
END;
$$
LANGUAGE plpgsql;
```

PostgreSQL function holding a CURSOR and EXECUTE for dynamically controlling the cursor datasets:

```
CREATE OR REPLACE FUNCTION fetch_messages ()
RETURNS VARCHAR AS $$

DECLARE
    refcur refcursor;
    c_id integer;
    text varchar (10);
```



```
msg_num varchar (10);
msg_info VARCHAR(1000) := '';

BEGIN
OPEN refcur FOR EXECUTE('SELECT text, msg_num,
                           FROM msg_table');

LOOP
  FETCH refcur INTO text, msg_num;
  IF NOT FOUND THEN
    exit;
  END IF;
  msg_info:= msg_info||rpad(text,20)||rpad(msg_num,20);
END LOOP;
CLOSE REFCUR;
RETURN msg_info;
END;
$$
LANGUAGE plpgsql
```

See the following for more information:

- [Oracle - PL/SQL Dynamic SQL](#)
- [PostgreSQL DEALLOCATE](#)
- [PostgreSQL PREPARE](#)
- [PostgreSQL - Executing Dynamic Commands](#)

## Oracle DBMS\_SCHEDULER Conversion to Scheduled Azure Functions

**General compatibility level:**



**Ora2pg automation capability:**

N/A

### Differences Summary:

- PostgreSQL does not have an internal scheduler to support database-level jobs.
- Migrate Oracle DBMS\_SCHEDULER to Azure Functions as an alternative solution

The DBMS\_SCHEDULER package contains a collection of scheduling functions, either executable or called from PL/SQL. When creating an Oracle DBMS\_SCHEDULER job, two additional objects need to be addressed: PROGRAM and SCHEDULE. A program defines what is executed when the job calls it, and the Scheduler runs database program units or external supported executables (for example, filesystem scripts).

Jobs have three execution methods: time-based scheduling, event-based jobs, and dependency jobs (chained):

- Time-based scheduling
  - Specify the program to run and schedule the program execution.
- Event-based jobs
  - Create a scheduled procedure that starts a job whenever the scheduler receives an event indicating that a file arrived to execute the event-base job.
- Dependency jobs (chained)
  - Scheduled tasks that are linked together for a combined objective.
  - Use chains to implement dependency-based scheduling, in which jobs are started depending on the outcomes of one or more previous jobs.

### Oracle DBMS\_SCHEDULER Examples

- Time based scheduling - Can be configured with or without specifying the following parameters (partial list):
  - JOB\_TYPE - Job action type ('PLSQL\_BLOCK', 'STORED\_PROCEDURE', 'EXECUTABLE', 'CHAIN', 'EXTERNAL\_SCRIPT', 'SQL\_SCRIPT', and 'BACKUP\_SCRIPT')
  - JOB\_ACTION - Inline action of the job. This is either the code for an anonymous PL/SQL block or the name of a stored procedure, external executable, or chain
  - START\_DATE - Start date and time of the job.
  - REPEAT\_INTERVAL - Specifies the type of recurrence.

```
-- CREATE_PROGRAM
BEGIN
    DBMS_SCHEDULER.CREATE_PROGRAM (
        => '<JobName>',
        program_action => '<StoredProcedureName>',
        program_type => 'STORED_PROCEDURE',
        enabled => TRUE);
END;
```

```
/\n\n-- CREATE_SCHEDULE\n\nBEGIN\n\n    DBMS_SCHEDULER.CREATE_SCHEDULE(\n        schedule_name => '<ScheduleName>',\n        start_date => SYSTIMESTAMP,\n        repeat_interval => 'FREQ=HOURLY;INTERVAL=3',\n        comments => 'Every 3 hours');\n\n    END;\n/\n\n-- CREATE_JOB\n\nBEGIN\n\n    DBMS_SCHEDULER.CREATE_JOB (\n        job_name => '<>JobName',\n        program_name => '<ProgramName>',\n        schedule_name => '<ScheduleName>');\n\n    END;\n/\n\n-- Specifying all elements in a single scheduler job via (CREATE_JOB)\n\nBEGIN\n\n    DBMS_SCHEDULER.CREATE_JOB(\n        job_name=>'DB_FULL_BACKUP',\n        job_type => 'EXECUTABLE',\n        job_action => '/u01/dba/scripts/rman/daily_bck.sh',\n        start_date=> SYSDATE,\n        repeat_interval=>'FREQ=DAILY;BYHOUR=22',
```



```
    comments => 'Daily-backups');

END;
/

-- Altering Oracle scheduler job attributes

BEGIN

    DBMS_SCHEDULER.SET_ATTRIBUTE (
        name => 'DB_FULL_BACKUP',
        attribute => 'repeat_interval',
        value => 'FREQ=HOURLY');

END;
/

```

- Event-based jobs:

```
-- Creating an Event-Based Schedule

BEGIN

    DBMS_SCHEDULER.CREATE_EVENT_SCHEDULE (
        schedule_name => '<JobName>',
        start_date => systimestamp,
        event_condition => 'tab.user_data.object_owner = ''HR''
            and tab.user_data.event_name = ''FILE_ARRIVAL''
            and extract hour from tab.user_data.event_timestamp < 7',
        queue_spec => 'my_events_q');

END;
/

-- Creating an Event-Based Job

BEGIN

    DBMS_SCHEDULER.CREATE_JOB (
        job_name => my_job,
        program_name => my_program,
```



```
start_date => '01-JAN-20 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;
/

```

- Dependency jobs (chained):

```
-- Creating an Event-Based Schedule
BEGIN
    DBMS_SCHEDULER.CREATE_CHAIN (
        chain_name => 'Chain1',
        rule_set_name => NULL,
        evaluation_interval => NULL,
        comments => NULL);
    END;
/
BEGIN
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('Chain1','stepA',
        'my_program1');
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('Chain1','stepB',
        'my_program2');
    DBMS_SCHEDULER.DEFINE_CHAIN_STEP('Chain1','stepC',
        'my_program3');
    END;
/
BEGIN
    DBMS_SCHEDULER.DEFINE_CHAIN_RULE('Chain1', 'TRUE', 'START

```



```
        stepA');

DBMS_SCHEDULER.DEFINE_CHAIN_RULE('Chain1', 'stepA
                                    COMPLETED', 'Start stepB, stepC');

DBMS_SCHEDULER.DEFINE_CHAIN_RULE(
    'Chain1', 'stepB COMPLETED AND stepC COMPLETED', 'END');

END;
/

BEGIN
    DBMS_SCHEDULER.ENABLE('Chain1');
END;
/

BEGIN
    DBMS_SCHEDULER.CREATE_JOB (
        job_name => 'Chain1',
        job_type => 'CHAIN',
        job_action => 'Chain1',
        repeat_interval =>
        'freq=daily;byhour=19;byminute=30;bysecond=0',
        enabled => TRUE);
END;
/
```

### Azure Database for PostgreSQL Job Scheduling Considerations

There are differences regarding the supported extension between Azure Database for PostgreSQL - [Single Instance](#) and Azure Database for PostgreSQL - [Hyperscale \(Citus\)](#).

- For Azure Database for PostgreSQL - Single Instance, no extension exists to support database level scheduler, so consider these Azure services:
  - [Azure Logic Apps](#) for schedule-based and recurring automation workflows with a PostgreSQL connector.
  - Azure Functions can be used by a defined timer.



- Azure Database for PostgreSQL - Hyperscale (Citus) supports the PostgreSQL [pg\\_cron](#) extension to implement a cron-based job scheduler that runs inside the database. It uses the same syntax as regular Linux cron and allows you to schedule PostgreSQL commands directly from the database. In addition, pg\_cron can run multiple jobs in parallel. When Oracle DBMS\_SCHEDULER is used extensively, consider migrating to Azure Database for PostgreSQL - Hyperscale (Citus).

```
-- Delete old data on Saturday at 3:30am (GMT)

SELECT cron.schedule('30 3 * * 6', $$DELETE FROM events WHERE
    event_time < now() - interval '1 week'$$);

schedule
-----
42

-- Vacuum every day at 10:00am (GMT)

SELECT cron.schedule('0 10 * * *', 'VACUUM');

schedule
-----
43

-- Stop scheduling a job

SELECT cron.unschedule(43);

unschedule
-----
```

See the following for more information:

- [Oracle Scheduler Concepts](#)
- [Azure - Create a function in Azure that is triggered by a timer](#)

#### PostgreSQL 11 updates:

None

#### PostgreSQL 12 updates:



random() and setseed() now behave uniformly across platforms. The sequence of random() values generated following a setseed() call with a particular seed value is likely to be different now than before. However, it is also repeatable, which was not previously guaranteed because of interference from other uses of random numbers inside the server. The SQL random() function now has its own private per-session state to forestall that.

**PostgreSQL 13 updates:**

None

## Migrating from Oracle Global Temporary Tables

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- PostgreSQL global temporary table cannot read from multiple sessions.
- PostgreSQL global temporary table is dropped after the session ends.

Oracle supports temporary data in the form of global temporary tables that can hold the data for the duration of a session or transaction. Global temporary tables in Oracle can be defined as persistent database objects that can store data on disk that is visible to all sessions. Data stored in the global temporary table is private to the session; each session can only access its own data in the Oracle global temporary table concept.

### Oracle Global Temporary Table Main Features and Limitations

- The following objects and operations are supported on Oracle global temporary tables:
  - Indexes
  - Triggers
  - DDL operations, including ALTER TABLE and DROP TABLE
- Oracle global temporary tables store the data in the temporary tablespace.
- Processing DML operations on an Oracle global temporary table does not generate redo data. However, undo data for the rows and redo data for the undo data itself are generated.
- Oracle 12c and 18c enhancements for global temporary tables:
  - Oracle 12c supports collecting statistics for session-specific temporary tables.
  - Undo data can be stored in the temporary tablespace.
  - Oracle 18c introduced private temporary tables which are memory-based temporary tables that can be dropped at the end of the session or transaction, depending on the initial configuration.
- Oracle global temporary tables cannot be partitioned, clustered, or created as index-organized tables. In addition, they do not support parallel UPDATE, DELETE, and MERGE operations.
- Foreign key constraints cannot be created on global temporary tables.

### Global Temporary Tables Transaction Support

- ON-COMMIT - The data in the global temporary table persists for the duration of a transaction or duration of a session.
- PRESERVE ROWS - The data is truncated at the end of the session, but it persists beyond the end of the transaction (within the same session).
- DELETE ROWS - The default behavior. Data is truncated after each commit.

### Oracle Global Temporary Tables Examples

Global temporary table configured with ON COMMIT PRESERVE ROWS:

```
CREATE GLOBAL TEMPORARY TABLE ORDERS_TMP (
    ORDER_ID      NUMBER PRIMARY KEY,
    CUSTOMER_ID   VARCHAR2(60) NOT NULL,
```



```
ORDER_ITEMS_AVG NUMERIC NOT NULL)
ON COMMIT PRESERVE ROWS;

CREATE INDEX IDX_ORDERS_TEMP ON ORDERS_TMP(ORDER_ID);

INSERT INTO ORDER_TMP VALUES(123, 'Company-A', '10000');
COMMIT;

SELECT * FROM ORDERS_TMP;

ORDER_ID      CUSTOMER_ID          ORDER_ITEMS_AVG
-----  -----
1            Company-A           10000
```

Global temporary table configured with ON COMMIT DELETE ROWS:

```
CREATE GLOBAL TEMPORARY TABLE ORDERS_TMP (
    ORDER_ID      NUMBER PRIMARY KEY,
    CUSTOMER_ID   VARCHAR2(60) NOT NULL,
    ORDER_ITEMS_AVG INT NOT NULL
)
ON COMMIT DELETE ROWS;

INSERT INTO ORDER_TMP VALUES(123, 'Company-A', '10000');
COMMIT;

SELECT * FROM ORDERS_TMP;

no rows selected
```

For more information, see [Oracle Global Temporary Table](#).

## Migrate to PostgreSQL Temporary Tables

PostgreSQL supports Oracle's CREATE GLOBAL TEMPORARY TABLE syntax, with several differences in the core functionality. Oracle saves the structure of the global temporary table after the session ends (it stays persistent until it is dropped) with truncating only the data. In contrast, PostgreSQL supports different behavior; the table structure (DDL) is not stored in the database, and when the session ends, the temporary table is dropped along with its data. Oracle default global temporary tables behavior is ON COMMIT DELETE ROWS (when no other configuration is specified); in PostgreSQL, the default transaction behavior is ON COMMIT PRESERVE ROWS.

## PostgreSQL Temporary Table Main Features

- Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction.
- The default behavior in PostgreSQL is ON COMMIT PRESERVE ROWS.
- The ON COMMIT DROP option does not exist in SQL.
- Any indexes created on a temporary table are automatically temporary as well.
- Statistic collection on a PostgreSQL temporary table is allowed by the ANALYZE statement, and it is recommended to run it on the temporary table after it is populated if used with complex queries.
- Session-level:
  - PostgreSQL requires that every session creates its own temporary tables.
  - Optionally, each session can create its own "private" temporary tables using identical table names.
- Local/Global:
  - Temporary tables data in PostgreSQL cannot be shared between different sessions.
  - 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.
  - It is mentioned in the official documentation that PostgreSQL "Global" keyword is deprecated (please refer to [PostgreSQL Compatibility](#) for more information).

## PostgreSQL Temporary Tables ON COMMIT Clause

Specifies the state of the data during the transaction or a session.

- PRESERVE ROWS
  - The default PostgreSQL state.
  - When a session ends, all data is truncated but persists beyond the end of the session transaction.
- DELETE ROWS
  - The data is truncated after each commit.

## PostgreSQL Temporary Tables Examples

Create temporary table configured with ON COMMIT PRESERVE ROWS:

```
CREATE GLOBAL TEMPORARY TABLE ORDERS_TMP (
    ORDER_ID      NUMERIC PRIMARY KEY,
    CUSTOMER_ID   VARCHAR(60) NOT NULL,
    ORDER_ITEMS_AVG NUMERIC NOT NULL)
ON COMMIT PRESERVE ROWS;
```



```
WARNING: GLOBAL is deprecated in temporary table creation
```

```
INSERT INTO ORDERS_TMP VALUES(123, 'Company-A', '10000');  
COMMIT;
```

```
SELECT * FROM ORDERS_TMP;
```

```
order_id | customer_id | order_items_avg  
-----+-----+-----  
123 | Company-A | 10000
```

Create a temporary table configured with ON COMMIT DELETE ROWS:

```
CREATE GLOBAL TEMPORARY TABLE ORDERS_TMP (  
    ORDER_ID      NUMERIC PRIMARY KEY,  
    CUSTOMER_ID   VARCHAR(60) NOT NULL,  
    ORDER_ITEMS_AVG NUMERIC NOT NULL)  
ON COMMIT DELETE ROWS;
```

```
WARNING: GLOBAL is deprecated in temporary table creation
```

```
INSERT INTO ORDERS_TMP VALUES(123, 'Company-A', '10000');  
COMMIT;
```

```
SELECT * FROM ORDERS_TMP;
```

```
order_id | customer_id | order_items_avg  
-----+-----+-----  
(0 rows)
```

**Oracle and PostgreSQL Temporary Tables Summary**

Temporary Table Feature	Oracle	PostgreSQL
<b>ON COMMIT Default Behavior</b>	COMMIT DELETE ROWS	ON COMMIT PRESERVE ROWS
<b>PRESERVE ROWS Support</b>	Supported	Supported
<b>DELETE ROWS Support</b>	Supported	Supported
<b>Alter Table support</b>	Supported	Supported
<b>Collect Statistics</b>	Supported by DBMS_STATS.GATHER_TABLE_STATS	Supported by ANALYZE
<b>Foreign Key Support</b>	Supported	Supported
<b>Index Support</b>	Supported	Supported
<b>DDL persist at Session End or database Reboot</b>	Supported	Not Supported All temporary tables/indexes are deleted Create a regular table
<b>Data can be accessed from multiple session</b>	Supported	Not Supported
<b>Syntax</b>	CREATE GLOBAL TEMPORARY TABLE	CREATE TEMPORARY TABLE

For more information, see [Temporary Tables](#).

## Oracle Virtual Columns conversion to PostgreSQL Views and Functions

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- PostgreSQL version 11 does not support Virtual Columns; workarounds are available. Fully supported in version 12 and later.

Oracle Virtual Columns can be set to hold a calculated value based on predefined values or functions; the values are calculated instead of stored in the database. Virtual column values cannot be created based on other virtual columns and can only reference columns from the same table. When creating a virtual column, the data type can be either explicitly specified, or selected by the database based on the expression.

### Oracle Virtual Columns Syntax

- Note that GENERATED ALWAYS and VIRTUAL keywords are provided for clarity only.
- Virtual columns can be created by specifying the keyword (see code examples).

```
column_name [datatype] [GENERATED ALWAYS] AS (expression) [VIRTUAL]
```

### Oracle Virtual Columns Examples

The table holds two virtual columns; for the first one, the GENERATED ALWAYS AS keywords are not specified:

```
CREATE TABLE invoice_tax (
    invoice_id      NUMBER PRIMARY KEY,
    customer_id     NUMBER NOT NULL,
    invoice_total   NUMBER,
    invoice_total_tax AS (ROUND(invoice_total * 1.15, 2)),
    emp_commission  NUMBER GENERATED ALWAYS AS
                    (ROUND(invoice_total * 1.05, 2));

-- Perform an INSERT
INSERT INTO invoice_tax (invoice_id, customer_id, invoice_total)
VALUES(123, 345, 99.99);

-- Verify data
SELECT * FROM invoice_tax;

INVOICE_ID CUSTOMER_ID INVOICIE_TOTAL INVOICE_TOTAL_TAX EMP_COMMISSION
----- ----- ----- -----
345      99.99       114.99        123
          104.99
```

### Oracle Virtual Columns Notes/Limitations

- Virtual columns do not support index-organized, temporary, external, objects, or clusters tables.
- Virtual columns can be used with constraints, indexes, table partitioning, and foreign keys.
- Functions in expressions must be defined during 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.

- Creating indexes on virtual columns also creates a function-based index.
- The output of a virtual column expression must be a scalar value.

### PostgreSQL Virtual Columns Workarounds

PostgreSQL version 11 does not support the implementation of Oracle virtual columns. In PostgreSQL version 12 and later, virtual columns are [supported](#) using the `GENERATED ALWAYS AS` syntax (as in Oracle). As an alternative solution in PostgreSQL 11, consider using the following:

- Views – Use views as an alternative for Oracle virtual columns by specifying the same logic as the original virtual column in the PostgreSQL view.
- Function as a column - By implementing a user-defined function (scalar function), you can apply the same logical definition implemented in Oracle virtual columns.
- Table triggers – Apply a BEFORE INSERT OR UPDATE trigger to support DML operations for virtual columns; this solution also requires a function.

### PostgreSQL Views, Functions and Triggers as Alternative Solution Examples

Use a view to simulate the virtual columns logical implementation:

```
-- Create the source table with no virtual columns
CREATE TABLE invoice_tax (
    invoice_id      NUMERIC PRIMARY KEY,
    customer_id     NUMERIC NOT NULL,
    invoice_total   NUMERIC,
    invoice_total_tax NUMERIC,
    emp_commission  NUMERIC);

-- Create a view with the same logics
CREATE VIEW vw_invoice_tax
AS
SELECT
    invoice_id,
    customer_id,
    invoice_total,
    ROUND(invoice_total * 1.15, 2) AS invoice_total_tax,
    ROUND(invoice_total * 1.05, 2) AS emp_commission
FROM invoice_tax;

-- Verify data
SELECT * FROM vw_invoice_tax;

 invoice_id | customer_id | invoice_total | invoice_total_tax | emp_commission
-----+-----+-----+-----+
 123 |      345 |       99.99 |        114.99 |        104.99
(1 row)
```

Use functions to simulate the virtual columns logical implementation for query and DML operations:

```
-- Create the source table with no virtual columns
CREATE TABLE invoice_tax (
    invoice_id      NUMERIC PRIMARY KEY,
    customer_id     NUMERIC NOT NULL,
    invoice_total   NUMERIC,
    invoice_total_tax NUMERIC,
    emp_commission  NUMERIC);
```

```
-- Create a function for each virtual column
CREATE OR REPLACE FUNCTION func_invoice_tax
(p_invoice_total NUMERIC)
RETURNS NUMERIC AS $$ 
DECLARE v_invoice_tax NUMERIC;
BEGIN
    v_invoice_tax:=p_invoice_total * 1.15;
    RETURN round(v_invoice_tax, 2);
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION func_emp_comm (p_emp_comm NUMERIC)
RETURNS NUMERIC AS $$ 
DECLARE v_emp_comm NUMERIC;
BEGIN
    v_emp_comm:=p_emp_comm * 1.05;
    RETURN round(v_emp_comm, 2);
END;
$$ LANGUAGE plpgsql;

-- DML
INSERT INTO invoice_tax (invoice_id, customer_id,
                        invoice_total, invoice_total_tax, emp_commission)
VALUES(123, 345, 99.99, func_invoice_tax(99.99),
       func_emp_comm(99.99));

-- Verify data
SELECT * FROM invoice_tax;

invoice_id | customer_id | invoice_total | invoice_total_tax | emp_commission
-----+-----+-----+-----+
      123 |        345 |      99.99 |      114.99 |      104.99

-- Query
SELECT
    invoice_id,
    customer_id,
    invoice_total,
    FUNC_INVOICE_TAX(invoice_total_tax),
    FUNC_EMP_COMM(emp_commission)
FROM invoice_tax;

invoice_id | customer_id | invoice_total | func_invoice_tax | func_emp_comm
-----+-----+-----+-----+
      123 |        345 |      99.99 |      114.99 |      104.99
```

Use triggers to simulate the virtual columns logical implementation to support DML operations:

```
-- Create the source table with no virtual columns
CREATE TABLE invoice_tax (
    invoice_id      NUMERIC PRIMARY KEY,
    customer_id    NUMERIC NOT NULL,
    invoice_total   NUMERIC,
    invoice_total_tax NUMERIC,
    emp_commission  NUMERIC);
```

```
-- Create a function to support the emp_commission columns
CREATE OR REPLACE FUNCTION func_emp_comm_trg()
RETURNS TRIGGER AS $$$
BEGIN
    new.emp_commission = ROUND(new.invoice_total * 1.05, 2);
    RETURN new;
END;
$$ LANGUAGE plpgsql;

-- Create a trigger to launch the function
CREATE TRIGGER trg_emp_comm
BEFORE INSERT OR UPDATE
ON invoice_tax FOR EACH ROW
EXECUTE PROCEDURE func_emp_comm_trg();

-- Test by performing an INSERT statement
INSERT INTO invoice_tax (invoice_id, customer_id,
                        invoice_total, invoice_total_tax)
VALUES(124, 345, 99.99, func_invoice_tax(99.99));

-- Verify data
SELECT * FROM invoice_tax;

invoice_id | customer_id | invoice_total | invoice_total_tax | emp_commission
-----+-----+-----+-----+-----+
  124 |      345 |     99.99 |      114.99 |     104.99
```

See the following for more information:

- [Oracle Create Table](#)
- [Oracle Virtual Columns](#)
- [PostgreSQL Functions](#)
- [PostgreSQL Views](#)
- [PostgreSQL Triggers](#)

#### PostgreSQL 11 updates:

None

#### PostgreSQL 12 updates:

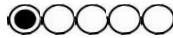
- [Generated Columns](#) - A special column that is always computed from other columns.

#### PostgreSQL 13 updates:

None

## Migrating from Oracle Index-Organized Tables

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

**Differences summary:**

- PostgreSQL does not support index-organized tables, but a partial workaround is possible.

Oracle index-organized tables (IOT) sort data based on a physical layer key, with all columns stored with the primary key. When all or most columns are required, this prevents unnecessary lookups in an index and can reduce the total disk space used by not having an additional index. The table data is stored in a B-tree index structure sorted by the primary key of each row.

### Oracle Index-Organized Table Creation

```
CREATE TABLE EMPLOYEES (
    EMP_ID      NUMBER,
    FIRST_NAME  VARCHAR2(50) NOT NULL,
    LAST_NAME   VARCHAR2(50) NOT NULL,
    HIRE_DATE   DATE NOT NULL,
    CONSTRAINT PK_EMP_ID PRIMARY KEY (EMP_ID)) ORGANIZATION INDEX;

INSERT INTO EMPLOYEES VALUES (65, 'Bob', 'West', '01-JAN-2020');
INSERT INTO EMPLOYEES VALUES (34, 'Eric', 'Valdez', '01-JAN-2020');
INSERT INTO EMPLOYEES VALUES (13, 'John', 'Smith', '01-JAN-2020');

COMMIT;

SELECT * FROM EMPLOYEES;
EMP_ID| FIRST_NAME | LAST_NAME|HIRE_DATE
-----|-----|-----|-----
13|John      |Smith     |2020-01-01 00:00:00|
34|Eric      |Valdez    |2020-01-01 00:00:00|
65|Bob       |West      |2020-01-01 00:00:00|
```



```
CREATE TABLE EMPS_NOT_IOT (
    EMP_ID NUMBER,
    EMP_FNAME VARCHAR2(50) NOT NULL,
    EMP_LNAME VARCHAR2(50) NOT NULL,
    START_DATE DATE NOT NULL,
    CONSTRAINT PK_EMPS_NOT_IOT_ID PRIMARY KEY (EMP_ID));

INSERT INTO EMPLOYEES VALUES (65, 'Bob', 'West', '01-JAN-2020');
INSERT INTO EMPLOYEES VALUES (34, 'Eric', 'Valdez', '01-JAN-2020');
INSERT INTO EMPLOYEES VALUES (13, 'John', 'Smith', '01-JAN-2020');

COMMIT;

SELECT * FROM EMPS_NOT_IOT;
EMP_ID| FIRST_NAME | LAST_NAME | HIRE_DATE
-----|-----|-----|-----
65|Bob|West|2020-01-01 00:00:00
34|Eric|Valdez|2020-01-01 00:00:00
13|John|Smith|2020-01-01 00:00:00
```

**Results:** The records are sorted in the reverse order from which they were inserted.

#### Migrate to Azure Database for PostgreSQL Cluster Table

Oracle IOT is not supported by PostgreSQL; however, its core functionality can be partially implemented by using PostgreSQL CLUSTER statement.

The CLUSTER command specifies table sorting based on an index already associated with the table. When you use the CLUSTER command, the data in the table is physically sorted based on the index.

Oracle IOT is defined during table creation and persists data sorting (the IOT always remains 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. To apply sorting, use the PostgreSQL CLUSTER statement as needed to re-cluster the table.

#### PostgreSQL CLUSTER Table Example

Create a new table with a simple primary key while inserting unsorted values, and issue the PostgreSQL CLUSTER statement to sort the unsorted values:

```
CREATE TABLE SYS_ERR_MSG (
    MSG_ID      NUMERIC,
    MSG_CODE     VARCHAR(10) NOT NULL,
    MSG_DESCRIPTION VARCHAR(200),
    MSG_TIME     DATE NOT NULL,
    CONSTRAINT PK_MSG_ID PRIMARY KEY(MSG_ID));

INSERT INTO SYS_ERR_MSG VALUES(7, '3b-45b-b2', 'Critical', '01-MAR-2020');
INSERT INTO SYS_ERR_MSG VALUES(3, '94-9c3-07', 'Warning', '01-MAR-2020');
INSERT INTO SYS_ERR_MSG VALUES(4, 'a4-64a-01', 'Info', '01-MAR-2020');

CLUSTER SYS_ERR_MSG USING PK_MSG_ID;

SELECT * FROM SYSTEM_EVENTS;
msg_id | msg_code | msg_description | msg_time
-----+-----+-----+-----
 3 | 94-9c3-07 | Warning | 01-MAR-2020
 4 | a4-64a-01 | Info | 01-MAR-2020
 7 | 3b-45b-b2 | Critical | 01-MAR-2020

INSERT INTO SYS_ERR_MSG VALUES(1, 'd4-c67-8c', 'Critical', '01-MAR-2020');

SELECT * FROM SYSTEM_EVENTS;
msg_id | msg_code | msg_description | msg_time
-----+-----+-----+-----
 3 | 94-9c3-07 | Warning | 01-MAR-2020
 4 | a4-64a-01 | Info | 01-MAR-2020
 7 | 3b-45b-b2 | Critical | 01-MAR-2020
 1 | d4-c67-8c | Critical | 01-MAR-2020

CLUSTER SYS_ERR_MSG USING PK_MSG_ID;
```



```
SELECT * FROM SYSTEM_EVENTS;

msg_id | msg_code    | msg_descipton | msg_time
-----+-----+-----+
 1 | d4-c67-8c | Critical     | 01-MAR-2020
 3 | 94-9c3-07 | Warning      | 01-MAR-2020
 4 | a4-64a-01 | Info         | 01-MAR-2020
 7 | 3b-45b-b2 | Critical     | 01-MAR-2020
```

See the following for more information:

- Oracle - [Managing Index-Organized Tables](#)
- Oracle - [Creating Index-Organized Tables](#)
- PostgreSQL - [CLUSTER Syntax](#)

## Oracle Constraints Conversion to PostgreSQL Constraints

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Constraints with REF/ENABLE/DISABLE options are not supported.

Oracle database constraints are used to enforce data integrity. Constraints can be specified at TABLE/VIEW creation at a column level (CREATE TABLE/CREATE VIEW) or post-creation (e.g. ALTER TABLE/ALTER VIEW). Oracle supports several types of constraints. Oracle views can only have a primary key, foreign key, and unique constraints. Creating constraints requires privileges on the table, and to create foreign key constraints, the REFERENCES privilege on the referenced table.

### Oracle Data Integrity Constraint Types

Constraint Type	Constraint Description
PRIMARY KEY	Enforce value uniqueness and column not-null values
FOREIGN KEY	Enforces that current table values correlate to the predefined referenced table values, using the parent-child relationship
CHECK	Validates values compliance according to predefined conditions
UNIQUE	Prevents column duplicate data, allowing only multiple null values
NOT NULL	Prevents a column from having null values
REF	References an object in a relational table or another object type; not supported in PostgreSQL.

### PRIMARY KEY Constraint

- **Description**
  - An RDBMS table can only have one primary key per table/view.
  - Each database table record has a unique identifier, so it can appear only once and cannot contain NULL values.
  - Primary keys can be created for a single table column or a collection of columns ; also known as a composite primary key.
  - When a new primary key constraint is created, a unique index is created on the primary key column (when no index already exists).
- **Limitations**
  - Primary keys cannot be created on columns with the following data types:
    - LOB
    - LONG
    - LONG RAW
    - VARRAY



- NESTED TABLE
  - BFILE
  - REF
  - TIMESTAMP WITH TIME ZONE
- A primary key-defined column is restricted from being defined as a unique constraint.
- Constraint Syntax Example
    - Set the primary key in-line at table creation. The primary key constraint name is generated by the system.

```
CREATE TABLE TBL (
    COL1 NUMBER PRIMARY KEY
);
```

- Set the primary key out-of-line at table creation and specify the primary key constraint name (PK\_COL1).

```
CREATE TABLE TBL (
    COL1 NUMBER,
    COL2 VARCHAR2(10),
    CONSTRAINT PK_COL1 PRIMARY KEY(COL1)
);
```

- Set the primary key constraint on an existing table, and set a composite primary key constructed by more than one column

```
ALTER TABLE TBL
    ADD CONSTRAINT PK_COL1 PRIMARY KEY(COL1, COL2);
```

## FOREIGN KEY Constraint

- Description
  - Foreign key constraints correlate their column records (defined with a foreign key constraint) against a referenced primary key or a unique column.
  - This enforces consistent values in table A and table B.
  - The referenced table is the parent table, and the foreign key table is the child table. Foreign keys created in child tables generally correlate up to a parent table with the same constraint.
- Oracle Foreign Key Clauses
  - ON DELETE Clause
    - The foreign key ON DELETE clause allows deletion of parent table values on the referenced child table records.



- If not specified, Oracle does not allow deletion of parent table key values in the affiliate child table.
- ON DELETE CASCADE - Both dependent child table foreign key values and original parent table values are removed.
- ON DELETE NULL - Dependent foreign key values in a child table are changed to NULL values.
- Constraint Limitations
  - Foreign keys cannot be created on the following data type columns: LOB, LONG, LONG RAW, VARRAY, NESTED TABLE, BFILE, REF, TIMESTAMP WITH TIME ZONE.
  - Composite (or multi-column) foreign key constraints are limited to 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.
- Constraint Syntax Example
  - Set a foreign key at table creation with inline reference. The constraint name is generated by the system.

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMBER PRIMARY KEY,
    LOG_NAME    VARCHAR2(20),
    LOG_DATE    DATE,
    LOG_EVENT   REFERENCES EVENTS(EVENT_ID)
);
```

- Set a foreign key at table creation with out-of-line reference. The constraint name is generated by the user (FK\_EVENT\_ID).

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMBER PRIMARY KEY,
    LOG_NAME    VARCHAR2(20),
    LOG_DATE    DATE,
    LOG_EVENT   INT,
    CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)
        REFERENCES EVENTS(EVENT_ID)
);
```

- Specify the ON DELETE CASCADE clause:

```
CREATE TABLE LOG_EVENTS (
```



```
LOG_ID    NUMBER PRIMARY KEY,  
LOG_NAME   VARCHAR2(20),  
LOG_DATE   DATE,  
LOG_EVENT  INT,  
CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)  
    REFERENCES EVENTS(EVENT_ID) ON DELETE CASCADE  
);
```

- Add a foreign key for an existing table:

```
ALTER TABLE LOG_EVENTS  
    ADD CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)  
        REFERENCES EVENTS(EVENT_ID);
```

## UNIQUE Constraint

- Description
  - Specifies that the values in a single column, or combination of columns, must be unique and cannot repeat in multiple rows, much like primary key constraints.
  - Columns defined with unique constraints can contain only multiple null values.
- Constraint 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 multi-column unique constraint can be up to 32 columns.
  - One-column or multi-column configurations prevent both primary key and unique constraints in the same column(s).
- Constraint Syntax Example
  - Set a unique constraint at table creation with inline reference. The constraint name is generated by the user.

```
CREATE TABLE LOG_EVENTS (  
    LOG_ID    NUMBER PRIMARY KEY,  
    LOG_NAME   VARCHAR2(20),
```



```
LOG_DATE DATE CONSTRAINT UNIQ_LOG_DATE UNIQUE,  
LOG_EVENT INT  
);
```

### CHECK Constraint

- Description
  - Used to validate data-entry criteria or conditions.
  - For example, a check constraint on an LOG\_NAME column can enforce that entries must include a defined prefix (for example, event\_<SequentialNum>.log).
  - If a record fails the check constraint validation, an error is raised, and the record is not inserted.
  - Check constraints help by using more of the database instead of the application for logical integrity validation.
- Constraint limitations
  - Cannot perform external table column validation.
  - Cannot be used with locked down, predefined functions like CURRENT\_DATE.
  - Cannot be combined with user-defined functions.
  - Cannot be used with pseudo columns such as CURRVAL, NEXTVAL, LEVEL, or ROWNUM.
- Constraint syntax example
  - Table DDL holding an inline check constraint that uses a regular expression to validate the log name prefix.

```
CREATE TABLE LOG_EVENTS (  
    LOG_ID      NUMBER PRIMARY KEY,  
    LOG_NAME    VARCHAR2(20)  
        CHECK(REGEXP_LIKE(LOG_NAME, 'event_[0-9]+.log')),  
    LOG_DATE    DATE,  
    LOG_EVENT   INT  
);
```

```
INSERT INTO LOG_EVENTS  
VALUES (1, 'event.log', sysdate, 1);
```

```
ORA-02290: check constraint (TST.SYS_C007125) violated
```

```
INSERT INTO LOG_EVENTS  
VALUES (1, 'event_123.log', sysdate, 1);
```



```
1 row created.
```

## NOT NULL Constraint

- Description
  - Prevents null values in a column.
  - The NOT NULL keyword must be inline or specified during table creation. Otherwise, the system defaults to allowing null values.
- Constraint Syntax Example
  - Set table columns with the not null constraint:

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMBER PRIMARY KEY,
    LOG_NAME    VARCHAR2(20) NOT NULL,
    LOG_DATE    DATE CONSTRAINT UNIQ_LOG_DATE UNIQUE NOT NULL,
    LOG_EVENT   INT NOT NULL
);
```

## REF Constraint

- Description
  - Define behaviors between REF columns and the objects they reference and can be created both inline and out-of-line.
  - Both methods permit defining a scope constraint, row ID constraint, or a referential integrity constraint based on the REF column.
- Constraint syntax example
  - Create a new Oracle type object:

```
CREATE TYPE EVENTS_TYPE AS OBJECT (
    EVENT_SEVIRITY VARCHAR2(30),
    EVENT_NAME      VARCHAR2(30)
);
```

- Create a table based on the previously created type object:

```
CREATE TABLE EVENTS OF EVENTS_TYPE;
```

- Create a table with a reference with one of the columns based on the type object:

```
CREATE TABLE EVENTS (
```



```
EVENT_ID INT PRIMARY KEY,  
EVENT_TYPE REF EVENTS_TYPE REFERENCES EVENTS  
);
```

### Oracle Constraints Enhancements

Oracle provides extensive control of database constraints, including temporarily disabling constraints while making table data modifications. Constraint states can be defined using the CREATE TABLE and ALTER TABLE statements.

#### Oracle constraints control options:

- 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 - (Default) Checks the constraint at the end of each subsequent SQL statement.
- 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 post-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.
- Primary keys can be created using an existing unique index rather than creating a new index by using the USING INDEX clause:

```
CREATE UNIQUE INDEX IDX_EVENT_ID ON EVENTS(EVENT_ID);  
  
ALTER TABLE EVENTS  
ADD CONSTRAINT PK_EVENT_ID  
PRIMARY KEY(EVENT_ID) USING INDEX IDX_EVENT_ID;
```

Oracle constraints metadata:

```
SELECT * FROM DBA_CONSTRAINTS;
```

### PostgreSQL Constraints

Like Oracle, PostgreSQL supports several constraint types to ensure database data integrity. PostgreSQL supports all Oracle constraints except REF, and provides an additional constraint named [EXCLUSION](#), which is not supported by Oracle. PostgreSQL users must have adequate table permissions to create constraints and must specifically



have the REFERENCES privilege for foreign key constraints. Like in Oracle, you can specify PostgreSQL constraint names, or they can be generated by the database system.

PostgreSQL supports the following types of table constraints:

- PRIMARY KEY
- FOREIGN KEY
- UNIQUE
- NOT NULL
- EXCLUSION

### **PRIMARY KEY Constraint**

- Constraint Syntax Examples
  - The primary key constraint name is generated by the system:

```
CREATE TABLE TBL (
    COL1 NUMERIC PRIMARY KEY
);

SELECT table_name, column_name, constraint_name
    FROM information_schema.constraint_column_usage
    WHERE table_name = 'tbl';

table_name | column_name | constraint_name
-----+-----+-----
tbl      | col1       | tbl_pkey
(1 row)
```

- Create a primary key constraint inline; the name is generated by the user:

```
CREATE TABLE TBL (
    COL1 NUMERIC CONSTRAINT PK_COL1 PRIMARY KEY,
    COL2 VARCHAR(10),
);
```

- Create a primary key constraint out-of-line; the name is generated by the user:

```
CREATE TABLE TBL (
```



```
COL1 NUMERIC,  
COL2 VARCHAR(10),  
CONSTRAINT PK_COL1 PRIMARY KEY(COL1)  
);
```

- Add a primary key constraint to an existing table:

```
ALTER TABLE TBL  
ADD CONSTRAINT PK_COL1 PRIMARY KEY(COL1, COL2);
```

- Drop the table primary key constraint:

```
ALTER TABLE TBL DROP CONSTRAINT PK_COL1;
```

## FOREIGN KEY Constraint

- Description
  - Foreign key constraints correlate their column records (defined with a foreign key constraint) against a referenced primary key or a unique column. This enforces consistent values in table A and table B.
  - Foreign keys use the same ANSI SQL syntax as Oracle and can be created inline or out-of-line.
  - The REFERENCES clause references the table cited by the foreign key constraint.
  - Where a column list is missing in the referenced table, the PRIMARY KEY of the referenced table is used as the referenced column or columns.
  - Tables can have multiple FOREIGN KEY constraints to describe its relationships with other tables.
  - ON DELETE clauses address FOREIGN KEY parent record deletions (including cascading deletes).
  - The database allows both manual and automatic foreign key constraint names.
- FOREIGN KEY clauses - PostgreSQL provides three main options to process data deletions from parent and child tables with the FOREIGN KEY constraints:
  - ON DELETE CASCADE - Any dependent foreign key values in both child and parent table are removed.
  - ON DELETE RESTRICT - Prevents deleting both parent (primary) and child (dependent) table values.
  - ON DELETE NO ACTION - The default option. System takes no action. NO ACTION is similar to the RESTRICT option but allows the check to be postponed until later in the transaction.
  - ON UPDATE
    - Oracle does not support the ON UPDATE clause.
    - The ON UPDATE clause can be used to handle FOREIGN KEY column updates, sharing the same options as the ON DELETE clause:
      - ON UPDATE CASCADE
      - ON UPDATE RESTRICT
      - ON UPDATE NO ACTION
- Constraint Syntax Examples



- Inline foreign key with constraint name set by the user (FK\_EVENT\_ID). Note that PostgreSQL foreign key columns must have a specified data type:

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMERIC PRIMARY KEY,
    LOG_NAME    VARCHAR(20),
    LOG_DATE    DATE,
    LOG_EVENT   INT REFERENCES EVENTS(EVENT_ID)
);
```

- Out-of-line foreign key with constraint name set by the user (FK\_EVENT\_ID):

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMERIC PRIMARY KEY,
    LOG_NAME    VARCHAR(20),
    LOG_DATE    DATE,
    LOG_EVENT   INT,
    CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)
        REFERENCES EVENTS(EVENT_ID)
);
```

- Foreign key using the ON DELETE CASCADE clause:

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMERIC PRIMARY KEY,
    LOG_NAME    VARCHAR(20),
    LOG_DATE    DATE,
    LOG_EVENT   INT,
    CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)
        REFERENCES EVENTS(EVENT_ID)
        ON DELETE CASCADE
);
```



- Add a foreign key for an existing table:

```
ALTER TABLE LOG_EVENTS
    ADD CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)
        REFERENCES EVENTS(EVENT_ID);
```

## UNIQUE Constraint

- Description
  - Ensures column values remain unique across the entire table.
  - Automatically creates a B-Tree index on the respective columns.
  - Generates an error message when duplicate values exist in the columns on which the constraint was defined.
  - Accepts multiple NULL values (same as Oracle unique constraint behavior).
  - Constraint name can be system generated or created manually.
  - Uses the same ANSI SQL syntax as Oracle.
- Constraint Syntax Examples
  - Create an inline constraint ensuring uniqueness of values for the LOG\_DATE column:

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMERIC PRIMARY KEY,
    LOG_NAME    VARCHAR(20),
    LOG_DATE    DATE CONSTRAINT UNIQ_LOG_DATE UNIQUE,
    LOG_EVENT   INT
);
```

## CHECK Constraint

- Description
  - Enforce that column values meet set requirements.
  - Restricted to Boolean data type for evaluating column values.
  - Constraint name can be system generated or manually created.
  - Uses the same ANSI SQL syntax as Oracle.
- Constraint syntax examples
  - Create an inline CHECK constraint, using a regular expression:

```
CREATE TABLE LOG_EVENTS (
    LOG_ID      NUMERIC PRIMARY KEY,
    LOG_NAME    VARCHAR(20)
        CHECK(LOG_NAME ~ 'event_[0-9]+.log'),
```



```
LOG_DATE DATE,  
LOG_EVENT INT  
);  
  
INSERT INTO LOG_EVENTS  
VALUES (1, 'event.log', oracle.sysdate(), 1);  
  
ERROR: new row for relation "log_events" violates check constraint  
"log_events_log_name_check"  
  
INSERT INTO LOG_EVENTS  
VALUES (1, 'event_123.log', oracle.sysdate(), 1);  
  
INSERT Ø 1
```

## NOT NULL Constraints

- Description
  - Prohibits columns from generating NULL values (otherwise allowed by default).
  - Defined exclusively inline during table creation (same as Oracle).
  - Constraint name can be system generated or manually created.
  - Uses the same ANSI SQL syntax as Oracle.
- Constraint syntax example
  - Set table columns with the NOT NULL constraint:

```
CREATE TABLE LOG_EVENTS (  
    LOG_ID    NUMERIC PRIMARY KEY,  
    LOG_NAME  VARCHAR2(20) NOT NULL,  
    LOG_DATE  DATE CONSTRAINT UNIQ_LOG_DATE UNIQUE NOT NULL,  
    LOG_EVENT INT NOT NULL  
);
```

## PostgreSQL Constraints Control Options

PostgreSQL provides advanced constraint controls using the SET CONSTRAINTS clause and DEFERRABLE, NOT DEFERRABLE, and IMMEDIATE settings.

- SET CONSTRAINTS sets the behavior of constraint checking within the current transaction:



- DEFERRABLE - Constraints are not checked until transaction commit.
- NOT DEFERRABLE - Runs immediately (IMMEDIATE) regardless of any SET CONSTRAINTS command.
- IMMEDIATE: - Constraints are enforced only at the end of each statement.
- Each constraint has its own IMMEDIATE or DEFERRED mode.
- VALIDATE CONSTRAINT validates a foreign key or checks THE constraint that was previously created as NOT VALID by scanning the table to ensure that there are no rows for which the constraint is not satisfied.
- NOT VALID - Allowed only for foreign key or check constraints, this clause prevents validating new records during constraint creation.

```
ALTER TABLE LOG_EVENTS
    ADD CONSTRAINT FK_EVENT_ID FOREIGN KEY(LOG_EVENT)
        REFERENCES EVENTS(EVENT_ID) NOT VALID;
```

As in Oracle, primary keys can be created using an existing unique index instead of creating a new index by using the USING INDEX clause.

```
CREATE UNIQUE INDEX IDX_EVENT_ID ON EVENTS(EVENT_ID);

ALTER TABLE EVENTS
    ADD CONSTRAINT PK_EVENT_ID
        PRIMARY KEY USING INDEX IDX_EVENT_ID;
```

#### PostgreSQL Constraints Metadata

```
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE;
or
SELECT * FROM PG_CONSTRAINT;
```

#### Oracle and PostgreSQL Constraints Comparison Table

Oracle Constraint	PostgreSQL Support	PostgreSQL Equivalent
Primary Key	Supported	Primary Key
Foreign Key	Supported	Foreign Key
Check	Supported	Check
Unique	Supported	Unique



Not Null	Supported	Not Null
REF	Not Supported	N/A
Constraint ENABLE	PostgreSQL default Not supported as keyword	N/A
Constraint DISABLE	Not supported as keyword	NOT VALID
Constraints on Views	Not Supported	N/A
ENABLE VALIDATE	PostgreSQL default Not supported as keyword	N/A
DISABLE VALIDATE	Not Supported	N/A
ENABLE NOVALIDATE	Not supported as keyword	NOT VALID
DISABLE NOVALIDATE	Not Supported	N/A

See the following for more information:

- [Oracle Constraint](#)
- [PostgreSQL Constraints](#)
- [PostgreSQL SET CONSTRAINTS](#)
- [PostgreSQL ALTER TABLE](#)

**PostgreSQL 11 updates:**

None

**PostgreSQL 12 updates:**

None

**PostgreSQL 13 updates:**

- Added ALTER [ COLUMN ] column\_name DROP EXPRESSION [ IF EXISTS ]
- When performing ALTER TABLE.. ADD COLUMN, existing rows are filled with the current time as the value of the new column, and then new rows receive the time of their insertion.

## Migrating from Oracle Table Partitions

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Foreign keys referencing to/from partitioned tables are supported on the individual tables in PostgreSQL.
- Some partition types are not supported by PostgreSQL.
  - Interval Partitioning
  - Virtual Columns Based Partitioning
  - Automatic List Partitioning
  - Partition Advisor

Oracle database partitioning supports efficient data handling of very large tables and indexes by logically splitting them into manageable partitions, or smaller portions of the table. Each partition has its own name and definitions. Partitions can be managed as standalone entities, or multiple partitions can be managed as a single object. At the application layer, partitions are “invisible”; this is, they have no impact on applications functionality, because partitioned tables, like non-partitioned tables, allow applications to access tables using unmodified SQL statements.

### Table Partitioning Benefits

- Data management - Partitions facilitate easier Information Lifecycle Management (ILM), including data migration, index creation/dropping/rebuilding, and backup/recovery.
- Performance improvements - Partitions improve query efficiency by accessing partition subsets instead of scanning a larger dataset. Additionally, partitions can be used with parallel query executions for DML and DDL operations.
- Maintenance - Partitions can significantly reduce downtime caused by table maintenance operations because maintenance can be performed on specific datasets.

### Oracle HASH Table Partitioning

Oracle HASH partitioning applies an even distribution of data between all the defined partitions in a table; this is particularly useful in cases when there is no certain range key.

You can set the HASH partition according to a column value or expression (such as a table column with a NUMBER data type). The number of partitions must be specified in the table DDL.

Oracle HASH partition DDL example:

```
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 5
STORE IN (tablespace1,
           Tablespace2,
           Tablespace3,
           Tablespace4);
```

### Oracle Range Table Partitioning

Oracle RANGE partitioning allows you to assign rows to partitions based on column values falling within a specified range. This means that each partition contains the rows for which the partitioning expression value lies within a given range.

Range table partitioning is one of the most frequently used types of partitioning. Dates ranges are often used in partitioning, but you can also use other value ranges.

Oracle RANGE partitioning DDL example:

```
CREATE TABLE ORDERS (
    ORDERID      NUMBER,
    ORDER_DATE   DATE,
    CUSTOMER_ID NUMBER,
    TOTAL        NUMBER
)
PARTITION BY RANGE(ORDER_DATE)
(
    PARTITION p0 VALUES LESS THAN (TO_DATE('01/01/2017',
                                             'DD/MM/YYYY')) TABLESPACE TS1,
    PARTITION p1 VALUES LESS THAN (TO_DATE('01/01/2018',
                                             'DD/MM/YYYY')) TABLESPACE TS2,
    PARTITION p2 VALUES LESS THAN (TO_DATE('01/01/2019',
                                             'DD/MM/YYYY')) TABLESPACE TS3,
    PARTITION p3 VALUES LESS THAN (TO_DATE('01/01/2020',
                                             'DD/MM/YYYY')) TABLESPACE TS4);
```



### Oracle List Table Partitioning

Using Oracle LIST partitioning allow you to specify a list of unique values for the table partitioning key defined by the user. The partitions are divided by specific values based on a predefined set of values.

This enables partition organizational control, using explicit values, like partitioning a table by state abbreviations values.

Oracle LIST partition DDL example:

```
CREATE TABLE CUSTOMERS (
    CUSTOMERID NUMBER PRIMARY KEY,
    BRANCHID  NUMBER
    STATE      VARCHAR(2)
    STATUS     VARCHAR2(20)
)
PARTITION BY LIST (STATE)
(
    PARTITION p_southwest    VALUES ('AZ', 'UT', 'NM'),
    PARTITION p_southeast    VALUES ('FL', 'GA'),
    PARTITION p_northeast    VALUES ('NY', 'VM', 'NJ'),
    PARTITION p_northwest    VALUES ('OR', 'WA'),
    PARTITION p_northcentral VALUES ('SD', 'WI'),
    PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

### Oracle Composite Partitioning

Oracle is compatible with combination partitioning. A table can be partitioned by one data partitioning method, and then each partition is further into sub-partitions using a second data distribution method. All sub-partitions for a given partition represent a logical subset of the data.

Oracle composite partitioning includes:

- Composite RANGE partitioning
  - Range-Range partitioning
  - Range-Hash partitioning
  - Range-List partitioning
- Composite LIST partitioning
  - List-List partitioning
  - List-Range partitioning
  - List-Hash partitioning

- Composite HASH partitioning
  - Hash-Hash partitioning
  - Hash-Range partitioning
  - Hash-List partitioning

### Oracle Partitioning Extensions

The following strategies enhance basic partitioning:

- Manageability extensions
  - Interval partitioning - An extension of range partitioning that instructs the database to automatically create partitions of a specified interval when data inserted into the table exceeds all of the existing range partitions.
  - Partition Advisor - Recommends a partitioning strategy for a table based, on a workload of SQL statements that can be supplied by the SQL Cache or a SQL Tuning set or can be defined by the user.
- Partitioning key extensions
  - Reference partitioning - Enables the partitioning of two tables that are related to one another by referential constraints.
  - Virtual column-based partitioning - A partitioning key can be defined by an expression using one or more existing columns of a table.

### Split Partitions

SPLIT PARTITION statements further separate the contents of an existing partition or sub-partition into multiple partitions or sub-partitions.

Oracle SPLIT PARTITIONS example:

```
ALTER TABLE PRODUCTS PARTITION p0 INTO
  (PARTITION p01 VALUES LESS THAN (30),
  PARTITION p02);
```

### Oracle Exchange Partitions

The EXCHANGE PARTITION statement enables swapping table partitions in or out of partitioned table.

Exchanging table partitions is useful to get data quickly in or out of a partitioned table. For example, in data warehousing environments, exchanging partitions facilitates high-speed data loading of new, incremental data into an existing partitioned table.

Oracle EXCHANGE PARTITION example:

```
ALTER TABLE SALES EXCHANGE
  PARTITION p_sale1 WITH TABLE sales_year_2019;
```

### Oracle Sub-Partitioning Tables

Partitions can be sub-partitioned to further split the table data.

Oracle SUBPARTITION example:

```
PARTITION BY RANGE(ORDER_DATE)
    SUBPARTITION BY HASH(STORE_ID)
    SUBPARTITION 6
        (PARTITION p1 VALUES LESS THAN (TO_DATE('01/01/2019',
            'DD/MM/YYYY')),
        PARTITION p1 VALUES LESS THAN (TO_DATE('01/01/2020',
            'DD/MM/YYYY')));
```

### Oracle Automatic List Partitioning

Introduced in Oracle 12c, Oracle provides automatic list partitioning, which enables automatic creation of new partitions for new values inserted into a list-partitioned table.

These are initially created with only one partition, but the database then auto-creates the additional table partitions.

Oracle automatic list partition example:

```
CREATE TABLE NETWORK_LOGS (
    LOG_ID    NUMBER NOT NULL,
    LOG_DATE  DATE NOT NULL,
    LOG_STR   VARCHAR2(500),
    LOG_MSG   VARCHAR2(10))
PARTITION BY LIST (LOG_MSG) AUTOMATIC
    (PARTITION log_msgs VALUES ('msg1', 'msg2', 'msg3'));
```

For more information, see [Oracle Partitioning](#).

### Migrating to Azure Database for PostgreSQL Partitions

Like Oracle, PostgreSQL provides table partitioning capabilities, but without some features and capabilities that Oracle partitions support.

Before PostgreSQL version 10, PostgreSQL partitioning was based only on the [table inheritance](#) concept, where each table partition was represented by a child table referenced up to a single parent table. That parent table remained empty and was only used to represent the entire table data set (for a metadata dictionary and query source).

PostgreSQL version 10 introduces a new partition architecture in the form of [declarative partitioning](#), supporting only the RANGE and the LIST partitions but with no overhead of creating additional triggers and functions to support the partitioning functionality.



Partition tables must be created manually, but do not require creating triggers or functions to redirect data to the right partition (as was required in previous versions). Additionally, some management operations are performed directly on the sub-partitions (sub-tables), and querying can be performed directly on the partitioned table itself.

PostgreSQL version 11 introduced support for HASH partitioning as well as support for PRIMARY KEY, FOREIGN KEY, INDEXES, and TRIGGERS on partitioned tables using the PostgreSQL declarative partitioning mechanism.

When dropping the main partitioned table using PostgreSQL version 11 declarative partitioning mechanism, all other partitioned tables and indexes are dropped as well.

Additional PostgreSQL partitions management can be achieved by using the pg\_partman (PG Partition Manager) extension for creating and managing both time-based and serial-based table partition sets. The pg\_partman extension is supported by Azure Database for PostgreSQL.

## PostgreSQL Declarative Partitioning Mechanism

This following section shows code examples for:

- LIST partitioning
- RANGE partitioning
- HASH partitioning
- Sub-Partitions
- Composite Partitions
- Partitions DDL/metadata

### PostgreSQL List Partition

The following example shows the creation of the main table specifying the partition type:

- Create partitioned tables holding the associate main table and partitions definitions.
- The default partition (added in PostgreSQL version 11) is used for values not defined by partition.
- Updating the default partition key moves the row to the correct partition table (not possible in previous versions).

```
CREATE TABLE customers (cust_id numeric NOT NULL,
    cust_name varchar(30) NOT NULL,
    cust_address text,
    cust_region text
)
PARTITION BY LIST(cust_region);

CREATE TABLE cust_eu PARTITION OF customers
    FOR VALUES IN ('europe');

CREATE TABLE cust_na PARTITION OF customers
    FOR VALUES IN ('united-states');
```

```
CREATE TABLE cust_gen PARTITION OF customers
    DEFAULT;

INSERT INTO customers VALUES (1234,'Company-A','Somewhere
                                St. CA','united-states');

INSERT INTO customers VALUES (2345,'Company-B','Somewhere
                                St. London','europe');

INSERT INTO customers VALUES (3456,'Company-C','Somewhere
                                St. unknown','unknown');

--Verify table records
SELECT * FROM customers;
```

cust_id	cust_name	cust_address	cust_region
1234	Company-A	Somewhere St. CA	united-states
2345	Company-B	Somewhere St. London	europe
3456	Company-C	Somewhere St. unknown	unknown

```
-- Querying the partitioned table directly is possible
SELECT * FROM cust_gen;



| cust_id | cust_name | cust_address          | cust_region |
|---------|-----------|-----------------------|-------------|
| 3456    | Company-C | Somewhere St. unknown | unknown     |



-- Updating the partition key values for the DEFAULT partition records
UPDATE customers
    SET~ cust_region = 'united-states'
```



```
WHERE cust_id = 3456;

-- Data verification
SELECT * FROM cust_gen;

cust_id | cust_name | cust_address | cust_region
-----+-----+-----+
(0 rows)
```

## PostgreSQL Range Partition

Create the main table specifying the partition type, and create partitioned tables holding the associated main table and partitions definitions.

```
CREATE TABLE orders (
    order_date DATE NOT NULL,
    item_id INT,
    item_price FLOAT DEFAULT 0
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2020q1 PARTITION OF orders (
    item_price DEFAULT 0)
    FOR VALUES FROM ('2020-01-01') TO ('2020-03-31');

CREATE TABLE orders_2020q2 PARTITION OF orders (
    item_price DEFAULT 0)
    FOR VALUES FROM ('2020-04-01') TO ('2020-06-30');

CREATE TABLE orders_2020q3 PARTITION OF orders (
    item_price DEFAULT 0)
    FOR VALUES FROM ('2020-07-01') TO ('2020-09-30');

-- Insert new records
INSERT INTO orders VALUES ('2020-01-01',1234, 299);
```



```
INSERT INTO orders VALUES ('2020-06-10',2345, 999);
```

```
INSERT INTO orders VALUES ('2020-09-29'),3456;
```

```
-- Data verification
```

```
SELECT * FROM orders;
```

```
order_date | item_id | item_price
```

order_date	item_id	item_price
2020-01-01	1234	299
2020-06-10	2345	999
2020-09-29	3456	0

```
(3 rows)
```

```
-- Query an individual partitioned table
```

```
SELECT * FROM orders_2020q1;
```

```
order_date | item_id | item_price
```

order_date	item_id	item_price
2020-01-01	1234	299

```
(1 row)
```

## PostgreSQL Hash Partition

Hash partitioning was introduced by PostgreSQL version 11 and solves data distribution issues when range or list partitions cannot be set.

The table is partitioned by specifying a modulus and a remainder for each partition. Each partition holds the rows for which the hash value of the partition key divided by the specified modulus produces the specified remainder.

The following example shows the creation of the main table specifying the partition type. Create partitioned tables holding the associate main table and partitions definitions.

```
CREATE TABLE orders_hash (
```



```
order_date DATE NOT NULL,
item_id INT,
item_price FLOAT DEFAULT 0
) PARTITION BY HASH (item_id);

CREATE TABLE p1 PARTITION OF orders_hash
FOR VALUES WITH (MODULUS 10, REMAINDER 0);

CREATE TABLE p2 PARTITION OF orders_hash
FOR VALUES WITH (MODULUS 10, REMAINDER 1);

CREATE TABLE p3 PARTITION OF orders_hash
FOR VALUES WITH (MODULUS 10, REMAINDER 2);

CREATE TABLE p4 PARTITION OF orders_hash
FOR VALUES WITH (MODULUS 10, REMAINDER 3);

CREATE TABLE p5 PARTITION OF orders_hash
FOR VALUES WITH (MODULUS 10, REMAINDER 4);

INSERT INTO orders_hash VALUES (('2020-01-01'),1234, 299);
INSERT INTO orders_hash VALUES (('2020-06-10'),2345, 999);
INSERT INTO orders_hash VALUES (('2020-09-29'),3456);

-- Data verification
SELECT * FROM orders_hash;

order_date | item_id | item_price
-----+-----+-----
2020-06-10 | 2345 | 999
2020-01-01 | 1234 | 299
```



```
2020-09-29 | 3456 | 0  
(3 rows)
```

## PostgreSQL Sub-Partitions

When creating a table with a PARTITION OF clause, you can use the PARTITION BY clause to create a sub-partition. A sub-partition can be the same type as the parent partition table or a different type.

```
CREATE TABLE orders (
    order_date DATE NOT NULL,
    item_id INT,
    item_price FLOAT DEFAULT 0
) PARTITION BY RANGE (order_date);

-- Create a partitioned table specifying the "PARTITION BY" clause
CREATE TABLE orders_2020q1 PARTITION OF orders
    FOR VALUES FROM ('2020-01-01') to ('2020-01-31')
    PARTITION BY RANGE(item_id);
```

## PostgreSQL Table and Partitions Metadata

(Some columns omitted)

```
-- Table & Range partitions metadata
\dt+ orders
          Partitioned table "public.orders"
  Column  |      Type       | Collation | Nullable | Default | Storage |
-----+-----+-----+-----+-----+-----+
order_date | date          |           | not null |         | plain   |
item_id    | integer        |           |          |         | plain   |
item_price | double precision |           |          | 0       | plain   |

Partition key: RANGE (order_date)
```



```
Partitions: orders_2020q1 FOR VALUES FROM ('2020-01-01') TO ('2020-03-31'),
            orders_2020q2 FOR VALUES FROM ('2020-04-01') TO ('2020-06-30'),
            orders_2020q3 FOR VALUES FROM ('2020-07-01') TO ('2020-09-30')

-- Table & Hash partitions metadata
\dt+ orders_hash
              Partitioned table "public.orders"
  Column   |      Type       | Collation | Nullable | Default | Storage |
-----+-----+-----+-----+-----+
order_date | date          |           | not null |         | plain    |
item_id    | integer        |           |           |         | plain    |
item_price | double precision |           |           | 0        | plain    |

Partition key: HASH (item_id)
Partitions: p1 FOR VALUES WITH (modulus 10, remainder 0),
            p2 FOR VALUES WITH (modulus 10, remainder 1),
            p3 FOR VALUES WITH (modulus 10, remainder 2),
            p4 FOR VALUES WITH (modulus 10, remainder 3),
            p5 FOR VALUES WITH (modulus 10, remainder 4)

-- The following SQL query can also be used for partitions details
SELECT * FROM pg_catalog.pg_class
  WHERE relkind = 'p';
```

## PostgreSQL Partitions Support for Foreign Keys

Starting with PostgreSQL 11, foreign keys in the partition table are supported, as demonstrated in this example:

```
-- Create FK child table
CREATE TABLE account_managers (
    account_man_id INT PRIMARY KEY
);
```

```
-- Create FK parent table specifying "partition by" Range
CREATE TABLE sales (
    sale_id      INT,
    account_man_id INT
        REFERENCES account_managers(account_man_id),
    sale_date     DATE NOT NULL,
    PRIMARY KEY(sale_id, sale_date))
PARTITION BY RANGE (sale_date);

\dt+ sales;
          Partitioned table "public.sales1"
 Column | Type   | Collation | Nullable | Default | Storage |
-----+-----+-----+-----+-----+
 sale_id | integer |           |          | plain   |
 account_man_id | integer |           |          | plain   |
 sale_date | date   |           | not null | plain   |

Partition key: RANGE (sale_date)
Foreign-key constraints:
  "sales1_account_man_id_fkey" FOREIGN KEY (account_man_id) REFERENCES
account_managers(account_man_id)
Number of partitions: 0
```

## PostgreSQL Partitions Index Support

Before PostgreSQL 11, you had to manually create an index for every partition child table.

Starting with PostgreSQL version 11, you can create an index for the master table, which then automatically creates the index with the same configuration for all existing and future child partitions.



```
CREATE INDEX idx_sales ON sales(sale_id);

SELECT tablename, indexname
  FROM pg_indexes
 WHERE tablename like '%sales%';

tablename | indexname
-----+-----
sales_2018 | sales_2018_sale_id_idx
sales_2019 | sales_2019_sale_id_idx
sales_2020 | sales_2020_sale_id_idx
(3 rows)
```

### PostgreSQL Composite or Multi-level Partitioning

Like Oracle, PostgreSQL supports composite partitioning, meaning that multiple partitions combinations can be applied.

The following Partition methods can be created in PostgreSQL declarative partitioning:

- List Partitioning
  - LIST-LIST
  - LIST-RANGE
  - LIST-HASH
- Range Partitioning
  - RANGE-RANGE
  - RANGE-LIST
  - RANGE-HASH
- Hash Partitioning
  - HASH-HASH
  - HASH-LIST
  - HASH-RANGE

```
CREATE TABLE order_status (
    order_id      INT,
    order_status  text,
    order_date    DATE)
PARTITION BY RANGE(order_date);
```



```
-- Create composite partition of Range-List
CREATE TABLE order_year_2019 PARTITION OF order_status
    FOR VALUES FROM ('2020-01-01') TO ('2020-12-31')
    PARTITION BY LIST(order_status);

-- Create composite partition of Range-List values definition
CREATE TABLE order_2019_stat PARTITION OF order_year_2019
    FOR VALUES IN('ACCEPTED','ACTIVE', 'CANCELLED');

-- Insert partition value which does not exists in the partition List
INSERT INTO order_status
    VALUES (1234, 'UNKNOWN', '2020-01-01');

ERROR: no partition of relation "order_year_2019" found for row
DETAIL: Partition key of the failing row contains(order_status)=(UNKNOWN).

-- Insert a valid partition List value
INSERT INTO order_status
    VALUES (1234, 'ACCEPTED', '2020-01-01');

-- Data verification
SELECT * FROM order_status;

order_id | order_status | order_date
-----+-----+-----
 1234 | ACCEPTED | 2020-01-01
(1 rows)

-- Partition metadata for the main table
\dt+ order_status
```

```

Partitioned table "public.order_status"
Column   | Type    | Collation | Nullable | Default | Storage |
-----+-----+-----+-----+-----+
order_id | integer |           |          |          | plain    |
order_status | text |           |          |          | extended |
order_date | date |           |          |          | plain    |

Partition key: RANGE (order_date)

Partitions: order_year_2019 FOR VALUES FROM ('2020-01-01') TO ('2020-12-31'),
PARTITIONED

-- Partition metadata for the Range-List table
\dt+ order_2019_stat

Table "public.order_2019_stat"
Column   | Type    | Collation | Nullable | Default | Storage |
-----+-----+-----+-----+-----+
order_id | integer |           |          |          | plain    |
order_status | text |           |          |          | extended |
order_date | date |           |          |          | plain    |

Partition of: order_year_2019 FOR VALUES IN ('ACCEPTED', 'ACTIVE', 'CANCELLED')
Partition constraint: ((order_date IS NOT NULL) AND (order_date >= '2020-01-01'::date) AND (order_date < '2020-12-31'::date) AND (order_status IS NOT NULL) AND (order_status = ANY (ARRAY['ACCEPTED'::text, 'ACTIVE'::text, 'CANCELLED'::text])))

```

## Implementing List Table Partitioning with PostgreSQL Inheritance Tables

PostgreSQL [table inheritance](#) is the old way of implementing partitions using PostgreSQL. It requires the creation of triggers and functions to support the partition overall logical implementation. In addition, this mechanism lacks many of PostgreSQL table partitioning features. The following are not supported:

- Hash partitions
- Foreign keys
- Sub-partitions
- SPLIT and EXCHANGE partition operations

PostgreSQL table inheritance example:

```
-- Create the partitions parent table
CREATE TABLE NETWORK_LOGS (
    MSG_NO NUMERIC NOT NULL,
    MSG_DATE      DATE NOT NULL,
    MSG_TXT       VARCHAR(500),
    MSG_ERR_CODE  VARCHAR(10)
);

-- Create child partitioned tables with check constraints
CREATE TABLE NETWORK_LOGS_INFO (
    CHECK (MSG_ERR_CODE IN('info1', 'info2', 'info3')))
    INHERITS (NETWORK_LOGS);

CREATE TABLE NETWORK_LOGS_WARNING(
    CHECK (MSG_ERR_CODE IN('warn1', 'warn2', 'warn3')))
    INHERITS (NETWORK_LOGS);

CREATE TABLE NETWORK_LOGS_ERROR(
    CHECK (MSG_ERR_CODE IN('err1', 'err2', 'err3')))
    INHERITS (NETWORK_LOGS);

-- Create indexes on each of the child tables
CREATE INDEX IDX_NETWORK_LOGS_INFO
```



```
ON NETWORK_LOGS(MSG_ERR_CODE);

CREATE INDEX IDX_NETWORK_LOGS_WARNING
    ON NETWORK_LOGS(MSG_ERR_CODE);

CREATE INDEX IDX_NETWORK_LOGS_ERROR
    ON NETWORK_LOGS(MSG_ERR_CODE);

-- Create a function to redirect data inserted into the parent table
CREATE OR REPLACE FUNCTION NETWORK_LOGS_MSG_CODE_INS()
RETURNS TRIGGER AS
$$
BEGIN
    IF (NEW.MSG_ERR_CODE IN('warn1', 'warn2', 'warn3')) THEN
        INSERT INTO NETWORK_LOGS_WARNING VALUES (NEW.*);

    ELSIF (NEW.MSG_ERR_CODE IN('info1', 'info2','info3')) THEN
        INSERT INTO NETWORK_LOGS_INFO VALUES (NEW.*);

    ELSIF (NEW.MSG_ERR_CODE IN('err1', 'err2', 'err3')) THEN
        INSERT INTO NETWORK_LOGS_ERROR VALUES (NEW.*);

    ELSE
        RAISE EXCEPTION 'ERROR - Values are out of range';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

```
-- Create a trigger function (created above) to log to the table
CREATE TRIGGER NETWORK_LOGS_MSG_TRIG
    BEFORE INSERT ON NETWORK_LOGS
    FOR EACH ROW EXECUTE PROCEDURE NETWORK_LOGS_MSG_CODE_INS();

-- Insert data to the parent table
INSERT INTO NETWORK_LOGS
    VALUES('1234', '2020-04-01', 'Some text 1', 'info1');

INSERT INTO NETWORK_LOGS
    VALUES(2345, '2020-05-20', 'Some text 2', 'warn2');

INSERT INTO NETWORK_LOGS
    VALUES(3456, '2020-06-30', 'Some text 3', 'err3');

-- Data verification
SELECT * FROM NETWORK_LOGS;

msg_no | msg_date | msg_txt | msg_err_code
-----+-----+-----+-----
1234 | 2020-04-01 | Some text 1 | info1
2345 | 2020-05-20 | Some text 2 | warn2
3456 | 2020-06-30 | Some text 3 | err3

-- Add out of range value
INSERT INTO NETWORK_LOGS
    VALUES(3456, '2020-06-30', 'Some text 3', 'message');

ERROR: ERROR - Values are out of range
CONTEXT: PL/pgSQL function network_logs_msg_code_ins() line 13 at RAISE
```

```
-- Select data directly from a child table
SELECT * FROM NETWORK_LOGS_ERROR;

msg_no | msg_date | msg_txt | msg_err_code
-----+-----+-----+
 3456 | 2020-06-30 | Some text 3 | err3
(1 row)

-- Get parent table metadata and partitions details
\dt+ NETWORK_LOGS

          Table "public.network_logs"
   Column    |      Type      | Nullable | Default | Storage  |
-----+-----+-----+-----+-----+
 msg_no     | numeric       | not null |        | main     |
 msg_date    | date          | not null |        | plain    |
 msg_txt     | character varying(500) |        |        | extended |
 msg_err_code | character varying(10) |        |        | extended |

Indexes:
"idx_network_logs_error" btree (msg_err_code)
"idx_network_logs_info" btree (msg_err_code)
"idx_network_logs_warning" btree (msg_err_code)

Triggers:
 network_logs_msg_trig BEFORE INSERT ON network_logs FOR EACH ROW EXECUTE
PROCEDURE network_logs_msg_code_ins()

Child tables: network_logs_error,
               network_logs_info,
               network_logs_warning

-- Drop the parent table (will drop all child tables)
```



```
DROP TABLE NETWORK_LOGS CASCADE;

NOTICE: drop cascades to 3 other objects
DETAIL: drop cascades to table network_logs_info
drop cascades to table network_logs_warning
drop cascades to table network_logs_error
```

#### Oracle and PostgreSQL Partitions Comparison Summary

Partition Feature	Oracle	PostgreSQL
<b>Range Partition</b>	Yes	Yes
<b>List Partition</b>	Yes	Yes
<b>Hash partition</b>	Yes	Yes
<b>Composite Partitioning (Sub-Partitioning)</b>	Yes	Yes
<b>Interval Partitioning</b>	Yes	No
<b>Partition Advisor</b>	Yes	No
<b>Reference Partitioning</b>	Yes	Yes
<b>Virtual Column Based Partitioning</b>	Yes	No
<b>Automatic List Partitioning</b>	Yes	No
<b>Split / Exchange Partitions</b>	Yes	Yes, by using: ATTACH PARTITION DETACH PARTITION <a href="#">sub-commands</a>

## Migrating from Oracle Local and Global Partition Indexes

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

### Differences Summary:

- PostgreSQL does not support domain index.

Oracle databases use local and global Indexes for partitioned tables, with each index created on a partitioned table 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 using the LOCAL clause.
  - Because each index partition is independent, index maintenance operations are easier because they can be performed independently.
  - Local partitioned indexes are managed automatically by Oracle during table partition creation or deletion.
- Global partitioned index
  - Each global index contains keys from multiple table partitions in a single index partition, 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, like index management and maintenance.
  - For example, dropping a table partition causes the global index to become unusable without an index rebuild.

The following example creates a local and global index on a partitioned table:

```
CREATE INDEX IDX_NETWORK_LOGS_GLOB
  ON NETWOEKLOGS (MSG_DATE)
  GLOBAL PARTITION BY RANGE(MSG_DATE) (
    PARTITION MSG_DATE1 VALUES LESS THAN
      (TO_DATE('01/01/2019', 'DD/MM/YYYY')),
    PARTITION MSG_DATE2 VALUES LESS THAN
      (TO_DATE('01/01/2020', 'DD/MM/YYYY')),
    PARTITION MSG_DATE3 VALUES LESS THAN (MAXVALUE);
```



For more information, see [Oracle local and global indexes](#).

## Migrating to Azure Database for PostgreSQL Partitioned Indexes

The PostgreSQL table partitioning mechanism differs from Oracle in the following ways:

- There is no direct equivalent for Oracle local and global indexes.
- Creating an index on the parent table, similar to Oracle global index; no effect.
- PostgreSQL version 10 and earlier required you to manually create an index for every partition child table. Starting in PostgreSQL version 11, you can create indexes on the master table, which then automatically creates the index with the same configuration on all existing and future child partition tables.

```
CREATE TABLE customers (
    cust_id numeric NOT NULL,
    cust_name varchar(30) NOT NULL,
    cust_address text,
    cust_region text
)
PARTITION BY LIST(cust_region);

CREATE TABLE cust_eu PARTITION OF customers
    FOR VALUES IN ('europe');

CREATE TABLE cust_na PARTITION OF customers
    FOR VALUES IN ('united-states');

CREATE TABLE cust_gen PARTITION OF customers
    DEFAULT;

-- Add index on the main table
CREATE INDEX idx_cust_id ON customers(cust_id);

-- Verify index creation on all partitioned tables
SELECT tablename, indexname
FROM pg_indexes
WHERE tablename like '%cust%';
```

Tablename	indexname
<hr/>	
cust_na	cust_na_cust_id_idx
cust_eu	cust_eu_cust_id_idx
cust_gen	cust_gen_cust_id_idx
(3 rows)	

See the following for more information:

- [PostgreSQL Table Partitioning](#)
- [PostgreSQL Inheritance](#)

### PostgreSQL Partitions Features by Release

- PostgreSQL Version 11
  - Add support for partitioning by a hash key.
  - Add support for PRIMARY KEY, FOREIGN KEY, indexes, and triggers on partitioned tables.
  - Allow creation of a “default” partition for storing data that does not match any of the remaining partitions.
  - UPDATE statements that change a partition key column now cause affected rows to be moved to the appropriate partitions.
  - Improve SELECT performance through enhanced partition elimination strategies during query planning and execution.
  - Allow the creation of partitions based on hashing a key column.
  - Support indexes on partitioned tables. An “index” on a partitioned table is not a physical index across the whole partitioned table, but instead a template for automatically creating similar indexes on each partition of the table. If the partition key is part of the index’s column set, a partitioned index can be declared UNIQUE. It represents a valid uniqueness constraint across the whole partitioned table, even though each physical index only enforces uniqueness within its own partition.
  - The new command ALTER INDEX ATTACH PARTITION causes an existing index on a partition to be associated with a matching index template for its partitioned table. This provides flexibility in setting up a new partitioned index for an existing partitioned table.
  - Allow foreign keys on partitioned tables.
  - Allow FOR EACH ROW triggers on partitioned tables. Creation of a trigger on a partitioned table automatically creates triggers on all existing and future partitions. This also allows deferred unique constraints on partitioned tables.
  - Allow partitioned tables to have a default partition
  - UPDATE statements that change a partition key column now cause affected rows to be moved to the appropriate partitions.



- Allow aggregate functions on partitioned tables to be evaluated separately for each partition, subsequently merging the results. This feature is disabled by default but can be enabled by changing `enable_partitionwise_aggregate`.
- PostgreSQL Version 12:
  - Partitioning performance enhancements, including improved query performance on tables with thousands of partitions, improved insertion performance with `INSERT` and `COPY`, and the ability to execute `ALTER TABLE ATTACH PARTITION` without blocking queries.
  - Improve performance of many operations on partitioned tables
  - Allow tables with thousands of child partitions to be processed efficiently by operations that only affect a small number of partitions.
  - Allow foreign keys to reference partitioned tables.
  - Allow partition bounds to be any expression. Such expressions are evaluated at partitioned-table creation time. Previously, only simple constants were allowed as partition bounds.
  - `ALTER TABLE ATTACH PARTITION` is now performed with reduced locking requirements.
  - Add partition introspection functions. The new function `pg_partition_root()` returns the top-most parent of a partition tree, `pg_partition_ancestors()` reports all ancestors of a partition, and `pg_partition_tree()` displays information about partitions.
  - Include partitioned indexes in the system view `pg_indexes`.
  - Add `psql` command `\dP` to list partitioned tables and indexes.
  - Improve `psql \d` and `\z` display of partitioned tables.
  - Fix bugs that could cause `ALTER TABLE DETACH PARTITION` to leave behind incorrect dependency state, allowing subsequent operations to perform incorrectly, for example by not dropping a former partition child index when its table is dropped.
- PostgreSQL Version 13:
  - Allow pruning of partitions in more cases.
  - Allow partitionwise joins in more cases. For example, partitionwise joins can now occur between partitioned tables, even when their partition bounds do not match exactly.
  - Support row-level BEFORE triggers on partitioned tables. However, such a trigger is not allowed to change which partition is the destination.
  - Allow logical replication into partitioned tables on subscribers. Previously, subscribers could only receive rows into nonpartitioned tables.
  - Allow whole-row variables (that is, `table.*`) to be used in partitioning expressions

## Oracle Large Objects (LOBs) Conversion to PostgreSQL LOBs

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Oracle Secure File LOBs are not supported by PostgreSQL.
- Expect different LOB data-type mapping from Oracle to PostgreSQL

Oracle Large Objects (LOBs) are a set of datatypes designed to store data that does not have the logical structure to fit a database table. This data can be unstructured datasets (for example, binary files of photos, audio, and video) or semi-structured data (such as XML documents) not suitable for an RDBMS table. In addition, the data object may have a large footprint that cannot be broken into any suitable structure for RDBMS storage. Oracle LOBs can hold data up to between 8 and 128 terabytes, depending on the database configuration.

Oracle 11g introduced [Secure File LOBS](#) that provide more efficient storage, encryption and compression, and can be created by specifying the SECUREFILE keyword as part of the CREATE TABLE statement.

### Oracle LOB Types

Oracle provides internal and external LOB datatypes.

- Internal types:
  - BLOB (Binary Large Object) - Stores up to 8 terabytes of binary data inside the database.
  - CLOB (Character Large Object) - Stores up to 8 terabytes of character data inside the database.
  - NCLOB (National Character Large Object) - Stores up to 8 terabytes of national character data inside the database.
- External types:
  - BFILE (Binary File) - Stores a pointer to a large binary file stored outside the database in the file system of the host computer.

### Oracle LOB Example

- CLOB can be specified as a data type on table creation.
- The TO\_CLOB function is used to convert a string literal to a CLOB value.
- The EMPTY\_CLOB function in the second INSERT statement initializes a LOB locator that points to a CLOB value but does not fill the CLOB value with data.
- The query returns the length of each CLOB value.

```
CREATE TABLE book_review (
    book_id      NUMBER PRIMARY KEY,
    book_review  CLOB
);

INSERT INTO book_review
```

```
VALUES (123, to_clob('A very long book review...'));

INSERT INTO book_review VALUES (124, EMPTY_CLOB());

INSERT INTO book_review VALUES (125, NULL);

SELECT book_id,
       book_review,
       LENGTH(book_review) AS clob_text_length
  FROM book_review;

BOOK_ID BOOK REVIEW          CLOB_TEXT_LENGTH
-----
123   A very long book review...      26
124                           0
125
```

## PostgreSQL LOBs

PostgreSQL LOBs provide different data types to support large database objects. PostgreSQL has an equivalent feature to Oracle Secure File LOBS, but as an alternative solution you can utilize data encryption through [Azure Data Encryption](#) and optimize compression using the [TOAST](#) table [storage strategy](#).

### PostgreSQL LOB Types

- BYTEA
  - A variable-length binary string with a storage size of 1 or 4 bytes plus the actual binary string.
  - Very similar to a simple character strings like varchar and text, but strings specifically can be stored as octets of value zero and other "nonprintable" octets.
- TEXT
  - A variable with unlimited length.
  - A data type for storing strings with unlimited length.
  - Ora2Pg converts an Oracle CLOB to PostgreSQL TEXT.

### PostgreSQL LOB Examples

The following table conversion was performed using Ora2Pg:

```
CREATE TABLE book_review (
    book_id bigint NOT NULL,
    book_review TEXT);

INSERT INTO book_review
    VALUES (123, 'A very long book review...');
```

```

INSERT INTO book_review VALUES (124, '');
INSERT INTO book_review VALUES (125, NULL);

SELECT book_id,
       book_review,
       LENGTH(book_review) AS clob_text_length
  FROM book_review;

book_id |      book_review      | clob_text_length
-----+---------------------+-----
  123  | A very long book review... |        26
  124  |                         |         0
  125  |                         |
(3 rows)

```

Oracle Table created with multiple LOB columns (see each column data type) and converted into Azure Database for PostgreSQL using Ora2Pg:

```

-- Oracle LOBs
CREATE TABLE TBL(
    COL1 CLOB,
    CLO2 BLOB,
    COL3 NCLOB,
    COL4 BFILE,
    COL5 CLOB)
LOB(COL5) STORE AS SECUREFILE;

-- Ora2Pg LOB columns conversion and data type mapping
CREATE TABLE TBL (
    col1 text,
    clo2 bytea,
    col3 text,
    col4 bytea,
    col5 text);

```

See the following for additional information:

- [Oracle LOBs](#)
- [PostgreSQL LOBs](#)

#### **PostgreSQL 11 updates:**

None

#### **PostgreSQL 12 updates:**

None

#### **PostgreSQL 13 updates:**

None



## Oracle Database Links Conversion into PostgreSQL DB Link and Foreign-Data Wrapper

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

### Differences Summary:

- Expect different implementation methods and different syntax.

Oracle database links (also known as Oracle DBLinks) enable you to establish a connection from a local database to a remote database server, thereby accessing data stored on different servers or database schemas. The remote server can be an Oracle database, or by using Oracle [Heterogeneous Services](#), you can connect to different types of databases, including Microsoft MSSQL, MySQL, PostgreSQL, and others. You can use Oracle database links to perform remote queries on tables/views and perform DML operations. To connect remotely using database links, you need credentials including remote hostname and port (if not set to Oracle default 1521), username, password, and the instance name. Credentials can be stored as connection details in the database link or in the TNSNAMES file.

### Oracle Database Link Examples

Create a new database link using the TNSNAMES alias:

```
CREATE DATABASE LINK <DBLinkName>
  CONNECT TO <UserName> IDENTIFIED BY <Password>
  USING '<TNSNAMES_ALIAS>';
```

Create a new database link using full connection string:

```
CREATE DATABASE LINK <DBLinkName>
  CONNECT TO <UserName> IDENTIFIED BY <Password>
  USING '(DESCRIPTION=(ADDRESS_LIST=(ADDRESS = (PROTOCOL =
    TCP)(HOST = <HostName>)(PORT = 1521)))
    (CONNECT_DATA =(SERVICE_NAME = <ServiceName>)))';
```

SELECT statement with a database link using "@":

```
SELECT * FROM <TableName>@<DBLinkName>;
```

Performing an UPDATE using a database link using "@":



```
UPDATE <TableName>@<DBLinkName>
SET <ColumnName> = 3000
WHERE <ColumnName> = 100;
```

### PostgreSQL DB Link and Foreign-Data Wrapper

You can establish a remote connection between a local PostgreSQL database and remote PostgreSQL database servers by using DB Link and by using the foreign-data wrapper `postgres_fdw`. Both types are applied as an extension using the `CREATE EXTENSION` statement.

#### PostgreSQL Remote Access Types

- DB Link provides a set of different functions to perform different tasks on a remote database server.
  - PostgreSQL extension installation:

```
CREATE EXTENSION dblink;
CREATE EXTENSION
```

- DBLINK syntax to verify the connection to a remote PostgreSQL database server:

```
SELECT dblink_connect
      (ConnName, 'dbname=<DB_Name> port=5432 host=<HostName>
User = <UserName> password = <Password>');

dblink_connect
-----
OK
(1 row)
```

- DBLINK syntax to run an SQL query to a remote PostgreSQL database server:

```
SELECT * FROM
dblink ('dbname = postgres port = 5432 host = <HostName>
user = <UserName> password = <Password>',
'SELECT 1, oracle.sysdate() from dual')
AS QueryTable(id INT, sysdate TIMESTAMP);
```

```
id |      sysdate
---+-----
 1 | 2020-10-15 15:42:02
(1 row)
```

- DBLINK performs DML operations on a remote table:

```
SELECT dblink_connect
      (ConnName, 'dbname=<DB_Name> port=5432 host=<HostName>
      User = <'UserName@AzureDBName'> password = <Password>');

dblink_connect
-----
OK
(1 row)

SELECT dblink_exec('ConnName',
                   INSERT INTO tbl VALUES (1);');

dblink_exec
-----
INSERT 0 1
(1 row)

SELECT * FROM tbl;

col1
-----
 1
(1 rows)
```

```
SELECT dblink_exec('ConnName',
                    UPDATE tbl SET col1 = col1 + 1;');

SELECT * FROM tbl;

col1
-----
 2
(1 rows)
```

- Close an open DBLINK connection:

```
SELECT dblink_disconnect('ConnName');

dblink_disconnect
-----
OK
(1 row)
```

- Foreign-data wrapper is a newer tool for applying remote database queries and operations and is much closer to Oracle database links.
  - PostgreSQL extension installation:

```
CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

- Create a foreign-data wrapper remote connection by specifying the remote server and remote database, using name `fdw_conn`:

```
CREATE SERVER fdw_conn
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '<HostName>', dbname 'postgres', port '5432');
```



- Create a mapping between a database user and the foreign-data wrapper configured server (from the example above - fdw\_conn):

```
CREATE USER MAPPING FOR CURRENT_USER  
    SERVER fdw_conn  
    OPTIONS (user '<UserName@AzureDBName>', password  
        'password');
```

- Create a dedicated schema to hold all the mapped foreign objects:

```
CREATE SCHEMA remote_database;
```

- Map a remote table to the foreign data wrapper configurations:

```
CREATE FOREIGN TABLE remote_database.tbl (  
    col1 int)  
    SERVER fdw_conn  
    OPTIONS (schema_name 'public', table_name 'tbl');
```

- Perform SELECT statement and DML operations using the foreign data wrapper configurations:

```
SELECT * FROM remote_database.tbl;  
  
col1  
-----  
1  
(1 rows)  
  
UPDATE remote_database.tbl SET col1 = col1 + 1;  
SELECT * FROM remote_database.tbl;  
  
col1  
-----  
2
```



(1 rows)

See the following for more information:

- [Oracle Database Links](#)
- [PostgreSQL dblink](#)
- [PostgreSQL postgres\\_fdw](#)

**PostgreSQL 11 updates:**

None

**PostgreSQL 12 updates:**

None

**PostgreSQL 13 updates:**

- Foreign Data Wrapper
  - SSL-KEY and SSL-CET can appear in either or both connection and user mapping. If both are present, the user mapping setting overrides the connection setting.
  - Only superusers can create or modify user mappings with the sslcert or sslkey settings.
  - A superuser can override this check on a per-user-mapping basis by setting the user mapping option password\_required to 'false'; for example, ALTER USER MAPPING FOR some\_non\_superuser SERVER loopback\_nopw OPTIONS (ADD password\_required 'false').
  - To prevent unprivileged users from exploiting the authentication rights of the unix user that the Postgres server is running as to escalate to superuser rights. Only the superuser can set this option on a user mapping.
  - Be careful to ensure that this does not allow the mapped user the ability to connect as superuser to the mapped database per CVE-2007-3278 and CVE-2007-6601. Do not set password\_required=false on the public role. The mapped user can potentially use any client certificates--.pgpass, .pg\_service.conf, etc.-- in the unix home directory of the system user the postgres server runs as. They can also use any trust relationship granted by authentication modes, like peer or ident authentication.

## Migrating from Oracle Triggers

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Different paradigm and syntax
- System triggers are not supported by PostgreSQL

An Oracle trigger is a PL/SQL code block that holds defined logical behavior; when a specific event occurs, it is executed automatically. The event and the executed trigger can be associated with a database table, database view, database schema, or at a database level. Oracle triggers support the BEFORE and AFTER statements and can be DISABLED and ENABLED.

Oracle supports the following event types to launch a trigger:

- DML (Data Manipulation Language) operations such as INSERT, UPDATE or DELETE.
- DDL (Data Definition Language) operations such as CREATE, ALTER, or DROP.
- Database level events such as:
  - User logon
  - User logoff
  - Database startup
  - Database shutdown
  - Server-level error

Oracle provides the following triggers types:

- DML triggers - FOR EACH ROW
  - Launched once for each row affected by a DML triggering event, such as INSERT, UPDATE, or DELETE.
  - DML triggers can be useful for data-related activities such as data logging, auditing, and data validation.
  - FOR EACH ROW must be specified in the trigger PL/SQL code.

```
:OLD.column_name  
:NEW.column_name
```
- Statement-level triggers - Launched after a trigger event occurs on a table, regardless of how many rows are affected, and will be executed only once for each transaction.
- INSTEAD OF triggers
  - Allows you to automatically skip the original DML operation and execute an "instead of" DML operation as an alternative action.
  - The INSTEAD OF trigger supports VIEWS only and cannot be defined on TABLEs.
  - Is fired for each row of the view that gets modified.
- SYSTEM event triggers - Can be defined against database-level events; useful for logging or auditing.



## Oracle Triggers Examples

A DML trigger using the FOR EACH ROW functionality. Each DML modification on the EMPLOYEES table (specific columns) launches the trigger to update the JOB\_HISTORY table (as a logging table):

```
CREATE OR REPLACE TRIGGER EMP_JOB_HISTORY
  AFTER UPDATE OF JOB_ID, DEPARTMENT_ID ON EMPLOYEES
  FOR EACH ROW
  BEGIN
    INSERT INTO JOB_HISTORY(EMPLOYEE_ID, START_DATE,
                           END_DATE, JOB_ID, DEPARTMENT_ID)
    VALUES(:OLD.EMPLOYEE_ID, :OLD.HIRE_DATE, SYSDATE,
           :OLD.JOB_ID, :OLD.DEPARTMENT_ID);
  END;
/
-- Perform a DML operation to fire the trigger
UPDATE EMPLOYEES
  SET JOB_ID = 'AC_MGR'
 WHERE EMPLOYEE_ID = 120;
```

Verify the data in the JOB\_HISTORY table:

```
SELECT * FROM JOB_HISTORY WHERE EMPLOYEE_ID = 120;

EMPLOYEE_ID START_DATE   END_DATE     JOB_ID      DEPARTMENT_ID
-----  -----
120        18-JUL-04    01-JAN-20    ST_MAN      50
```

A trigger to prevent table dropping using the BEFORE statement:

```
CREATE OR REPLACE TRIGGER TBL_DROP_TRIG
  BEFORE DROP ON HR.SCHEMA
  BEGIN
```



```
RAISE_APPLICATION_ERROR (num => -20001,
    msg => 'Operation is not allowed - Drop Table');

END;
/

-- Perform a DROP TABLE operation
DROP TABLE HR.EMPLOYEES;

Error report -
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: Operation is not allowed - Drop Table
```

Oracle triggers metadata:

```
SELECT * FROM DBA_TRIGGERS;
```

For more information, see [Oracle Triggers](#).



## Migrating to PostgreSQL Triggers

PostgreSQL triggers have the same functionality as Oracle triggers: perform defined logic automatically, based on specified database event occurrences. PostgreSQL triggers carry only the trigger launch definition; the code logic is stored on a PostgreSQL function (using PL/pgSQL). This is different than Oracle triggers, which can hold both the trigger definition and the logical implementation of the trigger (using PL/SQL). Like Oracle, PostgreSQL uses the .NEW and .OLD keywords to reference old and new values inside the trigger functionality.

You can set PostgreSQL triggers for the following events types:

- BEFORE - Fires the PostgreSQL trigger before attempting a DML operation such as INSERT, UPDATE, or DELETE.
- AFTER - Fires the PostgreSQL trigger after the operation has completed.
- Instead of the operation - Performs an alternative operation to the original DML operation.

In addition, PostgreSQL supports the following statements:

- FOR EACH ROW - The trigger is called once for each row modified by a DML operation.
- FOR EACH STATEMENT - The trigger is executed only once for any performed operation. Triggers can be defined to fire for TRUNCATE, but only FOR EACH STATEMENT.

## PostgreSQL CREATE TRIGGER Syntax

```
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 { FUNCTION | PROCEDURE } function_name ( arguments )
```

where event can be one of:

```
INSERT

UPDATE [ OF column_name [, ... ] ]

DELETE

TRUNCATE
```

### PostgreSQL Triggers Support for Row-Level and Statement-Level Operations

Launched	Event	Row-Level	Statement-Level
<b>BEFORE</b>	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, foreign tables
	TRUNCATE	N/A	N/A
<b>AFTER</b>	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, foreign tables
	TRUNCATE	N/A	Tables
<b>INSTEAD OF</b>	INSERT/UPDATE/DELETE	Views	N/A
	TRUNCATE	N/A	N/A

### PostgreSQL DML Triggers Examples

The following conversion is performed using Ora2Pg. Note that Oracle trigger was converted into PostgreSQL trigger and function.

```

CREATE OR REPLACE FUNCTION trigger_fct_emp_job_history()
RETURNS trigger AS $BODY$
BEGIN
    INSERT INTO JOB_HISTORY(EMPLOYEE_ID, START_DATE, END_DATE,
                           JOB_ID, DEPARTMENT_ID)
    VALUES(OLD.EMPLOYEE_ID, OLD.HIRE_DATE, LOCALTIMESTAMP,
          OLD.JOB_ID, OLD.DEPARTMENT_ID);

    RETURN NEW;
END
$BODY$
LANGUAGE 'plpgsql' SECURITY DEFINER;

-- REVOKE ALL ON FUNCTION trigger_fct_emp_job_history() FROM PUBLIC;
CREATE TRIGGER emp_job_history
AFTER UPDATE ON public."Employees" FOR EACH ROW
EXECUTE PROCEDURE trigger_fct_emp_job_history();

-- Perform a DML operation to fire the trigger

```



```
UPDATE public."Employees"
    SET JOB_ID = 'AC_MGR'
  WHERE EMPLOYEE_ID = 120;
```

Verify the converted trigger functionality: inserting to the logging table (JOB\_HISTORY) after an UPDATE modification on the EMPLOYEES table:

```
SELECT * FROM public.JOB_HISTORY WHERE EMPLOYEE_ID = 120;

employee_id | start_date | end_date | job_id | department_id
-----+-----+-----+-----+
 120 | 2004-07-18 | 2020-01-01 | ST_MAN | 50
(1 row)
```

### PostgreSQL Event Triggers

PostgreSQL also provides an additional type of trigger called [event triggers](#), which support Oracle database/system level triggers. While PostgreSQL DML (row/statement level) triggers are attached to a single table and capture only DML events, event triggers are global to a particular database and can capture DDL events at the database level.

PostgreSQL event triggers supported events:

- `ddl_command_start` - Triggered before the execution of CREATE, ALTER, DROP, SECURITY LABEL, COMMENT, GRANT, REVOKE, or SELECT INTO operations.
- `ddl_command_end` - Triggered after the command completes and before the transaction commits.
- `sql_drop` - Triggered only for DROP DDL commands and before the `ddl_command_end` trigger.

#### Example: PostgreSQL DDL trigger to prevent table dropping using the BEFORE statement

Create PostgreSQL function to hold the PL/pgSQL logic:

```
CREATE OR REPLACE FUNCTION prevent_drop_tb1()
RETURNS EVENT_TRIGGER
AS $$

BEGIN

    RAISE EXCEPTION 'The % Command is not allowed, tg_tag';

END; $$

LANGUAGE PLPGSQL;
```



CREATE FUNCTION

Create the trigger:

```
CREATE EVENT TRIGGER trg_prevent_drop_tbl
    ON DDL_COMMAND_START
    WHEN TAG IN ('DROP TABLE')
    EXECUTE PROCEDURE prevent_drop_tbl();

-- Test the DDL trigger
drop table emp;

ERROR: The DROP TABLE Command is not allowed
CONTEXT: PL/pgSQL function prevent_drop_tbl() line 3 at RAISE
```

Oracle Triggers metadata:

```
SELECT * FROM information_schema.triggers;
```

For more information, see [PostgreSQL triggers](#).

#### Oracle and PostgreSQL Triggers Comparison

Trigger Operation Type	Oracle	PostgreSQL
Enable a trigger	ALTER TRIGGER <TriggerName> ENABLE;	ALTER TABLE <TableName> ENABLE TRIGGER <TriggerName>;
Disable a trigger	ALTER TRIGGER <TriggerName> DISABLE;	ALTER TABLE <TableName> DISABLE TRIGGER <TriggerName>;
Trigger Alteration	CREATE OR REPLACE TRIGGER...;	CREATE OR REPLACE TRIGGER...; CREATE OR REPLACE FUNCTION...;
Trigger Drop	DROP TRIGGER <TriggerName>;	DROP TRIGGER <TriggerName> ON <TableName>;



Addressing OLD and NEW values	:NEW.<ColumnName> :OLD.<ColumnName>	.NEW.<ColumnName> .OLD.<ColumnName>
Row-Level	... BEFORE UPDATE ON <TableName> FOR EACH ROW ...	... UPDATE ON <TableName> FOR EACH ROW EXECUTE PROCEDURE <FuncName>();
Statement-Level	... BEFORE UPDATE ON <TableName> ...	UPDATE ON <TableName> EXECUTE PROCEDURE <FuncName>();
DDL Level	... BEFORE DROP ON <SchemaName> ...	... ON ddl_command_start EXECUTE PROCEDURE <FuncName>;
Database/System Level	CREATE TRIGGER <TrigName> BEGIN ON DATABASE SHUTDOWN ...	N/A

#### **PostgreSQL 11 updates:**

The EXECUTE FUNCTION statement was added to call functions from within a PostgreSQL trigger. Previously (before PostgreSQL version 11), only the EXECUTE PROCEDURE statement was supported for executing functions within a PostgreSQL trigger.

#### **PostgreSQL 12 and PostgreSQL 13 updates:**

None.

## Oracle Common Data Types Conversion to PostgreSQL Common Data Types

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Some Oracle data types are not supported as-is by PostgreSQL. Alternative data types can be used.
- Some data types have low or medium compatibility. These require using specific data types on a case-by-case basis.
- Spatial data types usage might require installing the [PostGIS](#) extension, which is [supported](#) by Azure Database for PostgreSQL.

Oracle primitive and proprietary data types and their corresponding PostgreSQL equivalents or alternatives are described in the following tables.

### Oracle Date and Time Data Types

Data Type	Data Type Specification	PostgreSQL Support (No Conversion)	PostgreSQL Data Type/Alternative Data Type
DATE	DATE data type stores date and time data (year, month, day, hour, minute and second)	Yes	TIMESTAMP(0)
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)

### Oracle Numerical Data Types

Data Type	Data Type Specification	PostgreSQL Support (No Conversion)	PostgreSQL Data Type/Alternative Data Type



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)
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

#### Oracle Character Data Types

Data Type	Data Type Specification	PostgreSQL Support (No Conversion)	PostgreSQL Data Type/ Alternative Data Type
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) Oracle 11g	Maximum size of 4000 bytes maximum size of 32 KB in PL/SQL	No	VARCHAR(n)
VARCHAR2(n) Oracle 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
LONG RAW	Maximum size of 2GB	No	BYTEA

#### Oracle Large Object Data Types

Data Type	Data Type Specification	PostgreSQL Support (No Conversion)	PostgreSQL Data Type/ Alternative Data Type
BFILE	Pointer to binary file maximum file size 4 GB	No	VARCHAR (255) / CHARACTER VARYING (255)
BLOB	Binary large object maximum file size of 4 GB	No	BYTEA
CLOB	Character large object maximum file size of 4 GB	No	TEXT
NCLOB	Variable-length Unicode string maximum file size of 4 GB	No	TEXT

#### Oracle Various Data Types

Data Type	Data Type Specification	PostgreSQL Support (No Conversion)	PostgreSQL Data Type/ Alternative Data Type
XMLTYPE	XML data	No	XML
BOOLEAN	Values TRUE / FALSE and NULL. Cannot be assigned to a database table column.	Yes	BOOLEAN
ROWID	Physical row address	No	CHARACTER (255)
UROWID(n)	Universal row ID logical row addresses	No	CHARACTER VARYING
SDO_GEOGRAPHY	The geometric description of a spatial object	No	See PostgreSQL <a href="#">PostGIS</a> extension
SDO_TOPO_GEOGRAPHY	Describes a topology geometry	No	See PostgreSQL <a href="#">PostGIS</a> extension



SDO_GEORASTER	A raster grid or image object is stored in a single row	No	See PostgreSQL <a href="#">PostGIS</a> extension
---------------	---	----	--

### Oracle Data Types Additional Information

Oracle supports both BYTE and CHAR semantics for column size, that is, the amount of allocated storage for CHAR and VARCHAR columns.

- VARCHAR2(10 BYTE)
  - 10 bytes maximum storage. Note that the physical size of some non-English characters exceeds one byte, and this can impact the ten-character storage limit.
- VARCHAR2(10 CHAR)
  - 10-character storage regardless of size.
- Oracle defaults to BYTE semantics, such that when using a multibyte character set (like UTF8), you must either use the CHAR modifier in the VARCHAR2/CHAR column definition, or modify the session or system parameter NLS\_LENGTH\_SEMANTICS to change the default from BYTE to CHAR.

### PostgreSQL Data Types

PostgreSQL only supports CHAR at the column level. A VARCHAR (10) defined field can store 10 characters regardless of how many bytes it takes to store each non-English character, while VARCHAR(n) stores strings up to "n" characters (not bytes) in length.

### Oracle Data Type Conversion Considerations

- Like Oracle, PostgreSQL provides multiple data types that in most cases are equivalent to the Oracle data types and can be automatically converted using Ora2Pg.
- While automations can convert Oracle data types to PostgreSQL equivalents, the following data types are not native to PostgreSQL and require manual processing:
  - ROWID (physical row addresses inside Oracle's ROWID pseudo column)
    - Use CHAR as a partial datatype equivalent and determine if code must be modified to accommodate ROWID datatypes.
  - BFILE (pointers to binary files)
    - 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 the database.
  - UROWID (supports logical and physical ROWIDs of foreign table ROWIDs)
    - Use VARCHAR(n) as a partial datatype equivalent and determine if code must be modified to accommodate UROWID datatypes.

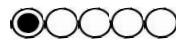
See the following for more information:

- [Oracle Data Types](#)
- [Oracle Built-in Data Types](#)
- [PostgreSQL Data Types](#)
- [PostgreSQL System Columns](#)

## Oracle Views Conversion to PostgreSQL Views

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:** none

Oracle views, like other RDBMS and PostgreSQL views, are used to store a defined SQL statement from one or more physical tables. Views enable access control to specific columns or datasets and provide the ability to display a data structure different than the original structure. Also known as virtual/logical table, a view does not store the actual data; instead, it saves a predefined SQL statement that is associated with the view name. Views can be based on database tables and previously created views.

Views provide the following advantages:

- Data structure - Alter the original table/data structure to suit more use cases.
- Data accessibility - Manage data access by providing only the required columns/datasets.
- Analytics – Summarizes datasets from multiple tables and can act as analytics reports.

### Oracle VIEW Associated Privileges

- Create a VIEW from user tables - Requires the CREATE VIEW privilege.
- Create a view for other users/schemas - Requires the CREATE ANY VIEW privilege.
- A user must have the required privileges on the source tables to create a view.

### Oracle VIEW Syntax

- CREATE VIEW <ViewName>
  - Creates a new view.
- CREATE OR REPLACE <ViewName>
  - Overwrites an existing view and modifies the view definition while retaining the prior granted permissions. Saves time by not requiring you to manually drop and re-create the original view.

### Oracle Common View Parameters

- 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.
- NO FORCE - Create the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default
- 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.

### Oracle Simple VIEW

Simple views are associated only with one table without specifying scalar/group functions that will change the form of the original data.

```
CREATE OR REPLACE FORCE VIEW "VW_INVOICE_REPORT"  
("INVOICEID", "CUSTOMERID", "TOTAL")
```



```
AS
SELECT
    INVOICEID,
    CUSTOMERID,
    TOTAL
FROM INVOICE;

-- Perform update operation on the view
UPDATE VW_INVOICE_REPORT
    SET TOTAL = TOTAL * 1.1 -- Add 10% Tax
    WHERE INVOICEID = 320;

1 row updated.

-- Altering the view with the WITH READ ONLY statement
CREATE OR REPLACE FORCE VIEW "VW_INVOICE_REPORT"
    ("INVOICEID", "CUSTOMERID", "TOTAL")
    AS
SELECT
    INVOICEID,
    CUSTOMERID,
    TOTAL
FROM INVOICE
    WITH READ ONLY;

-- Perform update operation on view with the WITH READ ONLY statement
UPDATE VW_INVOICE_REPORT
    SET TOTAL = TOTAL * 1.1 -- Add 10% Tax
    WHERE INVOICEID = 320;
```



```
ORA-42399: cannot perform a DML operation on a read-only view
```

## Oracle Complex VIEW

Complex views can hold data from several tables, and the original data form can be altered using scalar/group functions DML operations on complex views cannot be applied directly, but you can use INSTEAD OF triggers as a workaround.

```
CREATE OR REPLACE FORCE VIEW "VW_CUST_INVOICE_MONTH"
(
    "CUSTOMERID", "CUSTNAME", "COMPANY", "INVOICE_YEAR",
    "INVOICE_MONTH", "MONTH_TOTAL", "RANK_BY_MONTH"
)
AS
SELECT
    I.CUSTOMERID,
    C.FIRSTNAME || ' ' || C.LASTNAME AS CUSTNAME,
    NVL(C.COMPANY, 'No Company') AS COMPANY,
    EXTRACT(YEAR FROM I.INVOICEDATE) AS INVOICE_YEAR,
    TO_CHAR(I.INVOICEDATE, 'Month') AS INVOICE_MONTH,
    SUM(I.TOTAL) AS MONTH_TOTAL,
    RANK() OVER (PARTITION BY C.FIRSTNAME || ' ' || C.LASTNAME
                ORDER BY SUM(I.TOTAL) DESC) AS RANK_BY_MONTH
FROM INVOICE I JOIN CUSTOMER C
ON I.CUSTOMERID = C.CUSTOMERID
GROUP BY I.CUSTOMERID, C.FIRSTNAME, C.LASTNAME, C.COMPANY,
TO_CHAR(I.INVOICEDATE, 'Month'),
EXTRACT(YEAR FROM I.INVOICEDATE);

-- Perform update operation on the view
UPDATE VW_CUST_INVOICE_MONTH
SET MONTH_TOTAL = MONTH_TOTAL * 1.1 --Add 10% tax
WHERE CUSTOMERID = 320;
```



```
ORA-01732: data manipulation operation not legal on this view
```

```
Oracle VIEW Metadata (can be query by DBA_, ALL_ or USER_)

SELECT * FROM DBA_VIEWS;
```

## PostgreSQL VIEWS

PostgreSQL views have the same purpose as views in Oracle. CREATE VIEW defines a view on existing tables/views, and the actual view data is not physically materialized (stored in database data files); instead, the view's underlying query is run every time the view is referenced.

From a syntax perspective, PostgreSQL VIEWS support Oracle syntax in the form of CREATE VIEW... and CREATE OR REPLACE VIEW... for altering existing view attributes.

### PostgreSQL VIEW Syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name
[ , ... ] ) ]
[ WITH ( view_option_name [= view_option_value] [ , ... ] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

## PostgreSQL VIEWS Associated Privileges

PostgreSQL view privileges require that a role or user be granted SELECT and DML privileges on the base tables or views to create a view.

For more information, see [PostgreSQL GRANT](#).

## PostgreSQL VIEWS Associated Parameters

- CREATE [OR REPLACE] VIEW
  - 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).
  - A simpler approach is to simply use the CREATE VIEW statement instead.
  - Note that the IF EXISTS parameter is optional.
- WITH [ CASCADED | LOCAL ] CHECK OPTION
  - DML INSERT and UPDATE operations are verified against 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 consists of LOCAL and CASCADED configurations:
    - LOCAL - Verifies against the view without a hierarchical check.
    - CASCADED - Verifies all underlying base views using a hierarchical check.

## Executing DML Commands on Views

PostgreSQL simple views update automatically, and no restrictions exist when performing DML operations against views. An updatable view can contain a combination of updatable and non-updatable columns.



Updateable columns reference an updatable column of the underlying base table. Non-updatable columns are read-only, and an error is raised if an INSERT or UPDATE statement is attempted on the column.

### PostgreSQL VIEW Conversion Examples

The following VIEWS were converted using Ora2Pg.

```
CREATE OR REPLACE VIEW vw_invoice_report
  (invoiceid, customerid, total)
AS
  SELECT "InvoiceId", "CustomerId", "Total"
  FROM "Invoice";

-- Perform update operation on the view
UPDATE vw_invoice_report
  SET total = total * 1.1 -- Add 10% Tax
  WHERE invoiceid = 320;
UPDATE 1

-- Add the WITH LOCAL CHECK OPTION statement to the view
CREATE OR REPLACE VIEW vw_invoice_report
  (invoiceid, customerid, total)
AS
  SELECT "InvoiceId", "CustomerId", "Total"
  FROM "Invoice"
  WHERE "Total" BETWEEN 1 AND 10
  WITH LOCAL CHECK OPTION;

-- Perform update operation on the view
UPDATE vw_invoice_report
  SET total = total + 10
  WHERE invoiceid = 100;
```



```
ERROR: new row violates check option for view "vw_invoice_report"
```

PostgreSQL complex VIEW conversion (converted with Ora2Pg):

```
CREATE OR REPLACE VIEW vw_cust_invoice_month
  (customerid, custname, company, invoice_year, invoice_month,
   month_total, rank_by_month)
AS
SELECT
  I."CustomerId",
  C."FirstName" || ' ' || C."LastName" AS CUSTNAME,
  COALESCE(C."Company", 'No Company') COMPANY,
  EXTRACT(YEAR FROM I."InvoiceDate") INVOICE_YEAR,
  TO_CHAR(I."InvoiceDate", 'Month') INVOICE_MONTH,
  SUM(I."Total") AS MONTH_TOTAL,
  RANK() OVER (PARTITION BY C."FirstName" || ' ' || C."LastName"
               ORDER BY SUM(I."Total") DESC) AS RANK_BY_MONTH
FROM "Invoice" I JOIN "Customer" C
ON I."CustomerId" = C."CustomerId"
GROUP BY I."CustomerId", C."FirstName", C."LastName",
         C."Company",
         TO_CHAR(I."InvoiceDate", 'Month'),
         EXTRACT(YEAR FROM I."InvoiceDate");

-- Perform update operation on the view
UPDATE VW_CUST_INVOICE_MONTH
  SET MONTH_TOTAL = MONTH_TOTAL * 1.1 --Add 10% tax
 WHERE CUSTOMERID = 320;

ERROR: cannot update view "vw_cust_invoice_month"
```



```
DETAIL: Views containing GROUP BY are not automatically updatable.
```

PostgreSQL VIEW metadata:

```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE TABLE_TYPE='VIEW';
```

See the following for more information:

- [Oracle CREATE VIEW](#)
- [PostgreSQL Views](#)
- [PostgreSQL CREATE VIEW](#)

**PostgreSQL 11 updates:**

- Allow views to be locked by locking the underlying tables.

**PostgreSQL 12 updates:**

None

**PostgreSQL 13 updates:**

- Support syntax for ALTER VIEW to RENAME COLUMN

## Oracle Sequences Conversion to PostgreSQL Sequences

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- Expect different syntax and features (described below) with PostgreSQL

Oracle sequences are database objects that generate unique identity values (also known as autoincrement functionality), such as primary key values for one or more tables or columns and are treated as independent objects. For example, the `INCREMENT BY` clause defines generated sequence intervals, so if multiple users are using the same sequence, each user encounters visible gaps in the generated values (ensuring sequential, nonoverlapping usage). You can use several different parameters to configure Oracle sequences.

### Oracle Basic Sequence Syntax Example

```
CREATE SEQUENCE <SequenceName>
    MINVALUE      value
    MAXVALUE      value
    START WITH    value
    INCREMENT BY value
    CACHE         value;
```

### Oracle Sequence Specifications

Both initial and increment sequence values default to "1", with no limit.

Additionally, the following options are available:

Oracle Sequence Property	Property Description
START WITH	<ul style="list-style-type: none"><li>● Defines the beginning sequence value of a sequence.</li><li>● Default value is 1.</li></ul>
MINVALUE/NOMINVALUE	<ul style="list-style-type: none"><li>● Sets the bottom value limit (MINVALUE) or designates a "negative 10^26-value limit" if NOMINVALUE is selected.</li><li>● With MINVALUE, the entry must be less than or equal to the START WITH parameter and must be less than the MAXVALUE parameter. NOMINVALUE is the default.</li></ul>
INCREMENT BY	<ul style="list-style-type: none"><li>● Controls the number of increases or decreases in the increment.</li><li>● The default is one.</li><li>● Negative numbers are permitted, and zero is prohibited.</li></ul>



CYCLE/NOCYCLE	<ul style="list-style-type: none"><li>Enables a sequence to keep generating values even once the MAXVALUE or MINVALUE was reached.</li><li>At that point, the system generates a new value based on the original MAXVALUE or MINVALUE (that is, restarts at that set value).</li><li>NOCYCLE is the default</li></ul>
CACHE/NOCACHE	<ul style="list-style-type: none"><li>Designates how many values are cached. CACHE stores a minimum of two values, and NOCACHE retains none.</li><li>By default, the system caches 20 values.</li><li>If a database fail, the "unused" sequence values can potentially be lost, which can cause gaps in sequence values.</li></ul>

## Oracle Sequence Examples

Create a basic sequence:

```
CREATE SEQUENCE ORDER_ID_SEQ
    START WITH 100000
    INCREMENT BY 1
    MAXVALUE 99999999999
    CACHE 20
    NOCYCLE;
```

Sequence and INSERT statement interaction:

```
CREATE TABLE ORDERS (
    ORDER_ID      NUMBER PRIMARY KEY,
    CUSTOMER_ID  VARCHAR2(30));
/
INSERT INTO ORDERS
    VALUES(ORDER_ID_SEQ.NEXTVAL, '100001');
```

Retrieve the latest sequence value:

```
SELECT ORDER_ID_SEQ.CURRVAL FROM DUAL;
```

Alter current sequence value manually using the NEXTVAL statement:



```
SELECT ORDER_ID_SEQ.NEXTVAL FROM DUAL;
```

Alter existing sequence properties:

```
ALTER SEQUENCE ORDER_ID_SEQ MAXVALUE 999999999;
```

Drop an existing sequence:

```
DROP SEQUENCE ORDER_ID_SEQ;
```

Sequence metadata:

```
SELECT * FROM DBA_SEQUENCES;
```

## Oracle 12c Sequence Enhancements

Oracle version 12c introduced several additions to previous sequence functionality:

- From Oracle 12c, sequences can be associated with a table column on table creation and not only when performing DMP operations:

```
CREATE TABLE ORDERS (
    ORDER_ID      NUMBER DEFAULT ORDER_ID_SEQ PRIMARY KEY,
    CUSTOMER_ID VARCHAR2(30));

-- Check the sequence as default value
INSERT INTO ORDERS VALUES (123);

SELECT * FROM ORDERS;

ORDER_ID      CUSTOMER_ID
-----
100000          124
```

- Oracle 12c session and global session sequences:
  - Oracle 12c defaults to global-level (GLOBAL) sequences, but also allows sequences to be session-level (SESSION), added to the CREATE SEQUENCE statement.



- The default GLOBAL setting creates global sequences, ensuring consistent results across database sessions. Session sequences only return a unique sequence range within that session.

```
CREATE SEQUENCE SESSION_LEVEL_SEQ SESSION;
CREATE SEQUENCE SESSION_LEVEL_SEQ GLOBAL;
```

- Identity Columns
  - Oracle 12c allows sequences to be used as an IDENTITY type, automatically creating and associating a sequence with the table column (rather than being manual in prior versions). Though automatic, this type can also be reconfigured.
  - Oracle identity support syntax:

```
GENERATED
[ ALWAYS | BY DEFAULT [ ON NULL ] ]
AS IDENTITY [ ( identity_options ) ]
```

- GENERATED
  - Mandatory keyword for using Oracle identity.
- GENERATED ALWAYS
  - Oracle always tries to generate a value for the identity column.
  - Attempts to insert a value into the identity column cause an error.
- GENERATED BY DEFAULT
  - Oracle generates a value for the identity column if you provide no value. If you provide a value, Oracle inserts that value into the identity column.
  - Oracle issues an error if you insert a NULL value into the identity column.
- GENERATED BY DEFAULT ON NULL
  - Oracle generates a value for the identity column if you provide a NULL value or no value at all.

To insert records using an Oracle 12c IDENTITY column (explicitly/implicitly), use the following commands:

```
CREATE TABLE identity_tbl (
    col_id    NUMBER GENERATED ALWAYS AS IDENTITY,
    col_desc VARCHAR2(100) NOT NULL
);

INSERT INTO identity_tbl (col_desc)
VALUES ('identity test');
```



```
1 row created.

INSERT INTO identity_tbl (col_id, col_desc)
VALUES (NULL, 'identity test');

ERROR at line 1:
ORA-32795: cannot insert into a generated always identity column

INSERT INTO identity_tbl (col_id, col_desc)
VALUES (123, 'identity test');

ERROR at line 1:
ORA-32795: cannot insert into a generated always identity column

-- Same table only with the GENERATED DEFAULT AS IDENTITY statement
CREATE TABLE identity_tbl (
    col_id    NUMBER GENERATED DEFAULT AS IDENTITY,
    col_desc VARCHAR2(100) NOT NULL
);

INSERT INTO identity_tbl (col_id, col_desc)
VALUES (123, 'identity test');

1 row created.
```

## PostgreSQL Sequences

PostgreSQL sequences are highly compatible with Oracle sequences because they generate automatic numeric identifiers and are external objects to a database table owned by the user that created it. In addition, both PostgreSQL and Oracle utilize the same CREATE SEQUENCE command and sequences can be created and used inside CREATE TABLE statements.

### PostgreSQL Sequence Syntax

```
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 ] [ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

#### PostgreSQL Sequence Parameters

Sequence Property	Property Description
START WITH	Optional parameter that uses the MINVALUE for ascending sequences and the MAXVALUE for descending sequences. Default value is 1.
INCREMENT BY	Optional parameter that changes the value sequentially up or down, depending on positive/ascending or negative/descending values added. Default value is 1.
TEMPORARY/TEMP	Temporary (session-long) sequence, meaning the sequence is automatically dropped at the end of the session.
IF NOT EXISTS	Creates a sequence that overwrites any other sequences with the same name.
MAXVALUE/NO MAXVALUE	Sets a top value (MAXVALUE) or disables that feature (NO MAXVALUE). By default, ascending upper limit is $2^{63}-1$ and descending lower limit is -1.
MINVALUE/NO MINVALUE	Sets bottom value (MINVALUE) or disables that feature (NO MINVALUE). By default, ascending upper limit is 1 and descending lower limit is $-2^{63}+1$ .
CYCLE/NO CYCLE	Enables a sequence to keep generating values even when the MAXVALUE or MINVALUE is reached. At that point, the system generates a new value based on the original MAXVALUE or MINVALUE (restarts at the set value). NOCYCLE is the default.
CACHE	Designates how many values are cached. By default, CACHE stores a minimum of 1 value. If no CACHE parameter is specified, no sequence values are pre-cached into memory, which is equivalent to the Oracle NOCACHE parameter.
OWNED BY/OWNED BY NONE	Specifies associations between the sequence object and a specific column in a table. Dropping this sequence returns an error because of the sequence/table association. OWNED BY NONE is the default and specifies that there are no associations.

AS	New option for specifying the sequence data type: SMALLINT, INTEGER, and BIGINT (default), while setting the maximum and minimum values.
----	--

## PostgreSQL Sequence Examples

Basic syntax:

```
CREATE SEQUENCE pg_seq_1
    START WITH 100
    INCREMENT BY 1
    MAXVALUE 99999999999
    CACHE 20
    NO CYCLE;

-- PostgreSQL sequences metadata
SELECT * FROM INFORMATION_SCHEMA.SEQUENCES;

-- Drop a sequence
DROP SEQUENCE SEQ_1;

-- Retrieve current sequence value
SELECT CURRVAL('SEQ_1');

-- Alter current sequence value
SELECT NEXTVAL('pg_seq_1');
SELECT SETVAL(' pg_seq_1', 999);

-- Alter exiting sequence using the ALTER statement
ALTER SEQUENCE pg_seq_1 MAXVALUE 1000000;
```

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



```
CREATE TABLE sequence_tbl (
    col_id    NUMERIC DEFAULT NEXTVAL('pg_seq_1') PRIMARY KEY,
    col_desc VARCHAR(100) NOT NULL
);

INSERT INTO sequence_tbl (col_desc)
VALUES('pg_seq_test');

SELECT * FROM sequence_tbl;
col_id | col_desc
-----+-----
100   | pg_seq_test
```

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

```
CREATE SEQUENCE pg_seq_1
    START WITH 100
    INCREMENT BY 1
    OWNED BY sequence_tbl.col_id;
```

### Generating PostgreSQL Sequence by SERIAL Type

The PostgreSQL SERIAL pseudo-type is a special kind of database object that generates a sequence of integers to serve identify columns such as PRIMARY KEY columns.

Assigning PostgreSQL SERIAL pseudo-type to the table column performs the following:

- Creates a new sequence object and sets the next generated value by the sequence as the default value for the column.
- Adds a NOT NULL constraint to the ID column because a sequence always generates an integer, which is a non-null value.
- Assigns the owner of the sequence to the ID column, and as a result the sequence object is deleted when the ID column or table is dropped.

```
CREATE TABLE sequence_tbl (
    col_id    SERIAL PRIMARY KEY,
    col_desc VARCHAR(100) NOT NULL
```

```
);

INSERT INTO sequence_tbl (col_desc)
VALUES('pg_seq_test');

SELECT * FROM sequence_tbl;

col_id | col_desc
-----+-----
 1 | pg_seq_test
(1 row)

-- Sequence metadata
\dt sequence_tbl

      List of relations
 Schema |     Name      | Type  | Owner
-----+-----+-----+
 public | sequence_tbl | table | naya
(1 row)
```

See the following for more information:

- [Oracle CREATE SEQUENCE](#)
- [CREATE SEQUENCE](#)
- [PostgreSQL Sequence Manipulation Functions](#)
- [PostgreSQL Numeric Types](#)

**PostgreSQL 11 updates:**

None

**PostgreSQL 12 updates:**

None



**PostgreSQL 13 updates:**

None

## Oracle Character Sets Conversion to PostgreSQL Character Set

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- UTF16 character and NCHAR/NVARCHAR data types are not supported.

Oracle supports both the Unicode standard and most national and international encoded character set standards. Specifically, Oracle provides two major 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 this data type to store either Unicode or Non-Unicode characters.
- NVARCHAR2:
  - A quantitative data type used to store Unicode data; supports AL16UTF16 or UTF8, which is ID-specified during database creation.

### Oracle Character Sets Control

In Oracle 11g, character sets are defined at the instance level, while in 12c R2, they are defined at the Pluggable Database level. Prior to 12cR2, root container and all Pluggable Databases character sets were required to be identical.

### Oracle Encoding

Oracle uses the AL32UTF8 Character Set, encoding ASCII characters as single-byte for Latin characters, two-byte for some European and Middle Eastern languages, and three-byte for certain South and East-Asian characters. As a result, Unicode storage requirements are usually higher compared non-Unicode character sets.

### Migration Considerations

During migration, you have two options 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.

The CSALTER utility for character set migrations is no longer a best practice.

### Notes:

- Oracle Database 12c Release 1 (12.1.0.1) complies with the Unicode Standard version 6.1, while Release 2 (12.1.0.2) extends the compliance to version 6.2.
- 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.



PostgreSQL supports a variety of different encodings (character sets) for single- and multi-byte languages. The default character set is defined using the initdb command, when initializing the PostgreSQL database cluster. Each created database on the PostgreSQL cluster can have individual character sets.

In some scenarios, UTF-8 characters are not supported fully in open-source PostgreSQL on Windows, which affects Azure Database for PostgreSQL. Please see the thread on [Bug #15476 in the postgresql-archive](#) for more information.

Azure DB for PostgreSQL runs on Windows OS, so collate should be defined by the language\_territory.codeset format.

To combine the collate, you can use the following:

- [List of all languages](#)
- [List of all territories](#)
- [List of all Windows codesets](#)

Clients can use all supported character sets, but client-side-only characters may not be supported on the server-side.

PostgreSQL does not natively support the NVARCHAR data type and has no UTF-16 support.

Encoding defines the rules on how alphanumeric characters are represented in binary format, for example, Unicode Encoding.

Locale acts as a superset, including LC\_COLLATE and LC\_CTYPE. LC\_COLLATE defines how strings are sorted and must be a subset supported by the database.

Create a database named utf8\_db that uses the UTF8 encoding English United States locale:

```
CREATE DATABASE utf8_db
  WITH ENCODING 'UTF8'
    LC_COLLATE  'English_United States.1252'
    LC_CTYPE    'English_United States.1252'
  TEMPLATE    template0;
```

View the character sets configured for each database by querying the System Catalog:

```
select datname, datcollate, datctype from pg_database;
```

On-the-fly database encoding modifications are not supported. Instead, export all data, create a new database with the new encoding, and reimport the data.

Using the client\_encoding parameter overrides the use of PGCLIENTENCODING and can be useful in the import process.



PostgreSQL supports conversion of character sets between server and client for specific character set combinations as described in the pg\_conversion system catalog.

You can create a new conversion using the SQL command CREATE CONVERSION. Create a conversion from UTF8 to LATIN1 using a custom-made myfunc1 function:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc1;

Method 1
=====
psql \encoding SJIS

Method 2
=====
SET CLIENT_ENCODING TO 'value';

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 sort orders and character classifications on a per-column level.

Specify specific collations for individual table columns:

```
CREATE TABLE names (name text COLLATE "it", description text COLLATE
"it");
```

## Oracle and PostgreSQL Character Set Summary

- View database character set:
  - Oracle: SELECT \* FROM NLS\_DATABASE\_PARAMETERS; (Oracle);
  - PostgreSQL: select datname, pg\_encoding\_to\_char(encoding), datcollate, datatype from pg\_database;
- Modify the database character set:
  - Oracle:
    - Full Export/Import.
    - When converting to Unicode, use the Oracle DMU utility.
  - PostgreSQL:
    - 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
  - Oracle: Instance (11g + 12cR1); database (Oracle 12cR2)
  - PostgreSQL: Database
- UTF8
  - Oracle: Supported via VARCHAR2 and NVARCHAR data types
  - PostgreSQL: Supported via VARCHAR datatype
- UTF16
  - Oracle: Supported via NVARCHAR2 datatype
  - PostgreSQL: Not Supported
- NCHAR/NVARCHAR data types
  - Oracle: Supported
  - PostgreSQL: Not Supported

See the following for additional information:

- Oracle - [Character Set Support](#)
- Oracle - [Character Sets](#)
- PostgreSQL - [Character Set Support](#)

#### **PostgreSQL 11 updates:**

None

#### **PostgreSQL 12 updates:**

- Adding [jsonpath Type](#) - adds support for the SQL/JSON path language in PostgreSQL to efficiently query JSON data.

#### **PostgreSQL 13 updates:**

- Adding the pg\_snapshot as a new PostgreSQL [data type](#) - user-level transaction ID snapshot.

## Migrate from Oracle User-Defined Types

[back to summary](#)

**General compatibility Level:**



**Ora2pg automation capability:**



**Differences Summary:**

- FORALL statement and DEFAULT option are not supported by PostgreSQL.
- PostgreSQL does not support constructors of the "collection" type.

Oracle user-defined types (UDTs) enable the creation of application-dedicated, complex data types that are based on and extend the built-in Oracle data types. Oracle user-defined types are known as OBJECT TYPES and are managed by Oracle PL/SQL. For example, employee addresses can be constructed from multiple columns (Country, State, Street, etc.), and the relational model forces the user to interact with all of the related values for the employee address when performing data manipulation. User-defined enable you to interact with a group of related values as one entity.

The following Oracle user-defined types can be created using the CREATETYPE statement:

- Object Types
- INDEX BY table
- Nested Table
- VARRAY

### Oracle User-Defined Types Examples

```
-- Creating User Defined Type as OBJECT
CREATE TYPE EMPLOYEE_ADDR_TYPE AS OBJECT (
    STREET      VARCHAR2(20),
    CITY        VARCHAR2(20),
    STATE_NAME   VARCHAR2(2),
    ZIP_CODE     NUMBER(6)
);

-- Creating a table specifying the UDF as a column and type
CREATE TABLE EMPLOYEE_ADDRESS (
    EMPLOYEE_ID    NUMBER PRIMARY KEY,
    EMPLOYEE_ADDRESS EMPLOYEE_ADDR_TYPE NOT NULL
);
```

```
-- Performing INSERT specifying the UDF
INSERT INTO EMPLOYEE_ADDRESS
VALUES (1, EMPLOYEE_ADDR_TYPE('2971 Anmoore Road', 'CENTRAL', 'AK', 99730));

-- Verifying the data
SELECT EMPLOYEE_ID,
       ed.EMPLOYEE_ADDRESS.STREET,
       ed.EMPLOYEE_ADDRESS.CITY,
       ed.EMPLOYEE_ADDRESS.STATE_NAME,
       ed.EMPLOYEE_ADDRESS.ZIP_CODE
  FROM EMPLOYEE_ADDRESS ed;

EMPLOYEE_ID EMPLOYEE_ADDRESS.STR EMPLOYEE_ADDRESS.CIT EM EMPLOYEE_ADDRESS.ZIP_CODE
-----
1          2971 Anmoore Road      CENTRAL           AK  99730

-- Applying Oracle records type, table type within a function + bulk collect
CREATE OR REPLACE TYPE ORA_TYPE AS OBJECT (
  COL1 NUMBER,
  COL2 VARCHAR2(100)
);

-- Create a table type which is based on the object type
CREATE OR REPLACE TYPE TYPE_TABLE AS TABLE OF ORA_TYPE;

-- Create a function to return the user defined type
CREATE OR REPLACE FUNCTION FUNC_TABLE_VALUES
  RETURN TYPE_TABLE AS
  V_RESULTS TYPE_TABLE;
```



```
BEGIN
  V_RESULTS := TYPE_TABLE();
  V_RESULTS.EXTEND;
  V_RESULTS(V_RESULTS.count) := ORA_TYPE(1, 'A');
  RETURN V_RESULTS;
END;
/

-- Running the function to return a variable based on the user defined type and table type
SELECT * FROM TABLE(FUNC_TABLE_VALUES());
  COL1 COL2
-----
  1   A

-- Example of Oracle SYS REF CURSOR to return result set from a function
CREATE OR REPLACE FUNCTION FUNC_SYS_REFCURSOR
  RETURN SYS_REFCURSOR
AS
  SYSREFCUR SYS_REFCURSOR;
BEGIN
  OPEN SYSREFCUR FOR
    SELECT EMPLOYEE_ID, FIRST_NAME ||' '|| LAST_NAME
    FROM HR.EMPLOYEES WHERE EMPLOYEE_ID < 105;
  RETURN SYSREFCUR;
  CLOSE SYSREFCUR;
END;
/
```



```
VAR rc refcursor;
BEGIN
:rc := func_sys_refcursor;
END;
/
PRINT rc;

EMPLOYEE_ID FIRST_NAME||"||LAST_NAME
-----
100 Steven King
101 Neena Kochhar
102 Lex De Haan
103 Alexander Hunold
104 Bruce Ernst

5 rows selected.
```

For more information, see [Oracle Create Type](#).



## Migrating to Azure Database for PostgreSQL User Defined Types

PostgreSQL user-defined can be implemented using the CREATE TYPE statement. PostgreSQL UDTs are owned by their creator, and if a schema name is specified, the type is created under that specified schema.

PostgreSQL supports multiple user-defined types:

- Composite types
  - Stores a single named attribute that is attached to a data type or multiple attributes as an attribute collection.
  - Like Oracle, in PostgreSQL you can use of the CREATE TYPE object to allow association with a table.
- Enumerated types (ENUM)
  - Stores a static ordered set of values; for example, state abbreviations:

```
-- Creating User Defined Type as ENUM
CREATE TYPE STATE_ABBREVIATIONS AS ENUM
    ('AL', 'AK', 'AZ');
```

- Range types - PostgreSQL [Range Types](#) store a range of values. In the following example, a range of timestamps is used to represent the ranges of time a course is scheduled:

```
-- Creating User Defined Type as RANGE
CREATE TYPE float8_range AS RANGE
    (subtype = float8, subtype_diff = float8mi);
```

- Base types - Abstract, system core types implemented in a low-level language such as C.
- Array types – Support the definition of multidimensional array columns. An array column can be created with a built-in type or a user-defined base type, ENUM type, or composite.

```
CREATE TABLE ORDER_ITEMS (
    ORDER_ID    NUMERIC PRIMARY KEY,
    CUTOMER_ID VARCHAR(60),
    ORDER_ITEMS text[]
);

INSERT INTO ORDER_ITEMS
    VALUES (1, 123, '{"item-1", "item-2", "item-3"}');

SELECT * FROM ORDER_ITEMS;
```



order_id	customer_id	order_items
1	123	{item-1,item-2,item-3}

## Oracle and PostgreSQL Type Main Differences

PostgreSQL syntax differs from Oracle's CREATE TYPE statement in two major ways:

- PostgreSQL does not support the CREATE OR REPLACE TYPE command.
- PostgreSQL does not accept the AS OBJECT command.

## PostgreSQL CREATE TYPE Syntax

```
CREATE TYPE name AS  
  ( [ attribute_name data_type [ COLLATE collation ] [ , ... ] ] )  
  
CREATE TYPE name AS ENUM  
  ( [ 'Label' [ , ... ] ] )  
  
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 ]
)

CREATE TYPE name
```

## PostgreSQL User-Defined Type Conversion Example

The following conversion example convert Oracle user-defined types as OBJECT. The conversion was done with Ora2Pg conversion tool, and the outcome is identical to Oracle results.

```
CREATE TYPE EMPLOYEES_ADDRESS_TYPE
as (
    STREET VARCHAR(20),
    CITY VARCHAR(20),
    STATE VARCHAR(2),
    ZIP_CODE NUMERIC
);

CREATE TABLE EMPLOYEES_ADDRESS (
    EMP_ID NUMERIC PRIMARY KEY,
    EMP_ADDRESS EMPLOYEES_ADDRESS_TYPE NOT NULL
```



```
);

INSERT INTO EMPLOYEES_ADDRESS
VALUES (1, ('2971 Anmoore Road', 'CENTRAL', 'AK', 99730));

SELECT a.EMP_ID,
       (a.EMP_ADDRESS).STATE,
       (a.EMP_ADDRESS).CITY,
       (a.EMP_ADDRESS).STREET,
       (a.EMP_ADDRESS).ZIP_CODE
FROM EMPLOYEES_ADDRESS a;

emp_id |state |city      |street              |zip_code |
-----|-----|-----|-----|-----|
1      |AK    |CENTRAL |2971  Anmoore Road |99730    |

-- Create a Type
CREATE TYPE postgres_type AS (
COL1 numeric,
COL2 varchar(100)
);

-- Create a function to return the user defined type
CREATE OR REPLACE FUNCTION func_table_values()
RETURNS postgres_type AS $body$
DECLARE
  V_RESULTS postgres_type;
BEGIN
  V_RESULTS:=(1, 'A');
RETURN V_RESULTS;
```

```
END;
$body$
LANGUAGE PLPGSQL
SECURITY DEFINER;

-- Verify
select func_table_values();

func_table_values
-----
(1A)

-- PostgreSQL function to return result set by refcursor
CREATE OR REPLACE FUNCTION FUNC_SYS_REFCURSOR(sysrefcur refcursor)
RETURNS refcursor AS $$

BEGIN
    OPEN sysrefcur FOR SELECT "EmployeeId", "FirstName" || ' ' || "LastName" as
emp_name FROM "Employee";
    RETURN sysrefcur;
END;
$$ LANGUAGE plpgsql;

BEGIN;
SELECT FUNC_SYS_REFCURSOR('sysrefcur');
FETCH ALL IN sysrefcur;

EmployeeId |      emp_name
-----+-----
      1 | Andrew Adams
      2 | Nancy Edwards
      3 | Jane Peacock
```



```
4 | Margaret Park
```

```
5 | Steve Johnson
```

```
(5 rows)
```

```
COMMIT;
```

For more information, see [PostgreSQL Create Type](#).

## Performance Topics

### Migrating from Oracle Statistics Collection

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** N/A

#### Differences Summary:

- Expect different architecture, syntax, and configuration methods.

Like many other relational database systems, Oracle uses table statistics which are collected from table columns values and table indexes to describe the distribution, uniqueness and selectivity of the data stored in a specific database table.

The collected statistical information will be represented by a histogram (counts of the occurrences of specific values taken from multiple data points distributed across the data evenly), while the main objective of the table statistics is to serve the database Optimizer decision making for producing effective execution plans.

With no column-based statistics, the database Optimizer may assume a uniform distribution of rows across the values in a column (to accurately reflect a non-uniform data distribution a histogram is required) and uses statistics such as:

- Table total number of records
- Table cardinality and distinct values
- Table overall data distribution

#### Oracle Statistics Gathering Methods

- Automatic
  - Oracle manages table and index statistics collection automatically during predefined maintenance windows using the database scheduler and automated maintenance tasks.
  - Any DML operation of INSERT, UPDATE or DELETE will be tracked by Oracle automatic statistics collection system which affects the table and index statistics accordingly.
- Manual
  - In addition, Oracle offers the ability to gather table and index statistics manually in cases that the optimizer execution plans are not optimal for a specific use-case. The following are the statistics levels that can be collected manually using Oracle DBMS\_STATS system package:
    - GATHER\_TABLE\_STATS
    - GATHER\_SCHEMA\_STATS
    - GATHER\_INDEX\_STATS
    - GATHER\_DATABASE\_STATS
    - GATHER\_DICTIONARY\_STATS

#### Oracle Statistics Collection Examples

Collect statistics at a column level:

- Schema Name - SALES
- TABLE - ORDERS



- Column - ORDER\_ID

```
BEGIN  
DBMS_STATS.GATHER_TABLE_STATS('SALES','ORDERS',  
    METHOD_OPT=>'FOR COLUMNS ORDER_ID');  
END;  
/
```

Collect statistics at a table level:

- Schema Name - SALES
- TABLE - ORDERS

```
BEGIN  
DBMS_STATS.GATHER_TABLE_STATS(SALES,ORDERS);  
END;  
/
```

For more information, see [Oracle Statistics collection](#).

### Migrating to Azure Database for PostgreSQL Statistics Collection

Similar to Oracle, PostgreSQL uses statistics based on the contents of tables in the database and stores the results in the **pg\_statistic** system catalog to serve the database Optimizer (Query Planner in PostgreSQL). The query planner uses the collected statistics to determine the most efficient execution plan for a specific query.

### PostgreSQL Statistics Collection Methods

- Automated
  - Like Oracle, PostgreSQL collects statistics automatically using the PostgreSQL Routine Vacuuming (using the Autovacuum daemon) which automatically applies the ANALYZE statement when the content of a table has changed sufficiently. For the query planner to provide accurate execution plans, the table should be collected and updated.
  - PostgreSQL Autovacuum daemon automates several database background operations, including collecting and updating tables statistics. You use several parameters to configure the daemon:

Microsoft Azure | Microsoft

Autovacuum

Parameter name	Value	Parameter Type	Description
<code>autovacuum</code>	ON	Dynamic	Starts the <code>autovacuum</code> subprocesses.
<code>autovacuum_analyze_scale_factor</code>	0.05	Dynamic	Number of tuple inserts, updates, or deletes prior to analyze as...
<code>autovacuum_analyze_threshold</code>	50	Dynamic	Minimum number of tuple inserts, updates, or deletes prior to ...
<code>autovacuum_freeze_max_age</code>	200000000	Static	Age at which to <code>autovacuum</code> a table to prevent transaction ID...
<code>autovacuum_max_workers</code>	3	Static	Sets the maximum number of simultaneously running <code>autovacuum</code> ...
<code>autovacuum_multivax_freeze_max_age</code>	400000000	Static	Multivax age at which to <code>autovacuum</code> a table to prevent multiv...
<code>autovacuum_naptime</code>	15	Dynamic	Time to sleep between <code>autovacuum</code> runs. Unit is s.
<code>autovacuum_vacuum_cost_delay</code>	20	Dynamic	Vacuum cost delay in milliseconds, for <code>autovacuum</code> .
<code>autovacuum_vacuum_cost_limit</code>	-1	Dynamic	Vacuum cost amount available before napping, for <code>autovacuum</code> .
<code>autovacuum_vacuum_scale_factor</code>	0.05	Dynamic	Number of tuple updates or deletes prior to vacuum as a fract...
<code>autovacuum_vacuum_threshold</code>	50	Dynamic	Minimum number of tuple updates or deletes prior to vacuum.
<code>autovacuum_work_mem</code>	-1	Dynamic	Sets the maximum memory to be used by each <code>autovacuum</code> w...
<code>log_autovacuum_min_duration</code>	-1	Dynamic	Sets the minimum execution time above which <code>autovacuum</code> ac...

- Manual
  - ANALYZE collects statistics about the contents of tables in the database and stores the results in the pg\_statistic system catalog. Subsequently, the PostgreSQL query planner uses the collected statistics to select the most efficient execution plans for queries.
  - PostgreSQL allows collecting statistics on demand using the ANALYZE command at a database level, table level, or column-level.
  - Collecting statistics for indexes using the ANALYZE statement is not supported.
  - ANALYZE applies a read-lock on the target table.

### PostgreSQL ANALYZE Statement examples

Collect statistics at the table-level; the VERBOSE keyword displays additional information:

```
ANALYZE VERBOSE "Invoice";
INFO:  analyzing "public.Invoice"
INFO:  "Invoice": scanned 6 of 6 pages, containing 412 live rows and 0 dead rows; 412 rows in sample, 412 estimated total rows
ANALYZE
```

Collect statistics at the column level:

```
ANALYZE VERBOSE "Customer"("CustomerId");
INFO:  analyzing "public.Customer"
```



```
INFO: "Customer": scanned 2 of 2 pages, containing 59 live rows and 0 dead  
rows; 59 rows in sample, 59 estimated total rows  
  
ANALYZE
```

Collect statistics at the database level:

```
ANALYZE;  
  
ANALYZE
```

View PostgreSQL statistics metadata information:

```
SELECT schemaname,  
       relname,  
       last_analyze  
  FROM pg_stat_all_tables;
```

#### Oracle and PostgreSQL Statistics Collection Comparison

Statistics Property	Oracle	PostgreSQL
Gather statistics at schema level	begin execute dbms_stats.gather_schema_stats(ownname => user); end;	ANALYZE;
Gather statistics at table level	begin dbms_stats.gather_table_stats(ownname => user, tabname => tableName); end;	ANALYZE <TableName>;
Sampling only a segment from table total rows	begin dbms_stats.gather_schema_stats(estimate_percent => 60, ownname => user);	set default_statistics_target to 150;  ANALYZE <TableName>;



	end;	
Statistics Metadata View	SELECT * FROM DBA_TAB_STATS_HISTORY;	SELECT * FROM PG_STAT_ALL_TABLES;

For more information, see [PostgreSQL Routine Vacuuming](#).

### Migrate from Oracle Execution Plans

Oracle EXPLAIN PLAN statement provides granular details and insight about the execution plans selected by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE operations. Each Oracle SELECT and DML operation can be described by its related execution plan in a sequence of operations that Oracle performs to run the actual statement. The EXPLAIN PLAN results provide insight into whether the optimizer selects a particular execution plan. It also helps you understand the optimizer decisions, such as why the optimizer chose a nested loops join instead of a hash join, and provides information about query performance.

Oracle execution plans have the following information:

- An ordering of the tables referenced by the statement.
- An access method for each table mentioned in the statement.
- A join method for tables affected by join operations in the statement.
- Data operations like filter, sort, or aggregation.

Oracle Execution plan also provides the following details for performance analysis:

- Optimization, such as the cost and cardinality of each operation.
- Partitioning, such as the set of accessed partitions.
- Parallel execution, such as the distribution method of join inputs.

### Oracle Generate and View Execution Plan Examples

SET AUTOTRACE TRACEONLY EXPLAIN instructs Oracle (SQLPLUS session) to show the execution plan only without running the actual query.

```
-- Create unindexed sample table (based on DBA_OBJECTS)

CREATE TABLE DB_OBJECTS
AS
SELECT OBJECT_NAME FROM DBA_OBJECTS;

-- Generate EXECUTION PLAN - FULL TABLE SCAN

SET AUTOTRACE TRACEONLY EXPLAIN
SELECT OBJECT_NAME FROM DB_OBJECTS
WHERE OBJECT_NAME = 'EMPLOYEES';
```

```
Execution Plan
-----
Plan hash value: 159947100
-----
-- 
| Id | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time
|   |
-----  
--  
| 0 | SELECT STATEMENT   |                | 18708 | 1205K | 19 (0)    | 00:00:01  
|  
| 1 |  TABLE ACCESS FULL | DB_OBJECTS    | 18708 | 1205K | 19 (0)    | 00:00:01  
|  
--  
  
Predicate Information (identified by operation id):  
-----
  1 - filter("OBJECT_NAME"='EMPLOYEES')  
  
-- Creating an index on the OBJECT_NAME column  
CREATE INDEX IDX_OBJECT_NAME ON DB_OBJECTS(OBJECT_NAME);  
  
-- Generate Execution Plan again -  
SET AUTOTRACE TRACEONLY EXPLAIN  
SELECT OBJECT_NAME FROM DB_OBJECTS  
  WHERE OBJECT_NAME = 'EMPLOYEES';  
  
-- The Optimizer will now perform INDEX RANGE SCAN  
Execution Plan
-----
Plan hash value: 4163328303
```

```
--  
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |  
--  
| 0 | SELECT STATEMENT | | 1 | 66 | 1 (0)| 00:00:01 |  
|* 1 | INDEX RANGE SCAN| IDX_OBJECT_NAME | 1 | 66 | 1 (0)| 00:00:01 |  
--  
  
Predicate Information (identified by operation id):  
-----  
 1 - access("OBJECT_NAME"='EMPLOYEES')
```

For more information, see [Generating and Displaying Execution Plans](#).

### Migrating to Azure Database for PostgreSQL Execution Plans

Like Oracle Optimizer, PostgreSQL query planner generate the estimated execution plan for operations such as SELECT, INSERT, UPDATE and DELETE. It then builds a structured tree of plan operations representing the different actions taken; the (->) sign represents a root line in the PostgreSQL execution plan. In addition, PostgreSQL EXPLAIN provides statistical information regarding each action, such as cost, rows, time, and loops.

### PostgreSQL EXPLAIN and EXPLAIN ANALYZE

When using the PostgreSQL EXPLAIN command as part of a SQL statement, the statement does not execute, and the execution plan is an estimation.

When using the PostgreSQL EXPLAIN ANALYZE command, the statement is executed the execution plan is displayed.

PostgreSQL supports several scan types for processing and retrieving data from tables, including:

- Sequential scans
- Index scans
- Bitmap index scans

The sequential scan (Seq Scan) is PostgreSQL equivalent for Oracle "Table access full" (full table scan).

PostgreSQL EXPLAIN ANALYZE statement runs the command and shows actual run times and other statistics.

### PostgreSQL EXPLAIN and EXPLAIN ANALYZE Examples

EXPLAIN shows that idx\_invoice\_date is used for the following SQL statement:

```
EXPLAIN
```

```
SELECT * FROM public."Invoice"
WHERE "InvoiceDate"
BETWEEN '2009-01-01 00:00:00' AND '2010-01-01 00:00:00';
```

QUERY PLAN

---

```
Index Scan using idx_invoice_date on "Invoice"
(cost=0.27..5.14 rows=83 width=66)
Index Cond: ("InvoiceDate" >= '2009-01-01 00:00:00'::timestamp
without time zone) AND ("InvoiceDate" <= '2010-01-01
00:00:00'::timestamp without time zone))
(2 rows)
```

The following is a FULL TABLE SCAN example:

```
EXPLAIN
SELECT * FROM public."Invoice" WHERE "Total" > 100;

QUERY PLAN
-----
Seq Scan on "Invoice" (cost=0.00..11.15 rows=1 width=66)
Filter: ("Total" > '100'::numeric)
(2 rows)
```

Apply PostgreSQL ANALYZE statement to get information in addition to the execution plan:

```
EXPLAIN ANALYZE
SELECT * FROM public."Invoice" WHERE "Total" > 100;

QUERY PLAN
-----
Seq Scan on "Invoice" (cost=0.00..11.15 rows=1 width=66) (actual
time=0.177..0.177 rows=0 loops=1)
```

```

Filter: ("Total" > '100'::numeric)
Rows Removed by Filter: 412
Planning Time: 0.101 ms
Execution Time: 0.199 ms
(5 rows)

```

#### PostgreSQL versions 12 and 13 new features description:

Feature/parameter	Description	PostgreSQL 12	PostgreSQL 13
EXPLAIN	Added configuration parameter SETTINGS.  Include information on configuration parameters. Specifically, include options affecting query planning with value different from the built-in default value. This parameter defaults to FALSE.	Supported	Supported
EXPLAIN	Added configuration parameter WAL.  Include information on WAL record generation. Specifically, include the number of records, number of full-page images (fpi), and amount of WAL bytes generated. In text format, only nonzero values are printed. This parameter can only be used when ANALYZE is also enabled. It defaults to FALSE.	N/A	Supported
ANALYZE	Added configuration parameter SKIP_LOCKED.  Specifies that ANALYZE should not wait for any conflicting locks to be released when beginning work on a relation.	Supported	Supported



ANALYZE	<p>Added configuration parameter Boolean.</p> <p>Specifies whether the selected option should be turned on or off. You can write TRUE, ON, or 1 to enable the option, and FALSE, OFF, or 0 to disable it. The boolean value can be omitted, in which case TRUE is assumed.</p>	Supported	Supported
---------	--	-----------	-----------

For more information, see [PostgreSQL EXPLAIN](#).

## Oracle Database Optimizer Hints Conversion to PostgreSQL Query Planning

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

### Differences Summary:

- The ability to influence optimizer execution plans in PostgreSQL is limited compared to Oracle.

Oracle optimizer hints provide advanced options to affect the optimizer execution plan for a specific SQL statement. You can use Oracle optimizer hints to control various query performance parameters, such as indexes selection, indexes ignore, join order, parallel execution, and others. Oracle optimizer hints provide a method to direct the optimizer to choose a certain query execution plan, based on specific criteria, to improve a query performance. Hints are embedded directly in SQL statements following the SELECT keyword, using the following format:

```
SELECT /*+ <Optimizer_Hint> */...
```

Oracle supports multiple database hints categorized into several groups. Each optimizer hint can have 0 or more arguments:

- Access path hints
- Join order hints
- Parallel execution hints

### Oracle Optimizer Hints Examples

The following SQL query execution plan shows that the optimizer has selected the IDX\_CUST\_ID index:

```
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT * FROM invoice inv
  WHERE invoicedate BETWEEN '01-JAN-2013' AND '31-DEC-2013'
    AND customerid BETWEEN 100 AND 300;

-----
-----| Id | Operation          | Name      | Rows | Bytes | Cost
(%CPU)| Time   |
-----| 0 | SELECT STATEMENT   |          | 1   | 64   |
| 2 (0) | 00:00:01           |          |
-----| 1 | TABLE ACCESS BY INDEX ROWID| INVOICE    | 1   | 64   |
| 00:00:01 |          |
```



```
|* 2 | INDEX RANGE SCAN          | IDX_CUST_ID | 1      |      | 1 (0)  |
00:00:01  |
```

---

---

Add the INDEX optimizer hint to force the optimizer to use the `IDX_INVOICE_DATE` index instead:

```
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT /*+ INDEX (inv idx_invoice_date) */ * FROM INVOICE INV
  WHERE INVOICEDATE BETWEEN '01-JAN-2013' AND '31-DEC-2013'
    AND CUSTOMERID BETWEEN 100 AND 300;
```

---

---

(%CPU)	Time	Id	Operation	Name	Rows	Bytes	Cost
3 (0)	00:00:01	0	SELECT STATEMENT		1	64	
00:00:01		* 1	TABLE ACCESS BY INDEX ROWID	INVOICE	1	64	3 (0)
		* 2	INDEX RANGE SCAN	IDX_INV_DATE	80		2 (0)

---

---

## PostgreSQL Query Planning

Although PostgreSQL does not support Oracle optimizer hints, it does provide a wide selection of *query planning parameters* to influence the behavior of the optimizer-selected execution plans at a session level. The execution plans are generated from the SQL queries.

PostgreSQL provides the following [query planning options](#):

- Planner method configuration - Configuration parameters to influence the query plans chosen by the query optimizer when not optimal.
- Planner cost constants - Set of parameters that can provide optimizer cost optimization.
- Genetic query optimizer - An algorithm that plans queries using heuristic searching.
- Additional planner options - An additional set of parameters to control various optimization.

## PostgreSQL Query Planning Examples

The following examples describe some of the PostgreSQL query planning parameters.

- `enable_indexscan` (boolean)
  - Enables or disables the query planner's use of index-scan plan types.
  - On by default.

```
-- Set the enable_indexscan to off
set enable_indexscan=off;

-- Check current parameter value
show enable_indexscan;

enable_indexscan
-----
off
```

- `enable_mergejoin` (boolean)
  - Enables or disables the query planner's use of merge-join plan types.
  - On by default.

```
-- Set the enable_mergejoin to off
set enable_mergejoin=off;

-- Check current parameter value
show enable_mergejoin;
```

```
enable_mergejoin
-----
off
(1 row)
```

- `seq_page_cost` (floating point)
  - Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches
  - The default is 1.0

```
-- Check current parameter value
show seq_page_cost;

seq_page_cost
-----
1
(1 row)

-- Alter the parameter value
set seq_page_cost to 2;
SET

-- Check parameter value again
show seq_page_cost;

seq_page_cost
-----
2
(1 row)
```

- Control PostgreSQL optimizer using the query planning parameters:
  - The following SQL query execution plan shows that the chosen query plan is `HashAggregate`, which enables or disables the query planner's use of hashed aggregation plan types.



```
EXPLAIN
SELECT inv."InvoiceId",
       SUM(inv1."UnitPrice"),
       COUNT(inv1."Quantity")
  FROM "Invoice" inv JOIN "InvoiceLine" inv1
    ON inv."InvoiceId" = inv1."InvoiceId"
 GROUP BY inv."InvoiceId";

-- SQL query execution plan
          QUERY PLAN
-----
HashAggregate  (cost=96.41..101.56 rows=412 width=44)
  Group Key: inv."InvoiceId"
    -> Hash Join  (cost=21.27..79.61 rows=2240 width=13)
      Hash Cond: (inv1."InvoiceId" = inv."InvoiceId")
      -> Seq Scan on "InvoiceLine" inv1  (cost=0.00..52.40 rows=2240 width=13)
      -> Hash  (cost=16.12..16.12 rows=412 width=4)
        -> Seq Scan on "Invoice" inv  (cost=0.00..16.12 rows=412 width=4)
(7 rows)
```

- Run the same query while setting enable\_hashagg to OFF. This causes the execution plan to implement the GroupAggregate method.

```
SET ENABLE_HASHAGG=OFF;

EXPLAIN
SELECT inv."InvoiceId",
       SUM(inv1."UnitPrice"),
       COUNT(inv1."Quantity")
  FROM "Invoice" inv JOIN "InvoiceLine" inv1
    ON inv."InvoiceId" = inv1."InvoiceId"
```

```
GROUP BY inv."InvoiceId";\n\n-- SQL query execution plan\n\n          QUERY PLAN\n-----\n\nGroupAggregate  (cost=204.25..231.80 rows=412 width=44)\n  Group Key: inv."InvoiceId"\n    -> Sort  (cost=204.25..209.85 rows=2240 width=13)\n      Sort Key: inv."InvoiceId"\n        -> Hash Join  (cost=21.27..79.61 rows=2240 width=13)\n          Hash Cond: (invl."InvoiceId" = inv."InvoiceId")\n            -> Seq Scan on "InvoiceLine" invl  (cost=0.00..52.40 rows=2240\nwidth=13)\n            -> Hash  (cost=16.12..16.12 rows=412 width=4)\n              -> Seq Scan on "Invoice" inv  (cost=0.00..16.12 rows=412\nwidth=4)\n(9 rows)
```

- PostgreSQL index scan versus a seq scan (full table scan):

```
-- Index scan\n\nEXPLAIN\n\n  SELECT * FROM "Invoice"\n  WHERE "CustomerId" BETWEEN 1 AND 10;\n\n          QUERY PLAN\n-----\n\nIndex Scan using "IFK_InvoiceCustomerId" on "Invoice"  (cost=0.27..9.37 rows=70\nwidth=66)\n  Index Cond: ((\"CustomerId\" >= 1) AND (\"CustomerId\" <= 10))\n(2 rows)
```

```
-- Seq scan (full table scan)
SET enable_indexscan=OFF;

QUERY PLAN
-----
Seq Scan on "Invoice"  (cost=0.00..18.18 rows=70 width=66)
  Filter: ((CustomerId" >= 1) AND ("CustomerId" <= 10))
(2 rows)
```

See the following for more information:

- [Oracle Influencing the Optimizer](#)
- [PostgreSQL Query Planning](#)

#### PostgreSQL 11 updates:

None

#### PostgreSQL 12 updates:

- Add [planner support function](#) interfaces to improve optimizer estimates, inlining, and indexing for functions. This allows extensions to create planner support functions that can provide function-specific selectivity, cost, and row-count estimates that depend on the function's arguments. Support functions can also supply simplified representations and index conditions, greatly expanding optimization possibilities.
- Allow [CREATE STATISTICS](#) to create most-common-value statistics for multiple columns. This improves optimization for queries that test several columns, requiring an estimate of the combined effect of several WHERE clauses. If the columns are correlated and have non-uniform distributions, then multi-column statistics allow much better estimates.
- Allow [common table expressions](#) (CTEs) to be inlined into the outer query. Specifically, CTEs are automatically inlined if they have no side-effects, are not recursive, and are referenced only once in the query. Inlining can be prevented by specifying MATERIALIZED or forced for multiply-referenced CTEs by specifying NOT MATERIALIZED. Previously, CTEs were never inlined and were always evaluated before the rest of the query.
- Allow control over when generic plans are used for prepared statements. This is controlled by the [plan\\_cache\\_mode](#) server parameter.

#### PostgreSQL 13 updates:

- Improve the optimizer's [selectivity](#) estimation for containment/match operators.
- Allow setting the [statistics target](#) for [extended statistics](#). This is controlled with the new command option ALTER STATISTICS ... SET STATISTICS. Previously this was computed based on more general statistics target settings.
- Allow use of multiple extended statistics objects in a single query.
- Allow use of extended statistics objects for OR clauses and [IN/ANY](#) constant lists.
- Allow functions in FROM clauses to be pulled up (inlined) if they evaluate to constants.



## Oracle Indexes conversion to PostgreSQL Indexes

[back to summary](#)

**General compatibility Level:** 

**Ora2pg automation capability:** 

### Differences Summary:

- Index PostgreSQL does not support BITMAP index. (BRIN index can be used in as an alternative solution in some scenarios.)
- PostgreSQL does not support invisible indexes. A workaround is available.

Like many other database platforms, Oracle provides a wide selection of indexes for data query use cases. Database indexes provide a direct, fast access to table records. An index is created as a database object based on a table column or multiple columns (a composite index).

### Oracle Indexes Types and Syntax Examples

- B-Tree index
  - Most commonly used database index.
  - Used with table primary keys, suited for a variety of workloads, and can be configured in ascending (default) and descending ordering.

```
CREATE INDEX IDX_EMP_ID ON EMPLOYEES(EMP_ID ASC);
```

- Bitmap index
  - Stores a bitmap for each index key. Each index key can store pointers to multiple rows.
  - Best suited for providing fast data retrieval for OLAP workloads.

```
CREATE BITMAP INDEX IDX_EMP_ID ON EMPLOYEES(HIREDATE DESC);
```

- Function-Based Index
  - Calculates and stores the result of a function that involves one or more columns, and stores that result in the index.
  - In the following example, the index stores all employee last names in uppercase using the UPPER function.

```
CREATE INDEX IDX_LASTNAME ON EMPLOYEES(UPPER(LASTNAME));
```

- Unique index - A special type of B-Tree index enforcing a UNIQUE constraint on the indexed values per column.

```
CREATE UNIQUE INDEX IDX_EMP_PHONE ON EMPLOYEES(EMP_PHONE);
```



- Composite index – Created over a number of columns to improve the performance of data retrieval when the column sequence in the index is a key factor. Index access is by the defined column sequence inside the index.

```
CREATE INDEX IDX_EMPID_HIREDATE ON EMPLOYEES(EMP_ID, HIREDATE);
```

- Invisible index
  - Oracle 11g introduced a new feature allowing indexes to be marked as invisible.
  - Invisible indexes are maintained like any other index, but they can be ignored by the optimizer.
  - Indexes can be created as invisible by using the INVISIBLE keyword, and you manage their visibility using the ALTER INDEX command.

```
CREATE INDEX IDX_HIREDATE ON EMPLOYEES(HIREDATE) INVISIBLE;
```

```
ALTER INDEX IDX_HIREDATE VISIBLE;
```

- Index-organized table
  - A unique type of table stored in the B-tree index structure and specified with the CREATE TABLE statement.
  - The Index-organized table index stores the primary key values of an Oracle indexed-organized tables row, and each index entry in the B-tree stores the nonkey column values.
  - This can provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key.

```
CREATE TABLE EMPLOYEES (
    EMP_ID NUMBER,
    EMP_NAME VARCHAR2(10) NOT NULL,
    EMP_DEPARTMENT VARCHAR2(200),
    EVENT_TIME DATE NOT NULL,
    CONSTRAINT PK_EMP_ID PRIMARY KEY(EMP_ID))
    ORGANIZATION INDEX;
```

- Local and Global indexes
  - Local and global indexes are used for partitioned tables in Oracle databases.
  - Each index type can be created on a partitioned table and can be specified as either LOCAL or GLOBAL.
  - Local index
    - Keeps one-to-one relationship between the index partitions and the table partitions.
    - For each table partition, a separate index partition is created.



- Created using the LOCAL clause.

```
CREATE INDEX IDX_ORDERDATE ON ORDERS(ORDER_DATE) LOCAL  
  (PARTITION P_ORDER_DATE2018,  
   PARTITION P_ORDER_DATE2019,  
   PARTITION P_ORDER_DATE2020);
```

- Global index
  - A global index can be partitioned or non-partitioned (default).
  - Each global index contains keys from multiple table partitions in a single index partition.
  - Created using the GLOBAL clause during index creation.

```
CREATE INDEX IDX_ORDER_DATE ON ORDERS(ORDER_DATE) GLOBAL  
  PARTITION BY RANGE(ORDER_DATE) (  
    PARTITION P_ORDER_DATE2017  
      VALUES LESS THAN (TO_DATE('01/01/2018','DD/MM/YYYY')),  
    PARTITION P_ORDER_DATE2018  
      VALUES LESS THAN (TO_DATE('01/01/2019','DD/MM/YYYY')),  
    PARTITION P_ORDER_DATE2019  
      VALUES LESS THAN (TO_DATE('01/01/2020','DD/MM/YYYY')),  
    PARTITION P_ORDER_DATE2020 VALUES LESS THAN (MAXVALUE));
```

- Partial indexes for partitioned tables - Oracle version 12c introduced the creation of global and local indexes on a subset of the partitions of a partitioned table using the INDEXING [ON | OFF] clause.

```
CREATE TABLE ORDERS (  
  ORDER_ID      NUMBER,  
  ORDER_CUST_ID NUMBER,  
  ORDER_DATE    DATE  
)  
PARTITION BY RANGE(ORDER_DATE) (  
  PARTITION p2019  
    VALUES LESS THAN(TO_DATE('01/01/2018', 'DD/MM/YYYY')),  
  PARTITION p2020
```



```
VALUES LESS THAN(TO_DATE('01/01/2016', 'DD/MM/YYYY'))  
INDEXING ON,  
PARTITION p2021  
VALUES LESS THAN(TO_DATE('01/01/2017', 'DD/MM/YYYY'))  
INDEXING OFF);
```

#### Oracle Drop Index Statement

```
DROP INDEX <INDEX_NAME>;
```

#### Oracle Indexes Metadata

```
SELECT * FROM DBA_INDEXES;  
SELECT * FROM DBA_PART_INDEXES;  
SELECT * FROM DBA_IND_COLUMNS;
```

#### PostgreSQL Indexes

Like Oracle, PostgreSQL supports multiple types of indexes that use different algorithms to provide performance benefits for different types of queries and use cases. The PostgreSQL default index type is B-Tree index when issuing the `CREATE INDEX` statement with no other index configuration supplementals.

#### PostgreSQL Index Syntax

```
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 ]
```

#### PostgreSQL Index Types

PostgreSQL Index Type	Index Description
B-Tree	<ul style="list-style-type: none"><li>PostgreSQL default index and the most commonly used.</li><li>Specifying a primary key or a unique key within the <code>CREATE TABLE</code> statement causes PostgreSQL to create a B-Tree index.</li></ul>

	<ul style="list-style-type: none"> <li>Applying a CREATE INDEX statement without the USING keyword forces PostgreSQL to create a B-Tree index.</li> <li>Can be specified as unique index and with sorting using the ASCENDING/DESCENDING keywords.</li> <li>The PostgreSQL optimizer using B-Tree indexes supports one or more of the following operators in queries: &gt;, &gt;=, &lt;, &lt;=, =</li> </ul>
BRIN	<ul style="list-style-type: none"> <li>PostgreSQL block range index (BRIN) is designed to handle very large tables in which certain columns have some natural correlation with their physical location in the table.</li> <li>The BRIN indexes contain only the MIN and MAX values contained in a group of database pages. It can rule out the presence of certain records and therefore reduce query run time.</li> </ul>
Hash	<ul style="list-style-type: none"> <li>PostgreSQL hash indexes can perform faster lookup operations than a B-Tree index.</li> <li>On the downside, the hash index is limited to equality operators that perform equation operations only.</li> </ul>
GIN	<ul style="list-style-type: none"> <li>PostgreSQL Generalized inverted index (GIN) is most beneficial when a datatype has multiple values in a single column.</li> <li>The GIN index is most useful when an SQL query needs to map a large amount of values to one row.</li> <li>PostgreSQL GIN indexes work well for full-text searches and for indexing array values.</li> </ul>
GiST	<ul style="list-style-type: none"> <li>PostgreSQL generalized search tree index (GiST) provides a balanced, tree-structured access method and acts as a base template in which to implement arbitrary indexing schemes.</li> <li>B-Trees, R-Trees, and many other indexing schemes can be implemented in GiST.</li> </ul>
SP-GiST	<ul style="list-style-type: none"> <li>The PostgreSQL SP-GiST index (a space-partitioned GiST) allows the indexing of a variety of nonbalanced data structures by using partitioned search trees.</li> <li>The SP-GiST index can be used with data that has a natural grouping of elements and a nonbalanced tree structure</li> </ul>
Multi-column indexes	<ul style="list-style-type: none"> <li>PostgreSQL support multi-column indexes for the B-tree, GiST, GIN, and BRIN indexes.</li> <li>Up to 32 columns can be specified when creating a multi-column index.</li> <li>Oracle composite indexes and PostgreSQL multi-column indexes share the same syntax.</li> </ul>
Expression indexes	<ul style="list-style-type: none"> <li>PostgreSQL supports expression indexes, which accept an attached function to the index definition.</li> <li>These can be used as an equivalent for Oracle function-based indexes.</li> </ul>



Partial Indexes	<ul style="list-style-type: none"><li>PostgreSQL partial indexes can be used to index only a subset of data while specifying a WHERE clause when creating the index.</li><li>Partial indexes reduce the overall index size, allowing users to index only relevant table data.</li><li>In some cases, partial indexes increase efficiency and reduce the size of the index (for example, temporary tables).</li></ul>
-----------------	--

### PostgreSQL Indexes Creation Examples

Create a B-Tree index with DESCENDING sorting, with and without specifying USING:

```
CREATE INDEX IDX_EMPID ON EMPLOYEES(EMPID DESC);
-- OR
CREATE INDEX IDX_EMPID ON EMPLOYEES USING BTREE (EMPID DESC);
```

PostgreSQL unique index:

```
CREATE UNIQUE INDEX IDX_EMP_PHONE ON EMPLOYEES(PHONE);
```

PostgreSQL BRIN index:

```
CREATE INDEX IDX_ORD_DATE ON ORDERS USING BRIN(ORDER_DATE);
```

PostgreSQL multi-column index:

```
CREATE INDEX IDX_ORDS ON ORDERS(ORDERID, ORDERDATE);
```

PostgreSQL expression index:

```
CREATE INDEX IDX_LASTNAME ON EMPLOYEES(UPPER(LASTNAME));
```

PostgreSQL expression index:

```
CREATE INDEX IDX_PART_INV_DATE ON INVOICE(INVOICEDATE)
WHERE INVOICEDATE > '2019-01-01 00:00:00';
```

PostgreSQL SP-GiST index:

```
CREATE INDEX IDX_LOCATION ON SHIPMENTS USING SPGIST(LOCATION);
```

Drop index syntax:

```
DROP INDEX <INDEX_NAME>;
```

### PostgreSQL Indexes - Administration Commands Examples

Rename an existing index:

```
ALTER INDEX IDX_EMP_PHONE RENAME TO IDX_EMP_CELL_PHONE;
```

PostgreSQL [REINDEX](#) - equivalent to Oracle REBUILD INDEX command:

```
REINDEX INDEX IDX_EMP_PHONE;
```

PostgreSQL [REINDEX](#) concurrently - equivalent to Oracle REBUILD INDEX ONLINE command:

```
CREATE INDEX CONCURRENTLY IDX_EMPID ON EMPLOYEES(EMPID DESC);
```

### Oracle and PostgreSQL Indexes Comparison

Oracle Indexes Types	PostgreSQL Index Support (Yes/No) or Equivalent Index
B-Tree Index	B-Tree Index
Index-Organized Tables	Not supported, consider <a href="#">PostgreSQL Cluster</a> as an alternative solution
Reverse key indexes	Not supported
Descending indexes	ASCENDING/ASC (default) or DESCENDING/DESC
Unique	Supported
Non-unique Indexes	
Function-Based Indexes	PostgreSQL Expression Indexes
Application Domain indexes	Not supported
BITMAP Index	Consider the BRIN index as an alternative solution
Composite Indexes	Multi-column indexes
Invisible Indexes	Use the supported <a href="#">hypopg</a> extension as an alternative solution
Local and Global Indexes	Not supported

Partial Indexes for Partitioned Tables (Oracle 12c)	Not supported
CREATE INDEX DROP INDEX	Supported, expect syntax minor differences
ALTER INDEX... (General Definitions)	Supported
INDEX REBUILD	REINDEX INDEX
REBUILD ONLINE	CONCURRENTLY
Index Metadata	SELECT * FROM PG_INDEXES;
Index Compression	Not direct equivalent to Oracle index key compression or advanced index compression

See the following for more information:

- [Oracle CREATE INDEX](#)
- [PostgreSQL Index Types](#)
- [PostgreSQL CREATE INDEX](#)
- [PostgreSQL REINDEX](#)
- [PostgreSQL Multicolumn Indexes](#)
- [Oracle Understand When to Create Multiple Indexes on the Same Set of Columns](#)

#### PostgreSQL 11 updates:

- Allow [btree\\_gin](#) to index bool, bpchar, name and uuid data types.
- CREATE INDEX can now use parallel processing while building a B-tree index.
- Covering indexes can now be created, using the INCLUDE clause of CREATE INDEX.
- Supports indexes on partitioned tables. An index on a partitioned table is not a physical index across the whole partitioned table, but instead a template for automatically creating similar indexes on each partition of the table. If the partition key is part of the index's column set, a partitioned index can be declared UNIQUE. It represents a valid uniqueness constraint across the whole partitioned table, even though each physical index only enforces uniqueness within its own partition.

The new command [ALTER INDEX ATTACH PARTITION](#) causes an existing index on a partition to be associated with a matching index template for its partitioned table. This provides flexibility in setting up a new partitioned index for an existing partitioned table.

- Allow B-tree indexes to include columns that are not part of the search key or unique constraint but are available to be read by index-only scans. This is enabled by the new INCLUDE clause of CREATE INDEX. It facilitates building "covering indexes" that optimize specific types of queries. Columns can be included even if their data types do not have B-tree support.



#### PostgreSQL 12 updates:

None

#### PostgreSQL 13 updates:

- More efficiently store [duplicates](#) in B-tree indexes. This allows efficient B-tree indexing of low-cardinality columns by storing duplicate keys only once. Users upgrading with [pg\\_upgrade](#) need to use [REINDEX](#) to make an existing index use this feature.
- Allow [GiST](#) and [SP-GIST](#) indexes on box columns to support ORDER BY *box* <-> *point* queries.
- Allow [GIN](#) indexes to more efficiently handle ! (NOT) clauses in tsquery searches.
- Allow [index operator classes](#) to take parameters.
- Allow CREATE INDEX to specify the GiST signature length and maximum number of integer ranges.
- Indexes created on four- and eight-byte [integer array](#), [tsvector](#), [pg\\_trgm](#), [ltree](#), and [hstore](#) columns can now control these GiST index parameters instead of using the defaults.
- Prevent indexes that use non-default collations from being [added](#) as a table's unique or primary key constraint.
- The index's collation must match that of the underlying column; ALTER TABLE previously failed to check this.

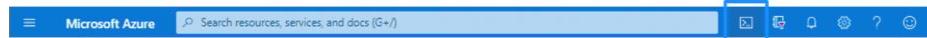
## Azure-Specific Topics

### Azure CLI - Common Usages with Azure DB

[Azure CLI](#) is Azure cloud's cross-platform command-line interface that provides commands to utilize and manage Azure resources, including the [Azure Database for PostgreSQL managed services](#). You can use Azure CLI in several ways, including installing on a local machine (different for each [operation system](#)) with all the required libraries, or by running in the Azure [Cloud Shell](#) environment through a web browser.

#### Deploying Azure Database for PostgreSQL using Azure CLI

To use Azure CLI with Cloud Shell, click the Cloud Shell icon on the upper-right tool bar of Azure Portal:



The following example uses the Azure CLI `az postgres server create` command to create an Azure Database for PostgreSQL server, with optional parameters:

```
az postgres server create [-h] [--verbose] [--debug]
                          [--output {json,jsonc,table,tsv}]
                          [--query JMESPATH]
                          --resource-group RESOURCE_GROUP_NAME --name SERVER_NAME
                          --sku-name SKU_NAME [--location LOCATION]
                          --admin-user ADMINISTRATOR_LOGIN
                          --admin-password ADMINISTRATOR_LOGIN_PASSWORD
                          [--backup-retention BACKUP_RETENTION]
                          [--geo-redundant-backup GEO_REDUNDANT_BACKUP]
                          [--ssl-enforcement {Enabled,Disabled}]
                          [--storage-size STORAGE_MB]
                          [--tags [TAGS [TAGS ...]]]
                          [--version VERSION]
                          [--subscription _SUBSCRIPTION]
```

Azure Database for PostgreSQL server create example:

```
az postgres server create -l northeurope -g testgroup -n testsvr -u username -p
password --sku-name GP_Gen5_1 --ssl-enforcement Enabled --minimal-tls-version TLS1_0
```



```
--public-network-access Disabled --backup-retention 10 --geo-redundant-backup  
Enabled --storage-size 51200 --version 11.0
```

Create Azure Database for PostgreSQL database command example:

```
az postgres db create -g testgroup -s testsrv -n testdb
```

See the following for more information:

- [Azure CLI for Azure Database for PostgreSQL Server Command Examples](#)
- [Azure CLI for Azure Database for PostgreSQL Database Command Examples](#)



## Oracle to Azure Database for PostgreSQL - Data Migration Paths

The following table describes the different online and offline approaches for data migration from an Oracle database hosted anywhere to an Azure Database for PostgreSQL managed service.

Objective	Description	Tool	Prerequisites	Pros and cons
Offline migration	Ora2Pg offers schema conversion and data migration functionality.	<a href="#">Ora2Pg</a>	Ora2Pg must be installed/deployed (using Docker) on a virtual machine and configured to support the migration process.	<b>Pros:</b> Ora2Pg migrates everything at once. It is less complex than online migration. <b>Cons:</b> Because full downtime is required, this process may be suitable only when extended downtime is possible.
Offline migration with parallel functionality	Azure Data Factory provides fast offline migrations that can be done using multiple parallel jobs.	<a href="#">Azure Data Factory</a>	None	<b>Pros:</b> Faster offline migrations.
Online migration	For information about online migration options, contact: <a href="mailto:AskAzureDBforPostgreSQL@service.microsoft.com">AskAzureDBforPostgreSQL@service.microsoft.com</a>			



## Right-Sizing PostgreSQL and Application VM Instances

Azure Database for PostgreSQL offers a wide range of options for different types of PostgreSQL database configuration and sizing. These options support multiple and dynamic workloads to help achieve pricing goals and performance requirements. You can deploy Azure Database for PostgreSQL using a wide selection of sizing layouts, choosing from the following elements:

- Single Instance
  - Compute
    - Compute is provisioned in virtual cores (vCores).
    - A vCore represents a logical CPU.
    - Compute Gen 5 CPUs are based on Intel E5-2673 v4 (Broadwell) 2.3 GHz processors.
  - Storage
    - Charges for the storage component as a part of the database server.
    - Storage can be provisioned up to 1 TB.
  - Backup
    - You can configure the server to use automated backup storage.
    - Increasing the backup retention period increases the backup storage consumed by the database server.
    - There is no additional charge for backup storage for up to 100% of the total provisioned server storage.
  - Read Replica
    - Read replicas allow workload distribution by separating read and write operations between the master and read replicas databases servers.
    - For each read replica you create, you are billed for the provisioned compute in vCores and provisioned storage in GB/month.
  - Each component can be configured for the following supported workloads:
    - Basic - Workloads requiring light compute and I/O performance. For example:
      - Min sizing: vCores: 1 | Memory: 2GB
      - Max sizing: vCores: 2 | Memory: 4GB
    - General Purpose - Most business workloads requiring balanced compute and memory with scalable I/O throughput. For example:
      - Min sizing: vCores: 2 | Memory: 10GB
      - Max sizing: vCores: 64 | Memory: 320GB



- Memory Optimized - High-performance database workloads requiring in-memory performance for faster transaction processing and higher concurrency. For example:
  - Min sizing: vCores: 2 | Memory: 20GB
  - Max sizing: vCores: 32 | Memory: 320GB
- Flexible Server (currently available in preview)
  - Burstable - Workloads with flexible compute requirements. For example:
    - Instance B1MS: vCores: 1 | Memory: 2GB
    - Instance B2S: vCores: 2 | Memory: 4GB
  - General Purpose - Most business workloads requiring balanced compute and memory with scalable I/O throughput. For example:
    - Min sizing D2 v3: vCores: 2 | Memory: 8GB
    - Max sizing D64 v3: vCores: 64 | Memory: 256GB
  - Memory Optimized - High-performance database workloads requiring in-memory performance for faster transaction processing and higher concurrency. For example:
    - Min sizing E2 v3: vCores: 2 | Memory: 16GB
    - Max sizing E64 v3: vCores: 64 | Memory: 432GB
- Hyperscale (Citus)
  - Worker Node Size (Compute & Memory)
    - Each Hyperscale (Citus) database cluster starts at 2 worker nodes.
    - As your workload grows, you can add more vCores and memory to your worker nodes and scale the PostgreSQL cluster by adding more worker nodes.
    - Example:
      - Min sizing: vCores: 4 | Memory: 16 GB
      - Max sizing: vCores: 64 | Memory: 432 GB
  - Coordinator Node Size (Compute & Memory)
    - Each Hyperscale (Citus) database cluster has 1 coordinator node.
    - Example:
      - Min sizing: vCores: 4 | Memory: 16 GB
      - Max sizing: vCores: 64 | Memory: 256 GB

See the following for more information:

- [Azure Database for PostgreSQL pricing](#) (according to sizing options)



- [Compute and Storage options in Azure Database for PostgreSQL - Flexible Server](#)
- [Monitor and tune Azure Database for PostgreSQL - Single Server](#)
- [Monitor and tune Azure Database for PostgreSQL - Hyperscale \(Citus\)](#)



## Azure Database for PostgreSQL Sizing Based on Oracle AWR

Oracle AWR (Automatic Workload Repository) is a tool for collecting information about an Oracle database instance running processes. It provides a detailed summary report for performance insights. You can use the AWR information when sizing the Azure Database for PostgreSQL target for migration. The Oracle AWR is a long, verbose report, and understanding and analyzing the complete report requires previous experience. The following section focuses on collecting only the information required accurately size the target Azure Database for PostgreSQL instance.

### AWR Sizing Assessment Key Points

- Use more than one AWR report from multiple time periods to gain a better understanding about a specific Oracle database workload and performance information. One AWR report does not represent accurately the overall workload.
- review an AWR from both an average workload timeframe and a high workload timeframe to identify different workload characteristics.
- For RAC environments, perform the AWR analysis separately on all the instances in the Oracle RAC cluster. This helps to determine if all instances are balanced from a workload perspective and helps prevent sizing assumptions based on a single Oracle database instance.
- An AWR report show the current CPU and RAM configuration and resources usage for a defined timeframe. To identify the current Oracle database storage size, use the following query:

```
SELECT SUM(BYTES/1024/1024) AS SIZEMB FROM DBA_DATA_FILES;
```

- When planning the sizing of the Azure Database for PostgreSQL, the Oracle AWR report provides a starting point. Because PostgreSQL utilizes a different architecture, different sizing configurations should be expected, and performance tests are crucial.

### Gathering AWR General and Sizing Specific Information

This section describes the specific AWS report sections that can provide useful information about the analyzed Oracle database sizing details for CPU and RAM usage.

#### AWR General Information

The first AWR section provide general information about the Oracle database instance and server, such as:

- Database name
- Database version
- Server CPU and memory configuration

When performing analysis for Oracle RAC environment, note that each database node has a unique name.

The memory values are for the database server and indicate the database memory allocation (see details below).

## WORKLOAD REPOSITORY report for

DB Name	DB Id	Instance	Inst num	Startup Time	Release	RAC
ORADB	3411413079	ORADB	1	01-Nov-20 08:49	11.2.0.4.0	NO

Host Name	Platform	CPUs	Cores	Sockets	Memory (GB)
oradb.srv.local	Linux x86 64-bit	4	4	1	31.35

### Database Memory and Additional Configuration

The values of the memory parameters configuration with additional parameters can be found in the "init.ora Parameters" section in the AWR report.

The memory values are represented in bytes. For example, 15770583040/1024/1024/1024 = 14.6 GB of memory allocated to the database.

Additional configuration for the "open\_cursors" and "processes" is also found in this section.

Parameter Name	Begin value
_system_trig_enabled	FALSE
audit_file_dest	/u01/app/oracle/admin/ORADB/adump
audit_trail	DB
compatible	11.2.0.0.0
control_files	/u01/app/oracle/oradata/ORADB/control01.ctl, /u01/app/oracle/flash_recovery_area/ORADB/control02.ctl
control_management_pack_access	DIAGNOSTIC+TUNING
db_block_size	8192
db_domain	
db_name	ORADB
db_recovery_file_dest	/u01/app/oracle/flash_recovery_area
db_recovery_file_dest_size	4070572032
diagnostic_dest	/u01/app/oracle
dispatchers	(PROTOCOL=TCP) (SERVICE=ORADBXDB)
memory_max_target	15770583040
memory_target	15770583040
open_cursors	300
processes	150
recyclebin	OFF
remote_login_passwordfile	EXCLUSIVE
resource_manager_plan	
undo_retention	86400
undo_tablespace	UNDOTBS1

### CPU and RAM

AWR reports show the operating system CPUs number, the CPU cores, CPU sockets, and memory allocation in the "Operating System Statistics" section:

## Operating System Statistics

- \*TIME statistic values are diffed. All others display actual values. End Value is displayed if different
- ordered by statistic type (CPU Use, Virtual Memory, Hardware Config), Name

Statistic	Value	End Value
BUSY_TIME	200,287	
IDLE_TIME	6,937,395	
IOWAIT_TIME	566,395	
NICE_TIME	0	
SYS_TIME	24,527	
USER_TIME	174,607	
LOAD	1	0
RSRC_MGR_CPU_WAIT_TIME	0	
PHYSICAL_MEMORY_BYTES	33,661,534,208	
NUM_CPUS	4	
NUM_CPU_CORES	4	
NUM_CPU_SOCKETS	1	
GLOBAL_RECEIVE_SIZE_MAX	4,194,304	
GLOBAL_SEND_SIZE_MAX	1,048,576	
TCP_RECEIVE_SIZE_DEFAULT	87,380	
TCP_RECEIVE_SIZE_MAX	4,194,304	
TCP_RECEIVE_SIZE_MIN	4,096	
TCP_SEND_SIZE_DEFAULT	16,384	
TCP_SEND_SIZE_MAX	4,194,304	
TCP_SEND_SIZE_MIN	4,096	

AWR reports offer several ways to identify specific CPU and memory usage patterns. These methods can provide a good baseline for the target sizing considerations. For example, the AWR can indicate if the current workload is fully utilized (CPU and memory at high usage), or if the current configuration supports low and high workloads:

### Host CPU (CPUs: 4 Cores: 4 Sockets: 2)

Load Average Begin	Load Average End	%User	%System	%WIO	%Idle
		39.4	21.5		39.1

### Instance CPU

%Total CPU	%Busy CPU	%DB time waiting for CPU (Resource Manager)
2.9	4.8	0.0

### Memory Statistics

	Begin	End
Host Mem (MB):	24,575.6	24,575.6
SGA use (MB):	808.0	808.0
PGA use (MB):	341.6	361.5
% Host Mem used for SGA+PGA:	4.68	4.76

AWR "Load Profile" shows the actual identified CPU usage for the AWR snapshot timeframe:

- DB CPU(s) - The amount of CPU time spent on user requests.



- DB time - The database time for user requests in microseconds; does not include background processes.
- Logical and physical reads - Show how many IOs (physical and logical) the database is performing. For example, logical reads constructed from gets + db block gets. Oracle checks if the data is available in buffer cache (SGA). If it exists, the logical read is increased by 1.

### Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	1.3	6.1	0.02	0.00
DB CPU(s):	0.1	0.6	0.00	0.00
Redo size:	2,332.7	11,005.8		
Logical reads:	8,648.1	40,802.9		
Block changes:	11.4	53.9		
Physical reads:	430.3	2,030.1		
Physical writes:	46.9	221.1		
User calls:	339.3	1,601.0		
Parses:	2.1	10.1		
Hard parses:	0.1	0.3		
W/A MB processed:	0.5	2.3		
Logons:	0.1	0.5		
Executes:	83.5	393.9		
Rollbacks:	0.1	0.6		
Transactions:	0.2			

### CPU Usage per SQL Statement

For performance comparison between Oracle as the source database and Azure Database for PostgreSQL as the target platform, see the AWR "SQL ordered by CPU Time" section to identify the current Oracle SQL statements from CPU usage performance perspectives. This can be very useful when performing migration performance tests.

### SQL ordered by CPU Time

- Resources reported for PL/SQL code includes the resources used by all SQL statements called by the code.
- %Total - CPU Time as a percentage of Total DB Time
- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Captured SQL account for 71.5% of Total CPU Time (s): 2,024
- Captured PL/SQL account for 5.8% of Total CPU Time (s): 2,024

CPU Time (s)	Executions	CPU per Exec (s)	%Total	Elapsed Time (s)	%CPU	%IO	SQL Id	SQL Module	SQL Text
245.22	12	20.43	12.12	4,102.55	5.98	98.07	fuwqushpzbqy9a	sqlservr.exe	SELECT Max(written) MaxWritten...
186.03	2	93.02	9.19	370.41	50.22	51.66	5gngrvxz5zz9x	JDBC Thin Client	select insideAlias.CNT from (S...
155.16	2	77.58	7.67	194.74	79.68	21.61	3n8nnnmccxp5	JDBC Thin Client	select insideAlias.CNT from (S...
151.39	2	75.69	7.48	706.21	21.44	81.02	b1zpwu30lcm05	JDBC Thin Client	select insideAlias.CNT from (S...
123.00	2	61.50	6.08	153.24	80.27	21.03	3fgxpdhy6t9g	JDBC Thin Client	select insideAlias.CNT from (S...
99.83	2	49.92	4.93	120.41	82.91	17.89	06983gt1v07zs	JDBC Thin Client	select insideAlias.CNT from (S...
94.79	2	47.39	4.68	126.60	74.87	26.75	7m9wkrqw825un	JDBC Thin Client	select insideAlias.CNT from (S...
83.95	1	83.95	4.15	173.87	48.28	50.64	b6usrg82hwsa3	DBMS_SCHEDULER	call dbms_stats.gather_databas...
64.42	2	32.21	3.18	83.81	76.87	24.45	52wzunvnvy4qz	JDBC Thin Client	select insideAlias.CNT from (S...
52.62	2	26.31	2.60	200.44	26.25	78.51	b09cu4b2sqtr1	JDBC Thin Client	select insideAlias.CNT from (S...
30.86	1	30.86	1.52	45.08	68.45	32.73	8zvfgs4ggmm9qd	JDBC Thin Client	select insideAlias.CNT from (S...
27.00	1	27.00	1.33	391.69	6.89	42.34	8mcpb06rctk0x	DBMS_SCHEDULER	call dbms_space.auto_space_adv...



For more information about Oracle to Azure sizing calculation formula, see [Estimate Tool for Sizing Oracle Workloads to Azure IaaS VMs](#).

For additional information, see [Oracle Automatic Workload Repository](#).