

Oracle® Database

Graph Developer's Guide for Property Graph



21.3
F44328-03
October 2021

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Oracle Database Graph Developer's Guide for Property Graph, 21.3

F44328-03

Copyright © 2016, 2021, Oracle and/or its affiliates.

Primary Author: Lavanya Jayapalan

Contributors: Prashant Kannan, Chuck Murray, Melliyal Annamalai, Korbinian Schmid, Albert Godfrind, Oskar van Rest, Jorge Barba, Ana Estrada, Steve Serra, Ryota Yamanaka, Bill Beauregard, Hector Briseno, Hassan Chafi, Eugene Chong, Souripriya Das, Juan Garcia, Florian Gratzer, Zazhil Herena, Sungpack Hong, Roberto Infante, Hugo Labra, Gabriela Montiel-Moreno, Eduardo Pacheco, Joao Paiva, Matthew Perry, Diego Ramirez, Siva Ravada, Carlos Reyes, Jane Tao, Edgar Vazquez, Zhe (Alan) Wu

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	XX
Documentation Accessibility	XX
Related Documents	XX
Conventions	XX

Changes in This Release for This Guide

Part I Getting Started with Oracle Property Graphs

1 Property Graph Support Overview

1.1	Introduction to Property Graphs	1-2
1.1.1	What Are Property Graphs?	1-2
1.1.2	About the Property Graph Feature of Oracle Database	1-3
1.1.2.1	In-Memory Graph Server (PGX)	1-4
1.1.2.2	Options for Property Graph Architecture	1-4
1.2	Before You Begin with Oracle Property Graphs	1-7
1.2.1	Database Compatibility and Restrictions	1-8
1.2.2	Downloading Oracle Graph Server and Client	1-8
1.2.3	Installing PL/SQL Packages in Oracle Database	1-9
1.3	Oracle Graph Server and Client Installation	1-10
1.3.1	Installing Oracle Graph Server	1-11
1.3.2	Deploying Oracle Graph Server to a Web Server	1-13
1.3.2.1	Deploying to Apache Tomcat	1-13
1.3.2.2	Deploying to Oracle WebLogic Server	1-14
1.3.3	Upgrading Oracle Graph Server	1-15
1.3.4	Uninstalling Oracle Graph Server	1-15
1.3.5	Installing Oracle Graph Client	1-15
1.3.5.1	Installing the Java Client	1-15
1.3.5.2	Installing the Python Client	1-16

1.3.5.3	Uninstalling the Python Client	1-18
1.3.5.4	Enabling the Graph Visualization Application	1-18
1.3.5.5	Deploying the Graph Visualization Application	1-19
1.3.5.6	Installing the Graph Zeppelin Interpreter Client	1-24
1.4	Setting Up Transport Layer Security	1-25
1.4.1	Generating a Self-Signed Server Certificate	1-25
1.4.2	Configuring the Graph Server (PGX)	1-26
1.4.3	Configuring a Client to Trust the Self-Signed Certificate	1-27
1.5	Adding Permissions to Publish the Graph	1-28
1.6	Security Best Practices with Graph Data	1-29
1.7	Interactive Graph Shell	1-29
1.8	Developing Applications Using Graph Server Functionality as a Library	1-32
1.9	Storing Graphs in Oracle Database and Loading Graphs into Memory	1-32
1.9.1	Two-Tier Mode	1-32
1.9.2	Three-Tier Mode	1-33
1.10	Using Oracle Graph with the Autonomous Database	1-33
1.10.1	Two-Tier Deployments of Oracle Graph with Autonomous Database	1-34
1.10.2	Three-Tier Deployments of Oracle Graph with Autonomous Database	1-35
1.11	Migrating Property Graph Applications from Before Release 21c	1-37
1.12	Upgrading From Graph Server and Client 20.4.x to 21.x	1-40
1.13	Using the Graph Zeppelin Interpreter Client	1-42
1.14	About Oracle Graph Server and Client Accessibility	1-43

2 Quick Starts for Using Oracle Property Graph

2.1	Quick Start: Interactively Analyze Graph Data	2-1
2.1.1	Quick Start: Create and Query a Graph in the Database, Load into In-Memory Graph Server (PGX) for Analytics	2-1
2.1.1.1	Create and Query a Graph in the Database	2-2
2.1.1.2	Load the Graph into Memory and Run Graph Analytics	2-6
2.1.2	Quick Start: Create, Query, and Analyze a Graph in In-Memory Graph Server (PGX)	2-9
2.1.3	Quick Start: Executing PGQL Queries in SQLcl	2-14
2.2	QuickStart: Run Graph Analytics Using the Python Shell	2-14
2.3	Quick Start: Using the Python Client as a Module	2-15
2.4	Oracle LiveLabs Workshops for Graphs	2-17

3 Property Graph Views on Oracle Database Tables

3.1	Loading a Graph into the Graph Server (PGX) from a Property Graph View	3-4
-----	--	-----

4 Using the In-Memory Graph Server (PGX)

4.1	Overview of the In-Memory Graph Server (PGX)	4-2
4.1.1	Design of the In-Memory Graph Server (PGX)	4-2
4.1.2	Usage Modes of the In-memory Graph Server (PGX)	4-4
4.2	User Authentication and Authorization	4-5
4.2.1	Privileges and Roles in Oracle Database	4-6
4.2.2	Basic Steps for Using an Oracle Database for Authentication	4-7
4.2.3	Prepare the Graph Server for Database Authentication	4-9
4.2.4	Store the Database Password in a Keystore	4-10
4.2.5	Token Expiration	4-16
4.2.6	Advanced Access Configuration	4-16
4.2.7	Customizing Roles and Permissions	4-17
4.2.7.1	Checking Graph Permissions Using API	4-18
4.2.7.2	Adding and Removing Roles	4-20
4.2.7.3	Defining Permissions for Individual Users	4-20
4.2.7.4	Defining Permissions to Use Custom Graph Algorithms	4-21
4.2.8	Revoking Access to the Graph Server	4-21
4.2.9	Examples of Custom Authorization Rules	4-21
4.2.10	Kerberos Enabled Authentication	4-23
4.2.10.1	Prerequisite Requirements	4-24
4.2.10.2	Prepare the Graph Server for Kerberos Authentication	4-24
4.2.10.3	Login to the Graph Server Using Kerberos Ticket	4-25
4.3	About Vertex and Edge IDs	4-26
4.4	Reading Graphs from Oracle Database into the Graph Server (PGX)	4-29
4.4.1	Creating a JSON Configuration to Load a Graph	4-29
4.4.2	Defining the Graph Configuration via Java	4-30
4.5	Keeping the Graph in Oracle Database Synchronized with the Graph Server	4-31
4.5.1	Examples of Synchronizing	4-32
4.6	Optimizing Graphs for Read Versus Updates in the In-Memory Graph Server (PGX)	4-36
4.7	Storing a Graph Snapshot on Disk	4-37
4.8	Executing Built-in Algorithms	4-38
4.8.1	About Built-In Algorithms in the In-Memory Graph Server (PGX)	4-39
4.8.2	Running the Triangle Counting Algorithm	4-40
4.8.3	Running the PageRank Algorithm	4-40
4.9	Using Custom PGX Graph Algorithms	4-41
4.9.1	Writing a Custom PGX Algorithm	4-41
4.9.1.1	Collections	4-42
4.9.1.2	Iteration	4-43
4.9.1.3	Reductions	4-43
4.9.2	Compiling and Running a PGX Algorithm	4-44

4.9.3	Example Custom PGX Algorithm: PageRank	4-45
4.10	Creating Subgraphs	4-46
4.10.1	About Filter Expressions	4-46
4.10.2	Using a Simple Filter to Create a Subgraph	4-47
4.10.3	Using a Complex Filter to Create a Subgraph	4-47
4.10.4	Using a Vertex Set to Create a Bipartite Subgraph	4-48
4.11	Using Automatic Delta Refresh to Handle Database Changes	4-50
4.11.1	Configuring the In-Memory Server for Auto-Refresh	4-50
4.11.2	Configuring Basic Auto-Refresh	4-51
4.11.3	Reading the Graph Using the In-Memory Graph Server (PGX) or a Java Application	4-51
4.11.4	Checking Out a Specific Snapshot of the Graph	4-52
4.11.5	Advanced Auto-Refresh Configuration	4-53
4.11.6	Special Considerations When Using Auto-Refresh	4-54
4.12	Starting the In-Memory Graph Server (PGX)	4-54
4.12.1	Starting and Stopping the Graph Server (PGX) Using the Command Line	4-54
4.12.2	Configuring the In-Memory Graph Server (PGX)	4-55
4.13	Connecting to the In-Memory Graph Server (PGX)	4-61
4.13.1	Connecting with the Graph Shell	4-61
4.13.2	Connecting with Java	4-64
4.13.2.1	Starting and Stopping the PGX Engine	4-64
4.13.3	Connecting with Python	4-65
4.14	Using Graph Server (PGX) as a Library	4-66
4.15	User-Defined Functions (UDFs) in PGX	4-67
4.16	Using HAProxy for PGX Load Balancing and High Availability	4-71

5 Using the Property Graph Schema

5.1	Property Graph Schema Objects for Oracle Database	5-2
5.1.1	Property Graph Tables (Detailed Information)	5-2
5.1.2	Default Indexes on Vertex (VT\$) and Edge (GE\$) Tables	5-7
5.1.3	Flexibility in the Property Graph Schema	5-7
5.2	Data Access Layer	5-7
5.3	Getting Started with Property Graphs	5-8
5.3.1	Required Privileges for Database Users	5-8
5.4	Using Java APIs for Property Graph Data	5-8
5.4.1	Overview of the Java APIs	5-9
5.4.1.1	Oracle Graph Property Graph Java APIs	5-9
5.4.1.2	Oracle Database Property Graph Java APIs	5-9
5.4.2	Parallel Loading of Graph Data	5-10
5.4.2.1	JDBC-Based Data Loading	5-10
5.4.2.2	External Table-Based Data Loading	5-19

5.4.2.3	SQL*Loader-Based Data Loading	5-23
5.4.3	Parallel Retrieval of Graph Data	5-26
5.4.4	Using an Element Filter Callback for Subgraph Extraction	5-28
5.4.5	Using Optimization Flags on Reads over Property Graph Data	5-31
5.4.6	Adding and Removing Attributes of a Property Graph Subgraph	5-33
5.4.7	Getting Property Graph Metadata	5-38
5.4.8	Merging New Data into an Existing Property Graph	5-39
5.4.9	Opening and Closing a Property Graph Instance	5-41
5.4.10	Creating Vertices	5-43
5.4.11	Creating Edges	5-43
5.4.12	Deleting Vertices and Edges	5-44
5.4.13	Reading a Graph from a Database into an Embedded In-Memory Analyst	5-44
5.4.14	Specifying Labels for Vertices	5-45
5.4.15	Building an In-Memory Graph	5-45
5.4.16	Dropping a Property Graph	5-47
5.4.17	Executing PGQL Queries	5-47
5.5	Managing Text Indexing for Property Graph Data	5-47
5.5.1	Configuring a Text Index for Property Graph Data	5-48
5.5.1.1	Configuring Text Indexes Using Oracle Text	5-48
5.5.2	Using Automatic Indexes for Property Graph Data	5-50
5.5.3	Using Manual Indexes for Property Graph Data	5-52
5.5.4	Executing Search Queries Over a Property Graph's Text Indexes	5-52
5.5.4.1	Executing Search Queries Over a Text Index Using Oracle Text	5-52
5.5.5	Handling Data Types	5-54
5.5.5.1	Handling Data Types on Oracle Text	5-54
5.5.6	Updating Configuration Settings on Text Indexes for Property Graph Data	5-55
5.5.7	Using Parallel Query on Text Indexes for Property Graph Data	5-55
5.5.7.1	Parallel Text Search Using Oracle Text	5-55
5.6	Access Control for Property Graph Data (Graph-Level and OLS)	5-57
5.6.1	Applying Oracle Label Security (OLS) on Property Graph Data	5-57
5.7	SQL-Based Property Graph Query and Analytics	5-62
5.7.1	Simple Property Graph Queries	5-63
5.7.2	Text Queries on Property Graphs	5-66
5.7.3	Navigation and Graph Pattern Matching	5-71
5.7.4	Navigation Options: CONNECT BY and Parallel Recursion	5-76
5.7.5	Pivot	5-79
5.7.6	SQL-Based Property Graph Analytics	5-80
5.7.6.1	Shortest Path Examples	5-81
5.7.6.2	Collaborative Filtering Overview and Examples	5-84
5.8	Creating Property Graph Views on an RDF Graph	5-90
5.9	Oracle Flat File Format Definition	5-93

5.9.1	About the Property Graph Description Files	5-93
5.9.2	Edge File	5-93
5.9.3	Vertex File	5-95
5.9.4	Encoding Special Characters	5-97
5.9.5	Example Property Graph in Oracle Flat File Format	5-98
5.9.6	Converting an Oracle Database Table to an Oracle-Defined Property Graph Flat File	5-98
5.9.7	Converting CSV Files for Vertices and Edges to Oracle-Defined Property Graph Flat Files	5-101

6 Property Graph Query Language (PGQL)

6.1	Creating a Property Graph using PGQL	6-1
6.2	Pattern Matching with PGQL	6-3
6.3	Edge Patterns Have a Direction with PGQL	6-4
6.4	Vertex and Edge Labels with PGQL	6-5
6.5	Variable-Length Paths with PGQL	6-5
6.6	Aggregation and Sorting with PGQL	6-5
6.7	Executing PGQL Queries Against the In-Memory Graph Server (PGX)	6-6
6.7.1	Getting Started with PGQL	6-6
6.7.2	Supported PGQL Features	6-8
6.7.2.1	Limitations on Quantifiers	6-8
6.7.2.2	Limitations on WHERE and COST Clauses in Quantified Patterns	6-8
6.7.3	Java APIs for Executing CREATE PROPERTY GRAPH Statements	6-9
6.7.4	Python APIs for Executing CREATE PROPERTY GRAPH Statements	6-9
6.7.5	Java APIs for Executing SELECT Queries	6-10
6.7.5.1	Executing SELECT Queries Against a Graph in the In-memory Graph Server (PGX)	6-10
6.7.5.2	Executing SELECT Queries Against a PGX Session	6-11
6.7.5.3	Iterating Through a Result Set	6-11
6.7.5.4	Printing a Result Set	6-13
6.7.6	Java APIs for Executing UPDATE Queries	6-14
6.7.6.1	Updatability of Graphs Through PGQL	6-14
6.7.6.2	Executing UPDATE Queries against a Graph in the in-memory Graph Server (PGX)	6-15
6.7.6.3	Executing UPDATE Queries Against a PGX Session	6-16
6.7.6.4	Altering the Underlying Schema of a Graph	6-16
6.7.7	Security Tools for Executing PGQL Queries	6-17
6.7.7.1	Using Bind Variables	6-17
6.7.7.2	Using Identifiers in a Safe Manner	6-18
6.7.8	Best Practices for Tuning PGQL Queries	6-19
6.7.8.1	Memory Allocation	6-19

6.7.8.2	Parallelism	6-19
6.7.8.3	Query Plan Explaining	6-20
6.8	Executing PGQL Queries Directly Against Oracle Database	6-20
6.8.1	Executing PGQL Queries Against Property Graph Schema Tables	6-21
6.8.1.1	PGQL Features Supported	6-22
6.8.1.2	Creating Property Graphs through CREATE PROPERTY GRAPH Statements	6-24
6.8.1.3	Dropping Property Graphs through DROP PROPERTY GRAPH Statements	6-31
6.8.1.4	Using the oracle.pg.rdbms.pgql Java Package to Execute PGQL Queries	6-32
6.8.1.5	Using the Python Client to Execute PGQL Queries	6-98
6.8.1.6	Performance Considerations for PGQL Queries	6-103
6.8.2	Executing PGQL Queries Against Property Graph Views	6-104
6.8.2.1	PGQL Features Supported in Property Graph Views	6-105
6.8.2.2	PGQL Limitations in Property Graph Views	6-106
6.8.2.3	Performance Considerations for PGQL Queries	6-107
6.8.2.4	Creating a Property Graph View	6-109
6.8.2.5	Executing PGQL SELECT Queries	6-111
6.8.2.6	Dropping A Property Graph View	6-114

7 Graph Visualization Application

7.1	About the Graph Visualization Application	7-1
7.2	How does the Graph Visualization Application Work	7-1
7.3	Using the Graph Visualization Application	7-2
7.3.1	Graph Visualization Modes	7-3
7.3.2	Graph Visualization Settings	7-3
7.3.3	Using the Geographical Layout	7-6
7.3.4	Using Live Search	7-8
7.3.5	Using URL Parameters to Control the Graph Visualization Application	7-9
7.4	REST Endpoints for the Graph Visualization Application	7-9
7.4.1	Login	7-10
7.4.2	List Graphs	7-10
7.4.3	Run a PGQL Query	7-11
7.4.4	Get User	7-13
7.4.5	Asynchronous REST Endpoints	7-13
7.4.5.1	Run a PGQL Query Asynchronously	7-13
7.4.5.2	Check a Query Completion	7-14
7.4.5.3	Cancel a Query Execution	7-14
7.4.5.4	Retrieve a Query Result	7-15
7.5	Kerberos Enabled Authentication for the Graph Visualization Application	7-17
7.5.1	Prerequisite Requirements for Kerberos Authentication	7-17

8 Using the Machine Learning Library (PgxML) for Graphs

8.1	Using the DeepWalk Algorithm	8-1
8.1.1	Loading a Graph	8-2
8.1.2	Building a Minimal DeepWalk Model	8-3
8.1.3	Building a Customized DeepWalk Model	8-3
8.1.4	Training a DeepWalk Model	8-4
8.1.5	Getting the Loss Value For a DeepWalk Model	8-5
8.1.6	Computing Similar Vertices for a Given Vertex	8-5
8.1.7	Computing Similar Vertices for a Vertex Batch	8-6
8.1.8	Storing a Trained DeepWalk Model	8-7
8.1.8.1	Storing a Trained Model in Another Database	8-8
8.1.9	Loading a Pre-Trained DeepWalk Model	8-9
8.1.9.1	Loading a Pre-Trained Model From Another Database	8-9
8.1.10	Destroying a DeepWalk Model	8-11
8.2	Using the Supervised GraphWise Algorithm	8-11
8.2.1	Loading a Graph	8-12
8.2.2	Building a Minimal GraphWise Model	8-13
8.2.3	Advanced Hyperparameter Customization	8-14
8.2.4	Training a Supervised GraphWise Model	8-16
8.2.5	Getting the Loss Value For a Supervised GraphWise Model	8-17
8.2.6	Inferring the Vertex Labels for a Supervised GraphWise Model	8-17
8.2.7	Evaluating the Supervised GraphWise Model Performance	8-18
8.2.8	Inferring Embeddings for a Supervised GraphWise Model	8-19
8.2.8.1	Inferring Embeddings for a Model in Another Database	8-20
8.2.9	Storing a Trained Supervised GraphWise Model	8-21
8.2.10	Loading a Pre-Trained Supervised GraphWise Model	8-21
8.2.11	Destroying a Supervised GraphWise Model	8-22
8.2.12	Explaining a Prediction of a Supervised GraphWise Model	8-23
8.3	Using the Unsupervised GraphWise Algorithm	8-25
8.3.1	Loading a Graph	8-26
8.3.2	Building a Minimal Unsupervised GraphWise Model	8-27
8.3.3	Advanced Hyperparameter Customization	8-28
8.3.4	Training a Unsupervised GraphWise Model	8-29
8.3.5	Getting the Loss Value for a Unsupervised GraphWise Model	8-29
8.3.6	Inferring Embeddings for a Unsupervised GraphWise Model	8-30
8.3.7	Storing a Unsupervised GraphWise Model	8-31
8.3.8	Loading a Pre-Trained Unsupervised GraphWise Model	8-32
8.3.9	Destroying a Unsupervised GraphWise Model	8-32

8.4	Using the Pg2vec Algorithm	8-33
8.4.1	Loading a Graph	8-34
8.4.2	Building a Minimal Pg2vec Model	8-35
8.4.3	Building a Customized Pg2vec Model	8-36
8.4.4	Training a Pg2vec Model	8-37
8.4.5	Getting the Loss Value For a Pg2vec Model	8-37
8.4.6	Computing Similar Graphlets for a Given Graphlet	8-38
8.4.7	Computing Similar for a Graphlet Batch	8-39
8.4.8	Inferring a Graphlet Vector	8-40
8.4.9	Inferring Vectors for a Graphlet Batch	8-41
8.4.10	Storing a Trained Pg2vec Model	8-41
8.4.11	Loading a Pre-Trained Pg2vec Model	8-42
8.4.12	Destroying a Pg2vec Model	8-43

9 OPG_APIS Package Subprograms

9.1	OPG_APIS.ANALYZE_PG	9-2
9.2	OPG_APIS.CF	9-4
9.3	OPG_APIS.CF_CLEANUP	9-7
9.4	OPG_APIS.CF_PREP	9-9
9.5	OPG_APIS.CLEAR_PG	9-10
9.6	OPG_APIS.CLEAR_PG_INDICES	9-11
9.7	OPG_APIS.CLONE_GRAPH	9-11
9.8	OPG_APIS.COUNT_TRIANGLE	9-12
9.9	OPG_APIS.COUNT_TRIANGLE_CLEANUP	9-13
9.10	OPG_APIS.COUNT_TRIANGLE_PREP	9-14
9.11	OPG_APIS.COUNT_TRIANGLE_RENUM	9-16
9.12	OPG_APIS.CREATE_EDGES_TEXT_IDX	9-17
9.13	OPG_APIS.CREATE_PG	9-18
9.14	OPG_APIS.CREATE_PG_SNAPSHOT_TAB	9-19
9.15	OPG_APIS.CREATE_PG_TEXTIDX_TAB	9-21
9.16	OPG_APIS.CREATE_STAT_TABLE	9-22
9.17	OPG_APIS.CREATE_SUB_GRAPH	9-23
9.18	OPG_APIS.CREATE_VERTICES_TEXT_IDX	9-24
9.19	OPG_APIS.DROP_EDGES_TEXT_IDX	9-26
9.20	OPG_APIS.DROP_PG	9-26
9.21	OPG_APIS.DROP_PG_VIEW	9-27
9.22	OPG_APIS.DROP_VERTICES_TEXT_IDX	9-27
9.23	OPG_APIS.ESTIMATE_TRIANGLE_RENUM	9-28
9.24	OPG_APIS.EXP_EDGE_TAB_STATS	9-30
9.25	OPG_APIS.EXP_VERTEX_TAB_STATS	9-31

9.26	OPG_APIS.FIND_CC_MAPPING_BASED	9-32
9.27	OPG_APIS.FIND_CLUSTERS_CLEANUP	9-33
9.28	OPG_APIS.FIND_CLUSTERS_PREP	9-34
9.29	OPG_APIS.FIND_SP	9-36
9.30	OPG_APIS.FIND_SP_CLEANUP	9-37
9.31	OPG_APIS.FIND_SP_PREP	9-38
9.32	OPG_APIS.GET_BUILD_ID	9-39
9.33	OPG_APIS.GET_GEOMETRY_FROM_V_COL	9-39
9.34	OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS	9-41
9.35	OPG_APIS.GET_LATLONG_FROM_V_COL	9-42
9.36	OPG_APIS.GET_LATLONG_FROM_V_T_COLS	9-43
9.37	OPG_APIS.GET_LONG_LAT_GEOMETRY	9-44
9.38	OPG_APIS.GET_LATLONG_FROM_V_COL	9-45
9.39	OPG_APIS.GET_LONGLAT_FROM_V_T_COLS	9-46
9.40	OPG_APIS.GET_SCN	9-47
9.41	OPG_APIS.GET_VERSION	9-47
9.42	OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL	9-48
9.43	OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS	9-49
9.44	OPG_APIS.GRANT_ACCESS	9-50
9.45	OPG_APIS.IMP_EDGE_TAB_STATS	9-51
9.46	OPG_APIS.IMP_VERTEX_TAB_STATS	9-52
9.47	OPG_APIS.PR	9-54
9.48	OPG_APIS.PR_CLEANUP	9-56
9.49	OPG_APIS.PR_PREP	9-57
9.50	OPG_APIS.PREPARE_TEXT_INDEX	9-58
9.51	OPG_APIS.RENAME_PG	9-58
9.52	OPG_APIS.SPARSIFY_GRAPH	9-59
9.53	OPG_APIS.SPARSIFY_GRAPH_CLEANUP	9-61
9.54	OPG_APIS.SPARSIFY_GRAPH_PREP	9-62

10 OPG_GRAPHOP Package Subprograms

10.1	OPG_GRAPHOP.POPULATE_SKELETON_TAB	10-1
------	-----------------------------------	------

Part II In-Memory Graph Server (PGX) Advanced User Guide

11 Configuring the In-Memory Graph Server (PGX)

11.1	Configuration Parameters for the Graph Server (PGX) Engine	11-1
11.1.1	Configuration of the Graph Server (PGX) Run-Time Parameters	11-11
11.1.2	Specifying the Configuration File to the In-Memory Graph Server (PGX)	11-14

11.1.3	Memory Consumption by the Graph Server (PGX)	11-15
11.1.3.1	Memory Management	11-15
11.2	Configuration Parameters for Connecting to the Graph Server (PGX)	11-17
11.3	Configuration Parameters for the Graph Client	11-17

12 Graphs Management

12.1	Loading a Graph Into the Graph Server (PGX)	12-1
12.1.1	API for Loading Graphs into Memory	12-1
12.1.2	Graph Configuration Options	12-2
12.1.3	Preloading a Graph	12-9
12.1.4	Data Loading Security Best Practices	12-10
12.1.5	Data Format Support Matrix	12-10
12.1.6	Immutability of Loaded Graphs	12-11
12.2	Publishing a Graph	12-11
12.3	Publishing a Preloaded Graph	12-15
12.4	Deleting a Graph	12-16

13 Namespaces and Sharing

13.1	Defining Graph Names	13-1
13.2	Retrieving Graphs by Name	13-1
13.3	Checking Used Names	13-2
13.4	Property Name Resolution and Graph Mutations	13-2

14 PGX Programming Guides

14.1	Design of the Graph Server (PGX) API	14-3
14.2	Data Types and Collections in the Graph Server (PGX)	14-4
14.2.1	Using Collections and Maps	14-7
14.2.1.1	Collection Data Types	14-7
14.2.1.2	Map Data Types	14-11
14.2.2	Using Datetime Data Types	14-15
14.2.2.1	Loading Datetime Data	14-16
14.2.2.2	Specifying Custom Datetime Formats	14-17
14.2.2.3	APIs for Accessing Datetime Data	14-19
14.2.2.4	Querying Datetime Data Using PGQL	14-19
14.2.2.5	Accessing Datetimes from PGQL Result Sets	14-21
14.3	Handling Asynchronous Requests in Graph Server (PGX)	14-23
14.3.1	Blocking Operation	14-23
14.3.2	Chaining Operation	14-24
14.3.3	Cancelling Operation	14-25

14.3.4	Handling Concurrent Asynchronous Operations	14-25
14.4	Graph Client Sessions	14-26
14.5	Graph Mutation and Subgraphs	14-27
14.5.1	Altering Graphs	14-28
14.5.1.1	Loading Or Removing Additional Vertex or Edge Providers	14-28
14.5.2	Simplifying and Copying Graphs	14-31
14.5.3	Transposing Graphs	14-32
14.5.4	Undirecting Graphs	14-33
14.5.5	Advanced Multi-Edge Handling	14-33
14.5.5.1	Picking	14-34
14.5.5.2	Merging	14-35
14.5.5.3	StrategyBuilder in General	14-36
14.5.6	Creating a Subgraph	14-37
14.5.7	Creating a Bipartite Subgraph	14-37
14.5.8	Creating a Sparsified Subgraph	14-38
14.6	Managing Transient Data	14-38
14.6.1	Managing Transient Properties	14-38
14.6.2	Managing Collections and Scalars	14-40
14.7	Graph Versioning	14-41
14.7.1	Configuring the Snapshots Source	14-42
14.7.2	Creating a Snapshot via Refreshing	14-42
14.7.3	Creating a Snapshot via ChangeSet	14-44
14.7.4	Checking Out the Latest Snapshots of a Graph	14-46
14.7.5	Checking Out Different Snapshots of a Graph	14-46
14.7.6	Directly Loading a Specific Snapshot of a Graph	14-47
14.8	Labels and Properties	14-48
14.8.1	Setting and Getting Property Values	14-49
14.8.2	Getting Label Values	14-50
14.9	Filter Expressions	14-50
14.9.1	Syntax	14-51
14.9.2	Type System	14-56
14.9.3	Path Finding Filters	14-56
14.9.4	Subgraph Filters	14-56
14.9.5	Operations on Filter Expressions	14-57
14.9.5.1	Defining Filter Expressions	14-57
14.9.5.2	Defining Result Set Filters	14-58
14.9.5.3	Creating a Subgraph from PGQL Result Set	14-59
14.9.5.4	Defining Collection Filters	14-60
14.9.5.5	Creating a Subgraph from Collection Filters	14-60
14.9.5.6	Combining Filter Expressions	14-61
14.10	Advanced Task Scheduling Using Execution Environments	14-62

14.10.1	Enterprise Scheduler Configuration Guide	14-63
14.10.2	Enabling Enterprise Scheduler Features	14-65
14.10.3	Retrieving and Inspecting the Execution Environment	14-65
14.10.4	Modifying and Submitting Tasks Under an Updated Environment	14-66
14.10.5	Using Lambda Syntax	14-67
14.11	Admin API	14-68
14.11.1	Get a Server Instance	14-68
14.11.2	Get Inspection Data	14-68
14.11.3	Get Active Sessions	14-69
14.11.4	Get Cached Graphs	14-71
14.11.5	Get Published Graphs	14-72
14.11.6	Get Currently Loading Graphs	14-72
14.11.7	Get Tasks	14-73
14.11.8	Get Available Memories	14-73
14.12	PgxFrames Tabular Data-Structure	14-73
14.12.1	Loading a PgxFrame from a Database	14-74
14.12.2	Loading a PgxFrame from Client-Side Data	14-76
14.12.3	Printing the Content of a PgxFrame	14-80
14.12.4	Destroying a PgxFrame	14-81
14.12.5	Storing a PgxFrame to a Database	14-81
14.12.6	Loading and Storing Vector Properties	14-82
14.12.7	Flattening Vector Properties	14-83
14.12.8	Union of PGX Frames	14-84
14.12.9	Joining PGX Frames	14-85
14.12.10	PgxFrame Helpers	14-86
14.12.11	PgxFrame-PgqlResultSet Conversions	14-88
14.12.12	Creating a Graph from Multiple PgxFrame Objects	14-89

15 Working with Files Using the Graph Server (PGX)

15.1	Loading Graph Data from Files	15-1
15.1.1	Graph Configuration for Loading from File	15-2
15.1.2	Specifying the File Path	15-7
15.1.3	Supported File Access Protocols	15-7
15.1.4	Plain Text Formats	15-8
15.1.4.1	Comma-Separated Values (CSV)	15-10
15.1.4.2	Adjacency List (ADJ_LIST)	15-13
15.1.4.3	Edge List (EDGE_LIST)	15-13
15.1.4.4	Two Tables (TWO_TABLES)	15-15
15.1.4.5	Flat File (FLAT_FILE)	15-16
15.1.5	XML File Formats	15-18

15.1.6	Binary File Formats	15-19
15.2	Loading Graph Data in Parallel from Multiple Files	15-25
15.3	Exporting Graphs Into a File	15-27
15.3.1	Exporting a Graph to Disk	15-28
15.4	Exporting a Graph into Multiple Files	15-29

16 Log Management in the Graph Server (PGX)

16.1	Configuring Log4j Logging	16-1
------	---------------------------	------

Part III Supplementary Information for Property Graph Support

A Handling Property Graphs Using a Two-Tables Schema

A.1	Preparing the Two-Tables Schema	A-2
A.2	Storing Data in a Property Graph Using a Two-Tables Schema	A-4
A.3	Reading Data from a Property Graph Using a Two-Tables Schema	A-7

B About Property Graph Data Formats

B.1	GraphSON Data Format	B-1
B.2	GraphML Data Format	B-2
B.3	GML Data Format	B-2
B.4	Oracle Flat File Format	B-3

C Mapping Graph Server Roles to Default Privileges

D Disabling Transport Layer Security (TLS) in Graph Server

Index

List of Figures

1-1	Simple Property Graph Example	1-3
1-2	Three-Tier Property Graph Architecture	1-5
1-3	Two-Tier Property Graph Architecture	1-6
1-4	Graph Visualization Login	1-20
1-5	PGQL on Graph Server (PGX)	1-23
1-6	PGQL on Database	1-24
1-7	Enabling Accessibility Mode in the Graph Visualization Application	1-44
4-1	Graph Server (PGX) Design	4-3
4-2	Session and Transient Properties	4-4
4-3	Remote Server Mode	4-4
4-4	PGX as a Library	4-5
4-5	Edges Matching <code>src.prop == 10</code>	4-46
4-6	Graph Created by the Simple Filter	4-47
4-7	Edges Matching the <code>outDegree</code> Filter	4-48
4-8	Graph Created by the <code>outDegree</code> Filter	4-48
5-1	Phones Graph for Collaborative Filtering	5-85
6-1	PGQL on Property Graph Schema Tables in Oracle Database (RDBMS)	6-21
6-2	PGQL on Property Graph Views in Oracle Database	6-104
7-1	Query Visualization	7-2
7-2	Graph Visualization Settings Window	7-4
7-3	Highlights Options for Vertices	7-5
7-4	Geographical Layout	7-6
7-5	Setting Geographical Layout	7-7
7-6	Selecting the Coordinates for the Geographical layout	7-8
8-1	Pg2vec - Visualization of Two Similar Graphlets	8-39
14-1	Picking Strategy	14-35
14-2	Merging Strategy	14-36

List of Tables

1-1	Graph Size Estimator	1-5
1-2	Overview of Tasks to Get Started with Property Graphs	1-7
1-3	Components in the Oracle Graph Server and Client Deployment	1-8
3-1	Metadata Tables for PG Views	3-1
4-1	Privileges and Roles in Oracle Database	4-6
4-2	Advanced Access Configuration Options	4-17
4-3	API for Checking Graph Permissions	4-18
4-4	Allowed Permissions	4-22
4-5	Overview of Built-In Algorithms	4-39
4-6	Configuration Parameters for the In-Memory Graph Server (PGX)	4-55
4-7	Fields for Each UDF	4-70
5-1	Edge File Record Format	5-94
5-2	Vertex File Record Format	5-96
5-3	Special Character Codes in the Oracle Flat File Format	5-97
6-1	CREATE PROPERTY GRAPH Statement Support	6-3
6-2	Type Casting Support in PGQL (From and To Types)	6-23
6-3	PGQL Translation and Execution Options	6-63
6-4	PGQL Statement Modification Options	6-94
7-1	Available URL Parameters	7-9
7-2	Parameters	7-10
7-3	Query Parameters	7-11
7-4	Location of WEB-INF/web.xml file	7-18
11-1	Configuration Parameters for the Graph Server (PGX) Engine	11-2
11-2	Graph Server (PGX) Run-Time Parameters	11-11
11-3	Configuration Parameters for the Graph Client	11-17
12-1	Graph Config JSON Fields	12-2
12-2	Provider Configuration JSON file Options	12-4
12-3	Property Configuration	12-6
12-4	Loading Configuration	12-7
12-5	Error Handling Configuration	12-8
12-6	Data Format Support Matrix	12-10
14-1	PGX API Interface	14-1
14-2	Overview of Data types	14-5
14-3	Overview of Datetime Data Types in PGX	14-15
14-4	Default Temporal Formats	14-52

14-5	Session Information Options	14-70
14-6	Graph Information	14-72
14-7	Mapping between In-Place and Out-Place Operations	14-74
15-1	Loading from File - Graph Configuration Options	15-2
15-2	CSV Specific Options	15-5
15-3	Mapping between PGX Property Type and Flat File value_type	15-17
15-4	Type Encoding	15-19
15-5	File Layout	15-20
15-6	Integer Vertex Keys	15-21
15-7	Long Vertex Keys	15-21
15-8	String Vertex Keys	15-21
15-9	String Key Element Layout	15-21
15-10	Primitive Type Layout	15-22
15-11	Vector Property Layout	15-22
15-12	String Type Layout	15-22
15-13	String Dictionary Layout	15-22
15-14	String Dictionary Element Layout	15-23
15-15	Vertex Labels Layout	15-23
15-16	Shared Pools Layout	15-23
15-17	Type == Enum	15-24
15-18	Type == Prefix	15-24
15-19	String Table for Shared Pools	15-24
15-20	Property Names Layout	15-24
15-21	Files CompressionScheme	15-27
15-22	Graph Configuration when Exporting Graph into Multiple Files	15-27
C-1	Mapping Graph Server Roles to Default Privileges	C-1

Preface

This document provides conceptual and usage information about Oracle Database support for working with property graph data.

- [Audience](#)
- [Documentation Accessibility](#)
- [Related Documents](#)
- [Conventions](#)

Audience

This document is intended for database and application developers in an Oracle Database environment.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Related Documents

For more information, see the following documents:

- *Oracle Spatial Developer's Guide*
- *Oracle Database Graph Developer's Guide for RDF Graph*
- *Oracle Spatial GeoRaster Developer's Guide*
- *Oracle Spatial Topology and Network Data Model Developer's Guide*
- *Oracle Big Data Spatial and Graph User's Guide and Reference*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for This Guide

The following changes apply to property graph support that is shipped with Oracle Graph Server and Client.

Oracle Graph Server and Client is required for using the property graph feature of Oracle Database (see [Oracle Graph Server and Client Installation](#)), and is released four times a year.

New Features

Significant New Features in Oracle Graph Server and Client 21.3

- Oracle Graph Server and Client installation is now supported on Oracle Linux 8.
- Added support for PGQL 1.4.

 **Note:**

Graph Server and Client 21.3.0 is backwards compatible with older versions of PGQL (1.0, 1.1, 1.2 and 1.3).

See [PGQL 1.4 Specification](#) for more information.

- Added API support to load property graph views by name.
See [Loading a Graph into the Graph Server \(PGX\) from a Property Graph View](#) for more information.
- Added support for the execution of recursive path queries against property graph views.
See [PGQL Features Supported in Property Graph Views](#) for more information on the supported variable-length path finding goals and horizontal aggregations in recursive queries.
- Added `GNNExplainer` support to leverage Graph ML explainability for Supervised GraphWise models in PgxML Library.
See [Explaining a Prediction of a Supervised GraphWise Model](#) for more information.
- Added support for creating a `PgxFrame` from client-side data. Functionality also added to create `PgxFrame` from `Pandas` dataframe.
See [Loading a PgxFrame from Client-Side Data](#) for more information.
- Added support for creating a graph from `PgxFrame(s)`.
See [Creating a Graph from Multiple PgxFrame Objects](#) for more information.
- Added new API methods to check graph permissions and to get the user name and the role name of the current user.
See [Checking Graph Permissions Using API](#) for more information.

- Added Python client support for PGQL on RDBMS Java library. See [Using the Python Client to Execute PGQL Queries](#) for more information.
- Added the following enhancements to the Graph Visualization application:
 - Added support for Oracle Database Kerberos authentication. See [Kerberos Enabled Authentication for the Graph Visualization Application](#) for more information.
 - Allow configuration of the PGQL driver (Graph Server (PGX) or Database) during login. See [Configuring Advanced Options for PGQL Driver Selection](#) for more information.
- Added support for Apache HDFS on Cloudera CDH7.

Migrating Property Graph Applications to Oracle Database 21c

From Release 21c onwards, Oracle Graph Server and Client must be installed separately. It is recommended to remove the older property graph libraries from `$ORACLE_HOME`. See [Uninstalling Previous Versions of Property Graph Libraries](#) for more details.

Deprecated Features

- **PL/SQL API OPG_APIS.GET_SCN Function**
The PL/SQL API OPG_APIS.GET_SCN function is deprecated. Instead, to retrieve the current SCN (system change number), use the DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER function:

```
SELECT dbms_flashback.get_system_change_number FROM DUAL;
```
- **Two-Table Support**
Support for the two-table format described in [Handling Property Graphs Using a Two-Tables Schema](#) was deprecated in 19c and will be removed in a future release.
- **Apache Tinkerpop API Support**
Apache Tinkerpop API support for Oracle Database was deprecated in 19c and is scheduled to be removed in a future release.
- **OraclePgqlResultSet**
The `oracle.pg.rdbms.OraclePgqlResultSet` interface was deprecated in 19c and will be removed in a future release. Instead, use the standardized interface `oracle.pgql.lang.ResultSet` to retrieve values from a PGQL result set.
- **Oracle NoSQL Database Support**
Property Graph support for data stored in Oracle NoSQL Database is deprecated and will be removed in a future release.

Desupported Features

- Graph property text search based on Apache Solr/Lucene is desupported. Instead, use Oracle Text or PGQL query expressions.
- The PGX property type DATE is desupported. Instead, use LOCAL_DATE or TIMESTAMP.
- Support for the Apache Groovy-based shell was deprecated in 19c and is now desupported.
- Support for Apache HBASE and Apache HDFS on Cloudera CDH5 is desupported.

Part I

Getting Started with Oracle Property Graphs

Part I provides the fundamental information to get you started on the property graph feature of Oracle Database.

This part covers the following:

- Overview of the property graph features
- Installing and configuring Oracle Graph Server and Client
- Graph data modeling using the in-memory graph server (PGX)
- Querying graph data using PGQL
- Visualizing property graphs using the Graph Visualization Application
- Applying machine learning algorithms to analyze relationships in graph data

Part I contains the following chapters:

- [Property Graph Support Overview](#)
- [Quick Starts for Using Oracle Property Graph](#)
This chapter contains quick start tutorials and other resources to help you get started on working with Oracle property graphs.
- [Property Graph Views on Oracle Database Tables](#)
You can create property graph views over data stored in Oracle Database. You can perform various graph analytics operations using PGQL on these views.
- [Using the In-Memory Graph Server \(PGX\)](#)
The in-memory Graph server of Oracle Graph supports a set of analytical functions.
- [Using the Property Graph Schema](#)
This chapter provides conceptual and usage information about creating, storing, and working with property graph data in an Oracle Database environment.
- [Property Graph Query Language \(PGQL\)](#)
PGQL is a SQL-like query language for property graph data structures that consist of *vertices* that are connected to other vertices by *edges*, each of which can have key-value pairs (properties) associated with them.
- [Graph Visualization Application](#)
The Graph Visualization application enables interactive exploration and visualization of property graphs. It can also visualize graphs stored in the database.
- [Using the Machine Learning Library \(PgXML\) for Graphs](#)
The in-memory graph server (PGX) provides a machine learning library `oracle.pgx.api.ml.lib`, which supports graph-empowered machine learning algorithms.
- [OPG_APIS Package Subprograms](#)
The OPG_APIS package contains subprograms (functions and procedures) for working with property graphs in an Oracle database.
- [OPG_GRAPHOP Package Subprograms](#)
The OPG_GRAPHOP package contains subprograms for various operations on property graphs in an Oracle database.

1

Property Graph Support Overview

This chapter provides an overview of Oracle Graph support for property graph features.

- [Introduction to Property Graphs](#)
Property graphs give you a different way of looking at your data.
- [Before You Begin with Oracle Property Graphs](#)
Before you begin to create an Oracle Property Graph, you may need to adhere to one or more of the prerequisites explained in this section.
- [Oracle Graph Server and Client Installation](#)
This section explains the various operations that you must perform to install, uninstall or upgrade Oracle Graph Server (PGX). It also includes the steps to install Oracle Graph Client.
- [Setting Up Transport Layer Security](#)
The graph server (PGX), by default, allows only encrypted connections using Transport Layer Security (TLS). TLS requires the server to present a server certificate to the client and the client must be configured to trust the issuer of that certificate.
- [Adding Permissions to Publish the Graph](#)
There are two ways by which you can view any graph in your graph server (PGX) session in the graph visualization application.
- [Security Best Practices with Graph Data](#)
Several security-related best practices apply when working with graph data.
- [Interactive Graph Shell](#)
Both the Oracle Graph server and client packages contain an interactive command-line application for interacting with all the Java APIs of the product, locally or on remote computers.
- [Developing Applications Using Graph Server Functionality as a Library](#)
The graph functions available with the graph server (PGX) can be used as a library in your application.
- [Storing Graphs in Oracle Database and Loading Graphs into Memory](#)
You can work with graphs in **two-tier mode** (graph client connects directly to Oracle Database), or **three-tier mode** (graph client connects to the graph server (PGX) on the middle-tier, which then connects to Oracle Database).
- [Using Oracle Graph with the Autonomous Database](#)
Oracle Graph with the Autonomous Database allows you to create property graphs from data in your Autonomous Database.
- [Migrating Property Graph Applications from Before Release 21c](#)
If you are migrating from a previous version of Oracle Spatial and Graph to Release 21c, you may need to make some changes to existing property graph-related applications.
- [Upgrading From Graph Server and Client 20.4.x to 21.x](#)
If you are upgrading from Graph Server and Client 20.4.x to 21.x version, you may need to create new roles in database and migrate authorization rules from `pgx.conf` file to the database. Also, starting from Graph Server and Client Release 21.1, TLS is enforced at the time of the RPM file installation.

- [Using the Graph Zeppelin Interpreter Client](#)
Oracle Graph provides an interpreter client implementation for Apache Zeppelin. This tutorial topic explains how to perform simple operations using the graph Zeppelin interpreter client.
- [About Oracle Graph Server and Client Accessibility](#)
This section provides information on the accessibility features for Oracle Graph Server and Client.

1.1 Introduction to Property Graphs

Property graphs give you a different way of looking at your data.

You can model your data as a graph by making data entities **vertices** in the graph, and relationships between them as **edges** in the graph. For example, in a bank customer accounts can be vertices, and cash transfer relationships between them can be edges.

When you view your data as a graph, you can analyze your data based on the connections and relationships between them. You can run graph analytics algorithms like PageRank to measure the relative importance of data entities based on the relationships between them, for example, links between webpages.

- [What Are Property Graphs?](#)
- [About the Property Graph Feature of Oracle Database](#)
The Property Graph feature delivers advanced graph query and analytics capabilities in Oracle Database.

1.1.1 What Are Property Graphs?

A property graph consists of a set of objects or **vertices**, and a set of arrows or **edges** connecting the objects. Vertices and edges can have multiple properties, which are represented as key-value pairs.

Each vertex has a unique identifier and can have:

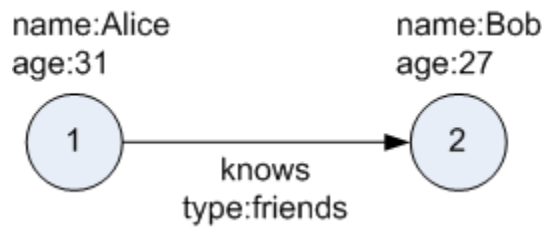
- A set of outgoing edges
- A set of incoming edges
- A collection of properties

Each edge has a unique identifier and can have:

- An outgoing vertex
- An incoming vertex
- A text label that describes the relationship between the two vertices
- A collection of properties

For vertices and edges, each property is identified with a unique name.

The following figure illustrates a very simple property graph with two vertices and one edge. The two vertices have identifiers 1 and 2. Both vertices have properties `name` and `age`. The edge is from the outgoing vertex 1 to the incoming vertex 2. The edge has a text label `knows` and a property `type` identifying the type of relationship between vertices 1 and 2.

Figure 1-1 Simple Property Graph Example

A property graph can have self-edges (that is, an edge whose source and destination vertex are the same), as well as multiple edges between the same source and destination vertices.

A property graph can also have different types of vertices and edges in the same graph. For example a graph can have a set of vertices with label `Person` and a set of vertices with label `Place`, with different properties relevant to these two sets of vertices.

The property graph data model is similar to the W3C standards-based Resource Description Framework (RDF) graph data model; however, the property graph data model is simpler and less precise than RDF.

The property graph data model features and analytic APIs make property graphs a good candidate for use cases such as these:

- Identifying influencers in a social network
- Predicting trends and customer behavior
- Discovering relationships based on pattern matching
- Identifying clusters to customize campaigns

 **Note:**

The property graph data model that Oracle supports at the database side does not allow labels for vertices. However, you can treat the value of a designated vertex property as one or more labels.

Related Topics

- [Specifying Labels for Vertices](#)

1.1.2 About the Property Graph Feature of Oracle Database

The Property Graph feature delivers advanced graph query and analytics capabilities in Oracle Database.

This feature supports graph operations, indexing, queries, search, and in-memory analytics.

Graphs manage networks of linked data as vertices, edges, and properties of the vertices and edges. Graphs are commonly used to model, store, and analyze relationships found in social networks, cybersecurity, utilities and telecommunications, life sciences and clinical data, and knowledge networks.

Typical graph analyses encompass graph traversal, recommendations, finding communities and influencers, and pattern matching. Industries including telecommunications, life sciences and healthcare, security, media, and publishing can benefit from graphs.

The property graph features of Oracle Special and Graph support those use cases with the following capabilities:

- A scalable graph database
- Developer-based APIs based upon PGQL and Java graph APIs
- Text search and query through integration with Oracle Text
- A parallel, in-memory graph server (PGX) for running graph queries and graph analytics
See [In-Memory Graph Server \(PGX\)](#) for more information.
- A fast, scalable suite of social network analysis functions that include ranking, centrality, recommender, community detection, and path finding
- Parallel bulk load and export of property graph data in Oracle-defined flat files format
- A powerful Graph Visualization (GraphViz) application
- Notebook support through integration with Apache Zeppelin
- [In-Memory Graph Server \(PGX\)](#)
- [Options for Property Graph Architecture](#)

1.1.2.1 In-Memory Graph Server (PGX)

The in-memory graph server layer enables you to analyze property graphs using parallel in-memory execution. It provides over 50 analytic functions. Examples of the categories and specific functions include:

- Centrality - Degree Centrality, Eigenvector Centrality, PageRank, Betweenness Centrality, Closedness Centrality
- Component and Community - Strongly Connected Components (Tarjan's and Kosaraju's). Weakly Connected Components
- Twitter's Who-To-Follow, Label Propagation.
- Path Finding - Single source all destination (Bellman-Ford), Dijkstra's shortest path, Hop Distance (Breadth-first search)
- Community Evaluation - Coefficient (Triangle Counting), Conductance, Modularity, Adamic-Adar counter.

See [Using the In-Memory Graph Server \(PGX\)](#) for more information on the in-memory graph server (PGX).

1.1.2.2 Options for Property Graph Architecture

You have two architecture options when using the property graph feature of Oracle Database:

- [Run Graph Query and Analytics in the In-Memory Graph Server \(PGX\) \(3-Tier\)](#)
- [Load the Graph into Oracle Database \(2-Tier\)](#)

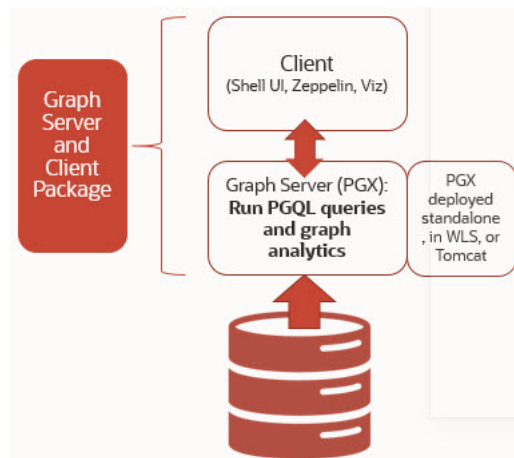
Both options let you use the Property Graph Query Language (PGQL).

Run Graph Query and Analytics in the In-Memory Graph Server (PGX) (3-Tier)

You can load your property graph into the in-memory graph server, which has a specialized architecture for graph computations. All query and analytics operations on this graph can be executed in-memory in the graph server. This graph can be created directly from relational tables or loaded from the property graph schema that stores the graph in the database. You can modify the graph in memory (insert, update, and delete vertices and edges, and create new properties for results of executing an algorithm). The graph server does not write the modifications back to the relational tables.

The in-memory graph server (PGX) typically in a server separate from the database, and can run standalone, or in a container like Oracle WebLogic Server or Apache Tomcat. This approach (load your property graph into the in-memory graph server) uses a three-tier architecture, as shown in the following figure.

Figure 1-2 Three-Tier Property Graph Architecture



Property Graph Sizing Recommendations

You can compute the memory required by the in-memory graph server (PGX) by using this calculator, [Graph Size Estimator](#).

For example, the following table shows the memory estimated by the calculator for the given input:

Table 1-1 Graph Size Estimator

Number of vertices	Number of Edges	Properties per Vertex	Properties per Edge	Estimated graph size
10M	100M	<ul style="list-style-type: none"> 4 - Integer Type 1 - String Type(15 characters) 	<ul style="list-style-type: none"> 4 - Integer Type 1 - String Type(15 characters) 	15 GB

Table 1-1 (Cont.) Graph Size Estimator

Number of vertices	Number of Edges	Properties per Vertex	Properties per Edge	Estimated graph size
100M	1B	<ul style="list-style-type: none"> 4 - Integer Type 1 - String Type(15 characters) 	<ul style="list-style-type: none"> 4 - Integer Type 1 - String Type(15 characters) 	140 GB



Note:

- Reading a graph into memory can take upto twice the amount of memory needed to represent it in memory. So when you calculate the memory required for running PGX it is recommended that you double the amount of memory of the estimated graph size.
- **CPU Processors:** The recommended number of CPU processors for a graph with 10M vertices and 100M edges is 2-4 processors, and up to 16 processors for more compute-intensive workloads. Increasing CPU processors will improve performance.

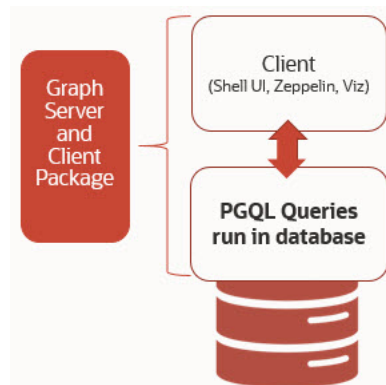
Load the Graph into Oracle Database (2-Tier)

If you do not need to load the graph into the in-memory graph server, you can use another approach: create a property graph from data in relational tables, and store it in the property graph schema (VT\$ and GE\$ tables). You can then run PGQL queries on this graph.

You can load this graph into memory for running analytics algorithms and PGQL queries not supported in the database. You can configure the in-memory graph server to periodically fetch updates from the data automatically in the graph to keep the data synchronized.

This approach uses a two-tier architecture, as shown in the following figure.

Figure 1-3 Two-Tier Property Graph Architecture



1.2 Before You Begin with Oracle Property Graphs

Before you begin to create an Oracle Property Graph, you may need to adhere to one or more of the prerequisites explained in this section.

You must perform the tasks listed in [Table 1-2](#) to get started on property graphs.

Table 1-2 Overview of Tasks to Get Started with Property Graphs

Sequence	Task	Description	More Information
1	Verify Oracle Database Requirements	Conform to the following Oracle Database prerequisites: <ul style="list-style-type: none"> Oracle Database 12.2 and higher AL16UTF16 (instead of UTF8) must be specified as the NLS_NCHAR_CHARACTERSET. AL32UTF8 (UTF8) should be the default character set, but AL16UTF16 must be the NLS_NCHAR_CHARACTERSET. 	Database Compatibility and Restrictions
2	Download Oracle Graph Server and Client	Download Oracle Graph Server and Client from Oracle Software Delivery Cloud or from Oracle Technology Network .	Downloading Oracle Graph Server and Client
3	Install the PL/SQL patch in your Oracle Database	Upgrade the PL/SQL Graph packages in your Oracle Database.	Installing PL/SQL Packages in Oracle Database
4	Install Oracle Graph Server	Install Oracle Graph server, which is available as a separate downloadable package.	Installing Oracle Graph Server
5	Download Oracle Graph Client	Install Oracle Graph Client to work with property graphs.	Installing the Java Client
6	Set up transport layer security	Configure the graph server and client to trust the self-signed certificate.	Setting Up Transport Layer Security
7	Add permissions to publish the graph	Grant permissions to publish graphs.	Adding Permissions to Publish the Graph

- [Database Compatibility and Restrictions](#)
- [Downloading Oracle Graph Server and Client](#)
- [Installing PL/SQL Packages in Oracle Database](#)

Oracle Graph Server and Client will work with Oracle Database 12.2 onward. However, you must install the updated PL/SQL packages that are part of the Oracle Graph Server and Client download.

1.2.1 Database Compatibility and Restrictions

Oracle Graph Server and Client will work with Oracle Database 12.2 onward. This includes working with the family of Oracle Autonomous Database -- all versions of Oracle Autonomous Data Warehouse (shared), Oracle Autonomous Database (shared), and Oracle Autonomous Database (dedicated).

For details, including any limitations and actions you should take to address them, see "[Database Compatibility Matrix for Oracle Graph Server and Client](#)".

1.2.2 Downloading Oracle Graph Server and Client

You can download **Oracle Graph Server and Client** from [Oracle Software Delivery Cloud](#) or from [Oracle Technology Network](#).

[Table 1-3](#) summarizes all the files contained in the Oracle Graph Server and Client deployment.

<ver> denoted in the file name in the [Table 1-3](#) reflects the downloaded Oracle Graph Server and Client version.

Table 1-3 Components in the Oracle Graph Server and Client Deployment

File	Component	Description
oracle-graph-<ver>.rpm	Oracle Graph Server	An rpm file to deploy Oracle Graph Server.
oracle-graph-client-<ver>.zip	Oracle Graph Client	A zip file containing Oracle Graph Client.
oracle-graph-zepplin- interpreter-<ver>.zip	Oracle Graph Apache Zeppelin Client	A zip file containing libraries to use Apache Zeppelin to work with Oracle Graph.
oracle-graph-hdfs-connector- <ver>.zip	Oracle Graph HDFS Connector	A zip file containing libraries to connect Oracle Graph Server with the Apache Hadoop Distributed Filesystem (HDFS).
oracle-graph-sqlcl-plugin- <ver>.zip	Oracle Graph PGQL Plugin for SQLcl	A plugin for SQLcl to run PGQL queries in SQLcl.
oracle-graph-webapps-<ver>.zip	Oracle Graph Web Applications	A zip file containing .war files for deploying graph servers in an application server.

Table 1-3 (Cont.) Components in the Oracle Graph Server and Client Deployment

File	Component	Description
oracle-graph-plsql-<ver>.zip	Oracle Graph PL/SQL Patch	A zip file containing PL/SQL packages. It is recommended to update the PL/SQL Graph packages in your database with these packages. Instructions are in the README file.

1.2.3 Installing PL/SQL Packages in Oracle Database

Oracle Graph Server and Client will work with Oracle Database 12.2 onward. However, you must install the updated PL/SQL packages that are part of the Oracle Graph Server and Client download.



Note:

You can skip this section if you are using Graph Server and Client with Oracle Autonomous Database. You only need to create roles and assign permissions by executing step-5 and step-6 in [Basic Steps for Using an Oracle Database for Authentication](#). You can run these steps using Database Actions in Oracle Cloud Infrastructure Console.

1. Download the Oracle Graph PL/SQL patch component, which is a part of the Oracle Graph Server and Client download from [Oracle Software Delivery Cloud](#).
2. Unzip the file `oracle-graph-plsql-<ver>.zip` into a directory of your choice.
`<ver>` denotes the version downloaded for the Oracle Graph PL/SQL Patch for PL/SQL.
3. Install the PL/SQL packages:
 - There are two directories, one for users with Oracle Database 18c or below, and one for users with Oracle Database 19c or above. As a database user with DBA privileges, follow the instructions in the README.md file in the appropriate directory (that matches your database version). This has to be done for every PDB you will use the graph feature in. For example:

```
-- Connect as SYSDBA
SQL> ALTER SESSION SET CONTAINER=<YOUR_PDB_NAME>;
SQL> @opgremov.sql
SQL> @catopg.sql
```
4. Create a database user in the database for working with graphs:
 - a. As a database user with DBA privileges, create a user `<graphuser>`, and grant the necessary privileges.

- i. If you plan to use a three-tier architecture (graph queries and analytics executed in the in-memory graph server (PGX), then grant privileges as described in the following command:

```
SQL> GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW TO <graphuser>
```

- ii. If you plan to use a two-tier architecture and run graph queries in the database, then grant privileges as described in [Required Privileges for Database Users](#):

```
SQL> GRANT CREATE SESSION, ALTER SESSION, CREATE TABLE,  
CREATE PROCEDURE, CREATE TYPE, CREATE SEQUENCE, CREATE VIEW,  
CREATE TRIGGER TO <graphuser>
```

- b. As a <graphuser> in the database, check that the PL/SQL update is successful:

```
SQL> CONNECT <graphuser>/<password>  
SQL> SELECT opg_apis.get_opg_version() FROM DUAL;  
-- Should return 21.3 if you are using  
-- Graph Server and Client 21.3
```

5. Grant the appropriate roles (GRAPH_DEVELOPER or GRAPH_ADMINISTRATOR), to the database user created in step 4 for working with the graphs.

 **Note:**

- See [User Authentication and Authorization](#) for more information on authorization rules for Graph Server (PGX) and Client 21.3.
- See [Upgrading From Graph Server and Client 20.4.x to 21.x](#) for more information if you are migrating to Graph Server (PGX) and Client 21.1 from an earlier version.

```
SQL> GRANT GRAPH_DEVELOPER to <graphuser>  
SQL> GRANT GRAPH_ADMINISTRATOR to <adminuser>
```

1.3 Oracle Graph Server and Client Installation

This section explains the various operations that you must perform to install, uninstall or upgrade Oracle Graph Server (PGX). It also includes the steps to install Oracle Graph Client.

- [Installing Oracle Graph Server](#)
- [Deploying Oracle Graph Server to a Web Server](#)
- [Upgrading Oracle Graph Server](#)
- [Uninstalling Oracle Graph Server](#)
- [Installing Oracle Graph Client](#)

1.3.1 Installing Oracle Graph Server

The prerequisites for installing the Oracle Graph Server are:

- Oracle Linux 6 , 7 or 8 x64 or a similar Linux distribution such as RedHat ([Using the Machine Learning Library \(PgXML\) for Graphs](#) requires Oracle Linux 7 or later)
- Oracle JDK 8 or JDK 11

You can run Oracle Graph Server in standalone mode or using a webserver like Oracle WebLogic Server or Apache Tomcat.

The installation steps for installing Oracle Graph Server in standalone mode are as shown:

1. As a `root` user or using `sudo`, install the RPM file using the `rpm` command line utility:

```
sudo rpm -i oracle-graph-<version>.rpm
```

Where `<version>` reflects the version that you downloaded. (For example: `oracle-graph-21.3.0.0.0.x86_64.rpm`)

The `.rpm` file is the graph server.

The following post-installation steps are carried out at the time of the `RPM` file installation:

- Creation of a working directory in `/opt/oracle/graph/pgx/tmp_data`
- Creation of a log directory in `/var/log/oracle/graph`
- Installation of Python Client

 **Note:**

If Python is not installed in your system, then this step will be skipped.

- Automatic generation of self-signed TLS certificates in `/etc/oracle/graph`

 **Note:**

- You can also choose to configure and set up transport layer security (TLS) in graph server. See [Setting Up Transport Layer Security](#) for more details.
- For demonstration purposes, if you wish to disable transport layer security (TLS) in graph server, see [Disabling Transport Layer Security \(TLS\) in Graph Server](#) for more details.

2. As `root` or using `sudo`, add operating system users allowed to use the server installation to the operating system group `oraclegraph`. For example:

```
usermod -a -G oraclegraph <graphuser>
```

This adds the specified graph user to the group `oraclegraph`.

Note that `<graphuser>` must log out and log in again for this to take effect.

3. As `<graphuser>`, configure the server by modifying the files under `/etc/oracle/graph` by following the steps under [Prepare the Graph Server for Database Authentication](#).
4. Ensure that authentication is enabled for database users that will connect to the graph server, as explained in [User Authentication and Authorization](#).
5. As a `root` user or using `sudo`, start the graph server (PGX) by executing the following command:

```
sudo systemctl start pgx
```

You can verify if the graph server has started by executing the following command:

```
systemctl status pgx
```

- If the graph server has successfully started, the response may appear as:
 - `pgx.service - Oracle Graph In-Memory Server`
 - Loaded: loaded (`/etc/systemd/system/pgx.service`; disabled; vendor preset: disabled)
 - Active: active (running) since Wed 2021-01-27 10:06:06 EST; 33s ago
 - Main PID: 32127 (bash)
 - CGroup: `/system.slice/pgx.service`
 - └─32127 `/bin/bash start-server`
 - └─32176 `java -Dlog4j.configurationFile=/etc/oracle/graph/log4j2-server.xml -Doracle.jdbc.fanEnabled=false -cp /opt/oracle/graph/pgx/bin/../../pgx/server/lib/activat...`

The graph server is now ready to accept requests.

- If the graph server has not started, then you must check the log files in `/var/log/oracle/graph` for errors. Additionally, you can also run the following command to view any `systemd` errors:

```
journalctl -u pgx.service
```

Additional installation operations are required for specific use cases, such as:

- Analyze property graphs using Python (see [Installing the Python Client](#)).
- Deploy the graph server as a web application with Oracle WebLogic Server (see [Deploying to Oracle WebLogic Server](#)).
- Deploy GraphViz in Oracle WebLogic Server (see [Deploying the Graph Visualization Application in Oracle WebLogic Server](#)).
- Deploy the graph server as a web application with Apache Tomcat (see [Deploying to Apache Tomcat](#)).

For instructions to deploy the graph server in Oracle WebLogic Server or Apache Tomcat, see:

- [Deploying to Oracle WebLogic Server](#)
- [Deploying to Apache Tomcat](#)

1.3.2 Deploying Oracle Graph Server to a Web Server

You can deploy Oracle Graph Server to Apache Tomcat or Oracle WebLogic Server.

The following explains the deployment instructions:

- [Deploying to Apache Tomcat](#)
The example in this topic shows how to deploy the graph server as a web application with Apache Tomcat.
- [Deploying to Oracle WebLogic Server](#)
The example in this topic shows how to deploy the graph server as a web application with Oracle WebLogic Server.


1.3.2.1 Deploying to Apache Tomcat

The example in this topic shows how to deploy the graph server as a web application with Apache Tomcat.

The graph server will work with Apache Tomcat 9.0.x.

1. Download the Oracle Graph Webapps zip file from [Oracle Software Delivery Cloud](#). This file contains ready-to-deploy Java web application archives (.war files). The file name will be similar to this: `oracle-graph-webapps-<version>.zip`.
2. Unzip the file into a directory of your choice.
3. Locate the .war file that follows the naming pattern: `graph-server-<version>-pgx<version>.war`.
4. Configure the graph server.
 - a. Modify authentication and other server settings by modifying the `WEB-INF/classes/pgx.conf` file inside the web application archive. See [User Authentication and Authorization](#) section for more information.
 - b. Optionally, change logging settings by modifying the `WEB-INF/classes/log4j2.xml` file inside the web application archive.
 - c. Optionally, change other servlet specific deployment descriptors by modifying the `WEB-INF/web.xml` file inside the web application archive.
5. Copy the .war file into the Tomcat `webapps` directory. For example:

```
cp graph-server-<version>-pgx<version>.war $CATALINA_HOME/webapps/pgx.war
```

 **Note:**

The name you give the war file in the Tomcat `webapps` directory determines the context path of the graph server application. It is recommended naming the war file as `pgx.war`.

6. Configure Tomcat specific settings, like the correct use of TLS/encryption.
7. Ensure that port 8080 is not already in use.

8. Start Tomcat:

```
cd $CATALINA_HOME
./bin/startup.sh
```

The graph server will now listen on localhost:8080/pgx.

You can connect to the server from JShell by running the following command:

```
$ <client_install_dir>/bin/opg4j --base_url https://
localhost:8080/pgx -u <graphuser>
```

Related Topics

- [The Tomcat documentation \(select desired version\)](#)

1.3.2.2 Deploying to Oracle WebLogic Server

The example in this topic shows how to deploy the graph server as a web application with Oracle WebLogic Server.

This example shows how to deploy the graph server with Oracle WebLogic Server. Graph server supports WebLogic Server version 12.1.x and 12.2.x.

1. Download the Oracle Graph Webapps zip file from [Oracle Software Delivery Cloud](#). This file contains ready-to-deploy Java web application archives (.war files). The file name will be similar to this: oracle-graph-webapps-<version>.zip.
2. Unzip the file into a directory of your choice.
3. Locate the .war file that follows the naming pattern: graph-server-<version>-pgx<version>.war.
4. Configure the graph server.
 - a. Modify authentication and other server settings by modifying the WEB-INF/classes/pgx.conf file inside the web application archive.
 - b. Optionally, change logging settings by modifying the WEB-INF/classes/log4j2.xml file inside the web application archive.
 - c. Optionally, change other servlet specific deployment descriptors by modifying the WEB-INF/web.xml file inside the web application archive.
 - d. Optionally, change WebLogic Server-specific deployment descriptors by modifying the WEB-INF/weblogic.xml file inside the web application archive.
5. Configure WebLogic specific settings, like the correct use of TLS/encryption.
6. Deploy the .war file to WebLogic Server. The following example shows how to do this from the command line:

```
. $MW_HOME/user_projects/domains/mydomain/bin/setDomainEnv.sh
. $MW_HOME/wlserver/server/bin/setWLSEnv.sh
java weblogic.Deployer -adminurl http://localhost:7001 -username
<username> -password <password> -deploy -source <path-to-war-file>
```

- [Installing Oracle WebLogic Server](#)

1.3.2.2.1 Installing Oracle WebLogic Server

To download and install the latest version of Oracle WebLogic Server, see

<http://www.oracle.com/technetwork/middleware/weblogic/documentation/index.html>

1.3.3 Upgrading Oracle Graph Server

To upgrade the graph server, make sure the graph server is shut down, then execute the following command with the newer RPM file as an argument.

- Run the following command as a `root` user or with `sudo`:

```
sudo rpm -U oracle-graph-21.3.0.0.0.x86_64.rpm
```

1.3.4 Uninstalling Oracle Graph Server

To uninstall the graph server, make sure the graph server is shut down.

- Run the following command as a `root` user or with `sudo`:

```
sudo rpm -e oracle-graph
```

1.3.5 Installing Oracle Graph Client

This sections explains in detail the installation steps for the various clients.

- [Installing the Java Client](#)
- [Installing the Python Client](#)
- [Uninstalling the Python Client](#)
This section describes how to uninstall the Python client.
- [Enabling the Graph Visualization Application](#)
- [Deploying the Graph Visualization Application](#)
This section describes the various methods to deploy the Graph Visualization Application.
- [Installing the Graph Zeppelin Interpreter Client](#)

1.3.5.1 Installing the Java Client

The prerequisites for installing the Java client are:

- A Unix-based operation system (such as Linux) or macOS or Microsoft Windows
 - Oracle JDK 11
1. Download Oracle Graph Client 21.3 from [Oracle Software Cloud](#).
 2. Unzip the file into a directory of your choice.
 3. Configure your client to trust the self-signed server certificate. See [Configuring a Client to Trust the Self-Signed Certificate](#) for more information.

4. Connect to the graph server (PGX) using the graph shell for Java as shown:

```
cd <CLIENT_INSTALL_DIR>  
./bin/opg4j --base_url https://<host>:7007 --username <graphuser>
```

In the preceding code:

- **<CLIENT_INSTALL_DIR>**: Directory where the shell executables are located.

 **Note:**

The shell executables are generally found in `/opt/oracle/graph/bin` after server installation, and `<CLIENT_INSTALL_DIR>/bin` after the client installation.

- **<host>**: Server host
- **<graphuser>**: Database user
You will be prompted for the database password.

 **Note:**

The default graph server (PGX) port is 7007. If needed, you can configure the graph server to listen on a different port by changing the port configuration in `server.conf` file. See [Configuring the In-Memory Graph Server \(PGX\)](#) for more information.

The Java shell starts and the following command line prompt appears as shown:

```
For an introduction type: /help intro  
Oracle Graph Server Shell 21.2.0  
Variables instance, session, and analyst ready to use.  
opg4j>
```

See [Interactive Graph Shell](#) for more information on the Java client.

1.3.5.2 Installing the Python Client

To install the Python client, you must ensure that your system meets the prerequisites mentioned in [Prerequisites for Installing the Python Client](#).

You can execute the following steps to install and connect using the Python client:

1. Download the Oracle Graph Client from [Oracle Software Cloud](#).
For example, `oracle-graph-client-21.3.0.zip`.
2. Unzip the file into a directory of your choice.
3. Install the client through `pip`.

For example,

```
pip3 install --user oracle-graph-client-21.3.0.zip
```


4. Configure your client to trust the self-signed server certificate. See [Configuring a Client to Trust the Self-Signed Certificate](#) for more information.
5. Start the shell by running one of the following commands:
 - a. To connect to the PGX server instance located at `https://localhost:7007` using base URL parameter:

```
./bin/opg4py --base_url https://localhost:7007
```

You are prompted to enter your username and password.

- b. Alternatively, you can also connect to the PGX server instance located at `https://localhost:7007` with username. For example :

```
./bin/opg4py --base_url https://localhost:7007 -u <graphuser>
```

You will be prompted to enter your password.

- c. To start the client shell, and to avoid establishing a connection to any graph server:

```
./bin/opg4py --no_connect
```

The Python shell starts as shown:

```
Oracle Graph Server Shell 21.3.0  
>>>
```

- [Prerequisites for Installing the Python Client](#)

1.3.5.2.1 Prerequisites for Installing the Python Client

You must ensure that the following prerequisites are met before you install the Python client:

1. Make sure that the following softwares are installed in your system:
 - Oracle JDK 8 or later
 - Python 3.5 or laterTo verify you are using the right version of the Python client, run the following command:

```
$> python3 --version  
Python 3.6.1
```

Note:

Python 2.x is not supported.
For more information on installing Python 3 on Oracle Linux, see [Python for Oracle Linux](#).

2. Ensure that `python3-devel` is installed in your system.

```
sudo yum install python3-devel
```

1.3.5.3 Uninstalling the Python Client

This section describes how to uninstall the Python client.

To uninstall the Python client, run the following command:

```
pip3 uninstall pypgx
```

1.3.5.4 Enabling the Graph Visualization Application

There are two ways you can use the Graph Visualization application:

- **Standalone mode**
If you install the Graph Server `rpm` file, the Graph Visualization application starts up by default when you start the PGX server.
- **Custom web container mode**
You can download the `oracle-graph-webapps-<version>.zip` package which contains a web application archive (`WAR`) file. You can deploy this file into your Oracle Weblogic 12.2 (or later) or Apache Tomcat (9.x or later) web containers.
See [Deploying the Graph Visualization Application](#) for more information.

The Graph Visualization application requires the Oracle Graph Server to be installed as a prerequisite component.

See [Installing Oracle Graph Server](#) for more information.

To start the Graph Visualization application in standalone mode:

1. Start the graph server (PGX) as shown:

```
sudo systemctl start pgx
```

The Graph Visualization application starts up by default.

2. Configure your Graph Visualization application to trust the self-signed server certificate. See [Configuring a Client to Trust the Self-Signed Certificate](#) for more information.
3. Connect to your browser for running the Graph Visualization application as shown

```
https://localhost:7007/ui
```

One of the following messages may appear:

- Your connection is not private
- Your connection is not secure

Click the Continue or Accept button to proceed.

1.3.5.5 Deploying the Graph Visualization Application

This section describes the various methods to deploy the Graph Visualization Application.

- [Deploying the Graph Visualization Application in Standalone Mode](#)
You can use the instructions in this section to deploy the Graph Visualization application in a standalone mode.
- [Deploying the Graph Visualization Application to Apache Tomcat](#)
- [Deploying the Graph Visualization Application in Oracle WebLogic Server](#)
The following instructions are for deploying the Graph Visualization application in Oracle WebLogic Server 12.2.1.3. You might need to make slight modifications, as appropriate, for different versions of the Weblogic Server.
- [Configuring Advanced Options for PGQL Driver Selection](#)

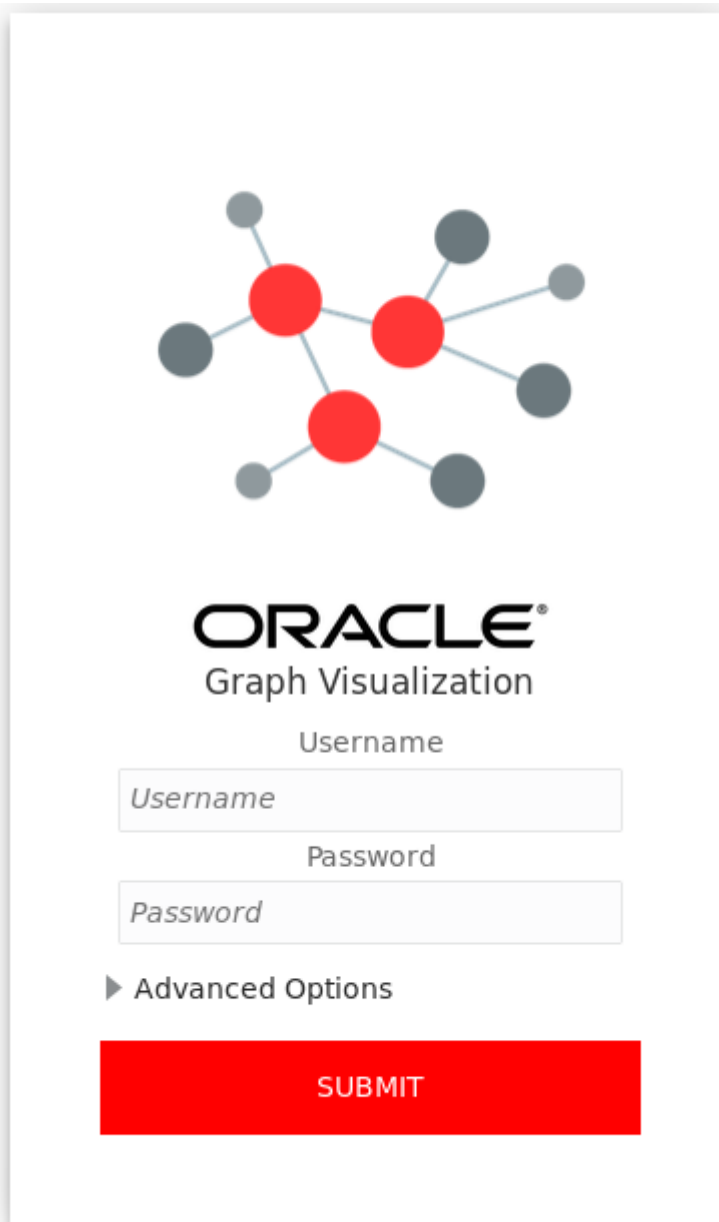
1.3.5.5.1 Deploying the Graph Visualization Application in Standalone Mode

You can use the instructions in this section to deploy the Graph Visualization application in a standalone mode.

The Graph Visualization application starts up by default when you start the graph server (PGX server).

- Navigate to `https://localhost:7007/ui` in your browser.
The Graph Visualization Login screen opens as shown:

Figure 1-4 Graph Visualization Login



The image shows a login form for Oracle Graph Visualization. At the top is a network graph icon with red and grey nodes. Below it is the Oracle logo and the text "ORACLE® Graph Visualization". The form includes two input fields: "Username" and "Password", each with a placeholder text "Username" and "Password" respectively. Below the password field is a link for "Advanced Options" with a right-pointing triangle icon. At the bottom is a large red "SUBMIT" button.

- Enter your database **Username** and **Password**.
- Select and configure the required PGQL Driver.
See [Configuring Advanced Options for PGQL Driver Selection](#) for more information.
- Click **Submit**.
You are now signed into the Graph Visualization application.

The title bar on the query visualization page displays the connection mode along with the relevant URL.

1.3.5.5.2 Deploying the Graph Visualization Application to Apache Tomcat

The following are the steps to deploy the Graph Visualization application to Apache Tomcat.

1. Download the Oracle Graph Webapps zip file from [Oracle Software Delivery Cloud](#). This file contains ready-to-deploy Java web application archives (.war files). The file name will be similar to this: `oracle-graph-webapps-<version>.zip`.
2. Configure Tomcat specific settings, like the correct use of TLS/encryption.
3. Ensure that port 8080 is not already in use.
4. Start Tomcat:

```
cd $CATALINA_HOME
./bin/startup.sh
```

The Graph Visualization application is now listening on `localhost:8080/ui`

Note:

The name you give the war file (`graphviz-<version>-pgviz<graphviz-version>-tomcat.war`) in the Tomcat webapps directory determines the context path of the graph server application. It is recommended naming the war file as `ui.war`.

5. Navigate to the Graph Visualization Application using the URL, `localhost:8080/ui` in your browser.

The Graph Visualization login page appears as shown in [Figure 1-4](#).

6. Enter your database credentials and configure the required PGQL driver.

See [Configuring Advanced Options for PGQL Driver Selection](#) for more information.

7. Click Submit.

You are now signed into the Graph Visualization application.

The title bar on the query visualization page displays the connection mode along with the relevant URL.

1.3.5.5.3 Deploying the Graph Visualization Application in Oracle WebLogic Server

The following instructions are for deploying the Graph Visualization application in Oracle WebLogic Server 12.2.1.3. You might need to make slight modifications, as appropriate, for different versions of the Weblogic Server.

1. Download the Oracle Graph Webapps zip file from [Oracle Software Delivery Cloud](#). This file contains ready-to-deploy Java web application archives (.war files). The file name will be similar to this: `oracle-graph-webapps-<version>.zip`
2. Start WebLogic Server.

```
# Start Server
cd $MW_HOME/user_projects/domains/base_domain
./bin/startWebLogic.sh
```

3. Enable tunneling.
In order to be able to deploy the Graph Visualization application WAR file over HTTP, you must enable tunneling first. Go to the WebLogic admin console (by default on <http://localhost:7001/console>). Select **Environment** (left panel) > **Servers** (left panel). Click the server that will run Graph Visualization (main panel). Select (top tab bar), check **Enable Tunneling**, and click **Save**.
4. Deploy the `graphviz-<version>-pgviz<graphviz-version>-wls.war` file.
To deploy the WAR file to WebLogic Server, use the following command, replacing the `<<...>>` markers with values matching your installation:

```
cd $MW_HOME/user_projects/domains/base_domain
source bin/setDomainEnv.sh
java weblogic.Deployer -adminurl <<admin-console-url>> -username
<<admin-user>> -password <<admin-password>> -deploy -upload <<path/
to>>/graphviz-<<version>>-pgviz<<graphviz-version>>.war
```

To undeploy, you can use the following command:

```
java weblogic.Deployer -adminurl <<admin-console-url>> -username
<<admin-user>> -password <<admin-password>> -name <<path/to>>/
graphviz-<<version>>-pgviz<<graphviz-version>>.war -undeploy
```

To test the deployment, navigate using your browser to: <https://<<fqdn-ip>>:<<port>>/ui>.

The Graph Visualization Login screen appears as shown in [Figure 1-4](#).

5. Enter your database credentials and configure the required PGQL driver.
See [Configuring Advanced Options for PGQL Driver Selection](#) for more information.
6. Click Submit.
You are now logged in and the Graph Visualization query user interface (UI) appears and the graphs from PGX are retrieved.

The title bar on the query visualization page displays the connection mode along with the relevant URL.

1.3.5.5.4 Configuring Advanced Options for PGQL Driver Selection

The Graph Visualization application can be configured to communicate either with the graph server (PGX) or to the Oracle Database. You can apply the required configuration at the time of login through the **Advanced Options** settings in the Graph Visualization login page.

You can dynamically change and configure the PGQL driver by following the instructions as appropriate for your preference:

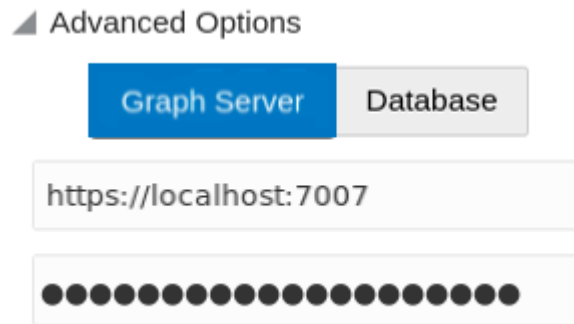
- [Configuring the Graph Visualization Application for PGQL on Graph Server \(PGX\)](#)
- [Configuring the Graph Visualization Application for PGQL on Database](#)

1.3.5.5.4.1 Configuring the Graph Visualization Application for PGQL on Graph Server (PGX)

To configure Graph Visualization application to communicate with a PGX deployment (PGQL on Graph Server):

1. Click **Advanced Options** in the Graph Visualization login page.
2. Select **Graph Server** as shown:

Figure 1-5 PGQL on Graph Server (PGX)



3. Optionally, modify your **PGX Base URL**.

 **Note:**

- By default, the Graph Visualization application connects to the graph server (PGX) using the PGX base URL defined in the `web.xml` file for your installation.
- If you wish to disable transport layer security (TLS) in graph server, see [Disabling Transport Layer Security \(TLS\) in Graph Server](#) for more details.

4. Optionally, enter **Session Id**.

When the Graph Visualization application is using PGQL on Graph Server (PGX), the application will use your Oracle Database as identity manager by default. This means that you log into the application using existing Oracle Database credentials (username and password), and the actions which you are allowed to do on the graph server are determined by the roles that have been granted to you in the Oracle Database.

 **Note:**

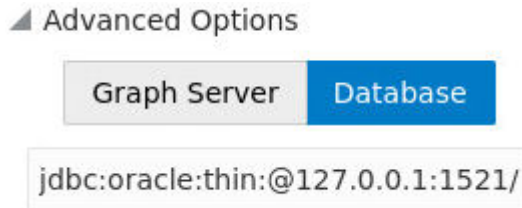
If you wish to enable Kerberos Authentication for the Graph Visualization Application, see [Kerberos Enabled Authentication for the Graph Visualization Application](#) for more information.

1.3.5.5.4.2 Configuring the Graph Visualization Application for PGQL on Database

To configure the Graph Visualization application to communicate with Oracle Database (PGQL on Database):

1. Click **Advanced Options** in the Graph Visualization login page.
2. Select **Database** as shown:

Figure 1-6 PGQL on Database



3. Optionally, modify the **JDBC URL** for your Oracle database.

 **Note:**

- By default, the Graph Visualization application connects to the database using the JDBC URL defined in the `web.xml` file for your installation.
- If you wish to enable Kerberos Authentication for the Graph Visualization Application, see [Kerberos Enabled Authentication for the Graph Visualization Application](#) for more information.
- If you wish to disable transport layer security (TLS) in graph server, see [Disabling Transport Layer Security \(TLS\) in Graph Server](#) for more details.

1.3.5.6 Installing the Graph Zeppelin Interpreter Client

To install the graph interpreter into your local Zeppelin installation:

 **Note:**

The following steps were tested with Zeppelin version 0.9, and might have to be modified with newer versions.

As a prerequisite, you must have Java 8 set in your system for installing Apache Zeppelin.

1. Download and install [Apache Zeppelin](#).
2. Download and install [Apache Groovy 2.4.x](#)
3. Copy the following libraries:

- a. Copy the libraries from the Oracle Graph Client for Apache Zeppelin package into `$ZEPPELIN_HOME/interpreter/pgx`.

```
unzip oracle-graph-zeppelin-interpreter-21.1.0.zip -d $ZEPPELIN_HOME/interpreter/pgx
```

- b. Copy the libraries inside `$GROOVY_HOME/lib` into `$ZEPPELIN_HOME/interpreter/pgx`.

```
cp $GROOVY_HOME/lib/* $ZEPPELIN_HOME/interpreter/pgx
```

4. Configure your graph Zeppelin interpreter client application to trust the self-signed server certificate. See [Configuring a Client to Trust the Self-Signed Certificate](#) for more information.
5. Restart Zeppelin.

1.4 Setting Up Transport Layer Security

The graph server (PGX), by default, allows only encrypted connections using Transport Layer Security (TLS). TLS requires the server to present a server certificate to the client and the client must be configured to trust the issuer of that certificate.

Starting with Graph Server and Client Release 21.1, the RPM file installation generates a self-signed certificate into `/etc/oracle/graph`, which the server uses to enable TLS by default. If self-signed certificates are sufficient for you to get started and if your connections are only to `localhost`, you can skip to [Configuring a Client to Trust the Self-Signed Certificate](#).

- [Generating a Self-Signed Server Certificate](#)
You can create a self-signed server certificate using the `openssl` command.
- [Configuring the Graph Server \(PGX\)](#)
You must specify the path to the server certificate and the server's private key in PEM format in the graph server (PGX) configuration file.
- [Configuring a Client to Trust the Self-Signed Certificate](#)
You must configure your client application to accept the self-signed graph server (PGX) certificate.

1.4.1 Generating a Self-Signed Server Certificate

You can create a self-signed server certificate using the `openssl` command.

The following steps show how to generate a self-signed server certificate.

1. Go to the following directory:

```
cd /etc/oracle/graph
```

2. Execute the following commands:

```
openssl req -new -newkey rsa:2048 -days 365 -nodes -x509 -subj "/C=US/ST=MyState/L=MyTown/O=MyOrganization/CN=ROOT" -keyout ca_key.pem -out ca_certificate.pem
openssl genrsa -out server_key_traditional.pem 2048
```

```
openssl pkcs8 -topk8 -in server_key_traditional.pem -inform pem -  
out server_key.pem -outform pem -nocrypt  
openssl req -new -subj "/C=US/ST=MyState/L=MyTown/O=MyOrganization/  
CN=localhost" -key server_key.pem -out server.csr  
chmod 600 server_key.pem  
openssl x509 -req -CA ca_certificate.pem -CAkey ca_key.pem -in  
server.csr -out server_certificate.pem -days 365 -CAcreateserial  
chown oraclegraph:oraclegraph server_key.pem
```

 **Note:**

- The certificate mentioned in the above example will only work for the host localhost. If you have a different domain, you must replace localhost with your domain name.
- The above self-signed certificate is valid only for 365 days.

1.4.2 Configuring the Graph Server (PGX)

You must specify the path to the server certificate and the server's private key in PEM format in the graph server (PGX) configuration file.

 **Note:**

If you deploy the graph server into your web server using the web applications download package, then this section does not apply. Please refer to the manual of your web server for instructions on how to configure TLS.

1. Edit the file at `/etc/oracle/graph/server.conf`, and specify the paths to the server certificate and the server's private key in PEM format, as shown:

```
{  
  "port": 7007,  
  "enable_tls": true,  
  "server_private_key": "/etc/oracle/graph/server_key.pem",  
  "server_cert": "/etc/oracle/graph/server_certificate.pem",  
  "enable_client_authentication": false,  
  "working_dir": "/opt/oracle/graph/pgx/tmp_data"  
}
```

2. Restart the graph server.

 **Note:**

- You should use a certificate issued by a certificate authority (CA) which is trusted by your organization. If you do not have a CA certificate, you can temporarily create a self-signed certificate and get started.
- Always use a valid certificate trusted by your organization. We do not recommend the usage of self-signed certificates for production environments.

1.4.3 Configuring a Client to Trust the Self-Signed Certificate

You must configure your client application to accept the self-signed graph server (PGX) certificate.

To configure a client to trust the self-signed certificate, the root certificate must be imported to your Java installation local trust store.

- For a Java or a Python client, you must import the root certificate to all the Java installations used by all the clients.

 **Note:**

The JShell client requires Java 11.

- For the Graph Visualization application, you must import the root certificate to the system Java installation of the environment running the graph server (PGX) or the web server serving the graph visualization application. That is, the JDK installation which is used by the OS user running the server that serves the Graph Visualization application.
- For the Graph Zeppelin interpreter client, you must import the root certificate to the Java installation used by the Zeppelin server.

You can import the root certificate as shown in the following step:

- Execute the following command as a `root` user or with `sudo`:

1. For Java 8 (make sure `JAVA_HOME` is set):

```
sudo keytool -import -trustcacerts -keystore $JAVA_HOME/jre/lib/  
security/cacerts -storepass changeit -alias pgx -file /etc/oracle/  
graph/ca_certificate.pem -noprompt
```

2. For Java 11 (make sure `JAVA11_HOME` is set):

```
sudo keytool -import -trustcacerts -keystore $JAVA11_HOME/lib/  
security/cacerts -storepass changeit -alias pgx -file /etc/oracle/  
graph/ca_certificate.pem -noprompt
```

where `changeit` is the sample keystore password. You can change this password to a password of your choice. Be sure to remember this password as you will need it to modify the certificate.

1. If you are upgrading the graph server from a previous release, you must first delete the certificate by executing the following command appropriate to your Java version. You must run the command using `sudo` or as a `root` user:

For Java 8:

```
sudo keytool -delete -alias pgx -keystore $JAVA_HOME/jre/lib/  
security/cacerts -storepass changeit
```

For Java 11:

```
sudo keytool -delete -alias pgx -keystore $JAVA11_HOME/lib/  
security/cacerts -storepass changeit
```

2. Import the new certificate as shown in the preceding [step](#).

1.5 Adding Permissions to Publish the Graph

There are two ways by which you can view any graph in your graph server (PGX) session in the graph visualization application.

When you log into the graph visualization tool in your browser, that will be a different session from your JShell session or application session. To visualize the graph you are working on in your JShell session or application session in your graph visualization session, you can perform one of the following two steps:

1. Get the session id of your working session using the `PgxSession` API, and use that session id when you log into the graph visualization application. This is the recommended option.

```
opg4j> session.getId();  
$2 ==> "898bdbc3-af80-49b7-9a5e-10ace6c9071c" //session id
```

or

2. Grant `PGX_SESSION_ADD_PUBLISHED_GRAPH` permission and then publish the graph as shown:
 - a. Grant `PGX_SESSION_ADD_PUBLISHED_GRAPH` role in the database to the user visualizing the graph as shown in the following statement:

```
GRANT PGX_SESSION_ADD_PUBLISHED_GRAPH TO <graphuser>
```

- b. Publish the graph when you are ready to visualize the graph using the `publish` API.

 **Note:**

- See [User Authentication and Authorization](#) for more information on authorization rules for Graph Server (PGX) and Client 21.1.
- See [Upgrading From Graph Server and Client 20.4.x to 21.x](#) for more information if you are migrating to Graph Server (PGX) and Client 21.3 from an earlier version.

1.6 Security Best Practices with Graph Data

Several security-related best practices apply when working with graph data.

Sensitive Information

Graph data can contain sensitive information and should therefore be treated with the same care as any other type of data. Oracle recommends the following considerations when using a graph product:

- Avoid storing sensitive information in your graph if that information is not required for analysis. If you have existing data, only model the relevant subset you need for analysis as a graph, either by applying a preprocessing step or by using subgraph and filtering techniques that are part of graph product.
- Model your graph in a way that vertex and edge identifiers are not considered sensitive information.
- Do not deploy the product into untrusted environments or in a way that gives access to untrusted client connections.
- Make sure all communication channels are encrypted and that authentication is always enabled, even if running within a trusted network.

Least Privilege Accounts

The database user account that is being used by the in-memory analyst (PGX) to read data should be a low-privilege, read-only account. PGX is an in-memory accelerator that acts as a read-only cache on top of the database, and it does not write any data back to the database.

If your application requires writing graph data and later analyzing it using PGX, make sure you use two different database user accounts for each component.

1.7 Interactive Graph Shell

Both the Oracle Graph server and client packages contain an interactive command-line application for interacting with all the Java APIs of the product, locally or on remote computers.

This interactive graph shell dynamically interprets command-line inputs from the user, executes them by invoking the underlying functionality, and can print results or process them further. The graph shell provides a lightweight and interactive way of exercising graph functionality without creating a Java application.

The graph shell is especially helpful if want to do any of the following:

- Quickly run a "one-off" graph analysis on a specific data set, rather than creating a large application
- Run getting started examples and create demos on a sample data set
- Explore the data set, trying different graph analyses on the data set interactively
- Learn how to use the product and develop a sense of what the built-in algorithms are good for
- Develop and test custom graph analytics algorithms

This graph shell is implemented on top of the Java Shell tool (JShell). As such, it inherits all features provided by JShell such as tab-completion, history, reverse search, semicolon inference, script files, and internal variables.

The graph shell connects to a graph server (PGX) specified by the `--base_url` parameter. When the `--base_url` parameter is not specified, the graph shell creates a local PGX instance, to run graph functions in the same JVM as the shell as described in [Developing Applications Using Graph Server Functionality as a Library](#).

Starting the Graph Shell

The Graph Shell uses JShell, which means the shell needs to run on Java 11 or later.

After installation, the shell executables are found in `/opt/oracle/graph/bin` after server installation, and `<CLIENT_INSTALL_DIR>/bin` after the client installation.

To launch the graph shell and connect to a graph server (PGX) enter the following in your terminal:

```
./bin/opg4j --base_url https://<host>:7007 --username <graphuser>
```

where :

- `<host>`: is the server host
- `<graphuser>`: is the database user

 **Note:**

You will be prompted for the database password.

 **Note:**

The graph server (PGX), listens on port 7007 by default. If needed, you can configure the graph server to listen on a different port by changing the port value in the server configuration file (`server.conf`). See [Configuring the In-Memory Graph Server \(PGX\)](#) for details.

When the shell has started, the following command line prompt appears:

```
opg4j>
```

If you have multiple versions of Java installed, you can easily switch between installations by setting the `JAVA_HOME` variable before starting the shell. For example:

```
export JAVA_HOME=/usr/lib/jvm/java-11-oracle
```

Command-line Options

To view the list of available command-line options, add `--help` to the `opg4j` command:

```
./bin/opg4j --help
```

Batch Execution of Scripts

The graph shell can execute a script by passing the path(s) to the script(s) to the `opg4j` command. For example:

```
./bin/opg4j /path/to/script.jsh
```

Predefined Functions

The graph shell provides the following utility functions:

- `println(String)`: A shorthand for `System.out.println(String)`.
- `loglevel(String loggerName, String levelName)`: A convenient function to set the `loglevel`.

The `loglevel` function allows you to set the log level for a logger. For example, `loglevel("ROOT", "INFO")` sets the level of the root logger to `INFO`. This causes all logs of `INFO` and higher (`WARN`, `ERROR`, `FATAL`) to be printed to the console.

Script Arguments

You can provide parameters to the script. For example:

```
./bin/opg4j /path/to/script.jsh script-arg-1 script-arg-2
```

In this example, the script `/path/to/script.jsh` can access the arguments via the `scriptArgs` system property. For example:

```
println(System.getProperty("scriptArgs"))// Prints: script-arg-1 script-arg-2
```

Staying in Interactive Mode

By default, the graph shell exits after it finishes execution. To stay in interactive mode after the script finishes *successfully*, pass the `--keep_running` flag to the shell. For example:

```
./bin/opg4j -b https://myserver.com:7007/ /path/to/script.jsh --keep_running
```

1.8 Developing Applications Using Graph Server Functionality as a Library

The graph functions available with the graph server (PGX) can be used as a library in your application.

After the rpm install of the graph server, all the jar files can be found in `/opt/oracle/graph/lib`. In this case, the server installation and the client user application are in the same machine.

For such use cases, development and testing can be done using the interactive Java shell or the Python shell in embedded (local) mode. This means a local PGX instance is created and runs in the same JVM as the client. If you start the shell without any parameters it will start a local PGX instance and run in embedded mode.

See [Using Graph Server \(PGX\) as a Library](#) for more information to obtain reference to a local PGX instance.

1.9 Storing Graphs in Oracle Database and Loading Graphs into Memory

You can work with graphs in **two-tier mode** (graph client connects directly to Oracle Database), or **three-tier mode** (graph client connects to the graph server (PGX) on the middle-tier, which then connects to Oracle Database).

Both modes for connecting to Oracle Database can be used regardless of whether the database is autonomous or not autonomous.

The database schema storing the graph must have the privileges listed in [Required Privileges for Database Users](#).

If you are using the Oracle Autonomous Database, see also [Using Oracle Graph with the Autonomous Database](#) for information about two-tier and three-tier deployments.

- [Two-Tier Mode](#)
In two-tier mode, the client graph application connects directly to Oracle Database.
- [Three-Tier Mode](#)
In three-tier mode, the client graph application connects to the graph server (PGX) in the middle tier, and the graph server connects to Oracle Database.

1.9.1 Two-Tier Mode

In two-tier mode, the client graph application connects directly to Oracle Database.

The graph is stored in the property graph schema (see [Property Graph Schema Objects for Oracle Database](#)).

You can use the PGQL DDL statement `CREATE PROPERTY GRAPH` to create a graph from database tables and store it in the property graph schema. You can then run PGQL queries on this graph from JShell shell, Java application, or the graph visualization tool.

The graph can be loaded from the property graph schema into memory in the graph server for faster processing and for using the analytics API.

1.9.2 Three-Tier Mode

In three-tier mode, the client graph application connects to the graph server (PGX) in the middle tier, and the graph server connects to Oracle Database.

The graph can be loaded from the property graph schema into the graph server, or directly from database tables into the graph server.

- **Loading a Graph from Property Graph Schema:**

Loading a graph from the property graph schema into memory in the graph server is the same as in the two-tier mode.

- **Loading a Graph Directly from Database Tables:**

When you load the graph from database tables into memory in the graph server, you create the graph in memory by directly reading data from the database tables. You do not create a graph in the property graph schema.

For more information about loading a graph from database tables into memory, see [Store the Database Password in a Keystore](#).

After the graph is loaded into memory, you can run PGQL queries on this graph from JShell shell, Java application, or the graph visualization tool. You can run graph analytics API from JShell shell or Java application, and visualize the results in the graph visualization application (GraphViz).

1.10 Using Oracle Graph with the Autonomous Database

Oracle Graph with the Autonomous Database allows you to create property graphs from data in your Autonomous Database.

When using Oracle Autonomous Database in a shared deployment, you can use Graph Studio, a powerful user interface for developing applications that use graph analysis. Using Graph Studio, you can automate the modeling of graphs from tables in Autonomous Database. You can interactively analyze and visualize the graph queries using advanced notebooks with multiple visualization options. You can execute nearly 60 built-in graph algorithms in Graph Studio to gain useful insights on your graph data. See [Using Graph Studio in Oracle Autonomous Database](#) for more information.

Alternatively, you can use Oracle Graph Server and Client with the family of Oracle Autonomous Database to create and work with property graphs.

This includes all versions of Oracle Autonomous Data Warehouse (shared), Oracle Autonomous Database (shared), and Oracle Autonomous Database (dedicated).

You can connect in two-tier mode (connect directly to Autonomous Database) or three-tier mode (connect to PGX on the middle tier, which then connects to Autonomous Database). (For basic information about two-tier and three-tier connection modes, see [Storing Graphs in Oracle Database and Loading Graphs into Memory](#).)

The database schema storing the graph must have the privileges listed in [Required Privileges for Database Users](#).

- [Two-Tier Deployments of Oracle Graph with Autonomous Database](#)
In two-tier deployments, the client graph application connects directly to the Autonomous Database.
- [Three-Tier Deployments of Oracle Graph with Autonomous Database](#)
In three-tier deployments, the client graph application connects to PGX in a middle tier, and PGX connects to the Autonomous Database.

1.10.1 Two-Tier Deployments of Oracle Graph with Autonomous Database

In two-tier deployments, the client graph application connects directly to the Autonomous Database.

1. Install Oracle Graph Client, as explained in [Installing the Java Client](#).
2. Establish a JDBC connection, as described in the [Oracle Autonomous Warehouse documentation](#).
You must download the wallet and unzip it to a secure location. You can then reference it when establishing the connection as shown in [Example 1-1](#).
3. Start the Java Shell as shown in the code:

```
/bin/opg-jshell --no_connect
```

4. Connect to your database as shown in [Example 1-1](#).

Note:

If you need to use the Graph Visualization Application, you must additionally install the Oracle Graph Server.

- See [Installing Oracle Graph Server](#) for more details.
- See [Deploying the Graph Visualization Application](#) for more details on deploying the Graph Visualization Application in Tomcat or Oracle WebLogic Server.

Example 1-1 Creating a Database Connection in a Two-Tier Graph Deployment with Autonomous Database

```
opg4j> var jdbcUrl = "jdbc:oracle:thin:@<tns_alias>?  
TNS_ADMIN=<wallet_location>" // jdbc url to the DB  
opg4j> var user = "<user>"  
opg4j> var pass = "<password>"  
opg4j> var conn = DriverManager.getConnection(jdbcUrl, user, pass) //  
connecting to the DB  
conn ==> oracle.jdbc.driver.T4CConnection@57e6cb01
```

In the preceding example:

- **<tns_alias>**: TNS alias used in `tnsnames.ora` file

- **<wallet_location>**: Path to the directory where the wallet is stored
- **<user>**: Name of the database user
- **<password>**: Password for the user

1.10.2 Three-Tier Deployments of Oracle Graph with Autonomous Database

In three-tier deployments, the client graph application connects to PGX in a middle tier, and PGX connects to the Autonomous Database.

The wallets downloaded from the Oracle Cloud Console are mainly *routing wallets*, meaning they are used to route the connection to the right database and to encrypt the connection. In most cases, they are not auto-login wallets, so they do not contain the password for the actual connection. The password usually needs to be provided separately to the wallet location.

The graph server does not support a wallet stored on the client file system or provided directly by remote users. The high level implications of this are:

- The server administrator provides the wallet and stores the wallet securely on the server's file system.
- Similar to Java EE connection pools, remote users will use that wallet when connecting. This means the server administrator trusts all remote users to use the wallet. As with any production deployments, the PGX server must be configured to enforce authentication and authorization to establish that trust.
- Remote users still need to provide a user name and password when sending a graph read request, just as with non-autonomous databases.
- You can only configure one wallet for each PGX server.

Having the same PGX server connecting to multiple Autonomous Databases is not supported. If you have that use case, start one PGX server for each Autonomous Database.

Pre-loaded graphs

To read a graph from Autonomous Database into PGX at server startup, follow the steps described in [Store the Database Password in a Keystore](#) to:

1. Create a Java Keystore containing the database password
2. Create a PGX graph configuration file describing the location and properties of the graph to be loaded
3. Update the `/opt/oracle/graph/pgx.conf` file to reference the graph configuration file

As root user, edit the service file at `/etc/systemd/system/pgx.service` and specify the environment variable under the `[Service]` directive:

```
Environment="JAVA_OPTS=-Doracle.net.tns_admin=/etc/oracle/graph/wallets"
```

Make sure that the directory (`/etc/oracle/graph/wallets` in the preceding code) is readable by the Oracle Graph user, which is the user that starts up the PGX server when using `systemd`.

In addition, edit the `ExecStart` command to specify the location of the keystore containing the password:

```
ExecStart=/bin/bash start-server --secret-store /etc/keystore.p12
```

 **Note:**

Please note that `/etc/keystore.p12` must not be password protected for this to work. Instead protect the file via file system permission that is only readable by `oraclegraph` user.

After the file is edited, reload the changes using:

```
systemctl daemon-reload
```

Finally start the server:

```
sudo systemctl start pgx
```

On-demand graph loading

To allow remote users of PGX to read from the Autonomous Database on demand, you can choose from two approaches:

- Provide the path to the wallet at server startup time via the `oracle.net.tns_admin` system property. Remote users have to provide the TNS address name, username and keystore alias (password) in their graph configuration files. The wallet is stored securely on the graph server's file system, and the server administrator trusts all remote users to use the wallet to connect to an Autonomous Database.

For example, the server administrator edits the service file at `/etc/systemd/system/pgx.service` and specifies the environment variable the under the `[Service]` directive:

```
Environment="JAVA_OPTS=-Doracle.net.tns_admin=/etc/oracle/graph/wallets"
```

and then start the server using

```
systemctl start pgx
```

The `/etc/oracle/graph/wallets/tnsnames.ora` file contains an address as follows:

```
sombrero_medium = (description= (retry_count=20)(retry_delay=3)
(address=(protocol=tcps)(port=1522)(host=adb.us-
ashburn-1.oraclecloud.com))
(connect_data=(service_name=l8lgholga0ujxsa_sombrero_medium.adwc.ora
clecloud.com))(security=(ssl_server_cert_dn="CN=adwc.uscom-
```

```
east-1.oraclecloud.com,OU=Oracle BMCS US,O=Oracle Corporation,L=Redwood  
City,ST=California,C=US"))
```

Now remote users can read data into the server by sending a graph configuration file with the following connection properties:

```
{  
  ...  
  "jdbc_url": "jdbc:oracle:thin:@sombbrero_medium",  
  "username": "hr",  
  "keystore_alias": "database1",  
  ...  
}
```

Note that the keystore still lives on the client side and should contain the password for the `hr` user referenced in the config object, as explained in [Store the Database Password in a Keystore](#). A similar approach works for Tomcat or WebLogic Server deployments.

- Use Java EE connection pools in your web application server. Remote users only have to provide the name of the datasource in their graph configuration files. The wallet and the connection credentials are stored securely in the web application server's file system, and the server administrator trusts all remote users to use a connection from the pool to connect to an Autonomous Database.

You can find instructions how to set up such a data source at the following locations:

- WebLogic Server: [Configuring a WebLogic Data Source to use ATP](#)
- Tomcat: <https://www.oracle.com/technetwork/database/application-development/jdbc/documentation/atp-5073445.html#Tomcat>

If you gave the data source the name `adb_ds`, you can reference them by sending a graph configuration file with the following connection properties:

```
{  
  ...  
  "datasource_id": "adb_ds",  
  ...  
}
```

1.11 Migrating Property Graph Applications from Before Release 21c

If you are migrating from a previous version of Oracle Spatial and Graph to Release 21c, you may need to make some changes to existing property graph-related applications.

Also note that Oracle Graph Server and Client is required for property graph applications. This can be downloaded from [Oracle Software Delivery Cloud](#) or from [Oracle Downloads](#) page.

Security-Related Changes

The Property Graph feature contains a series of enhancements to further strengthen the security of the property graph component of product. The following enhancements may require manual changes to existing graph applications so that they continue to work properly.

- **Graph configuration files now require sensitive information such as passwords to be stored in Java Keystore files**

If you use graph configuration files you are required to use Java Keystore files to store sensitive information such as passwords. (See [Store the Database Password in a Keystore](#) for how to create and reference such a keystore.)

All existing graph configuration files with secrets in them must be migrated to the keystore-based approach.

- **In a three-tier deployment, access to the PGX server file system requires a directories allowlist**

By default, the PGX server does not allow remote access to the local file system. This can be explicitly allowed, though, in `/etc/oracle/graph/pgx.conf` by setting `allow_local_filesystem` to `true`. If you set `allow_local_filesystem` to `true`, you must also specify a list of directories that are allowed to be accessed, by setting `datasource_dir_whitelist`. For example:

```
"allow_local_filesystem": true,  
"datasource_dir_whitelist": ["/scratch/data1", "/scratch/data2"]
```

This will allow remote users to read and write data on the server's file-system from and into `/scratch/data1` and `/scratch/data2`.

- **In a three-tier deployment, reading from remote locations into PGX is no longer allowed by default**

Previously, PGX allowed graph data to be read from remote locations over FTP or HTTP. This is no longer allowed by default and requires explicit opt-in by the server administrator. To opt-in, specify the `allowed_remote_loading_locations` configuration option in `/etc/oracle/graph/pgx.conf`. For example:

```
allowed_remote_loading_locations: ["*"]
```

In addition:

- The ftp and http protocols are no longer supported for loading or storing data because they are unencrypted and thus insecure.
- Configuration files can no longer be loaded from remote locations, but must be loaded from the local file system.

- **Removed shell command line options**

The following command line options of the Groovy-based `opg` shell have been removed and will no longer work:

- `--attach` - the shell no longer supports attaching to existing sessions via command line
- `--password` - the shell will prompt now for the password

Also note that the Groovy-based shell has been deprecated, and you are encouraged to use the new JShell-based shell instead (see [Interactive Graph Shell](#)).

- **Changes to PGX APIs**

The following APIs no longer return graph configuration information:

- `ServerInstance#getGraphInfo()`
- `ServerInstance#getGraphInfos()`
- `ServerInstance#getServerState()`

The REST API now identifies collections, graphs, and properties by UUID instead of a name.

The namespaces for graphs and properties are session private by default now. This implies that some operations that would previously throw an exception due to a naming conflict could succeed now.

`PgxGraph#publish()` throws an exception now if a graph with the given name has been published before.

Migrating Data to a New Database Version

Oracle Graph Server and Client works with older database versions. (See [Database Compatibility and Restrictions](#) for information.) If as part of your upgrade you also upgraded your Oracle Database, you can migrate your existing graph data that was stored using the Oracle Property Graph format by invoking the following helper script in your database after the upgrade:

```
sqlplus> EXECUTE mdsys.opg.migrate_pg_to_current(graph_name=>'mygraph');
```

The preceding example migrates the property graph *mygraph* to the current database version.

Uninstalling Previous Versions of Property Graph Libraries

This is only necessary if you are using Oracle Database versions 12.2, 18c, or 19c.

Use of the Property Graph feature of Oracle Database now requires Oracle Graph Server and Client that is installed separately. After you have completed the Graph Server and Client installation, complete the preceding migration steps (if needed), and confirmed that everything is working well, it is recommended that you remove the binaries of **older** graph installations from your Oracle Database installation by performing the following un-install steps:

1. Make sure the Property Graph mid-tier components are not in use on the target database host. For example, ensure that there is no application running which uses any files under `$ORACLE_HOME/md/property_graph`. Examples of such an application are a running PGX server on the same host as the database or a client application that references the JAR files under `$ORACLE_HOME/md/property_graph/lib`.

It is **not** necessary to shut down the database to perform the uninstall. The Oracle database itself does not reference or use any files under `$ORACLE_HOME/md/property_graph`.

2. Remove the files under `$ORACLE_HOME/md/property_graph` on your database host. On Linux, you can copy the following helper script to your database host and run it with as the DBA operating system user: `/opt/oracle/graph/scripts/patch-opg-oracle-home.sh`

1.12 Upgrading From Graph Server and Client 20.4.x to 21.x

If you are upgrading from Graph Server and Client 20.4.x to 21.x version, you may need to create new roles in database and migrate authorization rules from `pgx.conf` file to the database. Also, starting from Graph Server and Client Release 21.1, TLS is enforced at the time of the RPM file installation.

One of the main enhancements of Graph Server and Client Release 21.1 is moving the graph access permissions from the `pgx.conf` file to the database. A new set of graph roles with default permissions are created automatically in the database, at the time of the PL/SQL packages installation. See [Table C-1](#) in the appendix for more details on the default mappings.

In order to comply with this feature you must perform the database actions explained in the following sections:

Creating additional roles in the database

The roles in the database with additional privileges are created when you install the 21.x PL/SQL packages in your database as part of the upgrade. If you are not able to install the PL/SQL packages, for example if you are using an Autonomous Database, see [User Authentication and Authorization](#) for more information on manually creating these roles in the database with the default set of privileges.

Migrating authorization rules

You must execute database `GRANTS` for user-added mappings contained in the `pgx.conf` file when upgrading to 21.x.

The following examples explain the various scenarios where migration of authorization rules may or may not apply.

Example 1-2 Migrating user-added mappings to database

To migrate the following user-added mappings in `pgx.conf` file:

```
...
"authorization": [{
  "pgx_role": "GRAPH_DEVELOPER",
  "pgx_permissions": [{
    "grant": "PGX_SESSION_ADD_PUBLISHED_GRAPH"
  }],
  ...

```

```
GRANT
```

```
GRANT PGX_SESSION_ADD_PUBLISHED_GRAPH TO GRAPH_DEVELOPER
```


Example 1-3 Migrating user-added file system authorization rules to database

To migrate the following user-added file system authorization rules in `pgx.conf` file:

```
...
"file_locations": [{
  "name": "my_hdfs_graph_data",
  "location": "hdfs:/data/graphs"
}],
"authorization": [{
  "pgx_role": "GRAPH_DEVELOPER",
  "pgx_permissions": [{
    "file_location": "my_hdfs_graph_data",
    "grant": "read"
  }],
}],
...

GRANT

CREATE OR REPLACE DIRECTORY my_hdfs_graph_data AS 'hdfs:/data/graphs'
GRANT READ ON DIRECTORY my_hdfs_graph_data TO GRAPH_DEVELOPER
```

Example 1-4 User-added graph authorization rules for preloaded graphs**Note:**

No migration required for user-added graph authorization rules for preloaded graphs.

You must not migrate user-added graph authorization rules for preloaded graphs (as shown in the following code) as these rules continue to be configured in `pgx.conf` file.

```
"preload_graphs": [{
  "path": "/data/my-graph.json",
  "name": "global_graph"
}],
"authorization": [{
  "pgx_role": "GRAPH_DEVELOPER",
  "pgx_permissions": [{
    "preloaded_graph": "global_graph",
    "grant": "read"
  }],
}],
...
```

Self-signed TLS certificate now generated upon RPM installation

In Graph Server and Client 21.x the RPM installation generates a self-signed certificate into `/etc/oracle/graph`, which the server uses to enable TLS by default.

According to security best practices, access to the certificate is restricted to the `oraclegraph` operating system user. The implication of this is that you no longer can start the graph server via the `/opt/oracle/graph/pgx/bin/start-server` script, even if your user is part of the `oraclegraph` group. Instead, manage the lifecycle of the graph server via `systemctl` commands. For example:

```
sudo systemctl start pgx
```

Another possible option is to change the ownership of the certificate as shown:

```
sudo chown <youruser> /etc/oracle/graph/server_key.pem
```

Turning off TLS is not recommended as it reduces the security of your connection. However, if you must do so, see [Disabling Transport Layer Security \(TLS\) in Graph Server](#) for more details.

1.13 Using the Graph Zeppelin Interpreter Client

Oracle Graph provides an interpreter client implementation for Apache Zeppelin. This tutorial topic explains how to perform simple operations using the graph Zeppelin interpreter client.

See [Installing the Graph Zeppelin Interpreter Client](#) for more details to install the graph interpreter into your local Zeppelin installation.

Using the Interpreter

If you named the graph interpreter `pgx`, you can send paragraphs to the graph server by starting the paragraphs with the `%pgx` directive, just as with any other interpreter.

The interpreter acts like a client that talks to a remote graph server. You cannot run a graph server instance embedded inside the Zeppelin interpreter. You must provide the graph server base URL and connection information as illustrated in the following example:

```
%pgx
import oracle.pgx.api.*
import groovy.json.*

baseUrl = '<base-url>'
username = '<username>'
password = '<password>'

conn = new URL("${baseUrl}/auth/token").openConnection()
conn.setRequestProperty('Content-Type', 'application/json')
token = conn.with {
    doOutput = true
    requestMethod = 'POST'
    outputStream.withWriter { writer ->
        writer << JsonOutput.toJson([username: username, password:
password])
    }
    return new JsonSlurper().parseText(content.text).access_token
}
```

```
instance = Pgx.getInstance(baseUrl, token)
session = instance.createSession("my-session")
```

The in-memory analyst Zeppelin interpreter evaluates paragraphs in the same way that the in-memory analyst shell does, and returns the output. Therefore, any valid in-memory analyst shell script will run in the in-memory analyst interpreter, as in the following example:

```
%pgx
g_brands = session.readGraphWithProperties("/opt/data/exommerce/
brand_cat.json")
g_brands.getNumVertices()
rank = analyst.pagerank(g_brands, 0.001, 0.85, 100)
rank.getTopKValues(10)
```

The following figure shows the results of that query after you click the icon to execute it.



ID	value
Cell Phones & Accessories	0.10107276500035282
Cases	0.060593137960391966
Basic Cases	0.058782080785810285
Accessories	0.05657872563693525

As you can see in the preceding figure, the Zeppelin interpreter automatically renders the values returned by `rank.getTopKValues(10)` as a Zeppelin table, to make it more convenient for you to browse results.

Besides the property values (`getTopKValues()`, `getBottomKValues()`, and `getValues()`), the following return types are automatically rendered as table also if they are returned from a paragraph:

- `PgsqlResultSet` - the object returned by the `queryPgsql("...")` method of the `PgxGraph` class.
- `MapIterable` - the object returned by the `entries()` method of the `PgxMap` class

All other return types and errors are returned as normal strings, just as the in-memory analyst shell does.

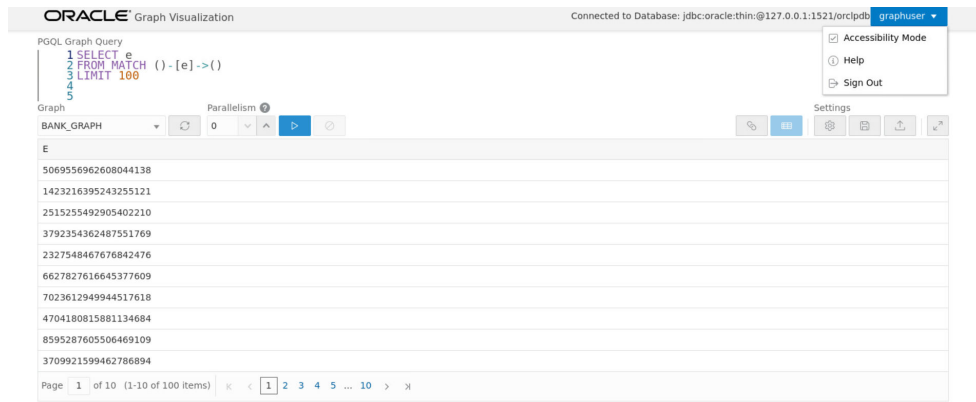
For more information about Zeppelin, see the [official Zeppelin documentation](#).

1.14 About Oracle Graph Server and Client Accessibility

This section provides information on the accessibility features for Oracle Graph Server and Client.

- For information on addressing accessibility for the Java Client and the Python Client which are installed on Oracle Linux, see [Working With Accessibility Features in Oracle Linux 7](#).
- For information on addressing accessibility for the Graph Visualization Application, which is based on Oracle JET, see [About Oracle JET and Accessibility](#).
- You can enable the accessibility mode for the Graph Visualization user interface by clicking the user menu on the top-right and selecting the **Accessibility Mode** checkbox. Once enabled, the query output is always displayed in a tabular layout as shown:

Figure 1-7 Enabling Accessibility Mode in the Graph Visualization Application



2

Quick Starts for Using Oracle Property Graph

This chapter contains quick start tutorials and other resources to help you get started on working with Oracle property graphs.

- [Quick Start: Interactively Analyze Graph Data](#)
This tutorial shows how you can quickly get started using property graph data and learn to execute PGQL queries and run graph algorithms on the data and display results.
- [QuickStart: Run Graph Analytics Using the Python Shell](#)
This tutorial shows how you can get started using property graph data using the Python shell.
- [Quick Start: Using the Python Client as a Module](#)
This section describes how to use the Python client as a module in Python applications.
- [Oracle LiveLabs Workshops for Graphs](#)
You can also explore Oracle Property Graph features using the graph workshops in Oracle LiveLabs.

2.1 Quick Start: Interactively Analyze Graph Data

This tutorial shows how you can quickly get started using property graph data and learn to execute PGQL queries and run graph algorithms on the data and display results.

The tutorials in this section are:

- [Quick Start: Create and Query a Graph in the Database, Load into In-Memory Graph Server \(PGX\) for Analytics](#)
This tutorial shows how you can get started using property graph data when you create a graph and persist it in the database. The graph can be queried in the database. This tutorial uses the JShell client.
- [Quick Start: Create, Query, and Analyze a Graph in In-Memory Graph Server \(PGX\)](#)
This tutorial shows how you can quickly get started using property graph data when using the in-memory graph server (PGX).
- [Quick Start: Executing PGQL Queries in SQLcl](#)
This tutorial provides you resources to get started on executing PGQL Queries in SQLcl.

2.1.1 Quick Start: Create and Query a Graph in the Database, Load into In-Memory Graph Server (PGX) for Analytics

This tutorial shows how you can get started using property graph data when you create a graph and persist it in the database. The graph can be queried in the database. This tutorial uses the JShell client.

See [Create and Query a Graph in the Database](#) for more information on creating and storing graphs in database.

- Convert existing relational data into a graph in the database.

- Query this graph using PGQL.

In [Load the Graph into Memory and Run Graph Analytics](#), you will run graph algorithms after loading the graph into the in-memory graph server (PGX).

- Load the graph into the in-memory graph server (PGX), run graph algorithms on this graph, and visualize results.

Prerequisites for the following quickstart are:

- An installation of Oracle Graph server (this is PGX, the in-memory graph server).
See [Oracle Graph Server and Client Installation](#) for information to download Oracle Graph Server and Client.

- An installation of Oracle Graph client

- Java 11

- The in-memory graph server can work with Java 8 or Java 11.
- The JShell client used in this example requires Java 11.

For Java downloads, see <https://www.oracle.com/technetwork/java/javase/overview/index.html>.

- Connection details for your Oracle Database. See [Database Compatibility and Restrictions](#) to identify any limitations. The Property Graph feature is supported for Oracle Database versions 12.2 and later.
- Basic knowledge about how to run commands on Oracle Database (for example, using `SQL*Plus` or `SQL Developer`).

Set up the example data

This example uses the HR (human resources) sample dataset.

- For instructions how to import that data into a user managed database, see: <https://github.com/oracle/db-sample-schemas>
- If you are using Autonomous Database, see: <https://www.thatjeffsmith.com/archive/2019/07/creating-hr-in-oracle-autonomous-database-w-sql-developer-web/>

Note that the database schema storing the graph must have the privileges listed in [Required Privileges for Database Users](#).

- [Create and Query a Graph in the Database](#)
In this section, you will use the Oracle Graph client to create a graph from relational tables and store it in the property graph schema in the database.
- [Load the Graph into Memory and Run Graph Analytics](#)

2.1.1.1 Create and Query a Graph in the Database

In this section, you will use the Oracle Graph client to create a graph from relational tables and store it in the property graph schema in the database.

Major tasks for this tutorial:

- [Start the shell](#)
- [Open a JDBC database connection](#)
- [Create a PGQL connection](#)

- [Write and execute the graph creation statement](#)
- [Run a few PGQL queries](#)

Start the shell

On the system where Oracle Graph client is installed, start the shell by as follows:

```
cd <client-install-dir>
./bin/opg4j --noconnect
```

The `--noconnect` option indicates that you are not connecting to the in-memory graph server (PGX). You will only be connecting to the database in this example.

Note that `JAVA_HOME` should be set to Java 11 before you start the shell. For example:

```
export JAVA_HOME=/usr/lib/jvm/java-11-oracle
```

See [Interactive Graph Shell](#) for details about the shell.

Open a JDBC database connection

Inside the shell prompt, use the standard JDBC Java API to obtain a database connection object. For example:

```
opg4j> var jdbcUrl = "<jdbc-url>" // for example:
jdbc:oracle:thin:@myhost:1521/myervice
opg4j> var user = "<db-user>" // for example: hr
opg4j> var pass = "<db-pass>"
opg4j> var conn = DriverManager.getConnection(jdbcUrl, user, pass)
conn ==> oracle.jdbc.driver.T4CConnection@57e6cb01
```

Connecting to an Autonomous Database works the same way: provide a JDBC URL that points to the local wallet. See [Using Oracle Graph with the Autonomous Database](#) for an example.

Create a PGQL connection

Convert the JDBC connection into a PGQL connection object. For example:

```
opg4j> conn.setAutoCommit(false)
opg4j> var pgql = PgqlConnection.getConnection(conn)
pgql ==> oracle.pg.rdbms.pgql.PgqlConnection@6fb3d3bb
```

Write and execute the graph creation statement

Using a text editor, write a `CREATE PROPERTY GRAPH` statement that describes how the HR sample data should be converted into a graph. Save this file as `create.pgql` at a location of your choice. For example:

```
CREATE PROPERTY GRAPH hr
  VERTEX TABLES (
    employees LABEL employee
    PROPERTIES ARE ALL COLUMNS EXCEPT ( job_id, manager_id,
```

```

department_id ),
  departments LABEL department
    PROPERTIES ( department_id, department_name ),
  jobs LABEL job
    PROPERTIES ARE ALL COLUMNS,
  job_history
    PROPERTIES ( start_date, end_date ),
  locations LABEL location
    PROPERTIES ARE ALL COLUMNS EXCEPT ( country_id ),
  countries LABEL country
    PROPERTIES ARE ALL COLUMNS EXCEPT ( region_id ),
  regions LABEL region
)
EDGE TABLES (
  employees AS works_for
    SOURCE employees
    DESTINATION KEY ( manager_id ) REFERENCES employees
    NO PROPERTIES,
  employees AS works_at
    SOURCE employees
    DESTINATION departments
    NO PROPERTIES,
  employees AS works_as
    SOURCE employees
    DESTINATION jobs
    NO PROPERTIES,
  departments AS managed_by
    SOURCE departments
    DESTINATION employees
    NO PROPERTIES,
  job_history AS for_employee
    SOURCE job_history
    DESTINATION employees
    LABEL for
    NO PROPERTIES,
  job_history AS for_department
    SOURCE job_history
    DESTINATION departments
    LABEL for
    NO PROPERTIES,
  job_history AS for_job
    SOURCE job_history
    DESTINATION jobs
    LABEL for
    NO PROPERTIES,
  departments AS department_located_in
    SOURCE departments
    DESTINATION locations
    LABEL located_in
    NO PROPERTIES,
  locations AS location_located_in
    SOURCE locations
    DESTINATION countries
    LABEL located_in
    NO PROPERTIES,

```



```

    countries AS country_located_in
    SOURCE countries
    DESTINATION regions
    LABEL located_in
    NO PROPERTIES
)

```

Then, back in your graph shell, execute the `CREATE PROPERTY GRAPH` statement by sending it to your PGQL connection. Replace `<path>` with the path to the directory containing the `create.pgql` file:

```

opg4j> pgql.prepareStatement(Files.readString(Paths.get("<path>/
create.pgql"))).execute()
$!6 ==> false

```

Run a few PGQL queries

Now that you have a graph named `hr`, you can use PGQL to run a few queries against it directly on the database. For example:

```

// define a little helper function that executes the query, prints the
// results and properly closes the statement
opg4j> Consumer<String> query = q -> { try(var s = pgql.prepareStatement(q))
{ s.execute(); s.getResultSet().print(); } catch(Exception e) { throw new
RuntimeException(e); } }
query ==> $Lambda$605/0x0000000100ae6440@6c9e7af2

// print the number of vertices in the graph
opg4j> query.accept("select count(v) from hr match (v)")
+-----+
| count(v) |
+-----+
| 215      |
+-----+

// print the number of edges in the graph
opg4j> query.accept("select count(e) from hr match ()-[e]->()")
+-----+
| count(e) |
+-----+
| 433      |
+-----+

// find the highest earning managers
opg4j> query.accept("select distinct m.FIRST_NAME, m.LAST_NAME, m.SALARY
from hr match (v:EMPLOYEE)-[:WORKS_FOR]->(m:EMPLOYEE) order by m.SALARY
desc")
+-----+
| m.FIRST_NAME | m.LAST_NAME | m.SALARY |
+-----+
| Steven       | King        | 24000.0  |
| Lex          | De Haan     | 17000.0  |
| Neena        | Kochhar     | 17000.0  |
| John         | Russell     | 14000.0  |

```

Karen	Partners	13500.0
Michael	Hartstein	13000.0
Alberto	Errazuriz	12000.0
Shelley	Higgins	12000.0
Nancy	Greenberg	12000.0
Den	Raphaely	11000.0
Gerald	Cambrault	11000.0
Eleni	Zlotkey	10500.0
Alexander	Hunold	9000.0
Adam	Fripp	8200.0
Matthew	Weiss	8000.0
Payam	Kaufling	7900.0
Shanta	Vollman	6500.0
Kevin	Mourgos	5800.0

```
// find the average salary of accountants in the Americas
opg4j> query.accept("select avg(e.SALARY) from hr match (e:EMPLOYEE) -
[h:WORKS_AT]-> (d:DEPARTMENT) -[:LOCATED_IN]-> (:LOCATION) -
[:LOCATED_IN]-> (:COUNTRY) -[:LOCATED_IN]-> (r:REGION) where
r.REGION_NAME = 'Americas' and d.DEPARTMENT_NAME = 'Accounting'")
+-----+
| avg(e.SALARY) |
+-----+
| 14500.0       |
+-----+
```

2.1.1.2 Load the Graph into Memory and Run Graph Analytics

Major tasks for this tutorial:

- [Load the graph from the property graph schema into memory](#)
- [Execute algorithms and query the algorithm results](#)
- [Share the Graph with Other Sessions](#)

Load the graph from the property graph schema into memory

In this section of the quickstart, you will load the graph stored in the Property Graphs schema in the database into the in-memory graph server (PGX). This will enable you to run a variety of different built-in algorithms on the graph and will also improve query performance for larger graphs.

First, start the JShell client and connect to the in-memory graph server (PGX):

```
./bin/opg4j --base_url https://<graph server host>:7007 --username
<graphuser>
```

<graphuser> is the database user you will use to for the PGX server authentication. You will be prompted for the database password.

 **Note:**

For demo purposes only, if you have set `enable_tls` to `false` in the `/etc/oracle/graph/server.conf` file you can use an `http` instead of `https` connection.

```
./bin/opg4j --base_url http://<graph server host>:7007 --username <graphuser>
```

This starts the shell and makes a connection to the graph server.

 **Note:**

Always use low-privilege read-only database user accounts for PGX, as explained in [Security Best Practices with Graph Data](#).

Next load the graph into memory in this server.

To load the graph into memory, create a PGX graph config object, using the PGX graph config builder API to do this directly in the shell.

The following example creates a PGX graph config object. It lists the properties to load into memory so that you can exclude other properties, thus reducing memory consumption.

```
Supplier<GraphConfig> pgxConfig = () -> { return
GraphConfigBuilder.forPropertyGraphRdbms()
.setName("hr")
.addVertexProperty("COUNTRY_NAME", PropertyType.STRING)
.addVertexProperty("DEPARTMENT_NAME", PropertyType.STRING)
.addVertexProperty("FIRST_NAME", PropertyType.STRING)
.addVertexProperty("LAST_NAME", PropertyType.STRING)
.addVertexProperty("EMAIL", PropertyType.STRING)
.addVertexProperty("PHONE_NUMBER", PropertyType.STRING)
.addVertexProperty("SALARY", PropertyType.DOUBLE)
.addVertexProperty("MIN_SALARY", PropertyType.DOUBLE)
.addVertexProperty("MAX_SALARY", PropertyType.DOUBLE)
.addVertexProperty("STREET_ADDRESS", PropertyType.STRING)
.addVertexProperty("POSTAL_CODE", PropertyType.STRING)
.addVertexProperty("CITY", PropertyType.STRING)
.addVertexProperty("STATE_PROVINCE", PropertyType.STRING)
.addVertexProperty("REGION_NAME", PropertyType.STRING)
.setPartitionWhileLoading(PartitionWhileLoading.BY_LABEL)
.setLoadVertexLabels(true)
.setLoadEdgeLabel(true)
.build(); }
```

Now that you have a graph config object, use the following API to read the graph into PGX:

```
opg4j> var graph = session.readGraphWithProperties(pgxConfig.get())
graph ==> PgxGraph[name=hr,N=215,E=433,created=1586996113457]
```

The session object is created for you automatically.

Execute algorithms and query the algorithm results

Now that you have the graph in memory, you can run any built-in algorithm using a single API invocation. For example, for `pagerank`:

```
opg4j> analyst.pagerank(graph)
$31==> VertexProperty[name=pagerank,type=double,graph=hr]
```

As you can see from the preceding outputs, each algorithm created a new vertex property on the graph holding the output of the algorithm. To print the most important people in the graph (according to `pagerank`), you can run the following query:

```
opg4j> session.queryPgql("select m.FIRST_NAME, m.LAST_NAME, m.pagerank
from hr match (m:EMPLOYEE) order by m.pagerank desc limit
10").print().close()
```

```
+-----+
| m.FIRST_NAME | m.LAST_NAME | m.pagerank |
+-----+
| Adam | Fripp | 0.002959240305566317 |
| John | Russell | 0.0028810951120575284 |
| Michael | Hartstein | 0.002181365227465801 |
| Alexander | Hunold | 0.002082616009054747 |
| Den | Raphaely | 0.0020378615199327507 |
| Shelley | Higgins | 0.002028946863425767 |
| Nancy | Greenberg | 0.0017419394483596667 |
| Steven | King | 0.0016622985848193119 |
| Neena | Kochhar | 0.0015252785582170803 |
| Jennifer | Whalen | 0.0014263044976976823 |
+-----+
```

Share the Graph with Other Sessions

After you load the graph into the in-memory graph server, you can use the `publish()` API to make the graph visible to other sessions, such as the graph visualization session. For example:

```
opg4j> graph.publish(VertexProperty.ALL, EdgeProperty.ALL)
```

The published graph will include any new properties you add to the graph by calling functions, such as `pagerank`.

You can use the [Graph Visualization Application](#) by navigating to `<my-server-name>:7007/ui/` in your browser.

You can connect to a particular client session by providing the session ID when you log into the Graph Visualization Application. You will then be able to visualize all graphs in the session, even if they have not been published.

```
opg4j> session
session ==> PgxSession[ID=5adf83ab-31b1-4a0e-8c08-
d6a95ba63ee0,source=pgxShell]
```

The session id is 5adf83ab-31b1-4a0e-8c08-d6a95ba63ee0.

**Note:**

You must create a server certificate to connect to the in-memory graph server (PGX) from the Graph Visualization Application. See [Setting Up Transport Layer Security](#) for more details.

2.1.2 Quick Start: Create, Query, and Analyze a Graph in In-Memory Graph Server (PGX)

This tutorial shows how you can quickly get started using property graph data when using the in-memory graph server (PGX).

This is for use cases where the graph is available as long as the in-memory graph server (PGX) session is active. The graph is not persisted in the database.

- Create a graph in the in-memory graph server (PGX), directly from existing relational data
- Query this graph using PGQL in the in-memory graph server (PGX)
- Run graph algorithms in the in-memory graph server (PGX) on this graph and display results

Prerequisites for the following quickstart are:

- An installation of Oracle Graph server (this is PGX, the in-memory graph server).
See [Installing Oracle Graph Server](#) for information to download Oracle Graph Server.
- An installation of Oracle Graph client.

See [Installing the Java Client](#) for information to download Oracle Graph Client.

You will authenticate yourself as the database user to the in-memory graph server, and these database credentials are used to access the database tables and create a graph.

- Java 11
 - The in-memory graph server can work with Java 8 or Java 11.
 - The JShell client used in this example requires Java 11.

For Java downloads, see <https://www.oracle.com/technetwork/java/javase/overview/index.html>.

Major tasks for this tutorial:

- [Set up the example data](#)
- [Start the shell](#)
- [Write and execute the graph creation statement](#)
- [Run a few PGQL queries](#)
- [Execute algorithms and query the algorithm results](#)
- [Share the Graph with Other Sessions](#)

Set up the example data

This example uses the HR (human resources) sample dataset.

- For instructions how to import that data into a user managed database, see: <https://github.com/oracle/db-sample-schemas>
- If you are using Autonomous Database, see: <https://www.thatjeffsmith.com/archive/2019/07/creating-hr-in-oracle-autonomous-database-w-sql-developer-web/>

Note that the database schema storing the graph must have the privileges listed in [Required Privileges for Database Users](#).

Start the shell

On the system where Oracle Graph Client is installed, start the shell as follows. This is an example of starting a shell in remote mode and connecting to the in-memory graph server (PGX):

```
./bin/opg4j --base_url https://<graph server host>:7007 --username  
<graphuser>
```

<graphuser> is the database user you will use to for the PGX server authentication. You will be prompted for the database password.

 **Note:**

For demo purposes only, if you have set `enable_tls` to `false` in the `/etc/oracle/graph/server.conf` file you can use an `http` instead of `https` connection.

```
./bin/opg4j --base_url http://<graph server host>:7007 --username  
<graphuser>
```

This starts the shell and makes a connection to the graph server.

Note that, `JAVA_HOME` should be set to Java 11 before you start the shell. For example:

```
export JAVA_HOME=/usr/lib/jvm/java-11-oracle
```

See [Interactive Graph Shell](#) for details about the shell.

Write and execute the graph creation statement

Create a graph with employees, departments, and “employee works at department”, by executing a `CREATE PROPERTY GRAPH` statement. The following statement creates a graph in the in-memory graph server (PGX):

```
opg4j> String statement =  
    "CREATE PROPERTY GRAPH hr_simplified "  
    + " VERTEX TABLES ( "  
    + "   hr.employees LABEL employee "
```

```

+ "      PROPERTIES ARE ALL COLUMNS EXCEPT ( job_id, manager_id,
department_id ), "
+ "      hr.departments LABEL department "
+ "      PROPERTIES ( department_id, department_name ) "
+ "    ) "
+ "  EDGE TABLES ( "
+ "    hr.employees AS works_at "
+ "    SOURCE KEY ( employee_id ) REFERENCES employees "
+ "    DESTINATION departments "
+ "    PROPERTIES ( employee_id ) "
+ "  )";
opg-jshell> session.executePgql(statement);

```

To get a handle to the graph, execute:

```
opg4j> PgxGraph g = session.getGraph("HR_SIMPLIFIED");
```

Run a few PGQL queries

You can use this handle to run PGQL queries on this graph. For example, to find the department that “Nandita Sarchand” works for, execute:

```

opg4j> String query =
    "SELECT dep.department_name "
+ "FROM MATCH (emp:Employee) -[:works_at]-> (dep:Department) "
+ "WHERE emp.first_name = 'Nandita' AND emp.last_name = 'Sarchand' "
+ "ORDER BY 1";
opg4j> PsqlResultSet resultSet = g.queryPgql(query);
opg4j> resultSet.print();
+-----+
| department_name |
+-----+
| Shipping        |
+-----+

```

To get an overview of the types of vertices and their frequencies, execute:

```

opg4j> String query =
    "SELECT label(n), COUNT(*) "
+ "FROM MATCH (n) "
+ "GROUP BY label(n) "
+ "ORDER BY COUNT(*) DESC";
opg4j> PsqlResultSet resultSet = g.queryPgql(query);
opg4j> resultSet.print();
+-----+
| label(n) | COUNT(*) |
+-----+
| EMPLOYEE | 107      |
| DEPARTMENT | 27      |
+-----+

```

To get an overview of the types of edges and their frequencies, execute:

```
opg4j> String query =
    "SELECT label(n) AS srcLbl, label(e) AS edgeLbl, label(m) AS
dstLbl, COUNT(*) "
    + "FROM MATCH (n) -[e]-> (m) "
    + "GROUP BY srcLbl, edgeLbl, dstLbl "
    + "ORDER BY COUNT(*) DESC";
opg4j> PsqlResultSet resultSet = g.queryPsql(query);
opg4j> resultSet.print();
```

```
+-----+
| srcLbl | edgeLbl | dstLbl | COUNT(*) |
+-----+
| EMPLOYEE | WORKS_AT | DEPARTMENT | 106 |
+-----+
```

Execute algorithms and query the algorithm results

Now that you have the graph in memory, you can run each built-in algorithms using a single API invocation. For example, for pagerank:

```
opg4j> analyst.pagerank(g)
$31==> VertexProperty[name=pagerank,type=double,graph=hr]
```

As you can see from the preceding outputs, each algorithm created a new vertex property on the graph holding the output of the algorithm. To print the most important people in the graph (according to pagerank), you can run the following query:

```
opg4j> session.queryPsql("select m.FIRST_NAME, m.LAST_NAME, m.pagerank
from HR_SIMPLIFIED match (m:EMPLOYEE) where m.FIRST_NAME = 'Nandita'
").print().close();
```

```
+-----+
| m.FIRST_NAME | m.LAST_NAME | m.pagerank |
+-----+
| Nandita | Sarchand | 0.001119402985074627 |
+-----+
```

In the following example, we order departments by their pagerank value. Departments with higher pagerank values have more employees.

```
opg4j> session.queryPsql("select m.DEPARTMENT_NAME, m.pagerank from
HR_SIMPLIFIED match (m:DEPARTMENT) order by m.pagerank
").print().close();
```

```
+-----+
| m.DEPARTMENT_NAME | m.pagerank |
+-----+
| Manufacturing | 0.001119402985074627 |
| Construction | 0.001119402985074627 |
| Contracting | 0.001119402985074627 |
| Operations | 0.001119402985074627 |
```



```

| IT Support | 0.001119402985074627 |
| NOC | 0.001119402985074627 |
| IT Helpdesk | 0.001119402985074627 |
| Government Sales | 0.001119402985074627 |
| Retail Sales | 0.001119402985074627 |
| Recruiting | 0.001119402985074627 |
| Payroll | 0.001119402985074627 |
| Treasury | 0.001119402985074627 |
| Corporate Tax | 0.001119402985074627 |
| Control And Credit | 0.001119402985074627 |
| Shareholder Services | 0.001119402985074627 |
| Benefits | 0.001119402985074627 |
| Human Resources | 0.0020708955223880596 |
| Administration | 0.0020708955223880596 |
| Public Relations | 0.0020708955223880596 |
| Marketing | 0.003022388059701493 |
| Accounting | 0.003022388059701493 |
| Executive | 0.003973880597014925 |
| IT | 0.005876865671641792 |
| Purchasing | 0.006828358208955224 |
| Finance | 0.006828358208955224 |
| Sales | 0.03347014925373134 |
| Shipping | 0.043936567164179076 |
+-----+

```

Share the Graph with Other Sessions

After you load the graph into the server, you can use the `publish()` API to make the graph visible to other sessions, such as the graph visualization session. For example:

```
opg4j> graph.publish(VertexProperty.ALL, EdgeProperty.ALL)
```

The published graph will include any new properties you add to the graph by calling functions, such as `pagerank`.

Ensure that the logged-in user has the privilege to publish graphs. You can do this by adding the privilege `PGX_SESSION_ADD_PUBLISHED_GRAPH` to the `GRAPH_DEVELOPER` role as explained in [Adding Permissions to Publish the Graph](#). We had given the `GRAPH_DEVELOPER` role to the database user in [Installing PL/SQL Packages in Oracle Database](#).

You can use the Graph Visualization Application by navigating to `<my-server-name>:7007/ui/` in your browser.

You can connect to a particular client session by providing the session ID when you log into the Graph Visualization Application. You will then be able to visualize all graphs in the session, even if they have not been published.

```
opg4j> session
session ==> PgxSession[ID=5adf83ab-31b1-4a0e-8c08-
d6a95ba63ee0,source=pgxShell]
```

The session id is `5adf83ab-31b1-4a0e-8c08-d6a95ba63ee0`.

 **Note:**

You must create a server certificate to connect to the in-memory graph server (PGX) from the Graph Visualization Application. See [Setting Up Transport Layer Security](#) for more details.

2.1.3 Quick Start: Executing PGQL Queries in SQLcl

This tutorial provides you resources to get started on executing PGQL Queries in SQLcl.

You can execute PGQL queries in SQLcl with a plugin that is available with Oracle Graph Server and Client.

See [Execute PGQL Queries in SQLcl](#) for more details.

You can also refer to PGQL Plug-in for SQLcl [PGQL Plug-in for SQLcl](#) section in the SQLcl documentation.

2.2 QuickStart: Run Graph Analytics Using the Python Shell

This tutorial shows how you can get started using property graph data using the Python shell.

As a prerequisite for this quick start, you must ensure that you have completed the following installations:

- [Installing Oracle Graph Server](#)
- [Installing the Python Client](#)

1. Start the Python shell as shown:

```
./bin/opg4py --base_url https://localhost:7007
```

You are prompted to enter your username and password.

2. Verify that the Python client is connected to a remote graph server (PGX) instance as shown:

```
Oracle Graph Server Shell 21.3.0
>>> instance
ServerInstance(embedded: False, base_url: https://localhost:7007,
version: <oracle.pgx.common.VersionInfo at 0x7fb71a1b2f68
jclass=oracle/pgx/common/VersionInfo jself=<LocalRef obj=0xadd938
at 0x7fb71a1808f0>>)
```

3. Create the graph using the graph builder Python API.

```
>>> graph = session.create_graph_builder().add_edge(1,
2).add_edge(2, 3).build("my_graph")
```

- 4. Execute any built-in algorithm on the graph. For example:

```
>>> analyst.pagerank(graph)
VertexProperty(name: pagerank, type: double, graph: my_graph)
```

- 5. Execute any PGQL queries and print the PGQL result set as shown:


```
>>> rs = session.query_pgql("select id(x), x.pagerank from match (x) on
my_graph")
>>> rs.print()
+-----+
| id(x) | pagerank          |
+-----+
| 1     | 0.0500000000000001 |
| 2     | 0.0925000000000003 |
| 3     | 0.1286250000000002 |
+-----+
```

Converting PGQL result set into pandas dataframe

Additionally, you can also convert the PGQL result set to a `pandas.DataFrame` object using the `to_pandas()` method. This makes it easier to perform various data filtering operations on the result set and it can also be used in Lambda functions. For example,

```
example_query = (
    "SELECT n.name as name, n.age as age "
    "WHERE (n)"
)
result_set = sample_graph.query_pgql(example_query)
result_df = result_set.to_pandas()

result_df['age_bin'] = result_df['age'].apply(lambda x: int(x)/20) #
create age bins based on age ranges
```

 **Note:**
To view the complete set of available Python APIs, see [Pygpx API](#).

2.3 Quick Start: Using the Python Client as a Module

This section describes how to use the Python client as a module in Python applications.

Embedded Server

You can use the python client as a module as illustrated in the following example.

 **Note:**

For this mode, the Python client and the Graph Server RPM package must be installed on the same machine.

```
import os
os.environ["PGX_CLASSPATH"] = "/opt/oracle/graph/lib/*"

import pypgx

session = pypgx.get_session()
graph = session.create_graph_builder().add_edge(1, 2).add_edge(2,
3).build("my_graph")
analyst = session.create_analyst()
analyst.pagerank(graph)
rs = session.query_pgql("select id(x), x.pagerank from match (x) on
my_graph")
rs.print()
```

To execute, save the above program into a file named `program.py` and run the following command.

```
python3 program.py
```

You will see the following output:

```
+-----+
| id(x) | pagerank |
+-----+
| 1      | 0.050000000000000001 |
| 2      | 0.092500000000000003 |
| 3      | 0.128625000000000002 |
+-----+
```

See [Converting PGQL result set into pandas dataframe](#) for more details on converting a PGQL result set into pandas dataframe.

 **Note:**

To view the complete set of available Python APIs, see [Pypgx API](#).

Remote Server

For this mode, all you need is the Python client to be installed. In your Python program, you must authenticate with the remote server before you can create a session as illustrated in the following example.

 **Note:**

Replace the `base_url`, `username`, and `password` with values to match your environment details.

```
import pypgx as pgx
import pypgx.pg.rdbms.graph_server as graph_server

base_url = "https://localhost:7007"
username = "scott"
password = "tiger"

instance = graph_server.get_instance(base_url, username, password)
session = instance.create_session("python_pgx_client")
print(session)
```

To execute, save the above program into a file named `program.py` and run the following command:

```
python3 program.py
```

After successful login, you'll see the following message indicating a PGX session was created:

```
PgxSession(id: 0bdd4828-c3cc-4cef-92c8-0fcd105416f0, name: python_pgx_client)
```

 **Note:**

To view the complete set of available Python APIs, see [Pypgx API](#).

2.4 Oracle LiveLabs Workshops for Graphs

You can also explore Oracle Property Graph features using the graph workshops in Oracle LiveLabs.

See the Oracle LiveLabs Workshop for a complete example on querying, analyzing and visualizing graphs using data stored in a free tier Autonomous Database instance. You will provision a new free tier Autonomous Database instance, load data into it, create a graph, and then query, analyze and visualize the graph.

3

Property Graph Views on Oracle Database Tables

You can create property graph views over data stored in Oracle Database. You can perform various graph analytics operations using PGQL on these views.

The `CREATE PROPERTY GRAPH` statement in PGQL can be used to create a view-like object that contains metadata about the graph. This graph can be queried using PGQL.

The property graph views are created directly over data that exists in the relational database tables. Since the graph is stored in the database tables it has a schema. This is unlike the graphs created with a flexible schema, where the data is copied from the source tables to property graph schema tables as described in [Property Graph Schema Objects for Oracle Database](#).

One of the main benefits of property graph views, is that all updates to the database tables are immediately reflected in the graph.

Metadata Tables for PG Views

Each time a `CREATE PROPERTY GRAPH` statement is executed, metadata tables are created in the user's own schema.

The following table describes the set of metadata tables that are created for each graph on executing `CREATE PROPERTY GRAPH` statement.

All columns shown underlined in the [Table 3-1](#) are part of the primary key of the table. Also all columns have a `NOT NULL` constraint.

Table 3-1 Metadata Tables for PG Views

Table Name	Description
<u>graphName_ELEM_TAB</u> <u>LE</u> \$	Metadata for graph element (vertex/edge) tables (one row per element table): <ul style="list-style-type: none">• <u>ET_NAME</u>: the name of the element table (the "alias")• <u>ET_TYPE</u>: either "VERTEX" or "EDGE"• <u>SCHEMA_NAME</u>: the name of the schema of the underlying table• <u>TABLE_NAME</u>: the name of underlying table
<u>graphName_LABEL</u> \$	Metadata on labels of element tables (one row per label; one label per element table): <ul style="list-style-type: none">• <u>LABEL_NAME</u>: the name of the label• <u>ET_NAME</u>: the name of the element table (the "alias")• <u>ET_TYPE</u>: either "VERTEX" or "EDGE"
<u>graphName_PROPERTY</u> \$	Metadata describing the columns that are exposed through a label (one row per property) <ul style="list-style-type: none">• <u>PROPERTY_NAME</u>: the name of the property• <u>ET_NAME</u>: the name of the element table (the "alias")• <u>ET_TYPE</u>: either "VERTEX" or "EDGE"• <u>LABEL_NAME</u>: the name of the label that this property belongs to• <u>COLUMN_NAME</u>: the name of the column (initially, only the case where property names equal column names is allowed)

Table 3-1 (Cont.) Metadata Tables for PG Views

Table Name	Description
graphName_ KEY \$	<p>Metadata describing a vertex/edge key (one row per column in the key)</p> <ul style="list-style-type: none"> <u>COLUMN_NAME</u>: the name of the column in the key <u>COLUMN_NUMBER</u>: the number of the column in the key For example, in KEY (a, b, c), "a" has number 1, "b" has number 2 and "c" has number 3. <u>KEY_TYPE</u>: either "VERTEX" or "EDGE" <u>ET_NAME</u>: the name of the element table (the "alias")
graphName_ SRC_DST_KEY \$	<p>Metadata describing the edge source/destination keys (one row per column of a key):</p> <ul style="list-style-type: none"> <u>ET_NAME</u>: the name of the element table (the "alias"), which is always an edge table <u>VT_NAME</u>: the name of the vertex table <u>KEY_TYPE</u>: either "EDGE_SOURCE" or "EDGE_DESTINATION" <u>ET_COLUMN_NAME</u>: the name of the key column <u>ET_COLUMN_NUMBER</u>: the number of the column in the key. For example, in KEY (a, b, c), "a" has number 1, "b" has number 2 and "c" has number 3.

 **Note:**

Currently, support is only for **SOURCE KEY (...) REFERENCES T1**. So only the edge source/destination key is stored.

Example 3-1 To create a Property Graph View

Consider the following CREATE PROPERTY GRAPH statement:

```
CREATE PROPERTY GRAPH student_network
  VERTEX TABLES(
    person
      KEY ( id )
      LABEL student
      PROPERTIES( name ),
    university
      KEY ( id )
      PROPERTIES( name )
  )
  EDGE TABLES(
    knows
      key (person1, person2)
      SOURCE KEY ( person1 ) REFERENCES person
      DESTINATION KEY ( person2 ) REFERENCES person
      NO PROPERTIES,
    person AS studentOf
      key (id, university)
      SOURCE KEY ( id ) REFERENCES person
      DESTINATION KEY ( university ) REFERENCES university
```

```

        NO PROPERTIES
    )
    OPTIONS (PG_VIEW)

```

The `OPTIONS` clause allows the creation of a property graph view instead of the creation of property graph schema graph. You must simply pass the `CREATE PROPERTY GRAPH` statement to the `execute` method:

 **Note:**

- You can create property graph views using the RDBMS Java API or through SQLcl.
- You can query property graph views using the graph visualization tool or SQLcl.
- Both creation and querying of property graph views are not supported when using Python API.

```
stmt.execute("CREATE PROPERTY GRAPH student_network ...");
```

This results in the creation of the following metadata tables:

```
SQL> SELECT * FROM STUDENT_NETWORK_ELEM_TABLE$;
```

ET_NAME	ET_TYPE	SCHEMA_NAME	TABLE_NAME
PERSON	VERTEX	SCOTT	PERSON
UNIVERSITY	VERTEX	SCOTT	UNIVERSITY
KNOWS	EDGE	SCOTT	KNOWS
STUDENTOF	EDGE	SCOTT	PERSON

```
SQL> SELECT * FROM STUDENT_NETWORK_LABEL$;
```

LABEL_NAME	ET_NAME	ET_TYPE
STUDENT	PERSON	VERTEX
UNIVERSITY	UNIVERSITY	VERTEX
KNOWS	KNOWS	EDGE
STUDENTOF	STUDENTOF	EDGE

```
SQL> SELECT * FROM STUDENT_NETWORK_PROPERTY$;
```

PROPERTY_NAME	ET_NAME	ET_TYPE	LABEL_NAME	COLUMN_NAME
NAME	PERSON	VERTEX	STUDENT	NAME
NAME	UNIVERSITY	VERTEX	UNIVERSITY	NAME

```
SQL> SELECT * FROM STUDENT_NETWORK_KEY$;
```

COLUMN_NAME	COLUMN_NUMBER	KEY_TY	ET_NAME
-------------	---------------	--------	---------


```

-----
ID                1 VERTEX PERSON
ID                1 VERTEX UNIVERSITY
PERSON1           1 EDGE KNOWS
PERSON2           2 EDGE KNOWS
ID                1 EDGE STUDENTOF
UNIVERSITY        2 EDGE STUDENTOF

SQL> SELECT * FROM STUDENT_NETWORK_SRC_DST_KEYS;

ET_NAME      VT_NAME      KEY_TYPE      ET_COLUMN_NAME
ET_COLUMN_NUMBER
-----
KNOWS        PERSON        EDGE_SOURCE
PERSON1      1
KNOWS        PERSON        EDGE_DESTINATION
PERSON2      1
STUDENTOF    PERSON        EDGE_SOURCE
ID           1
STUDENTOF    UNIVERSITY    EDGE_DESTINATION
UNIVERSITY  1

```

You can now run PGQL queries on the property graph view `student_network`.

See [Executing PGQL Queries Against Property Graph Views](#) for more details to create, query and drop property graph views.

- [Loading a Graph into the Graph Server \(PGX\) from a Property Graph View](#)
You can load a graph into the graph server (PGX) from a property graph view by name.

3.1 Loading a Graph into the Graph Server (PGX) from a Property Graph View

You can load a graph into the graph server (PGX) from a property graph view by name.

You can use the following `PgxSession` method to load a graph from a property graph view by name:

```
readGraphByName(String name, GraphSource source)
```

The arguments used in the method are as follows:

- `name`: Name of the property graph view.
- `source`: Source for the graph. In this case, `PG_VIEW`.

The `readGraphByName(String name, GraphSource source)` method reads the property graph view metadata tables and internally generates the graph configuration to load the graph. You must have `PGX_SESSION_NEW_GRAPH` permission to use this API.

For example you can load the graph from a property graph view using JShell as shown:

```
var pgview = session.readGraphByName("bankdataview", GraphSource.PG_VIEW)
$12 ==> PgxGraph[name=bankdataview,N=1000,E=5001,created=1625730942294]
```

Similarly, you can load the graph from a property graph view using Java as shown:

```
PgxGraph graph = session.readGraphByName("bankdataview",
GraphSource.PG_VIEW);
Graph: PgxGraph[name=bankdataview,N=1000,E=5001,created=1625732149262]
```

Also, you can load the graph from a property graph view using Python as shown:

```
>>> graph = session.read_graph_by_name('bankdataview', 'pg_view')
>>> graph
PgxGraph(name: bankdataview, v: 1000, e: 5001, directed: True, memory(Mb): 0)
```

4

Using the In-Memory Graph Server (PGX)

The in-memory Graph server of Oracle Graph supports a set of analytical functions.

This chapter provides examples using the in-memory Graph Server (also referred to as Property Graph In-Memory Analytics, and often abbreviated as PGX in the Javadoc, command line, path descriptions, error messages, and examples). It contains the following major topics.

- [Overview of the In-Memory Graph Server \(PGX\)](#)
The In-Memory Graph Server (PGX) is an in-memory graph server for fast, parallel graph query and analytics. The server uses light-weight in-memory data structures to enable fast execution of graph algorithms.
- [User Authentication and Authorization](#)
The Oracle Graph server (PGX) uses an Oracle Database as identity manager. Both username and password based as well as Kerberos based authentication is supported.
- [About Vertex and Edge IDs](#)
- [Reading Graphs from Oracle Database into the Graph Server \(PGX\)](#)
Once logged in to the graph server (PGX), you can now read graphs from the database into the graph server without specifying any connection information in the graph configuration.
- [Keeping the Graph in Oracle Database Synchronized with the Graph Server](#)
You can use the `FlashbackSynchronizer` API to automatically apply changes made to graph in the database to the corresponding `PgxGraph` object in memory, thus keeping both synchronized.
- [Optimizing Graphs for Read Versus Updates in the In-Memory Graph Server \(PGX\)](#)
- [Storing a Graph Snapshot on Disk](#)
After reading a graph into memory using either Java or the Shell, if you make some changes to the graph such as running the PageRank algorithm and storing the values as vertex properties, you can store this snapshot of the graph on disk.
- [Executing Built-in Algorithms](#)
The in-memory graph server (PGX) contains a set of built-in algorithms that are available as Java APIs.
- [Using Custom PGX Graph Algorithms](#)
A custom PGX graph algorithm allows you to write a graph algorithm in Java and have it automatically compiled to an efficient parallel implementation.
- [Creating Subgraphs](#)
You can create subgraphs based on a graph that has been loaded into memory. You can use filter expressions or create bipartite subgraphs based on a vertex (node) collection that specifies the left set of the bipartite graph.
- [Using Automatic Delta Refresh to Handle Database Changes](#)
You can automatically refresh (auto-refresh) graphs periodically to keep the in-memory graph synchronized with changes to the property graph stored in the property graph tables in Oracle Database (VT\$ and GE\$ tables).

- [Starting the In-Memory Graph Server \(PGX\)](#)
This section describes the commands to start and stop the in-memory graph server(PGX).
- [Connecting to the In-Memory Graph Server \(PGX\)](#)
This section explains how to connect to the in-memory graph server (PGX) running in remote mode or when deployed as a web application on Apache Tomcat or Oracle WebLogic Server.
- [Using Graph Server \(PGX\) as a Library](#)
- [User-Defined Functions \(UDFs\) in PGX](#)
User-defined functions (UDFs) allow users of PGX to add custom logic to their PGQL queries or custom graph algorithms, to complement built-in functions with custom requirements.
- [Using HAProxy for PGX Load Balancing and High Availability](#)
HAProxy is a high-performance TCP/HTTP load balancer and proxy server that allows multiplexing incoming requests across multiple web servers.

4.1 Overview of the In-Memory Graph Server (PGX)

The In-Memory Graph Server (PGX) is an in-memory graph server for fast, parallel graph query and analytics. The server uses light-weight in-memory data structures to enable fast execution of graph algorithms.

There are multiple options to load a graph into the graph server either from Oracle Database or from files.

The graph server can be deployed standalone (it includes an embedded Apache Tomcat instance), or deployed in Oracle WebLogic Server or Apache Tomcat.

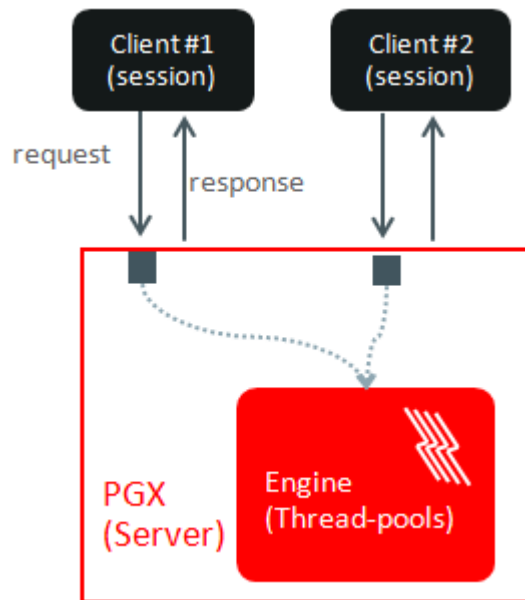
- [Design of the In-Memory Graph Server \(PGX\)](#)
- [Usage Modes of the In-memory Graph Server \(PGX\)](#)

4.1.1 Design of the In-Memory Graph Server (PGX)

The design of the in-memory graph server (PGX) is based on a Server-Client usage model. See [Usage Modes of the In-memory Graph Server \(PGX\)](#) for more details on the different graph server (PGX) execution modes.

The following figure shows the graph server (PGX) design:

Figure 4-1 Graph Server (PGX) Design



The core concepts of the graph server (PGX) design are as follows

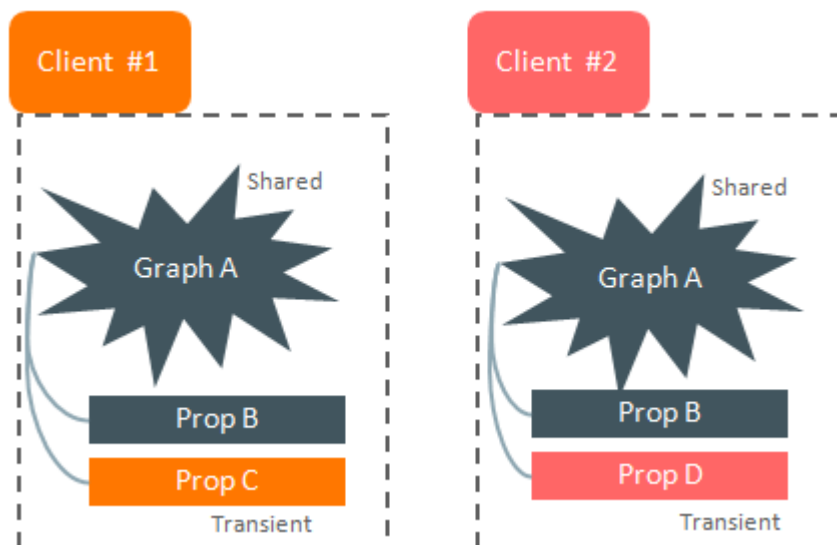
- Multiple graph clients can connect to the in-memory graph server at the same time.
- Each client request are processed by the graph server asynchronously. The client requests are queued up first and processed later, when resources are available. The client can poll the server to check if a request has been finished.
- Internally, the server maintains its own engine (thread pools) for running parallel graph algorithms and queries. The engine tries to process each analytics request concurrently with as many threads as possible.

Isolation Between Concurrent Clients

The graph server (PGX) supports data isolation between concurrent clients. Each client has its own private workspace, called session. Sessions are isolated from each other. Each client can load a graph instance into its own session, independently from other clients.

If multiple clients load the same graph instance the graph server can share one graph instance between multiple clients under the hood. Each client can add additional vertex or edge properties to a loaded graph in its own session. Such properties are transient properties, and are private to each session and not visible to another session as shown in the following figure:

Figure 4-2 Session and Transient Properties



Similarly, if a client creates a mutated version of the loaded graph, the graph server will create a private graph instance for that client.

4.1.2 Usage Modes of the In-memory Graph Server (PGX)

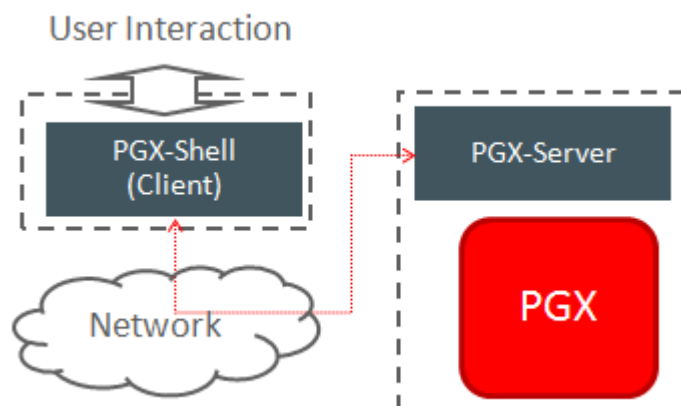
This section presents an overview of the different usage modes of the graph server (PGX). The graph server can be executed in one of the following usage modes.

Remote Server Mode

In the remote server mode, the main PGX execution engine is deployed as a RESTful application on a powerful server machine, and you can connect to it remotely from your machine using graph shell. Also, multiple clients can connect to the same graph server (PGX) at the same time and therefore the graph server is time-shared among these clients.

The following figure shows the graph server (PGX) in a remote execution mode:

Figure 4-3 Remote Server Mode



The remote server mode is useful for the following situations where you want to:

- Perform graph analysis on a large data set with a powerful server-class machine that has many cores and a large memory.
- The server-class machine is shared by multiple clients.

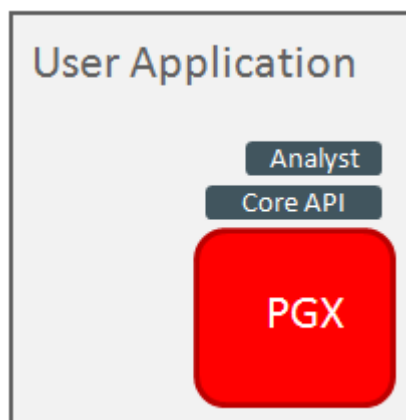
See [Starting the In-Memory Graph Server \(PGX\)](#) for instructions on how to start the graph server (PGX) in remote server mode.

Using Graph Server (PGX) as a Library

You can also include the graph server (PGX) as a normal Java library in your application.

The following figure shows the graph server (PGX) used as a library in an application:

Figure 4-4 PGX as a Library



The embedded mode is useful when you want to build an application having graph analysis as a part of its functionality.

See [Using Graph Server \(PGX\) as a Library](#) for more information.

Deploying Graph Server (PGX) as Servlet Web Application

You can deploy the graph server (PGX) as a web application using Apache Tomcat or Oracle WebLogic Server.

See [Deploying Oracle Graph Server to a Web Server](#) for instructions to deploy the graph server (PGX) in Apache Tomcat or Oracle WebLogic Server.

4.2 User Authentication and Authorization

The Oracle Graph server (PGX) uses an Oracle Database as identity manager. Both username and password based as well as Kerberos based authentication is supported.

The actions that you are allowed to do on the graph server are determined by the privileges enabled by roles that have been granted to you in the Oracle Database.

- [Privileges and Roles in Oracle Database](#)
All database users that work with graphs require the `CREATE SESSION` privilege in the database.

- [Basic Steps for Using an Oracle Database for Authentication](#)
You can follow the steps explained in this section to authenticate users to the graph server (PGX).
- [Prepare the Graph Server for Database Authentication](#)
Locate the `pgx.conf` file of your installation.
- [Store the Database Password in a Keystore](#)
- [Token Expiration](#)
By default, tokens are valid for 1 hour.
- [Advanced Access Configuration](#)
You can customize the following fields inside the `pgx_realm` block in the `pgx.conf` file to customize login behavior.
- [Customizing Roles and Permissions](#)
You can fully customize the permissions to roles mapping by adding and removing roles and specifying permissions for a role. You can also authorize individual users instead of roles.
- [Revoking Access to the Graph Server](#)
To revoke a user's ability to access the graph server, either drop the user from the database or revoke the corresponding roles from the user, depending on how you defined the access rules in your `pgx.conf` file.
- [Examples of Custom Authorization Rules](#)
You can define custom authorization rules for developers.
- [Kerberos Enabled Authentication](#)
The graph server (PGX) can authenticate users using an Oracle Database with Kerberos enabled as identity provider.

4.2.1 Privileges and Roles in Oracle Database

All database users that work with graphs require the `CREATE SESSION` privilege in the database.

Roles that are created for working with graphs are in [Table 4-1](#). These roles are created when you install the PL/SQL package of the Oracle Graph Server and Client distribution on the target database.

Table 4-1 Privileges and Roles in Oracle Database

Role	Operations enabled by this role	Used By
<code>PGX_SESSION_CREATE</code>	Create a new PGX session using the <code>ServerInstance.createSession</code> API.	Graph developers and graph users
<code>PGX_SERVER_GET_INFO</code>	Get status information on the PGX instance using the Admin API .	Users who administer PGX
<code>PGX_SERVER_MANAGE</code> (includes <code>PGX_SERVER_GET_INFO</code>)	Manage the PGX instance using the Admin API to stop or restart PGX.	Users who administer PGX
<code>PGX_SESSION_NEW_GRAPH</code>	Create a new graph in PGX by loading from the database using a config file, using the <code>CREATE PROPERTY GRAPH</code> statement in PGQL, creating a sub-graph from another graph, or using the <code>GraphBuilder</code> .	Graph developers and graph users

Table 4-1 (Cont.) Privileges and Roles in Oracle Database

Role	Operations enabled by this role	Used By
PGX_SESSION_GET_PUBLISHED_GRAPH	Query and view graphs published by another user to the public namespace.	Graph developers and graph users
PGX_SESSION_ADD_PUBLISHED_GRAPH (includes PGX_SESSION_GET_PUBLISHED_GRAPH)	Publish a graph to the public namespace.	Graph developers
PGX_SESSION_COMPILE_ALGORITHM	Compile an algorithm using the PGX Algorithm API.	Graph developers
PGX_SESSION_READ_MODEL	Load and use an ML model using PgxML.	Graph developers
PGX_SESSION_MODIFY_MODEL	Create, train, and store an ML model using PgxML.	Graph developers

Few additional roles are also created to group multiple roles together. They provide a convenient way to grant multiple roles to database users. See [Mapping Graph Server Roles to Default Privileges](#) for more information on these additional roles.

You can create additional groups that are useful for your application, as described in [Adding and Removing Roles](#) and [Defining Permissions for Individual Users](#).

4.2.2 Basic Steps for Using an Oracle Database for Authentication

You can follow the steps explained in this section to authenticate users to the graph server (PGX).

1. Use an Oracle Database version that is supported by Oracle Graph Server and Client: version 12.2 or later, including Autonomous Database.
2. Be sure that you have ADMIN access (or SYSDBA access for non-autonomous databases) to grant and revoke users access to the graph server (PGX).
3. Be sure that all existing users to which you plan to grant access to the graph server have at least the CREATE SESSION privilege granted.
4. Be sure that the database is accessible via JDBC from the host where the Graph Server runs.
5. As ADMIN (or SYSDBA on non-autonomous databases), run the following procedure to create the roles required by the graph server:

 **Note:**

You can skip this step if you install the PL/SQL packages as part of the Oracle Graph Server and Client installation. All the roles shown in the following code are created as part of the PL/SQL installation automatically. You need to add them separately only if you are using Oracle Graph Server and Client with Autonomous Database. You can run this code using Database Actions in Oracle Cloud Infrastructure Console.

```

DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  role_exists EXCEPTION;
  PRAGMA EXCEPTION_INIT(role_exists, -01921);
  TYPE graph_roles_table IS TABLE OF VARCHAR2(50);
  graph_roles graph_roles_table;
BEGIN
  graph_roles := graph_roles_table(
    'GRAPH_DEVELOPER',
    'GRAPH_ADMINISTRATOR',
    'GRAPH_USER',
    'PGX_SESSION_CREATE',
    'PGX_SERVER_GET_INFO',
    'PGX_SERVER_MANAGE',
    'PGX_SESSION_READ_MODEL',
    'PGX_SESSION_MODIFY_MODEL',
    'PGX_SESSION_NEW_GRAPH',
    'PGX_SESSION_GET_PUBLISHED_GRAPH',
    'PGX_SESSION_COMPILE_ALGORITHM',
    'PGX_SESSION_ADD_PUBLISHED_GRAPH');
  FOR elem IN 1 .. graph_roles.count LOOP
    BEGIN
      dbms_output.put_line('create_graph_roles: ' || elem || ':
CREATE ROLE ' || graph_roles(elem));
      EXECUTE IMMEDIATE 'CREATE ROLE ' || graph_roles(elem);
    EXCEPTION
      WHEN role_exists THEN
        dbms_output.put_line('create_graph_roles: role already
exists. continue');
      WHEN OTHERS THEN
        RAISE;
    END;
  END LOOP;
EXCEPTION
  when others then
    dbms_output.put_line('create_graph_roles: hit error ');
    raise;
END;
/

```

- Assign default permissions to the roles GRAPH_DEVELOPER, GRAPH_USER and GRAPH_ADMINISTRATOR to group multiple permissions together.

 **Note:**

You can skip this step if you install the PL/SQL packages as part of the Oracle Graph Server and Client installation. All the grants shown in the following code are executed as part of the PL/SQL installation automatically. You need to execute these grants separately only if you are using Oracle Graph Server and Client with Autonomous Database. You can run this code using Database Actions in Oracle Cloud Infrastructure Console.

```
GRANT PGX_SESSION_CREATE TO GRAPH_ADMINISTRATOR;  
GRANT PGX_SERVER_GET_INFO TO GRAPH_ADMINISTRATOR;  
GRANT PGX_SERVER_MANAGE TO GRAPH_ADMINISTRATOR;  
GRANT PGX_SESSION_CREATE TO GRAPH_DEVELOPER;  
GRANT PGX_SESSION_NEW_GRAPH TO GRAPH_DEVELOPER;  
GRANT PGX_SESSION_GET_PUBLISHED_GRAPH TO GRAPH_DEVELOPER;  
GRANT PGX_SESSION_MODIFY_MODEL TO GRAPH_DEVELOPER;  
GRANT PGX_SESSION_READ_MODEL TO GRAPH_DEVELOPER;  
GRANT PGX_SESSION_CREATE TO GRAPH_USER;  
GRANT PGX_SESSION_GET_PUBLISHED_GRAPH TO GRAPH_USER;
```

7. Assign roles to all the database developers who should have access to the graph server (PGX). For example:

```
GRANT graph_developer TO <graphuser>
```

where <graphuser> is a user in the database. You can also assign individual permissions (roles prefixed with PGX_) to users directly.

8. Assign the administrator role to users who should have administrative access. For example:

```
GRANT graph_administrator to <administratoruser>
```

where <administratoruser> is a user in the database.

4.2.3 Prepare the Graph Server for Database Authentication

Locate the `pgx.conf` file of your installation.

If you installed the graph server via RPM, the file is located at: `/etc/oracle/graph/pgx.conf`

If you use the `webapps` package to deploy into Tomcat or WebLogic Server, the `pgx.conf` file is located inside the web application archive file (WAR file) at: `WEB-INF/classes/pgx.conf`

Tip: On Linux, you can use `vim` to edit the file directly inside the WAR file without unzipping it first. For example:

```
vim graph-server-<version>-pgx<version>.war
```

Inside the `pgx.conf` file, locate the `jdbc_url` line of the realm options:

```
...
"pgx_realm": {
  "implementation": "oracle.pg.identity.DatabaseRealm",
  "options": {
    "jdbc_url": "<REPLACE-WITH-DATABASE-URL-TO-USE-FOR-AUTHENTICATION>",
    "token_expiration_seconds": 3600,
  }
}
...
```

Replace the text with the JDBC URL pointing to your database that you configured in the previous step. For example:

```
...
"pgx_realm": {
  "implementation": "oracle.pg.identity.DatabaseRealm",
  "options": {
    "jdbc_url": "jdbc:oracle:thin:@myhost:1521/myService",
    "token_expiration_seconds": 3600,
  }
}
...
```

If you are using an Autonomous Database, specify the JDBC URL like this:

```
...
"pgx_realm": {
  "implementation": "oracle.pg.identity.DatabaseRealm",
  "options": {
    "jdbc_url": "jdbc:oracle:thin:@my_identifier_low?TNS_ADMIN=/etc/oracle/graph/wallet",
    "token_expiration_seconds": 3600,
  }
}
...
```

where `/etc/oracle/graph/wallet` is an example path to the unzipped wallet file that you downloaded from your Autonomous Database service console, and `my_identifier_low` is one of the connect identifiers specified in `/etc/oracle/graph/wallet/tnsnames.ora`.

Now, start the graph server. If you installed via RPM, execute the following command as a root user or with `sudo`:

```
sudo systemctl start pgx
```

4.2.4 Store the Database Password in a Keystore

PGX requires a database account to read data from the database into memory. The account should be a low-privilege account (see [Security Best Practices with Graph Data](#)).

As described in [Reading Graphs from Oracle Database into the Graph Server \(PGX\)](#), you can read data from the database into the graph server without specifying additional authentication as long as the token is valid for that database user. But if you

want to access a graph from a different user, you can do so, as long as that user's password is stored in a Java Keystore file for protection.

You can use the `keytool` command that is bundled together with the JDK to generate such a keystore file on the command line. See the following script as an example:

```
# Add a password for the 'database1' connection
keytool -importpass -alias database1 -keystore keystore.p12
# 1. Enter the password for the keystore
# 2. Enter the password for the database

# Add another password (for the 'database2' connection)
keytool -importpass -alias database2 -keystore keystore.p12

# List what's in the keystore using the keytool
keytool -list -keystore keystore.p12
```

If you are using Java version 8 or lower, you should pass the additional parameter `-storetype pkcs12` to the `keytool` commands in the preceding example.

You can store more than one password into a single keystore file. Each password can be referenced using the alias name provided.

- [Either, Write the PGX graph configuration file to load from the property graph schema](#)
- [Or, Write the PGX graph configuration file to load a graph directly from relational tables](#)
- [Read the data](#)
- [Secure coding tips for graph client applications](#)

Either, Write the PGX graph configuration file to load from the property graph schema

Next write a PGX graph configuration file in JSON format. The file tells PGX where to load the data from, how the data looks like and the keystore alias to use. The following example shows a graph configuration to read data stored in the Oracle property graph format.

```
{
  "format": "pg",
  "db_engine": "rdbms",
  "name": "hr",
  "jdbc_url": "jdbc:oracle:thin:@myhost:1521/orcl",
  "username": "hr",
  "keystore_alias": "database1",
  "vertex_props": [{
    "name": "COUNTRY_NAME",
    "type": "string"
  }, {
    "name": "DEPARTMENT_NAME",
    "type": "string"
  }, {
    "name": "SALARY",
    "type": "double"
  }],
  "partition_while_loading": "by_label",
  "loading": {
    "load_vertex_labels": true,
```

```

    "load_edge_label": true
  }
}

```

(For the full list of available configuration fields, including their meanings and default values, see [Graph Configuration Options](#).)

Or, Write the PGX graph configuration file to load a graph directly from relational tables

The following example loads a subset of the HR sample data from relational tables directly into PGX as a graph. The configuration file specifies a mapping from relational to graph format by using the concept of vertex and edge providers.



Note:

Specifying the `vertex_providers` and `edge_providers` properties loads the data into an optimized representation of the graph.

```

{
  "name": "hr",
  "jdbc_url": "jdbc:oracle:thin:@myhost:1521/orcl",
  "username": "hr",
  "keystore_alias": "database1",
  "vertex_id_strategy": "no_ids",
  "vertex_providers": [
    {
      "name": "Employees",
      "format": "rdbms",
      "database_table_name": "EMPLOYEES",
      "key_column": "EMPLOYEE_ID",
      "key_type": "string",
      "props": [
        {
          "name": "FIRST_NAME",
          "type": "string"
        },
        {
          "name": "LAST_NAME",
          "type": "string"
        },
        {
          "name": "EMAIL",
          "type": "string"
        },
        {
          "name": "SALARY",
          "type": "long"
        }
      ]
    }
  ],
  {

```

```
    "name": "Jobs",
    "format": "rdbms",
    "database_table_name": "JOBS",
    "key_column": "JOB_ID",
    "key_type": "string",
    "props": [
      {
        "name": "JOB_TITLE",
        "type": "string"
      }
    ]
  },
  {
    "name": "Departments",
    "format": "rdbms",
    "database_table_name": "DEPARTMENTS",
    "key_column": "DEPARTMENT_ID",
    "key_type": "string",
    "props": [
      {
        "name": "DEPARTMENT_NAME",
        "type": "string"
      }
    ]
  }
],
"edge_providers": [
  {
    "name": "WorksFor",
    "format": "rdbms",
    "database_table_name": "EMPLOYEES",
    "key_column": "EMPLOYEE_ID",
    "source_column": "EMPLOYEE_ID",
    "destination_column": "EMPLOYEE_ID",
    "source_vertex_provider": "Employees",
    "destination_vertex_provider": "Employees"
  },
  {
    "name": "WorksAs",
    "format": "rdbms",
    "database_table_name": "EMPLOYEES",
    "key_column": "EMPLOYEE_ID",
    "source_column": "EMPLOYEE_ID",
    "destination_column": "JOB_ID",
    "source_vertex_provider": "Employees",
    "destination_vertex_provider": "Jobs"
  },
  {
    "name": "WorkedAt",
    "format": "rdbms",
    "database_table_name": "JOB_HISTORY",
    "key_column": "EMPLOYEE_ID",
    "source_column": "EMPLOYEE_ID",
    "destination_column": "DEPARTMENT_ID",
    "source_vertex_provider": "Employees",
```

```

        "destination_vertex_provider": "Departments",
        "props": [
            {
                "name": "START_DATE",
                "type": "local_date"
            },
            {
                "name": "END_DATE",
                "type": "local_date"
            }
        ]
    }
}

```

Read the data

Now you can instruct PGX to connect to the database and read the data by passing in both the keystore and the configuration file to PGX, using one of the following approaches:

- **Interactively in the graph shell**

If you are using the graph shell, start it with the `--secret_store` option. It will prompt you for the keystore password and then attach the keystore to your current session. For example:

```

cd /opt/oracle/graph
./bin/opg4j --secret_store /etc/my-secrets/keystore.p12

enter password for keystore /etc/my-secrets/keystore.p12:

```

Inside the shell, you can then use normal PGX APIs to read the graph into memory by passing the JSON file you just wrote into the `readGraphWithProperties` API:

```

opg4j> var graph = session.readGraphWithProperties("config.json")
graph ==> PgxGraph[name=hr,N=215,E=415,created=1576882388130]

```

- **As a PGX preloaded graph**

As a server administrator, you can instruct PGX to load graphs into memory upon server startup. To do so, modify the PGX configuration file at `/etc/oracle/graph/pgx.conf` and add the path the graph configuration file to the `preload_graphs` section. For example:

```

{
    ...
    "preload_graphs": [{
        "name": "hr",
        "path": "/path/to/config.json"
    }],
    "authorization": [{
        "pgx_role": "GRAPH_DEVELOPER",
        "pgx_permissions": [{
            "preloaded_graph": "hr",

```



```

        "grant": "read"
    }
  },
  ....
]
}

```

As root user, edit the service file at `/etc/systemd/system/pgx.service` and change the `ExecStart` command to specify the location of the keystore containing the password:

```
ExecStart=/bin/bash start-server --secret-store /etc/keystore.p12
```

 **Note:**

Please note that `/etc/keystore.p12` must not be password protected for this to work. Instead protect the file via file system permission that is only readable by `oraclegraph` user.

After the file is edited, reload the changes using:

```
sudo systemctl daemon-reload
```

Finally start the server:

```
sudo systemctl start pgx
```

- **In a Java application**

To register a keystore in a Java application, use the `registerKeystore()` API on the `PgxSession` object. For example:

```

import oracle.pgx.api.*;

class Main {

    public static void main(String[] args) throws Exception {
        String baseUrl = args[0];
        String keystorePath = "/etc/my-secrets/keystore.p12";
        char[] keystorePassword = args[1].toCharArray();
        String graphConfigPath = args[2];
        ServerInstance instance = Pgx.getInstance(baseUrl);
        try (PgxSession session = instance.createSession("my-session")) {
            session.registerKeystore(keystorePath, keystorePassword);
            PgxGraph graph = session.readGraphWithProperties(graphConfigPath);
            System.out.println("N = " + graph.getNumVertices() + " E = " +
graph.getNumEdges());
        }
    }
}

```

You can compile and run the preceding sample program using the Oracle Graph Client package. For example:

```
cd $GRAPH_CLIENT
// create Main.java with above contents
javac -cp 'lib/*' Main.java
java -cp '.:conf:lib/*' Main http://myhost:7007 MyKeystorePassword
path/to/config.json
```

Secure coding tips for graph client applications

When writing graph client applications, make sure to never store any passwords or other secrets in clear text in any files or in any of your code.

Do not accept passwords or other secrets through command line arguments either. Instead, use `Console.html#readPassword()` from the JDK.

4.2.5 Token Expiration

By default, tokens are valid for 1 hour.

Internally, the graph client automatically renews tokens which are about to expire in less than 30 minutes. This is also configurable by re-authenticating your credentials with the database. By default, tokens can only be automatically renewed for up to 24 times, then you need to login again.

If the maximum amount of auto-renewals is reached, you can log in again without losing any of your session data by using the `GraphServer#reauthenticate(instance, "<user>", "<password>")` API.



Note:

If a session time out occurs before you re-authenticate, then you may lose your session data.

For example:

```
opg4j> var graph = session.readGraphWithProperties(config) // fails
because token cannot be renewed anymore
opg4j> GraphServer.reauthenticate(instance, "<user>", "<password>") //
log in again
opg4j> var graph = session.readGraphWithProperties(config) //
works now
```

4.2.6 Advanced Access Configuration

You can customize the following fields inside the `pgx_realm` block in the `pgx.conf` file to customize login behavior.

Table 4-2 Advanced Access Configuration Options

Field Name	Explanation	Default
token_expiration_seconds	After how many seconds the generated bearer token will expire.	3600 (1 hour)
connect_timeout_milliseconds	After how many milliseconds an connection attempt to the specified JDBC URL will time out, resulting in the login attempt being rejected.	10000
max_pool_size	Maximum number of JDBC connections allowed per user. If the number is reached, attempts to read from the database will fail for the current user.	64
max_num_users	Maximum number of active, signed in users to allow. If this number is reached, the graph server will reject login attempts.	512
max_num_token_refresh	Maximum amount of times a token can be automatically refreshed before requiring a login again.	24

To configure the refresh time on the client side before token expiration, use the following API to login:

```
int refreshTimeBeforeTokenExpiry = 900; // in seconds, default is 1800 (30 minutes)
ServerInstance instance = GraphServer.getInstance("https://localhost:7007", "<database user>", "<database password>",
    refreshTimeBeforeTokenExpiry);
```

**Note:**

The preceding options work only if the realm implementation is configured to be `oracle.pg.identity.DatabaseRealm`.

4.2.7 Customizing Roles and Permissions

You can fully customize the permissions to roles mapping by adding and removing roles and specifying permissions for a role. You can also authorize individual users instead of roles.

This topic includes examples of how to customize the permission mapping.

- [Checking Graph Permissions Using API](#)
- [Adding and Removing Roles](#)
You can add new role permission mappings or remove existing mappings by modifying the authorization list.

- [Defining Permissions for Individual Users](#)
In addition to defining permissions for roles, you can define permissions for individual users.
- [Defining Permissions to Use Custom Graph Algorithms](#)
You can define permissions to allow developers to compile custom graph algorithms.

4.2.7.1 Checking Graph Permissions Using API

You can view your roles and graph permissions using the following PGX API methods:

Table 4-3 API for Checking Graph Permissions

Class	Method	Description
ServerInstance	getPgxUsername()	Name of the current user
ServerInstance	getPgxUserRoles()	Role names of the current user
ServerInstance	getPgxGenericPermissions()	Non-graph (system) permissions of the current user: <ul style="list-style-type: none"> • Pgx system permissions • File-location permissions
PgxGraph	getPermission()	Permission on the graph instance for a current user

You can get all permission-related information using the API in JShell as shown:

```
/bin/opg4j -b "https://<host>:<port>" -u "<graphuser>"
opg4j> instance
instance ==> ServerInstance[embedded=false,baseUrl=https://
<host>:<port>,serverVersion=null]
opg4j>instance.getPgxUsername()
$2 ==> "ORACLE"
opg4j>instance.getPgxUserRoles()
$3 ==> [GRAPH_DEVELOPER]
opg4j>instance.getPgxGenericPermissions()
$4 ==> [PGX_SESSION_CREATE, PGX_SESSION_READ_MODEL,
PGX_SESSION_ADD_PUBLISHED_GRAPH, PGX_SESSION_NEW_GRAPH,
PGX_SESSION_GET_PUBLISHED_GRAPH, PGX_SESSION_MODIFY_MODEL]
opg4jvar g =
session.readGraphWithProperties("bank_graph_analytics.json")
g ==>
PgxGraph[name=bank_graph_analytics,N=1000,E=5001,created=1625697341555]
opg4j>g.getPermission() // To get graph permissions
$9 ==> MANAGE
```

The following example shows a Java code using the PGX API to obtain the graph permission information:

```
import oracle.pg.rdbms.*;
import java.sql.Connection;
import java.sql.Statement;
```

```
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.pgx.api.*;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;
import java.nio.file.Files;
import java.nio.file.Path;

/**
 * This example shows how to get all permissions.
 */
public class GetPermissions
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];
        String user           = args[idx++];
        String password       = args[idx++];
        String graph          = args[idx++];

        Connection conn = null;
        PgxPreparedStatement stmt = null;

        try {

            // Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + "/" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            ServerInstance instance = GraphServer.getInstance("http://
localhost:7007", user, password.toCharArray());
            PgxSession session = instance.createSession("my-session");

            var statement = Files.readString(Path.of("/media/sf_Linux/Java/create-
pg.pgql"));
            stmt = session.preparePgql(statement);
            stmt.execute();

            PgxGraph g = session.getGraph(graph);
            System.out.println("Graph: " + g);

            String userName = instance.getPgxUsername();
            var userRoles = instance.getPgxUserRoles();
            var genericPermissions = instance.getPgxGenericPermissions();
            String graphPermission = g.getPermission().toString();
```

```
        System.out.println("Username is " + userName);
        System.out.println("User Roles are " + userRoles);
        System.out.println("Generic permissions are " +
genericPermissions);
        System.out.println("Graph permission is " + graphPermission);

    }

    finally {
        // close the sql statment
        if (stmt != null) {
            stmt.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}
}
```

On execution, the code gives the following output:

```
Graph: PgxGraph[name=BANK_GRAPH_PG,N=1000,E=5001,created=1625731370402]
Username is ORACLE
User Roles are [GRAPH_DEVELOPER]
Generic permissions are [PGX_SESSION_MODIFY_MODEL, PGX_SESSION_CREATE,
PGX_SESSION_NEW_GRAPH, PGX_SESSION_READ_MODEL,
PGX_SESSION_ADD_PUBLISHED_GRAPH, PGX_SESSION_GET_PUBLISHED_GRAPH]
Graph permission is MANAGE
```

4.2.7.2 Adding and Removing Roles

You can add new role permission mappings or remove existing mappings by modifying the authorization list.

For example:

```
CREATE ROLE MY_CUSTOM_ROLE_1
GRANT PGX_SESSION_CREATE TO MY_CUSTOM_ROLE1
GRANT PGX_SERVER_GET_INFO TO MY_CUSTOM_ROLE1
GRANT MY_CUSTOM_ROLE1 TO SCOTT
```

4.2.7.3 Defining Permissions for Individual Users

In addition to defining permissions for roles, you can define permissions for individual users.

For example:

```
GRANT PGX_SESSION_CREATE TO SCOTT
GRANT PGX_SERVER_GET_INFO TO SCOTT
```

4.2.7.4 Defining Permissions to Use Custom Graph Algorithms

You can define permissions to allow developers to compile custom graph algorithms.

For example,

- Add the following static permission to the list of permissions:

```
GRANT PGX_SESSION_COMPILE_ALGORITHM TO GRAPH_DEVELOPER
```

4.2.8 Revoking Access to the Graph Server

To revoke a user's ability to access the graph server, either drop the user from the database or revoke the corresponding roles from the user, depending on how you defined the access rules in your `pgx.conf` file.

For example:

```
REVOKE graph_developer FROM scott
```

Revoking Graph Permissions

If you have the `MANAGE` permission on a graph, you can revoke graph access from users or roles using the `PgxGraph#revokePermission` API. For example:

```
PgxGraph g = ...  
g.revokePermission(new PgxRole("GRAPH_DEVELOPER")) // revokes previously  
granted role access  
g.revokePermission(new PgxUser("SCOTT")) // revokes previously granted user  
access
```

4.2.9 Examples of Custom Authorization Rules

You can define custom authorization rules for developers.

- [Example 4-1](#)
- [Example 4-2](#)
- [Example 4-3](#)

Example 4-1 Allowing Developers to Publish Graphs

Sharing of graphs with other users should be done in Oracle Database where possible. Use `GRANT` statements on the database tables so that other users can create graphs from the tables. If the graph is in the Property Graph schema use the `OPG_APIS.GRANT_ACCESS` API to share the graph.

In the in-memory graph server you can use the following permissions to share a graph that is already in memory, with other users connected to the graph server.

Table 4-4 Allowed Permissions

Permission	Actions Enabled by this Permission
READ	<ul style="list-style-type: none"> • READ the graph via the PGX API or in PGQL queries in PGX, create a subgraph, or clone the graph
MANAGE	<ul style="list-style-type: none"> • Publish the graph or snapshot • Includes READ and EXPORT • Grant or revoke READ and EXPORT permissions on the graph
EXPORT	<ul style="list-style-type: none"> • Export the graph to a file. • Includes READ permission.

The creator of the graph automatically gets the MANAGE permission granted on the graph. If you have the MANAGE permission, you can grant other roles or users READ or EXPORT permission on the graph. You **cannot** grant MANAGE on a graph. The following example of a user named userA shows how:

```
import oracle.pgx.api.*
import oracle.pgx.common.auth.*

PgxSession session = GraphServer.getInstance("<base-url>", "<userA>",
"<password-of-userA>").createSession("userA")
PgxGraph g = session.readGraphWithProperties("examples/sample-
graph.json", "sample-graph")
g.grantPermission(new PgxRole("GRAPH_DEVELOPER"),
PgxResourcePermission.READ)
g.publish()
```

Now other users with the GRAPH_DEVELOPER role can access this graph and have READ access on it, as shown in the following example of userB:

```
PgxSession session = GraphServer.getInstance("<base-url>", "<userB>",
"<password-of-userB>").createSession("userB")
PgxGraph g = session.getGraph("sample-graph")
g.queryPgql("select count(*) from match (v)").print().close()
```

Similarly, graphs can be shared with individual users instead of roles, as shown in the following example:

```
g.grantPermission(new PgxUser("OTHER_USER"),
PgxResourcePermission.EXPORT)
```

where OTHER_USER is the user name of the user that will receive the EXPORT permission on graph g.

Example 4-2 Allowing Developers to Access Preloaded Graphs

To allow developers to access preloaded graphs (graphs loaded during graph server startup), grant the read permission on the preloaded graph in the `pgx.conf` file. For example:

```
"preload_graphs": [{
  "path": "/data/my-graph.json",
  "name": "global_graph"
}],
"authorization": [{
  "pgx_role": "GRAPH_DEVELOPER",
  "pgx_permissions": [{
    "preloaded_graph": "global_graph"
    "grant": "read"
  }],
  ...
```

You can grant `READ`, `EXPORT`, or `MANAGE` permission.

Example 4-3 Allowing Developers Access to the Hadoop Distributed Filesystem (HDFS) or the Local File System

To allow developers to read files from HDFS, you must first declare the HDFS directory and then map it to a read or write permission. For example:

```
CREATE OR REPLACE DIRECTORY pgx_file_location AS 'hdfs:/data/graphs'
GRANT READ ON DIRECTORY pgx_file_location TO GRAPH_DEVELOPER
```

Similarly, you can add another permission with `GRANT WRITE` to allow write access. Such a write access is required in order to export graphs.

Access to the local file system (where the graph server runs) can be granted the same way. The only difference is that location would be an absolute file path without the `hdfs:` prefix. For example:

```
CREATE OR REPLACE DIRECTORY pgx_file_location AS '/opt/oracle/graph/data'
```

Note that in addition to the preceding configuration, the operating system user that runs the graph server process must have the corresponding directory privileges to actually read or write into those directories.

4.2.10 Kerberos Enabled Authentication

The graph server (PGX) can authenticate users using an Oracle Database with Kerberos enabled as identity provider.

You can log into the graph server using a Kerberos ticket and the actions which you are allowed to do on the graph server are determined by the roles that have been granted to you in the Oracle Database.

- [Prerequisite Requirements](#)
- [Prepare the Graph Server for Kerberos Authentication](#)

- [Login to the Graph Server Using Kerberos Ticket](#)

4.2.10.1 Prerequisite Requirements

In order to enable Kerberos authentication on the in-memory graph server (PGX), the following system requirements must be met:

- The database needs to have Kerberos authentication enabled. See [Configuring Kerberos Authentication](#) for more information.
- Both the database and the Kerberos Authentication Server need to be reachable from the host where the graph server runs.
- The database is prepared for graph server authentication. That is, relevant graph roles have been granted to users who will log into the graph server.

4.2.10.2 Prepare the Graph Server for Kerberos Authentication

The following are the steps to enable Kerberos authentication on the in-memory graph server (PGX):

1. Locate the `pgx.conf` file of your installation.

 **Note:**

If you installed the graph server via RPM, the file is located at: `/etc/oracle/graph/pgx.conf`

2. Locate the `krb5_conf_file` line of the realm options, inside the `pgx.conf` file:

```
"pgx_realm": {
  "implementation": "oracle.pg.identity.DatabaseRealm",
  "options": {
    ...
    "krb5_conf_file": "<REPLACE-WITH-KRB5-CONF-FILE-PATH-TO-ENABLE-
KERBEROS-AUTHENTICATION>",
    "krb5_ticket_cache_dir": "/dev/shm",
    "krb5_max_cache_size": 1024
  }
},
```

3. Replace the text with the `krb5.conf` file that you are using for the database and user authentication. For example:

```
"pgx_realm": {
  "implementation": "oracle.pg.identity.DatabaseRealm",
  "options": {
    ...
    "krb5_conf_file": "/etc/krb5.conf",
    "krb5_ticket_cache_dir": "/dev/shm",
    "krb5_max_cache_size": 1024
  }
},
```

 **Note:**

The file provided for the `krb5_conf_file` option needs to be valid and readable by the graph server. In case you don't replace the `krb5_conf_file` value or the value is empty, then the graph server will not use Kerberos authentication.

Also, you can set the cache directory that will be used for the graph server to temporarily store Kerberos tickets given by clients as well as the maximum cache size after which new login attempts will be rejected. The cache size represents the maximum amount of concurrent Kerberos sessions active on the graph server.

4.2.10.3 Login to the Graph Server Using Kerberos Ticket

The following are the steps to login to the in-memory graph server (PGX) using Kerberos ticket:

1. Create a new Kerberos ticket using the `okinit` command:

```
$ okinit <username>
```

This will prompt for your password and then create a new Kerberos ticket.

2. Connect to a remote graph server with only the base URL parameter using JShell:

```
$ opg4j -b https://localhost:7007
```

Or using Python client:

```
$ opg4py -b https://localhost:7007
```

On Linux, JShell and Python interactive client shells automatically detect the Kerberos ticket on your local file system and use that to authenticate with the graph server.

3. In case the auto-detection is not working, you can also explicitly pass in the ticket to the shell. Run the `oklist` command, to find the location of the ticket on the local file system.

```
$ oklist
```

```
Kerberos Utilities for Linux: Version 19.0.0.0.0 - Production on 31-  
MAR-2021 15:26:46
```

```
Copyright (c) 1996, 2019 Oracle. All rights reserved.
```

```
Configuration file : /etc/krb5.conf.  
Ticket cache: FILE:/tmp/krb5cc_54321  
Default principal: oracle@realm
```

4. Specify your Kerberos ticket path using the `--kerberos_ticket` parameter. For example, using JShell:

```
$ opg4j -b https://localhost:7007 --kerberos_ticket /tmp/krb5cc_54321
```

Or using Python Client:

```
$ opg4py -b https://localhost:7007 --kerberos_ticket /tmp/  
krb5cc_54321
```

If you are using a Java client program (or JShell on embedded mode), you can get a server instance using the following API:

```
...  
ServerInstance instance = GraphServer.getInstance("https://  
localhost:7007", "/tmp/krb5cc_54321");  
PgxSession session = instance.createSession("my-session");  
...
```

If you are using a Python Client program (or opg4py on embedded mode), you can get a server instance using the following API:

```
...  
instance = graph_server.get_instance("https://localhost:7007",  
"/tmp/krb5cc_54321")  
session = instance.create_session("my-session")  
...
```

If you are connecting to a remote graph server, all you need is the Oracle Graph Client to be installed. For example:

```
import sys  
import pypgx as pgx  
  
sys.path.append("/path/to/graph/client/oracle-graph-client-21.2.0/  
python/pypgx/pg/rdbms")  
  
import graph_server  
  
base_url = "https://localhost:7007"  
kerberos_ticket = "/tmp/krb5cc_54321"  
  
instance = graph_server.get_instance(base_url, kerberos_ticket)  
print(instance)
```

4.3 About Vertex and Edge IDs

Generating vertex and edge IDs when loading from database tables into PGX

PGX enforces by default the existence of a unique identifier for each vertex and edge in a graph, so that they can be retrieved by using `PgxGraph.getVertex(ID id)` and `PgxGraph.getEdge(ID id)` or by PGQL queries using the built-in `id()` method.

The ID generation strategies can be selected through the configuration parameters `vertex_id_strategy` and `edge_id_strategy`.

Using keys to generate IDs

The default strategy to generate the vertex IDs is to use the keys provided during loading of the graph (`keys_as_ids`). In that case, each vertex should have a vertex key that is unique across all providers.

For edges, by default no keys are required in the edge data, and edge IDs will be automatically generated by PGX (`unstable_generated_ids`). Note that the generation of edge IDs is not guaranteed to be deterministic. If required, it is also possible to load edge keys as IDs.

The `partitioned_ids` strategy requires keys to be unique only *within* a vertex or edge provider (data source). The keys do not have to be globally unique. Globally unique IDs are derived from a combination of the provider name and the key inside the provider, as `<provider_name><unique_key_within_provider>`. For example, `Account(1)`.

The `partitioned_ids` strategy can be set through the configuration fields `vertex_id_strategy` and `edge_id_strategy`. For example,

```
{
  "name": "bank_graph_analytics",
  "optimized_for": "updates",
  "vertex_id_strategy": "partitioned_ids",
  "edge_id_strategy": "partitioned_ids",
  "vertex_providers": [
    {
      "name": "Accounts",
      "format": "rdbms",
      "database_table_name": "BANK_NODES",
      "key_column": "ID",
      "key_type": "integer",
      "props": [
        {
          "name": "keyProp",
          "type": "long",
          "column": 1
        },
        {
          "name": "number",
          "type": "long",
          "column": 2
        }
      ]
    },
    {
      "name": "Transfers",
      "format": "rdbms",
      "database_table_name": "BANK_EDGES_AMT",
      "key_column": "ID",
      "source_column": "SRC_ID",

```

```

    "destination_column": "DEST_ID",
    "source_vertex_provider": "Accounts",
    "destination_vertex_provider": "Accounts",
    "props": [
      {
        "name": "keyProp",
        "type": "long",
        "column": 1
      },
      {
        "name": "amount",
        "type": "double",
        "column": 4
      }
    ],
    "loading": {
      "create_key_mapping" : true
    }
  }
]
}

```



Note:

All available key types are supported in combination with partitioned IDs.

After the graph is loaded, PGX maintains information about which property of a provider corresponds to the key of the provider. In the preceding example, the vertex property `keyProp` happens to correspond to the vertex key (`"column": 1`) and also the edge property `keyProp` happens to correspond to the edge key (again, `"column": 1`). Each provider can have at most one such "key property" and the property can have any name.

Key properties are used for internal optimizations as well as for providing keys for the vertex or edge or both when inserting new entities. Key properties are currently non-updatable. Trying to update a key property will result in an error. For example,

```
vertex key property ID cannot be updated
```

Using an auto-incrementer to generate IDs

It is recommended to always set `create_key_mapping` to `true` to benefit from performance optimizations. But if there are no single-column keys for edges, `create_key_mapping` can be set to `false`. Similarly, `create_key_mapping` can be set to `false` for vertex providers also. IDs will be generated via an auto-incrementer, for example `Accounts(1)`, `Accounts(2)`, `Accounts(3)`.

4.4 Reading Graphs from Oracle Database into the Graph Server (PGX)

Once logged in to the graph server (PGX), you can now read graphs from the database into the graph server without specifying any connection information in the graph configuration.

Your database user must exist and have read access on the graph data in the database.

There are several ways to read a graph into the graph server (PGX) from Oracle Database:

- Using a Property Graph View
See [Loading a Graph into the Graph Server \(PGX\) from a Property Graph View](#) for more information.
- Using the PGQL - CREATE PROPERTY GRAPH statement.
See [Creating a Property Graph using PGQL](#) for more details.
- Using a PGX graph configuration file in JSON format
See [Creating a JSON Configuration to Load a Graph](#) for more details.
- Using [GraphConfigBuilder](#) class to create Oracle RDBMS graph configs programmatically via Java methods.
See [Defining the Graph Configuration via Java](#) for more details.
- [Creating a JSON Configuration to Load a Graph](#)
- [Defining the Graph Configuration via Java](#)

4.4.1 Creating a JSON Configuration to Load a Graph

In order to load a graph into the graph server (PGX), you can create a graph configuration file, which contains metadata about the graph to be loaded. See [Graph Configuration Options](#) for more details on graph configuration options.

The following shows a sample graph configuration file:

```
{
  "name": "bank_graph_analytics",
  "vertex_providers": [
    {
      "name": "Accounts",
      "format": "rdbms",
      "database_table_name": "BANK_NODES",
      "key_column": "ID",
      "key_type": "integer"
    }
  ],
  "edge_providers": [
    {
      "name": "Transfers",
      "format": "rdbms",
      "database_table_name": "BANK_EDGES_AMT",
      "key_column": "ID",
      "source_column": "SRC_ID",
      "destination_column": "DEST_ID",
```

```

        "source_vertex_provider": "Accounts",
        "destination_vertex_provider": "Accounts",
        "props": [
            {
                "name": "AMOUNT",
                "type": "float"
            }
        ]
    }
]
}

```

You can now read the graph into the graph server (PGX) using the `PgxSession` API method as shown:

Loading a Graph Using JShell

```
opg4j> session.readGraphWithProperties("bank_graph_analytics.json")
```

Loading a Graph Using Java

```
PgxGraph g =
session.readGraphWithProperties("bank_graph_analytics.json")
```

Loading a Graph Using Python

```
g = session.read_graph_with_properties("bank_graph_analytics.json")
```

You can also create a graph configuration file using keystore details. See the example in [Loading Graph Configuration Using Keystore](#) for more information.

See [API for Loading Graphs into Memory](#) for more information on `PgxSession` API methods for reading graphs into memory.

4.4.2 Defining the Graph Configuration via Java

You can load a graph from Oracle Database by first defining the graph configuration object using the `GraphConfigBuilder` class and then reading the graph into the graph server (PGX).

Example 4-4 Reading a graph into the graph server (PGX) using `GraphConfigBuilder`

```

GraphConfig cfg = GraphConfigBuilder.forPropertyGraphRdbms()
    .setJdbcUrl("jdbc:oracle:thin:@<host>:<port>/<sid>")
    .setUsername("<username>")
    .setPassword("<password>")
    .setName("bank_graph_analytics")
    .addVertexProperty("id", PropertyType.INTEGER)
    .addEdgeProperty("amount", PropertyType.INTEGER)
    .setPartitionWhileLoading(PartitionWhileLoading.BY_LABEL)
    .setLoadVertexLabels(true)
    .setLoadEdgeLabel(true)

```



```

        .build();

PgxGraph "bank_graph_analytics" = session.readGraphWithProperties(cfg);

```

See [Property Graph Schema Objects for Oracle Database](#) for more information.

4.5 Keeping the Graph in Oracle Database Synchronized with the Graph Server

You can use the `FlashbackSynchronizer` API to automatically apply changes made to graph in the database to the corresponding `PgxGraph` object in memory, thus keeping both synchronized.

This API uses Oracle's Flashback Technology to fetch the changes in the database since the last fetch and then push those changes into the graph server using the `ChangeSet` API. After the changes are applied, the usual snapshot semantics of the graph server apply: each delta fetch application creates a new in-memory snapshot. Any queries or algorithms that are executing concurrently to snapshot creation are unaffected by the changes until the corresponding session refreshes its `PgxGraph` object to the latest state by calling the `session.setSnapshot(graph, PgxSession.LATEST_SNAPSHOT)` procedure.

For detailed information about Oracle Flashback technology, see the Database Development Guide.

Prerequisites for Synchronizing

The Oracle database must have Flashback enabled and the database user that you use to perform synchronization must have:

- Read access to all tables which need to be kept synchronized.
- Permission to use flashback APIs. For example:

```
GRANT EXECUTE ON DBMS_FLASHBACK TO <user>
```

The database must also be configured to retain changes for the amount of time needed by your use case.

Types of graphs that can be synchronized

Not all `PgxGraph` objects in PGX can be synchronized. The following limitations apply:

- Only the original creator of the graph can synchronize it. That is, the current user must have the `MANAGE` permission of the graph.
- Only graphs loaded from database tables ("partitioned graphs") can be synchronized. Graphs created from other formats or graphs created via the graph builder API cannot be synchronized.
- Only the *latest snapshot* of a graph can be synchronized.

Types of changes that can be synchronized

The synchronizer supports keeping the in-memory graph snapshot in sync with the following database-side modifications:

- insertion of new vertices and edges

- removal of existing vertices and edges
- update of property values of any vertex or edge

The synchronizer does not support schema-level changes to the input graph, such as:

- alteration of the list of input vertex or edge tables
- alteration of any columns of any input tables (vertex or edge tables)

Furthermore, the synchronizer does not support updates to vertex and edge keys.

For detailed examples, see the following topic:

- [Examples of Synchronizing](#)

4.5.1 Examples of Synchronizing

You can perform your graph synchronization using the following examples:

- [Example 4-5](#)
- [Example 4-6](#)

Example 4-5 Synchronizing Graphs Using CREATE PROPERTY GRAPH Statement

1. Assume you have the following Oracle Database tables, PERSONS and FRIENDSHIPS.

```
CREATE TABLE persons (
  person_id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1
  INCREMENT BY 1),
  name VARCHAR2(200),
  birthdate DATE,
  height FLOAT DEFAULT on null 0,
  CONSTRAINT person_pk PRIMARY KEY (person_id)
);
```

```
CREATE TABLE friendships (
  friendship_id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1
  INCREMENT BY 1),
  person_a NUMBER,
  person_b NUMBER,
  meeting_date DATE,
  CONSTRAINT fk_person_a_id FOREIGN KEY (person_a) REFERENCES
  persons(person_id),
  CONSTRAINT fk_person_b_id FOREIGN KEY (person_b) REFERENCES
  persons(person_id)
  CONSTRAINT fs_pk PRIMARY KEY (friendship_id)
);
```

2. You can add some sample data into these tables as shown:

```
INSERT INTO persons (name, height, birthdate) VALUES ('John', 1.80,
to_date('13/06/1963', 'DD/MM/YYYY'));
INSERT INTO persons (name, height, birthdate) VALUES ('Mary', 1.65,
to_date('25/09/1982', 'DD/MM/YYYY'));
INSERT INTO persons (name, height, birthdate) VALUES ('Bob', 1.75,
to_date('11/03/1966', 'DD/MM/YYYY'));
```

```
INSERT INTO persons (name, height, birthdate) VALUES ('Alice', 1.70,
to_date('01/02/1987', 'DD/MM/YYYY'));
```

```
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (1, 3,
to_date('01/09/1972', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (2, 4,
to_date('19/09/1992', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (4, 2,
to_date('19/09/1992', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (3, 2,
to_date('10/07/2001', 'DD/MM/YYYY'));
```

3. Write the corresponding CREATE PROPERTY GRAPH statement which describes how to load those tables as a graph as shown in the following Java code example:

```
session.executePsql(
    "CREATE PROPERTY GRAPH friends VERTEX TABLES (
      + " persons KEY (person_id) LABEL person PROPERTIES
      (name,height,birthdate)"
      + ")"
      + "EDGE TABLES (
      + " friendships "
      + " KEY (friendship_id) "
      + " SOURCE KEY (person_a) REFERENCES persons "
      + " DESTINATION KEY (person_b) REFERENCES persons "
      + " LABEL friendof PROPERTIES (meeting_date)"
      + ")"
    );
PgxGraph graph = session.getGraph("friends");
```

This creates a snapshot of the graph which is loaded into memory. You can now run algorithms and queries on the graph.

4. Now change the data in the input tables in the database. For example, add new persons to the PERSONS table and also add another edge. You can open a new JDBC connection to the database and run a few INSERT statements as shown in the following code:

```
Connection conn = DriverManager.getConnection("<jdbc-url>", "<user>",
"<pass>");
conn.createStatement().executeQuery("INSERT INTO persons(name, birthdate,
height) VALUES ('Mariana',to_date('21/08/1996','DD/MM/YYYY'),1.65)");
conn.createStatement().executeQuery("INSERT INTO persons (name,
birthdate, height) VALUES ('Francisco',to_date('13/06/1963','DD/MM/
YYYY'),1.75)");
conn.createStatement().executeQuery("INSERT INTO friendships (person_a,
person_b, meeting_date) VALUES (1, 6, to_date('13/06/2013','DD/MM/
YYYY'))");
conn.commit();
```

Committing the changes to the database causes the graph in memory to become out of sync with the database source tables.

5. You can synchronize the in-memory graph with the database by creating a new synchronizer object as shown in the following code:

```
Synchronizer synchronizer = new
Synchronizer.Builder<FlashbackSynchronizer>()
    .setType(FlashbackSynchronizer.class)
    .setGraph(graph)
    .setConnection(conn)
    .build();
```

Internally, the graph server keeps track of the Oracle system change number (SCN) the current graph snapshot belongs to. The synchronizer is a *client-side* component which connects to the database, detects changes by comparing state of the the original input tables using the current SCN via the flashback mechanism and then sends any changes to the graph server using the changeset API. In order to do so, the synchronizer needs to know how to connect to the database (`conn` parameter) as well as which graph to keep in sync (`graph` parameter).

- Alternatively, you can use this equivalent shortcut:

```
Synchronizer synchronizer =
graph.createSynchronizer(FlashbackSynchronizer.class, conn);
```

6. Call the `sync()` operation, to fetch the database changes and create a new in-memory snapshot:

```
graph = synchronizer.sync();
```

You will notice that the two new vertices and the new edge have been applied to the graph:

```
graph ==> PgxGraph[name=FRIENDS,N=6,E=5,created=1594754376861]
```

Splitting the Fetching and Applying of Changes

The `synchronizer.sync()` invocation in the preceding code, fetches the changes and applies them in one call. However, you can encode a more complex update logic by splitting this process into separate `fetch()` and `apply()` invocations. For example:

```
synchronizer.fetch() // fetches changes from the database
if (synchronizer.getGraphDelta().getTotalNumberOfChanges() > 100)
{ // only create snapshot if there have been more than 100 changes
    synchronizer.apply()
}
```

Example 4-6 Synchronizing Graphs Created Via Graph Configuration Objects

[Example 4-5](#) uses a `CREATE PROPERTY GRAPH` statement to create the graph which hides some of the more advanced graph configuration options.

Though synchronization of graphs created via graph configuration objects is supported in general, the following few limitations apply:

- Only partitioned graph configurations with all providers being database tables are supported.
- Each edge or vertex provider or both must specify the owner of the table by setting the username field. For example, if user SCOTT owns the table, then set the username accordingly in the provider block of that table:

```
"username": "scott"
```

- In the root loading block, the snapshot source must be set to change_set:

```
"loading": {
  "snapshots_source": "change_set"
}
```

- It is highly recommended to set the "optimized_for" field to "updates" to avoid memory exhaustion when creating many snapshots:

```
"optimized_for": "updates"
```

You can load the same graph shown in [Example 4-5](#) using the following graph configuration (JSON) file:

```
{
  "name": "friends",
  "optimized_for": "updates",
  "vertex_id_strategy": "partitioned_ids",
  "edge_id_strategy": "partitioned_ids",
  "edge_id_type": "long",
  "vertex_id_type": "long",
  "jdbc_url": "<jdbc_url>",
  "username": "<username>",
  "keystore_alias": "<keystore_alias>",
  "vertex_providers": [
    {
      "format": "rdbms",
      "username": "<username>",
      "key_type": "long",
      "name": "person",
      "database_table_name": "persons",
      "key_column": "person_id",
      "props": [
        ...
      ],
      "loading": {
        "create_key_mapping": true
      }
    }
  ],
  "edge_providers": [
    {
      "format": "rdbms",
      "username": "<username>",
      "name": "friendOf",
      "source_vertex_provider": "person",
```

```

    "destination_vertex_provider": "person",
    "database_table_name": "friendships",
    "source_column": "person_a",
    "destination_column": "person_b",
    "key_column": "friendship_id",
    "key_type": "long",
    "props": [
      ...
    ],
    "loading": {
      "create_key_mapping": true
    }
  },
  "loading": {
    "snapshots_source": "change_set"
  }
}

```

Note:

- In the preceding JSON file, replace the values <jdbc_url>, <username>, and <keystore_alias> with the values for connecting to your database.
- When using the graph configuration file, you can load the graph into memory using JShell (be sure to register the keystore containing the database password when starting it) :

```

var pgxGraph =
  session.readGraphWithProperties("<name_of_config_file>.json"
  );

```

4.6 Optimizing Graphs for Read Versus Updates in the In-Memory Graph Server (PGX)

The in-memory graph server (PGX) can store an optimized graph for other reads or updates. This is only relevant when the updates are made directly to a graph instance in the in-memory graph server.

Graph Optimized for Reads

Graphs optimized for reads will provide the best performance for graph analytics and PGQL queries. In this case there could be potentially higher latencies to update the graph (adding or removing vertex and edges or updating the property values of previously existing vertex or edges through `GraphChangeSet` API). There could also be higher memory consumption. When using graphs optimized for reads, each updated graph or graph snapshot consumes memory proportional to the size of the graph in terms of vertices and edges.

The `optimized_for` configuration property can be set to `reads` when loading the graph into the in-memory graph server (PGX) to create a graph instance that is optimized for reads.

Graph Optimized for Updates

Graphs optimized for updates use a representation enabling low-latency update of graphs. With this representation, the graph server can reach millisecond-scale latencies when updating graphs with millions of vertices and edges (this is indicative and will vary depending on the hardware configuration).

To achieve faster update operations, graph server avoids as much as possible doing a full duplication of the previous graph (snapshot) to create a new graph (snapshot). This also improves the memory consumption (in typical scenarios). New snapshots (or new graphs) will only consume additional memory proportional to the memory required for the changes applied.

In this representation, there could be lower performance of graph queries and analytics.

The `optimized_for` configuration property can be set to `updates` when loading the graph into the in-memory graph server (PGX) to create a graph instance that is optimized for reads.

4.7 Storing a Graph Snapshot on Disk

After reading a graph into memory using either Java or the Shell, if you make some changes to the graph such as running the PageRank algorithm and storing the values as vertex properties, you can store this snapshot of the graph on disk.

This is helpful if you want to save the state of the graph in memory, such as if you must shut down the in-memory graph server to migrate to a newer version, or if you must shut it down for some other reason.

(Storing graphs over HTTP/REST is currently not supported.)

A snapshot of a graph can be saved as a file in a binary format (called a PGB file) if you want to save the state of the graph in memory, such as if you must shut down the in-memory graph server to migrate to a newer version, or if you must shut it down for some other reason.

In general, we recommend that you store the graph queries and analytics APIs that had been executed, and that after the in-memory graph server has been restarted, you reload and re-execute the APIs. But if you must save the state of the graph, you can use the logic in the following example to save the graph snapshot from the shell.

In a three-tier deployment, the file is written on the server-side file system. You must also ensure that the file location to write is specified in the in-memory graph server. (As explained in [Three-Tier Deployments of Oracle Graph with Autonomous Database](#), in a three-tier deployment, access to the PGX server file system requires a list of allowed locations to be specified.)

```
opg4j> var graph =
session.createGraphBuilder().addVertex(1).addVertex(2).addVertex(3).addEdge(1
,2).addEdge(2,3).addEdge(3, 1).build()
graph ==> PgxGraph[name=anonymous_graph_1,N=3,E=3,created=1581623669674]

opg4j> analyst.pagerank(graph)
$3 ==> VertexProperty[name=pagerank,type=double,graph=anonymous_graph_1]

// Now save the state of this graph
```

```

opg4j> g.store(Format.PGB, "/tmp/snapshot.pgb")
$4 ==> {"edge_props":[],"vertex_uris":["/tmp/snapshot.pgb"],"loading":
{},"attributes":{},"edge_uris":[],"vertex_props":
[{"name":"pagerank","dimension":0,"type":"double"}],"error_handling":
{},"vertex_id_type":"integer","format":"pgb"}

// reload from disk
opg4j> var graphFromDisk = session.readGraphFile("/tmp/snapshot.pgb")
graphFromDisk ==> PgxGraph[name=snapshot,N=3,E=3,created=1581623739395]

// previously computed properties are still part of the graph and can
be queried
opg4j> graphFromDisk.queryPsql("select x.pagerank match
(x)").print().close()

```

The following example is essentially the same as the preceding one, but it uses partitioned graphs. Note that in the case of partitioned graphs, multiple PGB files are being generated, one for each vertex/edge partition in the graph.

```

opg4j> analyst.pagerank(graph)
$3 ==>
VertexProperty[name=pagerank,type=double,graph=anonymous_graph_1]//
store graph including all props to disk
// Now save the state of this graph
opg4j> var storedPgbConfig = g.store(ProviderFormat.PGB, "/tmp/
snapshot")
$4 ==> {"edge_props":[],"vertex_uris":["/tmp/snapshot.pgb"],"loading":
{},"attributes":{},"edge_uris":[],"vertex_props":
[{"name":"pagerank","dimension":0,"type":"double"}],"error_handling":
{},"vertex_id_type":"integer","format":"pgb"}
// Reload from disk
opg4j> var graphFromDisk =
session.readGraphWithProperties(storedPgbConfig)
graphFromDisk ==> PgxGraph[name=snapshot,N=3,E=3,created=1581623739395]
// Previously computed properties are still part of the graph and can
be queried
opg4j> graphFromDisk.queryPsql("select x.pagerank match
(x)").print().close()

```

4.8 Executing Built-in Algorithms

The in-memory graph server (PGX) contains a set of built-in algorithms that are available as Java APIs.

The following table provides an overview of the available algorithms, grouped by category.



Note:

These algorithms can be invoked through the `Analyst` interface. See the [Analyst Class](#) in Javadoc for more details.

Table 4-5 Overview of Built-In Algorithms

Category	Algorithms
Classic graph algorithms	Prim's Algorithm
Community detection	Conductance Minimization (Soman and Narang Algorithm), Infomap, Label Propagation, Louvain
Connected components	Strongly Connected Components, Weakly Connected Components (WCC)
Link prediction	WTF (Whom To Follow) Algorithm
Matrix factorization	Matrix Factorization
Other	Graph Traversal Algorithms
Path finding	All Vertices and Edges on Filtered Path, Bellman-Ford Algorithms, Bidirectional Dijkstra Algorithms, Compute Distance Index, Compute High-Degree Vertices, Dijkstra Algorithms, Enumerate Simple Paths, Fast Path Finding, Fattest Path, Filtered Fast Path Finding, Hop Distance Algorithms
Ranking and walking	Closeness Centrality Algorithms, Degree Centrality Algorithms, Eigenvector Centrality, Hyperlink-Induced Topic Search (HITS), PageRank Algorithms, Random Walk with Restart, Stochastic Approach for Link-Structure Analysis (SALSA) Algorithms, Vertex Betweenness Centrality Algorithms
Structure evaluation	Adamic-Adar index, Bipartite Check, Conductance, Cycle Detection Algorithms, Degree Distribution Algorithms, Eccentricity Algorithms, K-Core, Local Clustering Coefficient (LCC), Modularity, Partition Conductance, Reachability Algorithms, Topological Ordering Algorithms, Triangle Counting Algorithms

The following topics describe the use of the in-memory graph server (PGX) using Triangle Counting and PageRank analytics as examples.

- [About Built-In Algorithms in the In-Memory Graph Server \(PGX\)](#)
- [Running the Triangle Counting Algorithm](#)
- [Running the PageRank Algorithm](#)

4.8.1 About Built-In Algorithms in the In-Memory Graph Server (PGX)

The in-memory graph server (PGX) contains a set of built-in algorithms that are available as Java APIs. The details of the APIs are documented in the Javadoc that is included in the product documentation library. Specifically, see the `BuiltInAlgorithms` interface Method Summary for a list of the supported in-memory analyst methods.

For example, this is the PageRank procedure signature:

```
/**
 * Classic pagerank algorithm. Time complexity:  $O(E * K)$  with  $E$  = number of edges,  $K$ 
 * is a given constant (max
 * iterations)
 *
 * @param graph
 *       graph
 * @param e
 *       maximum error for terminating the iteration
 * @param d
 *       damping factor
```

```

* @param max
*         maximum number of iterations
* @return Vertex Property holding the result as a double
*/
public <ID extends Comparable<ID>> VertexProperty<ID, Double>
pagerank(PgxGraph graph, double e, double d, int max);

```

4.8.2 Running the Triangle Counting Algorithm

For triangle counting, the `sortByDegree` boolean parameter of `countTriangles()` allows you to control whether the graph should first be sorted by degree (`true`) or not (`false`). If `true`, more memory will be used, but the algorithm will run faster; however, if your graph is very large, you might want to turn this optimization off to avoid running out of memory.

Using the Shell to Run Triangle Counting

```

opg4j> analyst.countTriangles(graph, true)
==> 1

```

Using Java to Run Triangle Counting

```

import oracle.pgx.api.*;

Analyst analyst = session.createAnalyst();
long triangles = analyst.countTriangles(graph, true);

```

The algorithm finds one triangle in the sample graph.

Tip:

When using the graph shell, you can increase the amount of log output during execution by changing the logging level. See information about the `:loglevel` command with `:h :loglevel`.

4.8.3 Running the PageRank Algorithm

PageRank computes a rank value between 0 and 1 for each vertex (node) in the graph and stores the values in a `double` property. The algorithm therefore creates a *vertex property* of type `double` for the output.

In the in-memory graph server (PGX), there are two types of vertex and edge properties:

- **Persistent Properties:** Properties that are loaded with the graph from a data source are fixed, in-memory copies of the data on disk, and are therefore persistent. Persistent properties are read-only, immutable and shared between sessions.
- **Transient Properties:** Values can only be written to transient properties, which are private to a session. You can create transient properties by calling `createVertexProperty` and `createEdgeProperty` on `PgxGraph` objects, or by copying existing properties using `clone()` on `Property` objects.

Transient properties hold the results of computation by algorithms. For example, the PageRank algorithm computes a rank value between 0 and 1 for each vertex in the graph and stores these values in a transient property named `pg_rank`. Transient properties are destroyed when the Analyst object is destroyed.

This example obtains the top three vertices with the highest PageRank values. It uses a transient vertex property of type `double` to hold the computed PageRank values. The PageRank algorithm uses the following default values for the input parameters: error (tolerance = 0.001), damping factor = 0.85, and maximum number of iterations = 100.

Using the Shell to Run PageRank

```
opg4j> rank = analyst.pagerank(graph, 0.001, 0.85, 100);  
==> ...  
opg4j> rank.getTopKValues(3)  
==> 128=0.1402019732468347  
==> 333=0.12002296283541904  
==> 99=0.09708583862990475
```

Using Java to Run PageRank

```
import java.util.Map.Entry;  
import oracle.pgx.api.*;  
  
Analyst analyst = session.createAnalyst();  
VertexProperty<Integer, Double> rank = analyst.pagerank(graph, 0.001, 0.85, 100);  
for (Entry<Integer, Double> entry : rank.getTopKValues(3)) {  
    System.out.println(entry.getKey() + "=" + entry.getValue());  
}
```

4.9 Using Custom PGX Graph Algorithms

A custom PGX graph algorithm allows you to write a graph algorithm in Java and have it automatically compiled to an efficient parallel implementation.

For more detailed information that appears in the following subtopics, see the [PGX Algorithm Specification](#).

- [Writing a Custom PGX Algorithm](#)
- [Compiling and Running a PGX Algorithm](#)
- [Example Custom PGX Algorithm: PageRank](#)

4.9.1 Writing a Custom PGX Algorithm

A PGX algorithm is a regular `.java` file with a single class definition that is annotated with `@GraphAlgorithm`. For example:

```
import oracle.pgx.algorithm.annotations.GraphAlgorithm;  
  
@GraphAlgorithm  
public class MyAlgorithm {  
    ...  
}
```

A PGX algorithm class must contain exactly one public method which will be used as entry point. The class may contain any number of private methods.

For example:

```
import oracle.pgx.algorithm.PgxGraph;
import oracle.pgx.algorithm.VertexProperty;
import oracle.pgx.algorithm.annotations.GraphAlgorithm;
import oracle.pgx.algorithm.annotations.Out;

@GraphAlgorithm
public class MyAlgorithm {
    public int myAlgorithm(PgxGraph g, @Out VertexProperty<Integer>
distance) {
        System.out.println("My first PGX Algorithm program!");

        return 42;
    }
}
```

As with normal Java methods, a PGX algorithm method only supports primitive data types as return values (an integer in this example). More interesting is the `@Out` annotation, which marks the vertex property `distance` as output parameter. The caller passes output parameters by reference. This way, the caller has a reference to the modified property after the algorithm terminates.

- [Collections](#)
- [Iteration](#)
- [Reductions](#)

4.9.1.1 Collections

To create a collection you call the `.create()` function. For example, a `VertexProperty<Integer>` is created as follows:

```
VertexProperty<Integer> distance = VertexProperty.create();
```

To get the value of a property at a certain vertex `v`:

```
distance.get(v);
```

Similarly, to set the property of a certain vertex `v` to a value `e`:

```
distance.set(v, e);
```

You can even create properties of collections:

```
VertexProperty<VertexSequence> path = VertexProperty.create();
```

However, PGX Algorithm assignments are always *by value* (as opposed to *by reference*). To make this explicit, you *must* call `.clone()` when assigning a collection:

```
VertexSequence sequence = path.get(v).clone();
```

Another consequence of values being passed *by value* is that you can check for equality using the `==` operator instead of the Java method `.equals()`. For example:

```
PgxVertex v1 = G.getRandomVertex();  
PgxVertex v2 = G.getRandomVertex();  
System.out.println(v1 == v2);
```

4.9.1.2 Iteration

The most common operations in PGX algorithms are iterations (such as looping over all vertices, and looping over a vertex's neighbors) and graph traversal (such as breath-first/depth-first). All collections expose a `forEach` and `forSequential` method by which you can iterate over the collection in parallel and in sequence, respectively.

For example:

- To iterate over a graph's vertices in parallel:

```
G.getVertices().forEach(v -> {  
    ...  
});
```

- To iterate over a graph's vertices in sequence:

```
G.getVertices().forSequential(v -> {  
    ...  
});
```

- To traverse a graph's vertices from `r` in breadth-first order:

```
import oracle.pgx.algorithm.Traversal;  
  
Traversal.inBFS(G, r).forward(n -> {  
    ...  
});
```

Inside the `forward` (or `backward`) lambda you can access the current level of the BFS (or DFS) traversal by calling `currentLevel()`.

4.9.1.3 Reductions

Within these parallel blocks it is common to atomically update, or reduce to, a variable defined outside the lambda. These atomic reductions are available as methods on

Scalar<T>: `reduceAdd`, `reduceMul`, `reduceAnd`, and so on. For example, to count the number of vertices in a graph:

```
public int countVertices() {
    Scalar<Integer> count = Scalar.create(0);

    G.getVertices().forEach(n -> {
        count.reduceAdd(1);
    });

    return count.get();
}
```

Sometimes you want to update multiple values atomically. For example, you might want to find the smallest property value as well as the vertex whose property value attains this smallest value. Due to the parallel execution, two separate reduction statements might get you in an inconsistent state.

To solve this problem the `Reductions` class provides `argMin` and `argMax` functions. The first argument to `argMin` is the current value and the second argument is the potential new minimum. Additionally, you can chain `andUpdate` calls on the `ArgMinMax` object to indicate other variables and the values that they should be updated to (atomically). For example:

```
VertexProperty<Integer> rank = VertexProperty.create();
int minRank = Integer.MAX_VALUE;
PgxVertex minVertex = PgxVertex.NONE;

G.getVertices().forEach(n ->
    argMin(minRank, rank.get(n)).andUpdate(minVertex, n)
);
```

4.9.2 Compiling and Running a PGX Algorithm

To be able to compile and run a custom PGX algorithm, you must perform several actions:

1. Set two configuration parameters in the `conf/pgx.conf` file:
 - Set the `graph_algorithm_language` option to `JAVA`.
 - Set the `java_home_dir` option to the path to your Java home (use `<system-java-home-dir>` to have PGX infer Java home from the system properties).

```
{
    "graph_algorithm_language": "JAVA",
    "java_home_dir": "<system-java-home-dir>"
}
```

2. Create a session (either implicitly in the shell or explicitly in Java). For example:

```
cd $PGX_HOME
./bin/opg4j
```

3. Compile a PGX Algorithm. For example:

```
myAlgorithm = session.compileProgram("/path/to/MyAlgorithm.java")
```

4. Run the algorithm. For example:

```
graph = session.readGraphWithProperties("/path/to/config.edge.json")
property = graph.createVertexProperty(PropertyType.INTEGER)
myAlgorithm.run(graph, property)
```

4.9.3 Example Custom PGX Algorithm: PageRank

The following is an implementation of pagerank as a PGX algorithm:

```
import oracle.pgx.algorithm.PgxGraph;
import oracle.pgx.algorithm.Scalar;
import oracle.pgx.algorithm.VertexProperty;
import oracle.pgx.algorithm.annotations.GraphAlgorithm;
import oracle.pgx.algorithm.annotations.Out;

@GraphAlgorithm
public class Pagerank {
    public void pagerank(PgxGraph G, double tol, double damp, int max_iter,
        boolean norm, @Out VertexProperty<Double> rank) {
        Scalar<Double> diff = Scalar.create();
        int cnt = 0;
        double N = G.getNumVertices();

        rank.setAll(1 / N);
        do {
            diff.set(0.0);
            Scalar<Double> dangling_factor = Scalar.create(0d);

            if (norm) {
                dangling_factor.set(damp / N * G.getVertices().filter(v ->
                    v.getOutDegree() == 0).sum(rank::get));
            }

            G.getVertices().forEach(t -> {
                double in_sum = t.getInNeighbors().sum(w -> rank.get(w) /
                    w.getOutDegree());
                double val = (1 - damp) / N + damp * in_sum + dangling_factor.get();
                diff.reduceAdd(Math.abs(val - rank.get(t)));
                rank.setDeferred(t, val);
            });
            cnt++;
        } while (diff.get() > tol && cnt < max_iter);
    }
}
```

4.10 Creating Subgraphs

You can create subgraphs based on a graph that has been loaded into memory. You can use filter expressions or create bipartite subgraphs based on a vertex (node) collection that specifies the left set of the bipartite graph.

For information about reading a graph into memory, see [Loading a Graph Into the Graph Server \(PGX\)](#) for the various methods to load a graph into the in-memory graph server (PGX).

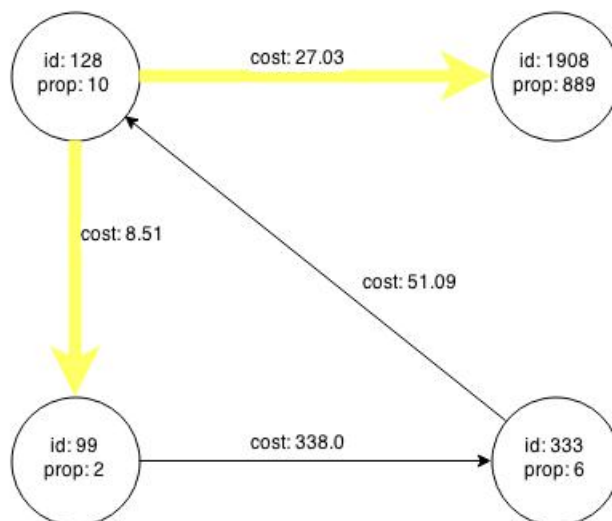
- [About Filter Expressions](#)
- [Using a Simple Filter to Create a Subgraph](#)
- [Using a Complex Filter to Create a Subgraph](#)
- [Using a Vertex Set to Create a Bipartite Subgraph](#)

4.10.1 About Filter Expressions

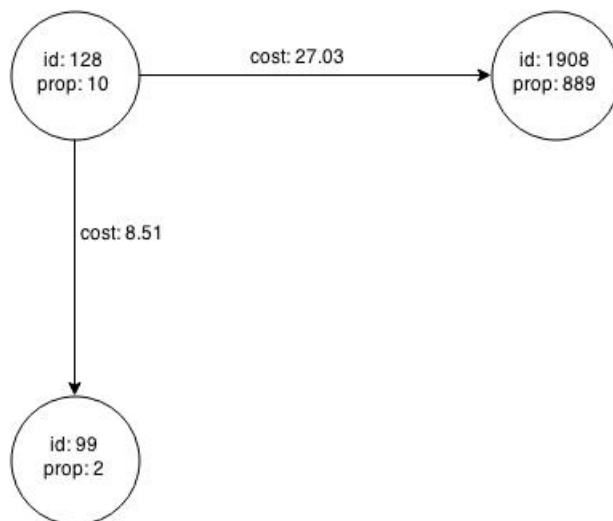
Filter expressions are expressions that are evaluated for each edge. The expression can define predicates that a vertex or an edge must fulfil to be contained in the result, in this case a subgraph.

Consider an example graph that consists of four vertices (nodes) and four edges. For an edge to match the filter expression `src.prop == 10`, the source vertex `prop` property must equal 10. Two edges match that filter expression, as shown in the following figure.

Figure 4-5 Edges Matching `src.prop == 10`



The following figure shows the graph that results when the filter is applied.

Figure 4-6 Graph Created by the Simple Filter

The vertex filter `src.prop == 10` filters out the edges associated with vertex 333 and the vertex itself.

4.10.2 Using a Simple Filter to Create a Subgraph

The following examples create the subgraph described in [About Filter Expressions](#).

Using the Shell to Create a Subgraph

```
subgraph = graph.filter(new VertexFilter("vertex.prop == 10"))
```

Using Java to Create a Subgraph

```
import oracle.pgx.api.*;
import oracle.pgx.api.filter.*;

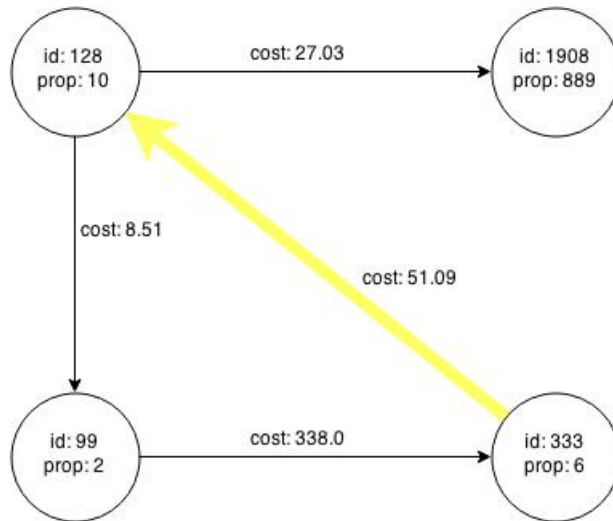
PgxGraph graph = session.readGraphWithProperties(...);
PgxGraph subgraph = graph.filter(new VertexFilter("vertex.prop == 10"));
```

4.10.3 Using a Complex Filter to Create a Subgraph

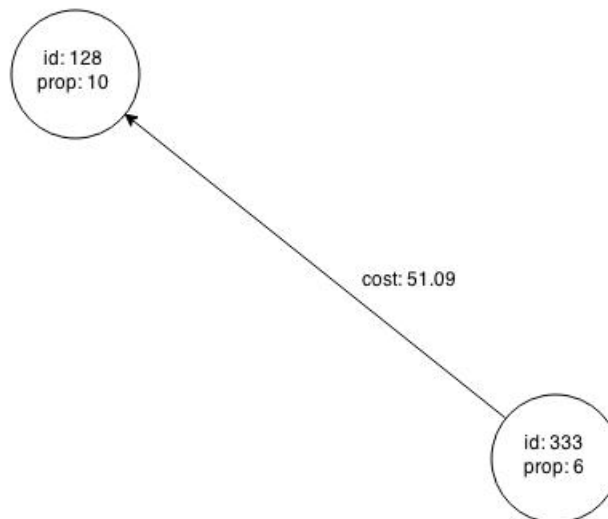
This example uses a slightly more complex filter. It uses the `outDegree` function, which calculates the number of outgoing edges for an identifier (source `src` or destination `dst`). The following filter expression matches all edges with a `cost` property value greater than 50 and a destination vertex (node) with an `outDegree` greater than 1.

```
dst.outDegree() > 1 && edge.cost > 50
```

One edge in the sample graph matches this filter expression, as shown in the following figure.

Figure 4-7 Edges Matching the outDegree Filter

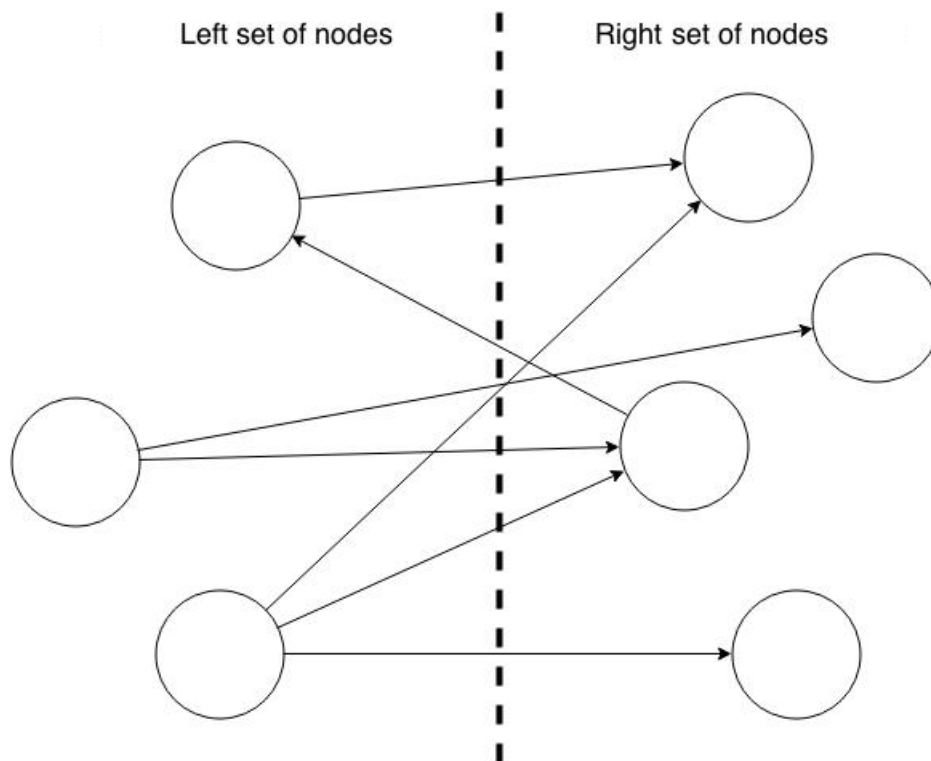
The following figure shows the graph that results when the filter is applied. The filter excludes the edges associated with the vertices 99 and 1908, and so excludes those vertices also.

Figure 4-8 Graph Created by the outDegree Filter

4.10.4 Using a Vertex Set to Create a Bipartite Subgraph

You can create a bipartite subgraph by specifying a set of vertices (nodes), which are used as the left side. A bipartite subgraph has edges only between the left set of vertices and the right set of vertices. There are no edges within those sets, such as between two nodes on the left side. In the in-memory graph server (PGX), vertices that are isolated because all incoming and outgoing edges were deleted are not part of the bipartite subgraph.

The following figure shows a bipartite subgraph. No properties are shown.



The following examples create a bipartite subgraph from the simple graph shown in [About Filter Expressions](#). They create a vertex collection and fill it with the vertices for the left side.

Using the Shell to Create a Bipartite Subgraph

```
opg4j> s = graph.createVertexSet()
==> ...
opg4j> s.addAll([graph.getVertex(333), graph.getVertex(99)])
==> ...
opg4j> s.size()
==> 2
opg4j> bGraph = graph.bipartiteSubGraphFromLeftSet(s)
==> PGX Bipartite Graph named sample-sub-graph-4
```

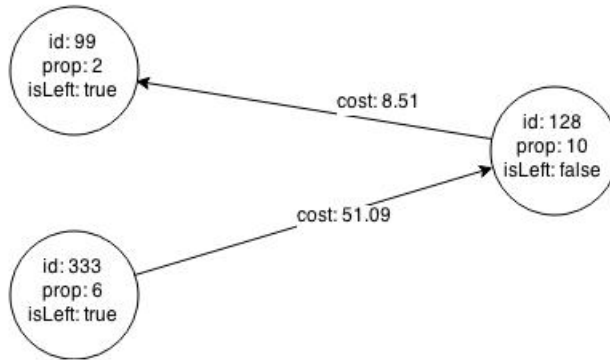
Using Java to Create a Bipartite Subgraph

```
import oracle.pgx.api.*;

VertexSet<Integer> s = graph.createVertexSet();
s.addAll(graph.getVertex(333), graph.getVertex(99));
BipartiteGraph bGraph = graph.bipartiteSubGraphFromLeftSet(s);
```

When you create a subgraph, the in-memory graph server (PGX) automatically creates a Boolean vertex (node) property that indicates whether the vertex is on the left side. You can specify a unique name for the property.

The resulting bipartite subgraph looks like this:



Vertex 1908 is excluded from the bipartite subgraph. The only edge that connected that vertex extended from 128 to 1908. The edge was removed, because it violated the bipartite properties of the subgraph. Vertex 1908 had no other edges, and so was removed as well.

4.11 Using Automatic Delta Refresh to Handle Database Changes

You can automatically refresh (auto-refresh) graphs periodically to keep the in-memory graph synchronized with changes to the property graph stored in the property graph tables in Oracle Database (VT\$ and GE\$ tables).

Note that the auto-refresh feature is not supported when loading a graph into PGX in memory directly from relational tables.

- [Configuring the In-Memory Server for Auto-Refresh](#)
- [Configuring Basic Auto-Refresh](#)
- [Reading the Graph Using the In-Memory Graph Server \(PGX\) or a Java Application](#)
- [Checking Out a Specific Snapshot of the Graph](#)
- [Advanced Auto-Refresh Configuration](#)
- [Special Considerations When Using Auto-Refresh](#)

4.11.1 Configuring the In-Memory Server for Auto-Refresh

Because auto-refresh can create many snapshots and therefore may lead to a high memory usage, by default the option to enable auto-refresh for graphs is available only to administrators.

To allow all users to auto-refresh graphs, you must include the following line into the in-memory graph server (PGX) configuration file (located in `$ORACLE_HOME/md/property_graph/pgx/conf/pgx.conf`):

```
{  
  "allow_user_auto_refresh": true  
}
```

4.11.2 Configuring Basic Auto-Refresh

Auto-refresh is configured in the loading section of the graph configuration. The example in this topic sets up auto-refresh to check for updates every minute, and to create a new snapshot when the data source has changed.

The following block (JSON format) enables the auto-refresh feature in the configuration file of the sample graph:

```
{
  "format": "pg",
  "jdbc_url": "jdbc:oracle:thin:@mydatabaseserver:1521/dbName",
  "username": "scott",
  "password": "<password>",
  "name": "my_graph",
  "vertex_props": [{
    "name": "prop",
    "type": "integer"
  }],
  "edge_props": [{
    "name": "cost",
    "type": "double"
  }],
  "separator": " ",
  "loading": {
    "auto_refresh": true,
    "update_interval_sec": 60
  },
}
```

Notice the additional `loading` section containing the auto-refresh settings. You can also use the Java APIs to construct the same graph configuration programmatically:

```
GraphConfig config = GraphConfigBuilder.forPropertyGraphRdbms()
    .setJdbcUrl("jdbc:oracle:thin:@mydatabaseserver:1521/dbName")
    .setUsername("scott")
    .setPassword("<password>")
    .setName("my_graph")
    .addVertexProperty("prop", PropertyType.INTEGER)
    .addEdgeProperty("cost", PropertyType.DOUBLE)
    .setAutoRefresh(true)
    .setUpdateIntervalSec(60)
    .build();
```

4.11.3 Reading the Graph Using the In-Memory Graph Server (PGX) or a Java Application

After creating the graph configuration, you can load the graph into the in-memory graph server (PGX) using the regular APIs.

```
opg4j> G = session.readGraphWithProperties("graphs/my-config.pg.json")
```

After the graph is loaded, a background task is started automatically, and it periodically checks the data source for updates.

4.11.4 Checking Out a Specific Snapshot of the Graph

The database is queried every minute for updates. If the graph has changed in the database after the time interval passed, the graph is reloaded and a new snapshot is created in-memory automatically.

You can "check out" (move a pointer to a different version of) the available in-memory snapshots of the graph using the `getAvailableSnapshots()` method of `PgxSession`. Example output is as follows:

```
opg4j> session.getAvailableSnapshots(G)
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669
(2016-01-20 10:38:42.669), creationTimestamp=1453315122685 (2016-01-20
10:38:42.685), vertexIdType=integer, edgeIdType=long]
==> GraphMetaData [getNumVertices()=5, getNumEdges()=5, memoryMb=3,
dataSourceVersion=1452083654000, creationRequestTimestamp=1453314938744
(2016-01-20 10:35:38.744), creationTimestamp=1453314938833 (2016-01-20
10:35:38.833), vertexIdType=integer, edgeIdType=long]
```

The preceding example output contains two entries, one for the originally loaded graph with 4 vertices and 4 edges, and one for the graph created by auto-refresh with 5 vertices and 5 edges.

To check out a specific snapshot of the graph, use the `setSnapshot()` methods of `PgxSession` and give it the `creationTimestamp` of the snapshot you want to load.

For example, if `G` is pointing to the newer graph with 5 vertices and 5 edges, but you want to analyze the older version of the graph, you need to set the snapshot to 1453315122685. In the graph shell:

```
opg4j> G.getNumVertices()
==> 5
opg4j> G.getNumEdges()
==> 5

opg4j> session.setSnapshot( G, 1453315122685 )
==> null

opg4j> G.getNumVertices()
==> 4
opg4j> G.getNumEdges()
==> 4
```

You can also load a specific snapshot of a graph directly using the `readGraphAsOf()` method of `PgxSession`. This is a shortcut for loading a graph with `readGraphWithProperty()` followed by a `setSnapshot()`. For example:

```
opg4j> G = session.readGraphAsOf( config, 1453315122685 )
```

If you do not know or care about what snapshots are currently available in-memory, you can also specify a time span of how “old” a snapshot is acceptable by specifying a maximum allowed age. For example, to specify a maximum snapshot age of 60 minutes, you can use the following:

```
opg4j> G = session.readGraphWithProperties( config, 60l, TimeUnit.MINUTES )
```

If there are one or more snapshots in memory younger (newer) than the specified maximum age, the youngest (newest) of those snapshots will be returned. If all the available snapshots are older than the specified maximum age, or if there is no snapshot available at all, then a new snapshot will be created automatically.

4.11.5 Advanced Auto-Refresh Configuration

You can specify advanced options for auto-refresh configuration.

Internally, the in-memory graph server (PGX) fetches the changes since the last check from the database and creates a new snapshot by applying the delta (changes) to the previous snapshot. There are two timers: one for fetching and caching the deltas from the database, the other for actually applying the deltas and creating a new snapshot.

Additionally, you can specify a threshold for the number of cached deltas. If the number of cached changes grows above this threshold, a new snapshot is created automatically. The number of cached changes is a simple sum of the number of vertex changes plus the number of edge changes.

The deltas are fetched periodically and cached on the in-memory graph server for two reasons:

- To speed up the actual snapshot creation process
- To account for the case that the database can “forget” changes after a while

You can specify both a threshold and an update timer, which means that both conditions will be checked before new snapshot is created. At least one of these parameters (threshold or update timer) must be specified to prevent the delta cache from becoming too large. The interval at which the source is queried for changes must not be omitted.

The following parameters show a configuration where the data source is queried for new deltas every 5 minutes. New snapshots are created every 20 minutes or if the cached deltas reach a size of 1000 changes.

```
{
  "format": "pg",
  "jdbc_url": "jdbc:oracle:thin:@mydatabaseserver:1521/dbName",
  "username": "scott",
  "password": "<your_password>",
  "name": "my_graph",

  "loading": {
    "auto_refresh": true,
    "fetch_interval_sec": 300,
    "update_interval_sec": 1200,
    "update_threshold": 1000,
    "create_edge_id_index": true,
    "create_edge_id_mapping": true
  }
}
```

```
    }
}
```

4.11.6 Special Considerations When Using Auto-Refresh

This section explains a few special considerations when you enable auto-refresh for graphs in the in-memory graph server (PGX):

- If you call `graph.destroy()`, auto-refresh does not immediately stop. It only stops once the graph is actually freed from the server memory.

This happens when all the following conditions are true:

 1. No other session is referencing that graph.
 2. PGX consumes more than `release_memory_threshold` memory. `release_memory_threshold` is a `pgx.conf` option that defaults to 85% of available system memory.
 3. The PGX "garbage collector" has been run. `memory_cleanup_interval` is a `pgx.conf` option which defaults to once every 10 minutes.
- If you configure the graph to be loaded with auto-refresh, you cannot omit the `jdbc url`, `username` and `keystore` parameters from the graph configuration file since auto-refreshed graphs are not "user bound". You cannot obtain the connection settings from the user who initiated it.

4.12 Starting the In-Memory Graph Server (PGX)

This section describes the commands to start and stop the in-memory graph server (PGX).

A preconfigured version of Apache Tomcat is bundled, which allows you to start the in-memory graph server (PGX) by running a script.

As a prerequisite to start the graph server in remote mode, you must ensure that Oracle graph server is installed in your system. See [Installing Oracle Graph Server](#) for instructions to install the graph server (PGX).



Note:

See [Usage Modes of the In-memory Graph Server \(PGX\)](#) for more information on the different graph server execution modes.

- [Starting and Stopping the Graph Server \(PGX\) Using the Command Line](#)
- [Configuring the In-Memory Graph Server \(PGX\)](#)

4.12.1 Starting and Stopping the Graph Server (PGX) Using the Command Line

PGX is integrated with `systemd` to run it as a Linux service in the background.

If you need to configure the server before starting it, see [Configuring the In-Memory Graph Server \(PGX\)](#) and [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information on the configuration options.

The commands to start and stop the graph server (PGX) and the PGX engine are as follows:

**Note:**

You can run the following commands without `sudo` if you are the root user.

To start the PGX server as a daemon process, run the following command:

```
sudo systemctl start pgx
```

To stop the server, run the following command:

```
sudo systemctl stop pgx
```

If the server does not start up, you can see if there are any errors by running:

```
journalctl -u pgx.service
```

For more information about how to interact with `systemd` on Oracle Linux, see the Oracle Linux administrator's documentation.

4.12.2 Configuring the In-Memory Graph Server (PGX)

You can configure the in-memory graph server (PGX) by modifying the `/etc/oracle/graph/server.conf` file. The following table shows the valid configuration options, which can be specified in JSON format.

Table 4-6 Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
<code>ca_certs</code>	array of string	List of files storing trusted certificates (PEM format). If <code>enable_tls</code> is set to <code>false</code> , this field has no effect.	<code>[]</code>

Table 4-6 (Cont.) Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
ciphers	array of string	List of cipher suites to be used by the server. For example, [cipher1, cipher2.]	["TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256" , "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384" , "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256" , "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" , "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256" , "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256" , "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384" , "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384" , "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256" , "TLS_DHE_DSS_WITH_AES_128_GCM_SHA256" , "TLS_DHE_RSA_WITH_AES_128_CBC_SHA256" , "TLS_DHE_DSS_WITH_AES_128_CBC_SHA256" , "TLS_DHE_DSS_WITH_AES_256_GCM_SHA384" , "TLS_DHE_RSA_WITH_AES_256_CBC_SHA256" , "TLS_DHE_DSS_WITH_AES_256_CBC_SHA256" , "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA" , "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA" , "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA" , "TLS_DHE_DSS_WITH_AES_128_CBC_SHA" , "TLS_DHE_RSA_WITH_AES_128_CBC_SHA" , "TLS_DHE_DSS_WITH_AES_256_CBC_SHA" , "TLS_DHE_RSA_WITH_AES_256_CBC_SHA" , "TLS_RSA_WITH_AES_128_GCM_SHA256" , "TLS_DH_DSS_WITH_AES_128_GCM_SHA256" , "TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256" , "TLS_RSA_WITH_AES_256_GCM_SHA384" , "TLS_DH_DSS_WITH_AES_256_GCM_SHA384"]

Table 4-6 (Cont.) Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
			A384", "TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384", "TLS_RSA_WITH_AES_128_CBC_SHA256", "TLS_DH_DSS_WITH_AES_128_CBC_SHA256", "TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256", "TLS_RSA_WITH_AES_256_CBC_SHA256", "TLS_DH_DSS_WITH_AES_256_CBC_SHA256", "TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384", "TLS_RSA_WITH_AES_128_CBC_SHA", "TLS_DH_DSS_WITH_AES_128_CBC_SHA", "TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA", "TLS_RSA_WITH_AES_256_CBC_SHA", "TLS_DH_DSS_WITH_AES_256_CBC_SHA", "TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA"]
context_path	string	This can be used to change the context path. For example, if you specify port as 7007 and context path as /pgx, the server will listen on https://localhost:7007/pgx	/
enable_tls	boolean	If true, the server enables transport layer security (TLS).	true
port	integer	Port the graph server (PGX) server should listen on.	7007

Table 4-6 (Cont.) Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
server_cert	string	The path to the server certificate to be presented to TLS clients (PEM format).	NULL


 **Note**: This file must only contain one certificate.

Table 4-6 (Cont.) Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
			f y o u r c e r t i f i c a t e i s a c h a i n a n d c o n t a i n s a r o o t c e r t i f i c a t e , a

Table 4-6 (Cont.) Configuration Parameters for the In-Memory Graph Server (PGX)

Parameter	Type	Description	Default
			default
server_private_key	string	If enable_tls is set to false, this field has no effect. This is the path to the file storing the private key of the server (PEM format). For security reasons, the file must have only Read and Write permissions only for the owner (600 permissions in a POSIX filesystem), otherwise an error will be thrown. If enable_tls is set to false, this field has no effect.	NULL
tls_version	string	TLS version to be used by the server. For example, TLSv1.2	TLSv1.2
working_dir	string	The working directory used by the server to store temporary files. Needs to be writeable by the process which started the server and should not be touched by any other process while the server is running.	

The in-memory graph server (PGX) enables two-way SSL/TLS (Transport Layer Security) by default. The server enforces TLS 1.2 and disables certain cipher suites known to be vulnerable to attacks. Upon a TLS handshake, both the server and the

client present certificates to each other, which are used to validate the authenticity of the other party. Client certificates are also used to authorize client applications.

The following is an example `server.conf` configuration file:

```
{
  "port": 7007,
  "enable_tls": true,
  "server_cert": "server_cert.pem",
  "server_private_key": "server_key.pem",
  "ca_certs": [
    "server_cert.pem"
  ]
}
```

4.13 Connecting to the In-Memory Graph Server (PGX)

This section explains how to connect to the in-memory graph server (PGX) running in remote mode or when deployed as a web application on Apache Tomcat or Oracle WebLogic Server.

The prerequisite requirement to connect to the graph server is to have the in-memory graph server (PGX) up and running. See [Starting and Stopping the Graph Server \(PGX\) Using the Command Line](#) for more information on the commands to start the graph server.



Note:

If you are using the graph server (PGX) as a library, see [Using Graph Server \(PGX\) as a Library](#) for more information.

- [Connecting with the Graph Shell](#)
- [Connecting with Java](#)
- [Connecting with Python](#)

4.13.1 Connecting with the Graph Shell

The simplest way to connect to a remote graph server (PGX) instance is to specify the base URL of the server along with the database user name required for the graph server (PGX) authentication as shown:

```
cd $PGX_HOME
./bin/opg4j --base_url https://<host>:<port> --username <graphuser>
```

where :

- `<host>`: is the server host name
- `<port>`: is the server port
- `<graphuser>`: is the database user

 **Note:**

You will be prompted for the database password.

See [User Authentication and Authorization](#) for more information.

However, the in-memory graph server (PGX), currently does not provide remote support for the [Admin API](#).

About Logging HTTP Requests

The graph shell suppresses all debugging messages by default. To see which HTTP requests are executed, set the log level for `oracle.pgx` to `DEBUG`, as shown in this example:

 **Note:**

Enabling these logs can lead to sensitive information like passwords getting printed on the screen.

```

opg4j> loglevel("oracle.pgx","DEBUG")
==> Log level of oracle.pgx logger set to DEBUG
opg4j> session.readGraphWithProperties("bank_graph_analytics.json",
"bank_graph");
06:29:03,702 DEBUG CommonsVfsProvider - resolve bank_graph_analytics.json
06:29:03,702 DEBUG AbstractConfigFactory - parse graph config from
bank_graph_analytics.json (parent: file:///opt/oracle/graph)
06:29:03,709 DEBUG RemoteUtils - create session cookie (session ID =
f5d029d7-2924-4cd4-86a9-6999clce5e3f)
06:29:03,713 DEBUG RemoteUtils - no value for the sticky cookie given
06:29:03,713 DEBUG RemoteUtils - create csrf token cookie (token =
36acbee2-6b78-4c13-b114-41040809833a)
06:29:03,713 DEBUG HttpRequestExecutor - Requesting POST https://localhost:7007/
core/v1/loadGraph HTTP/1.1 with payload
{"graphConfig":{"HRcBVfVc00dfXU9bUEhGEEYodkMUElZYRFpcZgBeDxBYcXFcRGY2c21wIBUBEE1AW
11HQV5vVhpYAFRIFAMbeC5+NithRx9EEkwUVRADRLFBXVhGf1BpT0ZfSEFPAlZBQ1RXG1kTKiEXSREVCU
AWU1dmElJfR1xKa11GDBJdSV1EQwMpd1paVhZfFxyXSREIB1gBEggbMEVMXEpUUUV9HQUGWSV1FFVBDV01
QVg1uAApZEF4IRA9GdHdqMGhkdhseFk1REBBdQ11CCFZDaU9cSxdUGzpFF1wQD1EBQhADRnZOUVZHW11H
QUGwQVdXBVBDURsDQkFSEQBUEVY5DVAdb19YFEdEXF4QDktVDxdRUBQUBVhZV1tYSgZuFwRXCvY5CFQJV
RADRnVsfHJtcWlzJjdrbHViQxUPXVxAZhdIEwAXXxEKCVsDeh4bAlhfX1hGFhcWEQBWQEsUHGQBFE9cSx
dUGzpFF1wQD1EBQkEbXmxWEFJXTXJXDahBQFYUWxtkchsVGw1QDgAXXxEnBVYLRVxNFxUBEFVdVulddQM
WFOMUAktIV01cZghUGjpYBEMWD1sDEghNFkJITxUQUExAAgZVX11pFVhPW1xmVwJcBkcPR3EnKH47fn19
IWQPHhtZUVRrFx1ESBoMQ1BDQ1xeXBETT0dTcKELB0FGChBLAFVAQRtPaQEWDQVZSBOMQ1tMWFJmXhFQE
w1qBF0HckwQWVFKRkotMjkyAE1r0w==" ,"graphName":"bank_graph", "_csrf_token":"36acbee2
-6b78-4c13-b114-41040809833a"}
06:29:03,788 DEBUG HttpRequestExecutor - received HTTP status 202
06:29:03,789 DEBUG HttpRequestExecutor -
{"futureId":"7f7a2206-8881-4c1e-909f-6e8778be617c"}
06:29:03,789 DEBUG PgxRemoteFuture - Requesting GET https://localhost:7007/
core/v1/futures/x-future-id/status HTTP/1.1
06:29:03,801 DEBUG PgxRemoteFuture - Requesting GET https://localhost:7007/
core/v1/futures/x-future-id/value HTTP/1.1
06:29:03,831 DEBUG RemoteUtils - received HTTP status 201
06:29:03,831 DEBUG RemoteUtils - {"id":"8B473228-0751-49A9-
A945-9A0E4011AB69", "links":[{"href":"https://localhost:7007/core/v1/graphs/x-

```



```

graph-id", "rel": "self", "method": "GET", "interaction": [{"async-polling"}]},
{"href": "https://localhost:7007/core/v1/graphs/x-graph-
id", "rel": "canonical", "method": "GET", "interaction": [{"async-
polling"}]}, "graphName": "bank_graph", "vertexTables": {"Accounts":
{"name": "Accounts", "metaData": {"name": "Accounts", "idType": "integer", "labels":
["Accounts"], "properties": [], "edgeProviderNamesWhereSource":
["Transfers"], "edgeProviderNamesWhereDestination":
["Transfers"], "id": null, "links": null}, "providerLabels":
["Accounts"], "entityKeyType": "integer", "isIdentityKeyMapping": false, "vertexProperties":
{"vertexLabels": {"id": "04156FFE-A3C1-4A6D-87E5-879A0895BBD4", "links":
[{"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-
name", "rel": "self", "method": "GET", "interaction": [{"async-polling"}]}, {"href": "https://
localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-
name", "rel": "canonical", "method": "GET", "interaction": [{"async-
polling"}]}], "dimension": -1, "propertyId": "04156FFE-
A3C1-4A6D-87E5-879A0895BBD4", "name": "__vertex_labels__", "entityType": "vertex", "type": "r
o_string_set", "namespace": "2C17C639-3771-3E30-88AE-34D6B380C5EC", "transient": false}, "tr
ansient": false}}, "edgeTables": {"Transfers": {"name": "Transfers", "metaData":
{"name": "Transfers", "idType": "long", "directed": true, "labels":
["Transfers"], "properties":
[{"name": "AMOUNT", "id": null, "propertyType": "float", "dimension": 0, "transient": true, "link
s": null, "propertyId": "AF2A2D0A-9C8C-478F-
BD74-3444A7DD7339"}], "sourceVertexProviderName": "Accounts", "destinationVertexProviderNa
me": "Accounts", "id": null, "links": null}, "providerLabels":
["Transfers"], "entityKeyType": "long", "isIdentityKeyMapping": true, "sourceVertexTableNa
me": "Accounts", "destinationVertexTableName": "Accounts", "edgeProperties": {"4046D845-
D0C6-4231-A69B-F69D4963CD91": {"id": "4046D845-D0C6-4231-A69B-F69D4963CD91", "links":
[{"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-
name", "rel": "self", "method": "GET", "interaction": [{"async-polling"}]}, {"href": "https://
localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-
name", "rel": "canonical", "method": "GET", "interaction": [{"async-
polling"}]}], "dimension": 0, "propertyId": "4046D845-D0C6-4231-A69B-
F69D4963CD91", "name": "AMOUNT", "entityType": "edge", "type": "float", "namespace": "2C17C639-
3771-3E30-88AE-34D6B380C5EC", "transient": false}}, "edgeLabel":
{"id": "9763546A-1860-49A4-9292-77D2AA04F4BB", "links
06:29:03,836 DEBUG PgxSession - engine reports latest snapshot is 621849 milli-seconds
old. Max age is 0 milli-seconds
06:29:03,836 DEBUG PgxSession - ==> try to check out newer snapshot
06:29:03,836 DEBUG RemoteUtils - create session cookie (session ID =
f5d029d7-2924-4cd4-86a9-6999c1ce5e3f)
06:29:03,836 DEBUG RemoteUtils - no value for the sticky cookie given
06:29:03,836 DEBUG RemoteUtils - create csrf token cookie (token = 36acbee2-6b78-4c13-
b114-41040809833a)
06:29:03,836 DEBUG HttpRequestExecutor - Requesting POST https://localhost:7007/
core/v1/graphs/x-graph-id/refresh HTTP/1.1 with payload
{"blockIfFull": false, "_csrf_token": "36acbee2-6b78-4c13-b114-41040809833a"}
06:29:03,878 DEBUG HttpRequestExecutor - received HTTP status 202
06:29:03,878 DEBUG HttpRequestExecutor - {"futureId": "898d546e-583f-4d37-9ca9-
d1e10134037f"}
06:29:04,135 DEBUG PgxRemoteFuture - Requesting GET https://localhost:7007/core/v1/
futures/x-future-id/status HTTP/1.1
06:29:04,828 DEBUG PgxRemoteFuture - Requesting GET https://localhost:7007/core/v1/
futures/x-future-id/value HTTP/1.1
06:29:04,858 DEBUG RemoteUtils - received HTTP status 201
06:29:04,859 DEBUG RemoteUtils - {"id": "BE960B34-E135-4CF8-AB2F-E1A6E2D7DB60", "links":
[{"href": "https://localhost:7007/core/v1/graphs/x-graph-
id", "rel": "self", "method": "GET", "interaction": [{"async-polling"}]}, {"href": "https://
localhost:7007/core/v1/graphs/x-graph-
id", "rel": "canonical", "method": "GET", "interaction": [{"async-
polling"}]}], "graphName": "bank_graph", "vertexTables": {"Accounts":
{"name": "Accounts", "metaData": {"name": "Accounts", "idType": "integer", "labels":

```

```

["Accounts"], "properties": [], "edgeProviderNamesWhereSource":
["Transfers"], "edgeProviderNamesWhereDestination":
["Transfers"], "id": null, "links": null, "providerLabels":
["Accounts"], "entityKeyType": "integer", "isIdentityKeyMapping": false, "vertexProperties": {}, "vertexLabels": {"id": "19D95502-40D5-47F2-9F45-B1CD09ECB989", "links": [{"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-name", "rel": "self", "method": "GET", "interaction": ["async-polling"]}, {"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-name", "rel": "canonical", "method": "GET", "interaction": ["async-polling"]}], "dimension": -1, "propertyId": "19D95502-40D5-47F2-9F45-B1CD09ECB989", "name": "__vertex_labels__", "entityType": "vertex", "type": "ro_string_set", "namespace": "2C17C639-3771-3E30-88AE-34D6B380C5EC", "transient": false}, "transient": false}, "edgeTables": {"Transfers": {"name": "Transfers", "metaData": {"name": "Transfers", "idType": "long", "directed": true, "labels": ["Transfers"], "properties": [{"name": "AMOUNT", "id": null, "propertyType": "float", "dimension": 0, "transient": true, "links": null, "propertyId": "9A49BC0C-F8AA-465A-B8D6-CA5A92BAE2C9"}], "sourceVertexProviderName": "Accounts", "destinationVertexProviderName": "Accounts", "id": null, "links": null}, "providerLabels": ["Transfers"], "entityKeyType": "long", "isIdentityKeyMapping": true, "sourceVertexTableName": "Accounts", "destinationVertexTableName": "Accounts", "edgeProperties": {"FED6FE43-D311-46B6-9A5A-E8DC0D7B56C6": {"id": "FED6FE43-D311-46B6-9A5A-E8DC0D7B56C6", "links": [{"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-name", "rel": "self", "method": "GET", "interaction": ["async-polling"]}, {"href": "https://localhost:7007/core/v1/graphs/x-graph-id/properties/x-property-name", "rel": "canonical", "method": "GET", "interaction": ["async-polling"]}], "dimension": 0, "propertyId": "FED6FE43-D311-46B6-9A5A-E8DC0D7B56C6", "name": "AMOUNT", "entityType": "edge", "type": "float", "namespace": "2C17C639-3771-3E30-88AE-34D6B380C5EC", "transient": false}}, "edgeLabel": {"id": "371D2AC6-4EC5-45AD-8885-B3590F56D944", "links
$5 ==> PgxGraph[name=bank_graph,N=1000,E=5001,created=1621160944599]

```

4.13.2 Connecting with Java

You can obtain a connection to a remote graph server (PGX) instance by simply passing the base URL of the remote PGX instance to the `getInstance()` method. By doing this, your application automatically uses the PGX client libraries to connect to a remotely-located graph server (PGX).

You can specify the base URL when you initialize the in-memory graph server (PGX) instance using Java. An example is as follows. A URL to an in-memory graph server (PGX) is provided to the `getInMemAnalyst` API call.

```

import oracle.pgx.api.*;
import oracle.pg.rdbms.api.*;
ServerInstance instance = GraphServer.getInstance("https://
<hostname>:<port>", "<username>", "<password>".toCharArray());
PgxSession session = instance.createSession("my-session");

```

- [Starting and Stopping the PGX Engine](#)

4.13.2.1 Starting and Stopping the PGX Engine

You can start the graph server (PGX) from the application by making a call to `instance.startEngine()` which takes a JSON object as an argument for PGX configuration.

 **Note:**

- See [Connecting with Java](#) for more information about connecting to a graph server (PGX) instance and obtaining a `ServerInstance` object.
- See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for the various configuration options for the graph server (PGX).

Stopping the PGX Engine

You can stop the PGX engine using one of the following APIs:

```
instance.shutdownEngineNow(); // cancels pending tasks, throws exception if
engine is not running
instance.shutdownEngineNowIfRunning(); // cancels pending tasks, only tries
to shut down if engine is running
if (instance.shutdownEngine(30, TimeUnit.SECONDS) == false) {
    // doesn't accept new tasks but finishes up remaining tasks
    // pending tasks didn't finish after 30 seconds
}
```

 **Note:**

Shutting down the PGX engine keeps the Apache Tomcat server alive, but new sessions cannot be created. Also, all the current sessions and tasks will be cancelled and terminated.

4.13.3 Connecting with Python

You can connect to a remote graph server (PGX) instance in your Python program. You must first authenticate with the remote server before you can create a session as illustrated in the following example:

```
import pypgx as pgx
import pypgx.pg.rdbms.graph_server as graph_server

base_url = "https://localhost:7007"
username = "scott"
password = "tiger"

instance = graph_server.get_instance(base_url, username, password)
session = instance.create_session("python_pgx_client")
print(session)
```

To execute, save the above program into a file named `program.py` and run the following command:

```
python3 program.py
```

After successful login, you'll see the following message indicating a PGX session was created:

```
PgxSession(id: 0bdd4828-c3cc-4cef-92c8-0fcd105416f0, name:
python_pgx_client)
```

**Note:**

To view the complete set of available Python APIs, see [Pypgx API](#).

4.14 Using Graph Server (PGX) as a Library

When you utilize PGX as a library in your application, the graph server (PGX) instance runs in the same JVM as the Java application and all requests are translated into direct function calls instead of remote procedure invocations.

In this case, you must install the graph server (PGX) using `RPM` in the same machine as the client applications. The shell executables provided by the graph server installation helps you to launch the Java or the Python shell in an embedded server mode. See [Installing Oracle Graph Server](#) for more information.

You can now start the Java shell without any parameters as shown:

```
cd /opt/oracle/graph
./bin/opg4j
```

The local PGX instance will try to load a PGX configuration file from:

```
/etc/oracle/graph/pgx.conf
```

You can change the location of the configuration file by passing the `--pgx_conf` command-line option followed by the path to the configuration file:

```
# start local PGX instance with custom config
./bin/opg4j --pgx_conf <path_to_pgx.conf>
```

You can also start the Python shell without any parameters as shown:

```
cd /opt/oracle/graph/
./bin/opg4py
```

When using Java, you can obtain a reference to the local PGX instance as shown:

```
import oracle.pg.rdbms.*;
...
ServerInstance instance = GraphServer.getEmbeddedInstance();
```

In a Python application, you can obtain a reference to the local PGX instance as shown:

```
import os
os.environ["PGX_CLASSPATH"] = "/opt/oracle/graph/lib/*"
import pypgx.pg.rdbms.graph_server as graph_server
...
instance = graph_server.get_embedded_instance()
```

Starting the PGX Engine

PGX provides a convenience mechanism to start the PGX Engine when using the graph server (PGX) as a library. That is, the graph server (PGX) is automatically initialized and starts up automatically when `ServerInstance.createSession()` is called the first time. This is provided that the engine is not already running at that time.

For this implicit initialization, PGX will configure itself with the PGX configuration file at the default locations. If the PGX configuration file is not found, PGX will configure itself using default parameter values as shown in [Configuration Parameters for the Graph Server \(PGX\) Engine](#).

Stopping the PGX Engine

When using the graph server (PGX) as a library, the `shutdownEngine()` method will be called automatically via a JVM shutdown hook on exit. Specifically, the shutdown hook is invoked once all the [non-daemon threads](#) of the application exit.

It is recommended that you do not terminate your PGX application forcibly with `kill -9`, as it will not clear the `temp` directory. See `tmp_dir` in [Configuration Parameters for the Graph Server \(PGX\) Engine](#).

4.15 User-Defined Functions (UDFs) in PGX

User-defined functions (UDFs) allow users of PGX to add custom logic to their PGQL queries or custom graph algorithms, to complement built-in functions with custom requirements.

▲ Caution:

UDFs enable running arbitrary code in the PGX server, possibly accessing sensitive data. Additionally, any PGX session can invoke any of the UDFs that are enabled on the PGX server. The application administrator who enables UDFs is responsible for checking the following:

- All the UDF code can be trusted.
- The UDFs are stored in a secure location that cannot be tampered with.

Furthermore, PGX assumes UDFs to be state-less and side-effect free.

PGX supports two types of UDFs:

- Java UDFs
- JavaScript UDFs

How to Use Java UDFs

The following simple example shows how to register a Java UDF at the PGX server and invoke it.

1. Create a class with a public static method. For example:

```
package my.udfs;

public class MyUdfs {
    public static String concat(String a, String b) {
        return a + b;
    }
}
```

2. Compile the class and compress into a JAR file. For example:

```
mkdir ./target
javac -d ./target *.java
cd target
jar cvf MyUdfs.jar *
```

3. Copy the JAR file into `/opt/oracle/graph/pgx/server/lib`.
4. Create a UDF JSON configuration file. For example, assume that `/path/to/my/udfs/dir/my_udfs.json` contains the following:

```
{
  "user_defined_functions": [
    {
      "namespace": "my",
      "language": "java",
      "implementation_reference": "my.udfs.MyUdfs",
      "function_name": "concat",
      "return_type": "string",
      "arguments": [
        {
          "name": "a",
          "type": "string"
        },
        {
          "name": "b",
          "type": "string"
        }
      ]
    }
  ]
}
```

5. Point to the directory containing the UDF configuration file in `/etc/oracle/graph/pgx.conf`. For example:

```
"udf_config_directory": "/path/to/my/udfs/dir/"
```

6. Restart the PGX server. For example:

```
sudo systemctl restart pgx
```

7. Try to invoke the UDF from within a PGQL query. For example:

```
graph.queryPgql("SELECT my.concat(my.concat(n.firstName, ' '),
n.lastName) FROM MATCH (n:Person)")
```

8. Try to invoke the UDF from within a PGX algorithm. For example:

 **Note:**

For each UDF you want to use, you need to create an abstract method with the same schema that gets annotated with the `@Udf` annotation.

```
import oracle.pgx.algorithm.annotations.Udf;
....

@GraphAlgorithm
public class MyAlgorithm {
    public void bomAlgorithm(PgxGraph g, VertexProperty<String> firstName,
VertexProperty<String> lastName, @Out VertexProperty<String> fullName) {

        ... fullName.set(v, concat(firstName.get(v), lastName.get(v))); ...

    }

    @Udf(namespace = "my")
    abstract String concat(String a, String b);
}
```

JavaScript UDFs

The requirements for a JavaScript UDF is as follows:

- The JavaScript source must contain all dependencies.
- The source must contain at least one valid export.
- The language parameter must be set to `javascript` in the UDF configuration file.

For example, consider a JavaScript source file `format.js` as shown:

```
//format.js
const fun = function(name, country) {
    if (country == null) return name;
    else return name + " (" + country + ")";
}

module.exports = {stringFormat: fun};
```

In order to load the UDF from `format.js`, the UDF configuration file will appear as follows:

```
{
  "namespace": "my",
```

```

"function_name": "format",
"language": "javascript",
"source_location": "format.js",
"source_function_name": "stringFormat",
"return_type": "string",
"arguments": [
  {
    "name": "name",
    "type": "string"
  },
  {
    "name": "country",
    "type": "string"
  }
]
}

```



Note:

In this case, since the name of the UDF and the implementing method differ, you need to set the name of the UDF in the `source_function_name` field. Also, you can provide the path of the source code file in the `source_location` field.

UDF Configuration File Information

A UDF configuration file is a JSON file containing an array of `user_defined_functions`. (An example of such a file is in the step to "Create a UDF JSON configuration file" in the preceding [How to Use Java UDFs](#) subsection.)

Each user-defined function supports the fields shown in the following table.

Table 4-7 Fields for Each UDF

Field	Data Type	Description	Required?
<code>function_name</code>	string	Name of the function used as identifier in PGX	Required
<code>language</code>	enum[java, javascript]	Source language for the function (java or javascript)	Required
<code>return_type</code>	enum[boolean, integer, long, float, double, string]	Return type of the function	Required
<code>arguments</code>	array of object	Array of arguments. For each argument: type, argument name, required?	[]
<code>implementation_reference</code>	string	Reference to the function name on the classpath	null
<code>namespace</code>	string	Namespace of the function in PGX	null
<code>source_code</code>	string	Source code of the function provided inline	null

Table 4-7 (Cont.) Fields for Each UDF

Field	Data Type	Description	Required?
source_function_name	string	Name of the function in the source language	null
source_location	string	Local file path to the function's source code	null

All configured UDFs must be unique with regard to the combination of the following fields:

- namespace
- function_name
- arguments

4.16 Using HAProxy for PGX Load Balancing and High Availability

HAProxy is a high-performance TCP/HTTP load balancer and proxy server that allows multiplexing incoming requests across multiple web servers.

You can use HAProxy with multiple instances of the in-memory analytics server (PGX) for high availability. The following example uses the `opg` shell to connect to PGX.

The following instructions assume you have already installed and configured the in-memory analyst server, as explained in [Starting the In-Memory Graph Server \(PGX\)](#).

1. If HAProxy is not already installed on Big Data Appliance or your Oracle Linux distribution, run this command:

```
yum install haproxy
```

2. Start the PGX servers.
For example, if you want to load balance PGX across 4 nodes (such as `bda02`, `bda03`, `bda04`, and `bda05`) in the Big Data Appliance, start PGX on each of these nodes. Configure PGX to listen for connections on port 7007.

3. Configure HAProxy.
In this example, you will configure HAProxy to run on host `bda01` and to listen for incoming connections on port 8888. Create a new file `haproxy.cfg` on host `bda01` with the following content:

```
global
    maxconn 50000
    log /dev/log local0

defaults
    mode http
    option httplog
    log global
    option forwardfor
    timeout connect 5s
    timeout client 5s
```

```
timeout server 5s
balance source
hash-type consistent

listen www
bind :8888
server web1 bda02:7007 check
server web2 bda03:7007 check
server web3 bda04:7007 check
server web4 bda05:7007 check
```

Specifying `balance source` maps the clients' IP addresses to corresponding servers' IP addresses. This is important because the PGX server relies on session stickiness during an analytics session. (For more information about configuring HAProxy, see the [HAProxy official documentation](#).)

4. Start the load balancer.
Start HAProxy on `bda01` by passing in configuration file that you created in the preceding step:

```
haproxy -f haproxy.cfg
```

5. Test the load balancer.
From any host you can test connectivity to the HAProxy server by passing in the host and port of the server running HAProxy as the `base_url` parameter to the `opg` client shell. For example:

```
cd /opt/oracle/oracle-spatial-graph/property_graph
./bin/opg --base_url http://bda01:8888
```

 **Note:**

The PGX in-memory state is lost if the server goes down. HAProxy will route commands to another server, but the client must reload all graph data.

It is recommended that you run a series of PGX commands to test session affinity. Kill a server and restart the `opg` shell to confirm that HAProxy redirects the request to a new server.

5

Using the Property Graph Schema

This chapter provides conceptual and usage information about creating, storing, and working with property graph data in an Oracle Database environment.

You can create a property graph and store it in the property graph schema in Oracle Database in one of the following ways:

1. Use the `CREATE PROPERTY GRAPH` statement to create and populate these property graph schema objects.
2. Use `OPG_APIS.CREATE_PG`, to create the property graph schema objects. Then load data from the database tables into the schema objects using SQL or using the Data Access Layer APIs. The property graph schema provides a flexible schema option for storing your graph.

Note:

The original database tables remain as-is and the data is copied from the original tables into the property graph schema tables.

- [Property Graph Schema Objects for Oracle Database](#)
The property graph PL/SQL and Java APIs use special Oracle Database schema objects.
- [Data Access Layer](#)
- [Getting Started with Property Graphs](#)
Follow these steps to get started with property graphs.
- [Using Java APIs for Property Graph Data](#)
Creating a property graph involves using the Java APIs to create the property graph and objects in it.
- [Managing Text Indexing for Property Graph Data](#)
Indexes in Oracle Spatial and Graph property graph support allow fast retrieval of elements by a particular key/value or key/text pair. These indexes are created based on an element type (vertices or edges), a set of keys (and values), and an index type.
- [Access Control for Property Graph Data \(Graph-Level and OLS\)](#)
Oracle Graph supports two access control and security models: graph level access control, and fine-grained security through integration with Oracle Label Security (OLS).
- [SQL-Based Property Graph Query and Analytics](#)
You can use SQL to query property graph data in Oracle Spatial and Graph.
- [Creating Property Graph Views on an RDF Graph](#)
With Oracle Graph, you can view RDF data as a property graph to execute graph analytics operations by creating property graph views over an RDF graph stored in Oracle Database.
- [Oracle Flat File Format Definition](#)
A property graph can be defined in two flat files, specifically description files for the vertices and edges.

5.1 Property Graph Schema Objects for Oracle Database

The property graph PL/SQL and Java APIs use special Oracle Database schema objects.

This topic describes objects related to the property graph schema approach to working with graph data. It is a more flexible approach than the deprecated two-tables schema approach described in [Handling Property Graphs Using a Two-Tables Schema](#), which has limitations.

Oracle Spatial and Graph lets you store, query, manipulate, and query property graph data in Oracle Database. For example, to create a property graph named `myGraph`, you can use either the Java APIs (`oracle.pg.rdbms.OraclePropertyGraph`) or the PL/SQL APIs (`MDSYS.OPG_APIS` package).

With the PL/SQL API:

```
BEGIN
    opg_apis.create_pg(
        'myGraph',
        dop => 4,                -- degree of parallelism
        num_hash_ptns => 8,     -- number of hash partitions used to
store the graph
        tbs => 'USERS',        -- tablespace
        options => 'COMPRESS=T'
    );
END;
/
```

With the Java API:

```
cfg = GraphConfigBuilder
    .forPropertyGraphRdbms()
    .setJdbcUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl")
    .setUsername("<your_user_name>")
    .setPassword("<your_password>")
    .setName("myGraph")
    .setMaxNumConnections(8)
    .setLoadEdgeLabel(false)
    .build();

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(cfg);
```

- [Property Graph Tables \(Detailed Information\)](#)
- [Default Indexes on Vertex \(VT\\$\) and Edge \(GE\\$\) Tables](#)
- [Flexibility in the Property Graph Schema](#)

5.1.1 Property Graph Tables (Detailed Information)

After a property graph is established in the database, several tables are created automatically in the user's schema, with the graph name as the prefix and VT\$ or GE\$

as the suffix. For example, for a graph named `myGraph`, table `myGraphVT$` is created to store vertices and their properties (K/V pairs), and table `myGraphGE$` is created to store edges and their properties.

Additional internal tables are created with `IT$` and `GT$` suffixes, to store text index metadata and graph skeleton (topological structure).

The definitions of tables `myGraphVT$` and `myGraphGE$` are as follows. They are important for SQL-based analytics and SQL-based property graph query. In both the `VT$` and `GE$` tables, `VT$`, `VTE`, and `FE` are reserved columns; column `SL` is for the security label; and columns `K`, `T`, `V`, `VN`, and `VT` together store all information about a property (K/V pair) of a graph element. In the `VT$` table, `VID` is a long integer for storing the vertex ID. In the `GE$` table, `EID`, `SVID`, and `DVID` are long integer columns for storing edge ID, source (from) vertex ID, and destination (to) vertex ID, respectively.

```
SQL> describe myGraphVT$
```

Name	Null?	Type
VID	NOT NULL	NUMBER
K		NVARCHAR2(3100)
T		NUMBER(38)
V		NVARCHAR2(15000)
VN		NUMBER
VT		TIMESTAMP(6) WITH TIME ZONE
SL		NUMBER
VT\$		DATE
VTE		DATE
FE		NVARCHAR2(4000)

```
SQL> describe myGraphGE$
```

Name	Null?	Type
EID	NOT NULL	NUMBER
SVID	NOT NULL	NUMBER
DVID	NOT NULL	NUMBER
EL		NVARCHAR2(3100)
K		NVARCHAR2(3100)
T		NUMBER(38)
V		NVARCHAR2(15000)
VN		NUMBER
VT		TIMESTAMP(6) WITH TIME ZONE
SL		NUMBER
VT\$		DATE
VTE		DATE
FE		NVARCHAR2(4000)

For simplicity, only simple graph names are allowed, and they are case insensitive.

In both the `VT$` and `GE$` tables, Columns `K`, `T`, `V`, `VN`, `VT` together store all information about a property (K/V pair) of a graph element, while `SL` is used for security label, and `VT$`, `VTE`, `FE` are reserved columns.

In the property graph schema design, a property value is stored in the VN column if the value has numeric data type (long, int, double, float, and so on), in the VT column if the value is a timestamp, or in the V column for Strings, boolean and other serializable data types. For better Oracle Text query support, a literal representation of the property value is saved in the V column even if the data type is numeric or timestamp. To differentiate all the supported data types, an integer ID is saved in the T column. (The possible T column integer ID values are those listed for the *value_type* field in the table in [Vertex File](#).)

The K column in both VT\$ and GE\$ tables stores the property key. Each edge must have a label of String type, and the labels are stored in the EL column of the GE\$ table.

The T column in both VT\$ and GE\$ tables is a number representing the data type of the value of the property it describes. For example 1 means the value is a string, 2 means the value is an integer, and so on. Some T column possible values and associated data types are as follows:

- 1: STRING
- 2: INTEGER
- 3: FLOAT
- 4: DOUBLE
- 5: DATE
- 6: BOOLEAN
- 7: LONG
- 8: SHORT
- 9: BYTE
- 10: CHAR
- 20: Spatial data

To support international characters, NVARCHAR columns are used in VT\$ and GE\$ tables. Oracle highly recommends UTF8 as the default database character set. In addition, the V column has a size of 15000, which **requires** the enabling of 32K VARCHAR (`MAX_STRING_SIZE = EXTENDED`).

The **VT\$ table** schema for storing vertices contains these columns:

- VID, a long column denoting the ID of the vertex.
- VL, a string column denoting the label of the vertex.
- K, a string column denoting the name of the property. If there is no property associated to the vertex, the value of this column should be a whitespace.
- T, a long column denoting the type of the property.
- V, a string column denoting the value of the property as a String. If the property type is numeric, a String format version of the value is stored in this column. Similarly, if the property is timestamp based, a String format version of the value is stored.
- VN, a numeric column denoting the value of a numeric property. This column stores the property value only if the property type is numeric.

- VT, a timestamp with time zone column storing the value of a date time property. This column stores the property value only if the property type is timestamp based.
- SL, a numeric column reserved for the security label set using Oracle Label Security (for further details on using Security Labels, see [Access Control for Property Graph Data \(Graph-Level and OLS\)](#)).
- VTS, a timestamp with time zone column reserved for future extensions.
- VTE, a timestamp with time zone column reserved for future extensions.
- FE, a string column reserved for future extensions.

The following example inserts rows into a table named CONNECTIONSVT\$. It includes T column values 1 through 10 (representing various data types).

```
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '1-STRING', 1,
'Some String', NULL, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '2-INTEGER', 2,
NULL, 21, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '3-FLOAT', 3,
NULL, 21.5, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '4-DOUBLE', 4,
NULL, 21.5, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '5-DATE', 5, NULL,
NULL, timestamp'2018-07-20 15:32:53.991000');
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '6-BOOLEAN', 6,
'Y', NULL, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '7-LONG', 7, NULL,
42, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '8-SHORT', 8,
NULL, 10, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '9-BYTE', 9, NULL,
10, NULL);
INSERT INTO connectionsvt$(vid,k,t,v,vn,vt) VALUES (2001, '10-CHAR', 10,
'A', NULL, NULL);
...
UPDATE connectionsvt$ SET V = coalesce(v,to_nchar(vn),to_nchar(vt)) WHERE
vid=2001;
COMMIT;
```

The **GES** table schema for storing edges contains these columns:

- EID, a long column denoting the ID of the edge.
- SVID, a long column denoting the ID of the outgoing (origin) vertex.
- DVID, a long column denoting the ID of the incoming (destination) vertex.
- EL, a string column denoting the label of the edge.
- K, a string column denoting the name of the property. If there is no property associated to the vertex, the value of this column should be a whitespace.
- T, a long column denoting the type of the property.
- V, a string column denoting the value of the property as a String. If the property type is numeric, a String format version of the value is stored in this column. Similarly, if the property is timestamp based, a String format version of the value is stored.

- VN, a numeric column denoting the value of a numeric property. This column stores the property value only if the property type is numeric.
- VT, a timestamp with time zone column storing the value of a date time property. This column stores the property value only if the property type is timestamp based.
- SL, a numeric column reserved for the security label set using Oracle Label Security (for further details on using Security Labels, see [Access Control for Property Graph Data \(Graph-Level and OLS\)](#)).
- VTS, a timestamp with time zone column reserved for future extensions.
- VTE, a timestamp with time zone column reserved for future extensions.
- FE, a string column reserved for future extensions.

In addition to the VT\$ and GE\$ tables, Oracle Spatial and Graph maintains other internal tables.

An internal graph skeleton table, defined with the **GT\$ suffix**, is used to store the topological structure of a graph, and contains these columns:

- EID, a long column denoting the ID of the edge.
- EL, a string column denoting the label of the edge.
- SVID, a long column denoting the ID of the outgoing (origin) vertex.
- DVID, a long column denoting the ID of the incoming (destination) vertex.
- ELH, a raw column specifying the hash value of an edge label.
- ELS, a integer column specifying the edge label size with respect to total of characters.

An internal text index metadata table, created with **IT\$ suffix**, is used to store metadata information on text indexes created using the Oracle Text search engine. It is automatically populated based on the text indexes created. The IT\$ table includes the following columns for general information about a text index:

- EIN, a string column denoting the name of the text index.
- ET, a numeric column denoting the entities used to build the text index, if it is a vertex (1) or edge (2) text index.
- IT, a numeric column denoting the type of the text index, if it is an automatic (1) or manual (2) text index.
- SE, a numeric column denoting the search engine used to index the entities properties (2 indicates Oracle Text).
- K, a string column denoting the property name used for text indexing.

For Oracle Text-based indexes, the following columns are used to describe the configuration of the text index (for further details on building an Oracle Text-based index, see [Configuring Text Indexes Using Oracle Text](#)):

- PO, a column denoting the preferred owner for the text index configuration settings. By default, the package owner is set to MDSYS.
- DS, a string column specifying the data store used to build the text index.
- FIL, a string column specifying the filter used to build the text index.
- STR, a string column specifying the storage property used to build the text index.
- WL, a string column specifying the word list used when building the text index.

- SL, a string column specifying the stop list used to build the text index.
- LXR, a string column specifying the lexer used by Oracle Text during text indexing.
- OPTS, a string column specifying additional configuration options.

An internal table, defined with the **SS\$ suffix**, is created for Oracle internal use only.

5.1.2 Default Indexes on Vertex (VT\$) and Edge (GE\$) Tables

For query performance, several indexes on property graph tables are created by default. The index names follow the same convention as the table names, including using the graph name as the prefix. For example, for the property graph `myGraph`, the following local (partitioned) indexes are created:

- A unique index `myGraphXQV$` on `myGraphVT$(VID, K)`
- A unique index `myGraphXQE$` on `myGraphGE$(EID, K)`
- An index `myGraphXSE$` on `myGraphGE$(SVID, DVID, EID, VN)`
- An index `myGraphXDE$` on `myGraphGE$(DVID, SVID, EID, VN)`

5.1.3 Flexibility in the Property Graph Schema

The property graph schema design does not use a catalog or centralized repository of any kind. Each property graph is separately stored and managed by a schema of user's choice. A user's schema may have one or more property graphs.

This design provides considerable flexibility to users. For example:

- Users can create additional indexes as desired.
- Different property graphs can have a different set of indexes or compression options for the base tables.
- Different property graphs can have different numbers of hash partitions.
- You can even drop the XSE\$ or XDE\$ index for a property graph; however, for integrity you should keep the unique constraints.

5.2 Data Access Layer

The data access layer provides a set of Java APIs that you can use to create and drop property graphs, add and remove vertices and edges, search for vertices and edges using key-value pairs, create text indexes, and perform other manipulations.

For more information, see:

- [Managing Text Indexing for Property Graph Data](#)
- [Using Java APIs for Property Graph Data](#)
- [Property Graph Schema Objects for Oracle Database \(PL/SQL and Java APIs\) and OPG_APIS Package Subprograms \(PL/SQL API\)](#).

5.3 Getting Started with Property Graphs

Follow these steps to get started with property graphs.

1. The first time you use property graphs, ensure that the software is installed and operational.
2. Interact with a graph using one or more of the following options:
 - Use Java APIs in your Java application. The Java APIs can also be run in the JShell Command line interface for prototype and demo purposes.
 - Run PGQL queries:
 - In the Java application, or
 - In the Graph visualization interface, or
 - In the SQLcl client
 - Run PGQL queries and execute Java APIs in the Apache Zeppelin interpreter
 - [Required Privileges for Database Users](#)
The database schema that contains the graph tables (either Property Graph schema objects or relational tables that will be directly loaded as a graph in memory) requires certain privileges.

Related Topics

- [Using Java APIs for Property Graph Data](#)
Creating a property graph involves using the Java APIs to create the property graph and objects in it.

5.3.1 Required Privileges for Database Users

The database schema that contains the graph tables (either Property Graph schema objects or relational tables that will be directly loaded as a graph in memory) requires certain privileges.

```
ALTER SESSION  
CREATE PROCEDURE  
CREATE SEQUENCE  
CREATE SESSION  
CREATE TABLE  
CREATE TRIGGER  
CREATE TYPE  
CREATE VIEW
```

5.4 Using Java APIs for Property Graph Data

Creating a property graph involves using the Java APIs to create the property graph and objects in it.

- [Overview of the Java APIs](#)
- [Parallel Loading of Graph Data](#)
- [Parallel Retrieval of Graph Data](#)

- [Using an Element Filter Callback for Subgraph Extraction](#)
- [Using Optimization Flags on Reads over Property Graph Data](#)
- [Adding and Removing Attributes of a Property Graph Subgraph](#)
- [Getting Property Graph Metadata](#)
- [Merging New Data into an Existing Property Graph](#)
- [Opening and Closing a Property Graph Instance](#)
- [Creating Vertices](#)
- [Creating Edges](#)
- [Deleting Vertices and Edges](#)
- [Reading a Graph from a Database into an Embedded In-Memory Analyst](#)
- [Specifying Labels for Vertices](#)
- [Building an In-Memory Graph](#)
- [Dropping a Property Graph](#)
- [Executing PGQL Queries](#)

5.4.1 Overview of the Java APIs

The Java APIs that you can use for property graphs include the following:

- [Oracle Graph Property Graph Java APIs](#)
- [Oracle Database Property Graph Java APIs](#)

5.4.1.1 Oracle Graph Property Graph Java APIs

Oracle Graph property graph support provides database-specific APIs for Oracle Database.

To use the Oracle Spatial and Graph API, import the following classes into your Java program:

```
import oracle.pg.common.*;
import oracle.pg.text.*;
import oracle.pg.rdbms.*;
import oracle.pg.rdbms.pgql.*;
import oracle.pgx.config.*;
import oracle.pgx.common.types.*;
```

To compile and run your Java applications, set your classpath to include the jar files in `<client-install-dir>/lib/`.

For example:

```
javac -cp ".:<client-install-dir>/lib/*" Main.java
java -cp ".:<client-install-dir>/lib/*" Main
```

5.4.1.2 Oracle Database Property Graph Java APIs

The Oracle Database property graph Java APIs enable you to create and populate a property graph stored in Oracle Database.

To use these Java APIs, import the classes into your Java program. For example:

```
import oracle.pg.rdbms.*;
import java.sql.*;
```

5.4.2 Parallel Loading of Graph Data

A Java API is provided for performing parallel loading of graph data.

Oracle Spatial and Graph supports loading graph data into Oracle Database. Graph data can be loaded into the property graph using the following approaches:

- Vertices and/or edges can be added incrementally using the `graph.addVertex(Object id)/graph.addEdge(Object id)` APIs.
- Graph data can be loaded from a file in Oracle flat-File format in parallel using the `OraclePropertyGraphDataLoader` API.
- A property graph in GraphML, GML, or GraphSON can be loaded using `GMLReader`, `GraphMLReader`, and `GraphSONReader`, respectively.

This topic focuses on the parallel loading of a property graph in Oracle-defined flat file format.

Parallel data loading provides an optimized solution to data loading where the vertices (or edges) input streams are split into multiple chunks and loaded into Oracle Database in parallel. This operation involves two main overlapping phases:

- Splitting. The vertices and edges input streams are split into multiple chunks and saved into a temporary input stream. The number of chunks is determined by the degree of parallelism specified
- Graph loading. For each chunk, a loader thread is created to process information about the vertices (or edges) information and to load the data into the property graph tables.

`OraclePropertyGraphDataLoader` supports parallel data loading using several different options:

- [JDBC-Based Data Loading](#)
- [External Table-Based Data Loading](#)
- [SQL*Loader-Based Data Loading](#)

5.4.2.1 JDBC-Based Data Loading

JDBC-based data loading uses Java Database Connectivity (JDBC) APIs to load the graph data into Oracle Database. In this option, the vertices (or edges) in the given input stream will be spread among multiple chunks by the splitter thread. Each chunk will be processed by a different loader thread that inserts all the elements in the chunk into a temporary work table using JDBC batching. The number of splitter and loader threads used is determined by the degree of parallelism (DOP) specified by the user.

After all the graph data is loaded into the temporary work tables, all the data stored in the temporary work tables is loaded into the property graph VT\$ and GE\$ tables.

The following example loads the graph data from a vertex and edge files in Oracle-defined flat-file format using a JDBC-based parallel data loading with a degree of parallelism of 48.

```
String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, 48 /* DOP */, 1000 /* batch
size */, true /* rebuild index flag */, "pddl=t,pdml=t" /* options */);
);
```

To optimize the performance of the data loading operations, a set of flags and hints can be specified when calling the JDBC-based data loading. These hints include:

- **DOP:** The degree of parallelism to use when loading the data. This parameter determines the number of chunks to generate when splitting the file as well as the number of loader threads to use when loading the data into the property graph VT\$ and GE\$ tables.
- **Batch Size:** An integer specifying the batch size to use for Oracle update statements in batching mode. The default batch size used in the JDBC-based data loading is 1000.
- **Rebuild index:** If this flag is set to `true`, the data loader will disable all the indexes and constraints defined over the property graph where the data will be loaded. After all the data is loaded into the property graph, all the indexes and constraints will be rebuilt.
- **Load options:** An option (or multiple options delimited by commas) to optimize the data loading operations. These options include:
 - **NO_DUP=T:** Assumes the input data does not have invalid duplicates. In a valid property graph, each vertex (edge) can at most have one value for a given property key. In an invalid property graph, a vertex (edge) may have two or more values for a particular key. As an example, a vertex, `v`, has two key/value pairs: `name/"John"` and `name/"Johnny"` and they share the same key.
 - **PDML=T:** Enables parallel execution for DML operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - **PDDL=T:** Enables parallel execution for DDL operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - **KEEP_WORK_TABS=T:** Skips cleaning and deleting the working tables after the data loading is complete. This is for debugging use only.
 - **KEEP_TMP_FILES=T:** Skips removing the temporary splitter files after the data loading is complete. This is for debug only.
- **Splitter Flag:** An integer value defining the type of files or streams used in the splitting phase to generate the data chunks used in the graph loading phase. The temporary files can be created as regular files (0), named pipes (1), or piped streams (2). By default, JDBC-based data loading uses

Piped streams to handle intermediate data chunks. Piped streams are for JDBC-based loader only. They are purely in-memory and efficient, and do not require any files created on the operating system.

Regular files consume space on the local operating system, while named pipes appear as empty files on the local operating system. Note that not every operating system has support for named pipes.

- **Split File Prefix:** The prefix used for the temporary files or pipes created when the splitting phase is generating the data chunks for the graph loading. By default, the prefix “OPG_Chunk” is used for regular files and “OPG_Pipe” is used for named pipes.
- **Tablespace:** The name of the tablespace where all the temporary work tables will be created.

Subtopics:

- JDBC-Based Data Loading with Multiple Files
- JDBC-Based Data Loading with Partitions
- JDBC-based Parallel Data Loading Using Fine-Tuning

JDBC-Based Data Loading with Multiple Files

JDBC-based data loading also supports loading vertices and edges from multiple files or input streams into the database. The following code fragment loads multiple vertex and edge files using the parallel data loading APIs. In the example, two string arrays szOPVFiles and szOPEFiles are used to hold the input files.

```
String[] szOPVFiles = new String[] {"../data/connections-
p1.opv",
                                   "../data/connections-
p2.opv"};
String[] szOPEFiles = new String[] {"../data/connections-
p1.ope",
                                   "../data/connections-
p2.ope"};
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFiles, szOPEFiles, 48 /* DOP */,
              1000 /* batch size */,
              true /* rebuild index flag */,
              "pddl=t,pdml=t" /* options */);
```

JDBC-Based Data Loading with Partitions

When dealing with graph data from thousands to hundreds of thousands elements, the JDBC-based data loading API allows loading the graph data in Oracle Flat file format into Oracle Database using logical partitioning.

Each partition represents a subset of vertices (or edges) in the graph data file of size is approximately the number of distinct element IDs in the file divided by the number of partitions. Each partition is identified by an integer ID in the range of [0, Number of partitions – 1].

To use parallel data loading with partitions, you must specify the total number of logical partitions to use and the partition offset (start ID) in addition to the base parameters used in the `loadData` API. To fully load a graph data file or input stream into the database, you must execute the data loading operation as many times as the defined

number of partitions. For example, to load the graph data from a file using two partitions, there should be two data loading API calls using an offset of 0 and 1. Each call to the data loader can be processed using multiple threads or a separate Java client on a single system or multiple systems.

Note that this approach is intended to be used with a single vertex file (or input stream) and a single edge file (or input stream). Additionally, this option requires disabling the indices and constraints on vertices and edges. These indices and constraints must be rebuilt after **all** partitions have been loaded.

The following example loads the graph data using two partitions. Each partition is loaded by one Java process `DataLoaderWorker`. To coordinate multiple workers, a coordinator process named `DataLoaderCoordinator` is used. This example does the following

1. Disables all indexes and constraints,
2. Creates a temporary working table, `loaderProgress`, that records the data loading progress (that is, how many workers have finished their work. All `DataLoaderWorker` processes start loading data after the working table is created.
3. Increments the progress by 1.
4. Keeps polling (using the `DataLoaderCoordinator` process) the progress until all `DataLoaderWorker` processes are done.
5. Rebuilds all indexes and constraints.

Note: In `DataLoaderWorker`, the flag `SKIP_INDEX` should be set to `true` and the flag `rebuildIndx` should be set to `false`.

```
// start DataLoaderCoordinator, set dop = 8 and number of partitions = 2
java DataLoaderCoordinator jdbcUrl user password pg 8 2
// start the first DataLoaderWorker, set dop = 8, number of partitions = 2,
partition offset = 0
java DataLoaderWorker jdbcUrl user password pg 8 2 0
// start the first DataLoaderWorker, set dop = 8, number of partitions = 2,
partition offset = 1
java DataLoaderWorker jdbcUrl user password pg 8 2 1
```

The `DataLoaderCoordinator` first disables all indexes and constraints. It then creates a table named `loaderProgress` and inserts one row with column `progress = 0`.

```
public class DataLoaderCoordinator {
    public static void main(String[] szArgs) {
        String jdbcUrl = szArgs[0];
        String user = szArgs[1];
        String password = szArgs[2];
        String graphName = szArgs[3];
        int dop = Integer.parseInt(szArgs[4]);
        int numLoaders = Integer.parseInt(szArgs[5]);

        Oracle oracle = null;
        OraclePropertyGraph opg = null;
        try {
            oracle = new Oracle(jdbcUrl, user, password);
            OraclePropertyGraphUtils.dropPropertyGraph(oracle, graphName);
            opg = OraclePropertyGraph.getInstance(oracle, graphName);
```

```
List<String> vIndices = opg.disableVertexTableIndices();
List<String> vConstraints =
opg.disableVertexTableConstraints();
List<String> eIndices = opg.disableEdgeTableIndices();
List<String> eConstraints =
opg.disableEdgeTableConstraints();

String szStmt = null;
try {
    szStmt = "drop table loaderProgress";
    opg.getOracle().executeUpdate(szStmt);
}
catch (SQLException ex) {
    if (ex.getErrorCode() == 942) {
        // table does not exist. ignore
    }
    else {
        throw new OraclePropertyGraphException(ex);
    }
}

szStmt = "create table loaderProgress (progress integer)";
opg.getOracle().executeUpdate(szStmt);
szStmt = "insert into loaderProgress (progress) values (0)";
opg.getOracle().executeUpdate(szStmt);
opg.getOracle().getConnection().commit();
while (true) {
    if (checkLoaderProgress(oracle) == numLoaders) {
        break;
    } else {
        Thread.sleep(1000);
    }
}

opg.rebuildVertexTableIndices(vIndices, dop, null);
opg.rebuildVertexTableConstraints(vConstraints, dop, null);
opg.rebuildEdgeTableIndices(eIndices, dop, null);
opg.rebuildEdgeTableConstraints(eConstraints, dop, null);
}
catch (IOException ex) {
    throw new OraclePropertyGraphException(ex);
}
catch (SQLException ex) {
    throw new OraclePropertyGraphException(ex);
}
catch (InterruptedException ex) {
    throw new OraclePropertyGraphException(ex);
}
catch (Exception ex) {
    throw new OraclePropertyGraphException(ex);
}
finally {
    try {
        if (opg != null) {
```



```
        opg.shutdown();
    }
    if (oracle != null) {
        oracle.dispose();
    }
}
catch (Throwable t) {
    System.out.println(t);
}
}

private static int checkLoaderProgress(Oracle oracle) {
    int result = 0;
    ResultSet rs = null;

    try {
        String szStmt = "select progress from loaderProgress";
        rs = oracle.executeQuery(szStmt);
        if (rs.next()) {
            result = rs.getInt(1);
        }
    }
    catch (Exception ex) {
        throw new OraclePropertyGraphException(ex);
    }
    finally {
        try {
            if (rs != null) {
                rs.close();
            }
        }
        catch (Throwable t) {
            System.out.println(t);
        }
    }
    return result;
}

public class DataLoaderWorker {

    public static void main(String[] szArgs) {
        String jdbcUrl = szArgs[0];
        String user = szArgs[1];
        String password = szArgs[2];
        String graphName = szArgs[3];
        int dop = Integer.parseInt(szArgs[4]);
        int numLoaders = Integer.parseInt(szArgs[5]);
        int offset = Integer.parseInt(szArgs[6]);

        Oracle oracle = null;
        OraclePropertyGraph opg = null;
```

```
try {
    oracle = new Oracle(jdbcUrl, user, password);
    opg = OraclePropertyGraph.getInstance(oracle, graphName, 8,
dop, null/*tbs*/, ",SKIP_INDEX=T,");
    OraclePropertyGraphDataLoader opgdal =
OraclePropertyGraphDataLoader.getInstance();

    while (true) {
        if (checkLoaderProgress(oracle) == 1) {
            break;
        } else {
            Thread.sleep(1000);
        }
    }

    String opvFile = "../../data/connections.opv";
    String opeFile = "../../data/connections.ope";
    opgdal.loadData(opg, opvFile, opeFile, dop, numLoaders,
offset, 1000, false, null, "pddl=t,pdml=t");

    updateLoaderProgress(oracle);
}
catch (SQLException ex) {
    throw new OraclePropertyGraphException(ex);
}
catch (InterruptedException ex) {
    throw new OraclePropertyGraphException(ex);
}
finally {
    try {
        if (opg != null) {
            opg.shutdown();
        }
        if (oracle != null) {
            oracle.dispose();
        }
    }
    catch (Throwable t) {
        System.out.println(t);
    }
}
}

private static int checkLoaderProgress(Oracle oracle) {
    int result = 0;
    ResultSet rs = null;

    try {
        String szStmt = "select count(*) from loaderProgress";
        rs = oracle.executeQuery(szStmt);
        if (rs.next()) {
            result = rs.getInt(1);
        }
    }
}
```

```

        catch (SQLException ex) {
            if (ex.getErrorCode() == 942) {
                // table does not exist. ignore
            } else {
                throw new OraclePropertyGraphException(ex);
            }
        }
    } finally {
        try {
            if (rs != null) {
                rs.close();
            }
        } catch (Throwable t) {
            System.out.println(t);
        }
    }
    return result;
}

private static void updateLoaderProgress(Oracle oracle) {
    ResultSet rs = null;

    try {
        String szStmt = "update loaderProgress set progress = progress +
1";

        oracle.executeUpdate(szStmt);
        oracle.getConnection().commit();
    } catch (Exception ex) {
        throw new OraclePropertyGraphException(ex);
    } finally {
        try {
            if (rs != null) {
                rs.close();
            }
        } catch (Throwable t) {
            System.out.println(t);
        }
    }
}
}

```

JDBC-based Parallel Data Loading Using Fine-Tuning

JDBC-based data loading supports fine-tuning the subset of data from a line to be loaded, as well as the ID offset to use when loading the elements into the property graph instance. You can specify the subset of data to load from a file by specifying the maximum number of lines to read from the file and the offset line number (start position) for both vertices and edges. This way, data will be loaded from the offset line number until the maximum number of lines has been read. If the maximum line number is -1, the loading process will scan the data until reaching the end of file.

Because multiple graph data files may have some ID collisions or overlap, the JDBC-based data loading allows you to define a vertex and edge ID offset. This way, the ID of each loaded vertex will be the sum of the original vertex ID and the given vertex ID offset. Similarly, the ID of each loaded edge will be generated from the sum of the original edge ID and the given edge ID offset. Note that the vertices and edge files must be correlated, because the in/out vertex ID for the loaded edges will be modified with respect to the specified vertex ID offset. This operation is supported only in data loading using a single logical partition.

The following code fragment loads the first 100 vertices and edges lines from the given graph data file. In this example, an ID offset 0 is used, which indicates no ID adjustment is performed.

```
String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";
// Run the data loading using fine tuning
long lVertexOffsetlines = 0;
long lEdgeOffsetlines = 0;
long lVertexMaxlines = 100;
long lEdgeMaxlines = 100;
long lVIDOffset = 0;
long lEIDOffset = 0;
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
OraclePropertyGraphDataLoader opgd1 =
OraclePropertyGraphDataLoader.getInstance();

opgd1.loadData(opg, szOPVFile, szOPEFile,
               lVertexOffsetlines /* offset of lines to start
loading from
partition, default 0 */,
               lEdgeOffsetlines /* offset of lines to start loading
from
partition, default 0 */,
               lVertexMaxlines /* maximum number of lines to start loading from
partition, default -1 (all lines in partition)
*/,
               lEdgeMaxlines /* maximum number of lines to start loading from
partition, default -1 (all lines in partition)
*/,
               lVIDOffset /* vertex ID offset: the vertex ID will be original
vertex ID + offset, default 0 */,
               lEIDOffset /* edge ID offset: the edge ID will be original edge
ID
+ offset, default 0 */,
               4 /* DOP */,
               1 /* Total number of partitions, default 1 */,
               0 /* Partition to load: from 0 to totalPartitions - 1, default 0
*/,
               OraclePropertyGraphDataLoader.PIPEDSTREAM /* splitter flag */,
               "chunkPrefix" /* prefix: the prefix used to generate split chunks
for regular files or named pipes */,
               1000 /* batch size: batch size of Oracle update in batching mode.
Default value is 1000 */,
               true /* rebuild index */);
```

```

null /* table space name*/,
"pddl=t,pdml=t" /* options: enable parallel DDL and DML */);

```

5.4.2.2 External Table-Based Data Loading

External table-based data loading uses an external table to load the graph data into Oracle Database. External table loading allows users to access the data in external sources as if it were in a regular relational table in the database. In this case, the vertices (or edges) in the given input stream will be spread among multiple chunks by the splitter thread. Each chunk will be processed by a different loader thread that is in charge of passing all the elements in the chunk to Oracle Database. The number of splitter and loader threads used is determined by the degree of parallelism (DOP) specified by the user.

After the external tables are automatically created by the data loading logic, the loader will read from the external tables and load all the data into the property graph schema tables (VT\$ and GE\$).

External-table based data loading requires a directory object where the files read by the external tables will be stored. This directory can be created by running the following scripts in a SQL*Plus environment:

```

create or replace directory tmp_dir as '/tmppath/';
grant read, write on directory tmp_dir to public;

```

The following code fragment loads the graph data from a vertex and edge files in Oracle Flat-file format using an external table-based parallel data loading with a degree of parallelism of 48.

```

String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";
String szExtDir = "tmp_dir";
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadDataWithExtTab(opg, szOPVFile, szOPEFile, 48 /*DOP*/,
true /*named pipe flag: setting the flag to
true will use
regular file
named pipe based splitting; otherwise,
based splitting would be used*/,
szExtDir /* database directory object */,
true /*rebuild index */,
"pddl=t,pdml=t,NO_DUP=T" /*options */);

```

To optimize the performance of the data loading operations, a set of flags and hints can be specified when calling the External table-based data loading. These hints include:

- **DOP:** The degree of parallelism to use when loading the data. This parameter determines the number of chunks to generate when splitting the file, as well as the number of loader threads to use when loading the data into the property graph VT\$ and GE\$ tables.

- **Rebuild index:** If this flag is set to `true`, the data loader will disable all the indexes and constraints defined over the property graph where the data will be loaded. After all the data is loaded into the property graph, all the indexes and constraints will be rebuilt.
- **Load options:** An option (or multiple options delimited by commas) to optimize the data loading operations. These options include:
 - `NO_DUP=T`: Chooses a faster way to load the data into the property graph tables as no validation for duplicate Key/value pairs will be conducted.
 - `PDML=T`: Enables parallel execution for DML operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - `PDDL=T`: Enables parallel execution for DDL operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - `KEEP_WORK_TABS=T`: Skips cleaning and deleting the working tables after the data loading is complete. This is for debugging use only.
 - `KEEP_TMP_FILES=T`: Skips removing the temporary splitter files after the data loading is complete. This is for debugging use only.
- **Splitter Flag:** An integer value defining the type of files or streams used in the splitting phase to generate the data chunks used in the graph loading phase. The temporary files can be created as regular files (0) or named pipes (1).

By default, External table-based data loading uses regular files to handle temporary files for data chunks. Named pipes can only be used on operating system that supports them. It is generally a good practice to use regular files together with DBFS.
- **Split File Prefix:** The prefix used for the temporary files or pipes created when the splitting phase is generating the data chunks for the graph loading. By default, the prefix “Chunk” is used for regular files and “Pipe” is used for named pipes.
- **Tablespace:** The name of the tablespace where all the temporary work tables will be created.

As with the JDBC-based data loading, external table-based data loading supports parallel data loading using a single file, multiple files, partitions, and fine-tuning.

Subtopics:

- External Table-Based Data Loading with Multiple Files
- External table-based Data Loading with Partitions
- External Table-Based Parallel Data Loading Using Fine-Tuning

External Table-Based Data Loading with Multiple Files

External table-based data loading also supports loading vertices and edges from multiple files or input streams into the database. The following code fragment loads multiple vertex and edge files using the parallel data loading APIs. In the example, two string arrays `szOPVFiles` and `szOPEFiles` are used to hold the input files.

```
String szOPVFile = "../../data/connections.opv";  
String szOPEFile = "../../data/connections.ope";  
String szExtDir = "tmp_dir";
```

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadDataWithExtTab(opg, szOPVFile, szOPEFile, 48 /* DOP */,
    true /* named pipe flag */,
    szExtDir /* database directory object */,
    true /* rebuild index flag */,
    "pddl=t,pdml=t" /* options */);
```

External table-based Data Loading with Partitions

When dealing with a very large property graph, the external table-based data loading API allows loading the graph data in Oracle flat file format into Oracle Database using logical partitioning. Each partition represents a subset of vertices (or edges) in the graph data file of size that is approximately the number of distinct element IDs in the file divided by the number of partitions. Each partition is identified by an integer ID in the range of [0, Number of partitions – 1].

To use parallel data loading with partitions, you must specify the total number of partitions to use and the partition offset besides the base parameters used in the `loadDataWithExtTab` API. To fully load a graph data file or input stream into the database, you must execute the data loading operation as many times as the defined number of partitions. For example, to load the graph data from a file using two partitions, there should be two data loading API calls using an offset of 0 and 1. Each call to the data loader can be processed using multiple threads or a separate Java client on a single system or multiple systems.

Note that this approach is intended to be used with a single vertex file (or input stream) and a single edge file (or input stream). Additionally, this option requires disabling the indexes and constraints on vertices and edges. These indices and constraints must be rebuilt after all partitions have been loaded.

The example for JDBC-based data loading with partitions can be easily migrated to work as external-table based loading with partitions. The only needed changes are to replace API `loadData()` with `loadDataWithExtTab()`, and supply some additional input parameters such as the database directory object.

External Table-Based Parallel Data Loading Using Fine-Tuning

External table-based data loading also supports fine-tuning the subset of data from a line to be loaded, as well as the ID offset to use when loading the elements into the property graph instance. You can specify the subset of data to load from a file by specifying the maximum number of lines to read from the file as well as the offset line number for both vertices and edges. This way, data will be loaded from the offset line number until the maximum number of lines has been read. If the maximum line number is -1, the loading process will scan the data until reaching the end of file.

Because graph data files may have some ID collisions, the external table-based data loading allows you to define a vertex and edge ID offset. This way, the ID of each loaded vertex will be obtained from the sum of the original vertex ID with the given vertex ID offset. Similarly, the ID of each loaded edge will be generated from the sum of the original edge ID with the given edge ID offset. Note that the vertices and edge files must be correlated, because the in/out vertex ID for the loaded edges will be modified with respect to the specified vertex ID offset. This operation is supported only in a data loading using a single partition.

The following code fragment loads the first 100 vertices and edges from the given graph data file. In this example, no ID offset is provided.

```
String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Run the data loading using fine tuning
long lVertexOffsetlines = 0;
long lEdgeOffsetlines = 0;
long lVertexMaxlines = 100;
long lEdgeMaxlines = 100;
long lVIDOffset = 0;
long lEIDOffset = 0;
String szExtDir = "tmp_dir";

OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();

opgdl.loadDataWithExtTab(opg, szOPVFile, szOPEFile,
lVertexOffsetlines /* offset of lines to
start loading
                                from partition,
                                default 0 */,
lEdgeOffsetlines /* offset of lines to
start loading from
                                partition, default 0
*/,
lVertexMaxlines /* maximum number of lines
to start
                                loading from partition,
                                default -1
                                (all lines in partition)
*/,
lEdgeMaxlines /* maximum number of lines
to start loading
                                from partition, default
-1 (all lines in
                                partition) */,
lVIDOffset /* vertex ID offset: the vertex
ID will be
                                original vertex ID + offset,
                                default 0 */,
lEIDOffset /* edge ID offset: the edge ID
will be
                                original edge ID + offset,
                                default 0 */,
4 /* DOP */,
1 /* Total number of partitions, default 1
*/,
0 /* Partition to load (from 0 to
totalPartitions - 1,
                                default 0) */,
OraclePropertyGraphDataLoader.NAMEDPIPE
```



```

/* splitter flag */,
"chunkPrefix" /* prefix */,
szExtDir /* database directory object */,
true /* rebuild index flag */,
"pddl=t,pdml=t" /* options */);

```

5.4.2.3 SQL*Loader-Based Data Loading

SQL*Loader-based data loading uses Oracle SQL*Loader to load the graph data into Oracle Database. SQL*Loader loads data from external files into Oracle Database tables. In this case, the vertices (or edges) in the given input stream will be spread among multiple chunks by the splitter thread. Each chunk will be processed by a different loader thread that inserts all the elements in the chunk into a temporary work table using SQL*Loader. The number of splitter and loader threads used is determined by the degree of parallelism (DOP) specified by the user.

After all the graph data is loaded into the temporary work table, the graph loader will load all the data stored in the temporary work tables into the property graph VT\$ and GE\$ tables.

The following code fragment loads the graph data from a vertex and edge files in Oracle flat-file format using a SQL-based parallel data loading with a degree of parallelism of 48. To use the APIs, the path to the SQL*Loader must be specified.

```

String szUser = "username";
String szPassword = "password";
String szDbId = "db18c"; /*service name of the database*/
String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";
String szSQLLoaderPath = "<YOUR_ORACLE_HOME>/bin/sqlldr";
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);

opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadDataWithSqlLdr(opg, szUser, szPassword, szDbId,
szOPVFile, szOPEFile,
48 /* DOP */,
true /*named pipe flag */,
szSQLLoaderPath /* SQL*Loader path: the path to
bin/sqlldr*/,
true /*rebuild index */,
"pddl=t,pdml=t" /* options */);

```

As with JDBC-based data loading, SQL*Loader-based data loading supports parallel data loading using a single file, multiple files, partitions, and fine-tuning.

Subtopics:

- SQL*Loader-Based Data Loading with Multiple Files
- SQL*Loader-Based Data Loading with Partitions
- SQL*Loader-Based Parallel Data Loading Using Fine-Tuning

SQL*Loader-Based Data Loading with Multiple Files

SQL*Loader-based data loading supports loading vertices and edges from multiple files or input streams into the database. The following code fragment loads multiple vertex and edge files using the parallel data loading APIs. In the example, two string arrays `szOPVFiles` and `szOPEFiles` are used to hold the input files.

```
String szUser = "username";
String szPassword = "password";
String szDbId = "db18c"; /*service name of the database*/
String[] szOPVFiles = new String[] {"../data/connections-
p1.opv",
                                   "../data/connections-
p2.opv"};
String[] szOPEFiles = new String[] {"../data/connections-
p1.ope",
                                   "../data/connections-
p2.ope"};
String szSQLLoaderPath = "../dbhome_1/bin/sqlldr";
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);

opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadDataWithSqlLdr (opg, szUser, szPassword, szDbId,
                          szOPVFiles, szOPEFiles,
                          48 /* DOP */,
                          true /* named pipe flag */,
                          szSQLLoaderPath /* SQL*Loader path */,
                          true /* rebuild index flag */,
                          "pddl=t,pdml=t" /* options */);
```

SQL*Loader-Based Data Loading with Partitions

When dealing with a large property graph, the SQL*Loader-based data loading API allows loading the graph data in Oracle flat-file format into Oracle Database using logical partitioning. Each partition represents a subset of vertices (or edges) in the graph data file of size that is approximately the number of distinct element IDs in the file divided by the number of partitions. Each partition is identified by an integer ID in the range of [0, Number of partitions – 1].

To use parallel data loading with partitions, you must specify the total number of partitions to use and the partition offset, in addition to the base parameters used in the `loadDataWithSqlLdr` API. To fully load a graph data file or input stream into the database, you must execute the data loading operation as many times as the defined number of partitions. For example, to load the graph data from a file using two partitions, there should be two data loading API calls using an offset of 0 and 1. Each call to the data loader can be processed using multiple threads or a separate Java client on a single system or multiple systems.

Note that this approach is intended to be used with a single vertex file (or input stream) and a single edge file (or input stream). Additionally, this option requires disabling the indexes and constraints on vertices and edges. These indexes and constraints must be rebuilt after all partitions have been loaded.

The example for JDBC-based data loading with partitions can be easily migrated to work as SQL*Loader-based loading with partitions. The only changes needed are to replace API `loadData()` with `loadDataWithSqlLdr()`, and supply some additional input parameters such as the location of SQL*Loader.

SQL*Loader-Based Parallel Data Loading Using Fine-Tuning

SQL Loader-based data loading supports fine-tuning the subset of data from a line to be loaded, as well as the ID offset to use when loading the elements into the property graph instance. You can specify the subset of data to load from a file by specifying the maximum number of lines to read from the file and the offset line number for both vertices and edges. This way, data will be loaded from the offset line number until the maximum number of lines has been read. If the maximum line number is -1, the loading process will scan the data until reaching the end of file.

Because graph data files may have some ID collisions, the SQL Loader-based data loading allows you to define a vertex and edge ID offset. This way, the ID of each loaded vertex will be obtained from the sum of the original vertex ID with the given vertex ID offset. Similarly, the ID of each loaded edge will be generated from the sum of the original edge ID with the given edge ID offset. Note that the vertices and edge files must be correlated, because the in/out vertex ID for the loaded edges will be modified with respect to the specified vertex ID offset. This operation is supported only in a data loading using a single partition.

The following code fragment loads the first 100 vertices and edges from the given graph data file. In this example, no ID offset is provided.

```
String szUser = "username";
String szPassword = "password";
String szDbId = "db18c"; /* service name of the database */
String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";
String szSQLLoaderPath = "../dbhome_1/bin/sqlldr";

// Run the data loading using fine tuning
long lVertexOffsetlines = 0;
long lEdgeOffsetlines = 0;
long lVertexMaxlines = 100;
long lEdgeMaxlines = 100;
long lVIDOffset = 0;
long lEIDOffset = 0;
OraclePropertyGraph opg = OraclePropertyGraph.getInstance( args,
szGraphName);
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();

opgdl.loadDataWithSqlLdr(opg, szUser, szPassword, szDbId,
szOPVFile, szOPEFile,
lVertexOffsetlines /* offset of lines to start
loading
                                from partition, default
0*/,
lEdgeOffsetlines /* offset of lines to start
loading from
                                partition, default 0*/,
lVertexMaxlines /* maximum number of lines to
start
```

```

loading from partition,
default -1
partition)*/,
to start loading
-1 (all lines in
ID will be
default 0 */,
will be
default 0 */,
48 /* DOP */,
1 /* Total number of partitions, default 1
*/,
0 /* Partition to load (from 0 to
default 0) */,
OraclePropertyGraphDataLoader.NAMEDPIPE
/* splitter flag */,
"chunkPrefix" /* prefix */,
szSQLLoaderPath /* SQL*Loader path: the
path to
bin/sqlldr*/,
true /* rebuild index */,
"pddl=t,pdml=t" /* options */);

```

5.4.3 Parallel Retrieval of Graph Data

The parallel property graph query provides a simple Java API to perform parallel scans on vertices (or edges). Parallel retrieval is an optimized solution taking advantage of the distribution of the data across table partitions, so each partition is queried using a separate database connection.

Parallel retrieval will produce an array where each element holds all the vertices (or edges) from a specific partition (split). The subset of shards queried will be separated by the given start split ID and the size of the connections array provided. This way, the subset will consider splits in the range of [start, start - 1 + size of connections array]. Note that an integer ID (in the range of [0, N - 1]) is assigned to all the splits in the vertex table with N splits.

The following code loads a property graph, opens an array of connections, and executes a parallel query to retrieve all vertices and edges using the opened connections. The number of calls to the `getVerticesPartitioned` (`getEdgesPartitioned`) method is controlled by the total number of splits and the number of connections used.

```

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args,
szGraphName);

```

```
// Clear existing vertices/edges in the property graph
opg.clearRepository();

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// This object will handle parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, dop);

// Create connections used in parallel query
Oracle[] oracleConns = new Oracle[dop];
Connection[] conns = new Connection[dop];
for (int i = 0; i < dop; i++) {
    oracleConns[i] = opg.getOracle().clone();
    conns[i] = oracleConns[i].getConnection();
}

long lCountV = 0;
// Iterate over all the vertices' partitionIDs to count all the vertices
for (int partitionID = 0; partitionID < opg.getVertexPartitionsNumber();
    partitionID += dop) {
    Iterable<Vertex>[] iterables
        = opg.getVerticesPartitioned(conns /* Connection array */,
                                     true /* skip store to cache */,
                                     partitionID /* starting partition */);
    lCountV += consumeIterables(iterables); /* consume iterables using
                                             threads */
}

// Count all vertices
System.out.println("Vertices found using parallel query: " + lCountV);

long lCountE = 0;
// Iterate over all the edges' partitionIDs to count all the edges
for (int partitionID = 0; partitionID < opg.getEdgeTablePartitionIDs();
    partitionID += dop) {
    Iterable<Edge>[] iterables
        = opg.getEdgesPartitioned(conns /* Connection array */,
                                   true /* skip store to cache */,
                                   partitionID /* starting partition */);
    lCountE += consumeIterables(iterables); /* consume iterables using
                                             threads */
}

// Count all edges
System.out.println("Edges found using parallel query: " + lCountE);

// Close the connections to the database after completed
for (int idx = 0; idx < conns.length; idx++) {
    conns[idx].close();
}
```

5.4.4 Using an Element Filter Callback for Subgraph Extraction

Oracle Spatial and Graph provides support for an easy subgraph extraction using user-defined element filter callbacks. An element filter callback defines a set of conditions that a vertex (or an edge) must meet in order to keep it in the subgraph. Users can define their own element filtering by implementing the `VertexFilterCallback` and `EdgeFilterCallback` API interfaces.

The following code fragment implements a `VertexFilterCallback` that validates if a vertex does not have a political role and its origin is the United States.

```
/**
 * VertexFilterCallback to retrieve a vertex from the United States
 * that does not have a political role
 */
private static class NonPoliticianFilterCallback
implements VertexFilterCallback
{
    @Override
    public boolean keepVertex(OracleVertexBase vertex)
    {
        String country = vertex.getProperty("country");
        String role = vertex.getProperty("role");

        if (country != null && country.equals("United States")) {
            if (role == null || !role.toLowerCase().contains("political")) {
                return true;
            }
        }

        return false;
    }

    public static NonPoliticianFilterCallback getInstance()
    {
        return new NonPoliticianFilterCallback();
    }
}
```

The following code fragment implements an `EdgeFilterCallback` that uses the `VertexFilterCallback` to keep only edges connected to the given input vertex, and whose connections are not politicians and come from the United States.

```
/**
 * EdgeFilterCallback to retrieve all edges connected to an input
 * vertex with "collaborates" label, and whose vertex is from the
 * United States with a role different than political
 */
private static class CollaboratorsFilterCallback
implements EdgeFilterCallback
{
    private VertexFilterCallback m_vfc;
    private Vertex m_startV;

    public CollaboratorsFilterCallback(VertexFilterCallback vfc,
        Vertex v)
    {
        m_vfc = vfc;
        m_startV = v;
    }
}
```

```

    }

    @Override
    public boolean keepEdge(OracleEdgeBase edge)
    {
        if ("collaborates".equals(edge.getLabel())) {
            if (edge.getVertex(Direction.IN).equals(m_startV) &&
                m_vfc.keepVertex((OracleVertex)
                    edge.getVertex(Direction.OUT))) {
                return true;
            }
            else if (edge.getVertex(Direction.OUT).equals(m_startV) &&
                m_vfc.keepVertex((OracleVertex)
                    edge.getVertex(Direction.IN))) {
                return true;
            }
        }

        return false;
    }

    public static CollaboratorsFilterCallback
    getInstance(VertexFilterCallback vfc, Vertex v)
    {
        return new CollaboratorsFilterCallback(vfc, v);
    }
}

```

Using the filter callbacks previously defined, the following code fragment loads a property graph, creates an instance of the filter callbacks and later gets all of Robert Smith's collaborators who are not politicians and come from the United States.

```

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
    args, szGraphName);

// Clear existing vertices/edges in the property graph
opg.clearRepository();

String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";

// This object will handle parallel data loading
OraclePropertyGraphDataLoader opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, dop);

// VertexFilterCallback to retrieve all people from the United States // who are not
politicians
NonPoliticianFilterCallback npvfc = NonPoliticianFilterCallback.getInstance();

// Initial vertex: Robert Smith
Vertex v = opg.getVertices("name", "Robert Smith").iterator().next();

// EdgeFilterCallback to retrieve all collaborators of Robert Smith
// from the United States who are not politicians
CollaboratorsFilterCallback cefc = CollaboratorsFilterCallback.getInstance(npvfc, v);

Iterable<<Edge> smithCollabs = opg.getEdges((String[])null /* Match any
of the properties */,
cefc /* Match the
EdgeFilterCallback */

```

```

);
Iterator<<Edge> iter = smithCollabs.iterator();

System.out.println("\n\n-----Collaborators of Robert Smith from " +
    " the US and non-politician\n\n");
long countV = 0;
while (iter.hasNext()) {
    Edge edge = iter.next(); // get the edge
    // check if smith is the IN vertex
    if (edge.getVertex(Direction.IN).equals(v)) {
        System.out.println(edge.getVertex(Direction.OUT) + "(Edge ID: " +
            edge.getId() + ")"); // get out vertex
    }
    else {
        System.out.println(edge.getVertex(Direction.IN) + "(Edge ID: " +
            edge.getId() + ")"); // get in vertex
    }

    countV++;
}

```

By default, all reading operations such as get all vertices, get all edges (and parallel approaches) will use the filter callbacks associated with the property graph using the methods `opg.setVertexFilterCallback(vfc)` and `opg.setEdgeFilterCallback(efc)`. If there is no filter callback set, then all the vertices (or edges) and edges will be retrieved.

The following code fragment uses the default edge filter callback set on the property graph to retrieve the edges.

```

// VertexFilterCallback to retrieve all people from the United States // who are
// not politicians
NonPoliticianFilterCallback npvfc = NonPoliticianFilterCallback.getInstance();

// Initial vertex: Robert Smith
Vertex v = opg.getVertices("name", "Robert Smith").iterator().next();

// EdgeFilterCallback to retrieve all collaborators of Robert Smith
// from the United States who are not politicians
CollaboratorsFilterCallback cefc =
CollaboratorsFilterCallback.getInstance(npvfc, v);

opg.setEdgeFilterCallback(cefc);

Iterable<Edge> smithCollabs = opg.getEdges();
Iterator<Edge> iter = smithCollabs.iterator();

System.out.println("\n\n-----Collaborators of Robert Smith from " +
    " the US and non-politician\n\n");
long countV = 0;
while (iter.hasNext()) {
    Edge edge = iter.next(); // get the edge
    // check if smith is the IN vertex
    if (edge.getVertex(Direction.IN).equals(v)) {
        System.out.println(edge.getVertex(Direction.OUT) + "(Edge ID: " +
            edge.getId() + ")"); // get out vertex
    }
    else {
        System.out.println(edge.getVertex(Direction.IN) + "(Edge ID: " +
            edge.getId() + ")"); // get in vertex
    }
}

```



```
countV++;
}
```

5.4.5 Using Optimization Flags on Reads over Property Graph Data

Oracle Spatial and Graph provides support for optimization flags to improve graph iteration performance. Optimization flags allow processing vertices (or edges) as objects with none or minimal information, such as ID, label, and/or incoming/outgoing vertices. This way, the time required to process each vertex (or edge) during iteration is reduced.

The following table shows the optimization flags available when processing vertices (or edges) in a property graph.

Optimization Flag	Description
DO_NOT_CREATE_OBJECT	Use a predefined constant object when processing vertices or edges.
JUST_EDGE_ID	Construct edge objects with ID only when processing edges.
JUST_LABEL_EDGE_ID	Construct edge objects with ID and label only when processing edges.
JUST_LABEL_VERTEX_EDGE_ID	Construct edge objects with ID, label, and in/out vertex IDs only when processing edges
JUST_VERTEX_EDGE_ID	Construct edge objects with just ID and in/out vertex IDs when processing edges.
JUST_VERTEX_ID	Construct vertex objects with ID only when processing vertices.

The following code fragment uses a set of optimization flags to retrieve only all the IDs from the vertices and edges in the property graph. The objects retrieved by reading all vertices and edges will include only the IDs and no Key/Value properties or additional information.

```
import oracle.pg.common.OraclePropertyGraphBase.OptimizationFlag;
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
    args, szGraphName);

// Clear existing vertices/edges in the property graph
opg.clearRepository();

String szOPVFile = "../data/connections.opv";
String szOPEFile = "../data/connections.ope";

// This object will handle parallel data loading
OraclePropertyGraphDataLoader opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, dop);

// Optimization flag to retrieve only vertices IDs
OptimizationFlag optFlagVertex = OptimizationFlag.JUST_VERTEX_ID;

// Optimization flag to retrieve only edges IDs
OptimizationFlag optFlagEdge = OptimizationFlag.JUST_EDGE_ID;

// Print all vertices
Iterator<Vertex> vertices =
opg.getVertices((String[])null /* Match any of the
properties */,
null /* Match the VertexFilterCallback */,
optFlagVertex /* optimization flag */
).iterator();
```

```

System.out.println("----- Vertices IDs-----");
long vCount = 0;
while (vertices.hasNext()) {
OracleVertex v = vertices.next();
System.out.println((Long) v.getId());
vCount++;
}
System.out.println("Vertices found: " + vCount);

// Print all edges
Iterator<Edge> edges =
opg.getEdges((String[])null /* Match any of the properties */,
null /* Match the EdgeFilterCallback */,
optFlagEdge /* optimization flag */
).iterator();

System.out.println("----- Edges -----");
long eCount = 0;
while (edges.hasNext()) {
Edge e = edges.next();
System.out.println((Long) e.getId());
eCount++;
}
System.out.println("Edges found: " + eCount);

```

By default, all reading operations such as get all vertices, get all edges (and parallel approaches) will use the optimization flag associated with the property graph using the method `opg.setDefaultVertexOptFlag(optFlagVertex)` and `opg.setDefaultEdgeOptFlag(optFlagEdge)`. If the optimization flags for processing vertices and edges are not defined, then all the information about the vertices and edges will be retrieved.

The following code fragment uses the default optimization flags set on the property graph to retrieve only all the IDs from its vertices and edges.

```

import oracle.pg.common.OraclePropertyGraphBase.OptimizationFlag;

// Optimization flag to retrieve only vertices IDs
OptimizationFlag optFlagVertex = OptimizationFlag.JUST_VERTEX_ID;

// Optimization flag to retrieve only edges IDs
OptimizationFlag optFlagEdge = OptimizationFlag.JUST_EDGE_ID;

opg.setDefaultVertexOptFlag(optFlagVertex);
opg.setDefaultEdgeOptFlag(optFlagEdge);

Iterator<Vertex> vertices = opg.getVertices().iterator();
System.out.println("----- Vertices IDs-----");
long vCount = 0;
while (vertices.hasNext()) {
OracleVertex v = vertices.next();
System.out.println((Long) v.getId());
vCount++;
}
System.out.println("Vertices found: " + vCount);

// Print all edges
Iterator<Edge> edges = opg.getEdges().iterator();

```

```

System.out.println("----- Edges -----");
long eCount = 0;
while (edges.hasNext()) {
    Edge e = edges.next();
    System.out.println((Long) e.getId());
    eCount++;
}
System.out.println("Edges found: " + eCount);

```

5.4.6 Adding and Removing Attributes of a Property Graph Subgraph

Oracle Spatial and Graph supports updating attributes (key/value pairs) to a subgraph of vertices and/or edges by using a user-customized operation callback. An operation callback defines a set of conditions that a vertex (or an edge) must meet in order to update it (either add or remove the given attribute and value).

You can define your own attribute operations by implementing the `VertexOpCallback` and `EdgeOpCallback` API interfaces. You must override the `needOp` method, which defines the conditions to be satisfied by the vertices (or edges) to be included in the update operation, as well as the `getAttributeKeyName` and `getAttributeKeyValue` methods, which return the key name and value, respectively, to be used when updating the elements.

The following code fragment implements a `VertexOpCallback` that operates over the `smithCollaborator` attribute associated only with Robert Smith collaborators. The value of this property is specified based on the role of the collaborators.

```

private static class CollaboratorsVertexOpCallback
implements VertexOpCallback
{
    private OracleVertexBase m_smith;
    private List<Vertex> m_smithCollaborators;

    public CollaboratorsVertexOpCallback(OraclePropertyGraph opg)
    {
        // Get a list of Robert Smith's Collaborators
        m_smith = (OracleVertexBase) opg.getVertices("name",
            "Robert Smith")
            .iterator().next();

        Iterable<Vertex> iter = m_smith.getVertices(Direction.BOTH,
            "collaborates");
        m_smithCollaborators = OraclePropertyGraphUtils.listify(iter);
    }

    public static CollaboratorsVertexOpCallback
    getInstance(OraclePropertyGraph opg)
    {
        return new CollaboratorsVertexOpCallback(opg);
    }

    /**
     * Add attribute if and only if the vertex is a collaborator of Robert
     * Smith
     */
    @Override
    public boolean needOp(OracleVertexBase v)
    {
        return m_smithCollaborators != null &&
            m_smithCollaborators.contains(v);
    }
}

```

```
@Override
public String getAttributeKeyName(OracleVertexBase v)
{
    return "smithCollaborator";
}

/**
 * Define the property's value based on the vertex role
 */
@Override
public Object getAttributeKeyValue(OracleVertexBase v)
{
    String role = v.getProperty("role");
    role = role.toLowerCase();
    if (role.contains("political")) {
        return "political";
    }
    else if (role.contains("actor") || role.contains("singer") ||
        role.contains("actress") || role.contains("writer") ||
        role.contains("producer") || role.contains("director")) {
        return "arts";
    }
    else if (role.contains("player")) {
        return "sports";
    }
    else if (role.contains("journalist")) {
        return "journalism";
    }
    else if (role.contains("business") || role.contains("economist")) {
        return "business";
    }
    else if (role.contains("philanthropist")) {
        return "philanthropy";
    }
    return " ";
}
}
```

The following code fragment implements an `EdgeOpCallback` that operates over the `smithFeud` attribute associated only with Robert Smith feuds. The value of this property is specified based on the role of the collaborators.

```
private static class FeudsEdgeOpCallback
implements EdgeOpCallback
{
    private OracleVertexBase m_smith;
    private List<Edge> m_smithFeuds;

    public FeudsEdgeOpCallback(OraclePropertyGraph opg)
    {
        // Get a list of Robert Smith's feuds
        m_smith = (OracleVertexBase) opg.getVertices("name",
            "Robert Smith")
            .iterator().next();

        Iterable<Vertex> iter = m_smith.getVertices(Direction.BOTH,
            "feuds");
        m_smithFeuds = OraclePropertyGraphUtils.listify(iter);
    }
}
```

```
public static FeudsEdgeOpCallback getInstance(OraclePropertyGraph opg)
{
    return new FeudsEdgeOpCallback(opg);
}

/**
 * Add attribute if and only if the edge is in the list of Robert Smith's
 * feuds
 */
@Override
public boolean needOp(OracleEdgeBase e)
{
    return m_smithFeuds != null && m_smithFeuds.contains(e);
}

@Override
public String getAttributeName(OracleEdgeBase e)
{
    return "smithFeud";
}

/**
 * Define the property's value based on the in/out vertex role
 */
@Override
public Object getAttributeValue(OracleEdgeBase e)
{
    OracleVertexBase v = (OracleVertexBase) e.getVertex(Direction.IN);
    if (m_smith.equals(v)) {
        v = (OracleVertexBase) e.getVertex(Direction.OUT);
    }
    String role = v.getProperty("role");
    role = role.toLowerCase();

    if (role.contains("political")) {
        return "political";
    }
    else if (role.contains("actor") || role.contains("singer") ||
        role.contains("actress") || role.contains("writer") ||
        role.contains("producer") || role.contains("director")) {
        return "arts";
    }
    else if (role.contains("journalist")) {
        return "journalism";
    }
    else if (role.contains("player")) {
        return "sports";
    }
    else if (role.contains("business") || role.contains("economist")) {
        return "business";
    }
    else if (role.contains("philanthropist")) {
        return "philanthropy";
    }
    return " ";
}
}
```

Using the operations callbacks defined previously, the following code fragment loads a property graph, creates an instance of the operation callbacks, and later adds the attributes

into the pertinent vertices and edges using the `addAttributeToAllVertices` and `addAttributeToAllEdges` methods in `OraclePropertyGraph`.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
    args, szGraphName);

// Clear existing vertices/edges in the property graph
opg.clearRepository();

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// This object will handle parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, dop);

// Create the vertex operation callback
CollaboratorsVertexOpCallback cvoc =
CollaboratorsVertexOpCallback.getInstance(opg);

// Add attribute to all people collaborating with Smith based on their role
opg.addAttributeToAllVertices(cvoc, true /** Skip store to Cache */, dop);

// Look up for all collaborators of Smith
Iterable<Vertex> collaborators = opg.getVertices("smithCollaborator",
"political");
System.out.println("Political collaborators of Robert Smith " +
getVerticesAsString(collaborators));

collaborators = opg.getVertices("smithCollaborator", "business");
System.out.println("Business collaborators of Robert Smith " +
getVerticesAsString(collaborators));

// Add an attribute to all people having a feud with Robert Smith to set
// the type of relation they have
FeudsEdgeOpCallback feoc = FeudsEdgeOpCallback.getInstance(opg);
opg.addAttributeToAllEdges(feoc, true /** Skip store to Cache */, dop);

// Look up for all feuds of Smith
Iterable<Edge> feuds = opg.getEdges("smithFeud", "political");
System.out.println("\n\nPolitical feuds of Robert Smith " +
getEdgesAsString(feuds));

feuds = opg.getEdges("smithFeud", "business");
System.out.println("Business feuds of Robert Smith " +
getEdgesAsString(feuds));
```

The following code fragment defines an implementation of `VertexOpCallback` that can be used to remove vertices having value `philanthropy` for attribute `smithCollaborator`, then call the API `removeAttributeFromAllVertices`; It also defines an implementation of `EdgeOpCallback` that can be used to remove edges having value `business` for attribute `smithFeud`, then call the API `removeAttributeFromAllEdges`.

```
System.out.println("\n\nRemove 'smithCollaborator' property from all the " +
"philanthropy collaborators");
PhilanthropyCollaboratorsVertexOpCallback pvoc =
PhilanthropyCollaboratorsVertexOpCallback.getInstance();

opg.removeAttributeFromAllVertices(pvoc);
```

```
System.out.println("\n\nRemove 'smithFeud' property from all the" + "business feuds");
BusinessFeudsEdgeOpCallback beoc = BusinessFeudsEdgeOpCallback.getInstance();
```

```
opg.removeAttributeFromAllEdges(beoc);
```

```
/**
 * Implementation of a EdgeOpCallback to remove the "smithCollaborators"
 * property from all people collaborating with Robert Smith that have a
 * philanthropy role
 */
private static class PhilanthropyCollaboratorsVertexOpCallback implements
VertexOpCallback
{
    public static PhilanthropyCollaboratorsVertexOpCallback getInstance()
    {
        return new PhilanthropyCollaboratorsVertexOpCallback();
    }

    /**
     * Remove attribute if and only if the property value for
     * smithCollaborator is Philanthropy
     */
    @Override
    public boolean needOp(OracleVertexBase v)
    {
        String type = v.getProperty("smithCollaborator");
        return type != null && type.equals("philanthropy");
    }

    @Override
    public String getAttributeKeyName(OracleVertexBase v)
    {
        return "smithCollaborator";
    }

    /**
     * Define the property's value. In this case can be empty
     */
    @Override
    public Object getAttributeKeyValue(OracleVertexBase v)
    {
        return " ";
    }
}

/**
 * Implementation of a EdgeOpCallback to remove the "smithFeud" property
 * from all connections in a feud with Robert Smith that have a business role
 */
private static class BusinessFeudsEdgeOpCallback implements EdgeOpCallback
{
    public static BusinessFeudsEdgeOpCallback getInstance()
    {
        return new BusinessFeudsEdgeOpCallback();
    }

    /**
     * Remove attribute if and only if the property value for smithFeud is
     * business
     */
    @Override
```

```

public boolean needOp(OracleEdgeBase e)
{
    String type = e.getProperty("smithFeud");
    return type != null && type.equals("business");
}

@Override
public String getAttributeKeyName(OracleEdgeBase e)
{
    return "smithFeud";
}

/**
 * Define the property's value. In this case can be empty
 */
@Override
public Object getAttributeKeyValue(OracleEdgeBase e)
{
    return " ";
}
}

```

5.4.7 Getting Property Graph Metadata

You can get graph metadata and statistics, such as all graph names in the database; for each graph, getting the minimum/maximum vertex ID, the minimum/maximum edge ID, vertex property names, edge property names, number of splits in graph vertex, and the edge table that supports parallel table scans.

The following code fragment gets the metadata and statistics of the existing property graphs stored in an Oracle database.

```

// Get all graph names in the database
List<String> graphNames = OraclePropertyGraphUtils.getGraphNames(dbArgs);

for (String graphName : graphNames) {
    OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args,
graphName);

    System.err.println("\n Graph name: " + graphName);
    System.err.println(" Total vertices: " +
        opg.countVertices(dop));

    System.err.println(" Minimum Vertex ID: " +
        opg.getMinVertexID(dop));
    System.err.println(" Maximum Vertex ID: " +
        opg.getMaxVertexID(dop));

    Set<String> propertyNamesV = new HashSet<String>();
    opg.getVertexPropertyNames(dop, 0 /* timeout,0 no timeout */,
        propertyNamesV);

    System.err.println(" Vertices property names: " +
        getPropertyNamesAsString(propertyNamesV));

    System.err.println("\n\n Total edges: " + opg.countEdges(dop));
    System.err.println(" Minimum Edge ID: " + opg.getMinEdgeID(dop));
    System.err.println(" Maximum Edge ID: " + opg.getMaxEdgeID(dop));

    Set<String> propertyNamesE = new HashSet<String>();

```



```

opg.getEdgePropertyNames(dop, 0 /* timeout,0 no timeout */,
    propertyNamesE);

System.err.println(" Edge property names: " +
    getPropertyNamesAsString(propertyNamesE));

System.err.println("\n\n Table Information: ");
System.err.println("Vertex table number of splits: " +
    (opg.getVertexPartitionsNumber()));
System.err.println("Edge table number of splits: " +
    (opg.getEdgePartitionsNumber()));
}

```

5.4.8 Merging New Data into an Existing Property Graph

In addition to loading graph data into an empty property graph in Oracle Database, you can merge new graph data into an existing (empty or non-empty) graph. As with data loading, data merging splits the input vertices and edges into multiple chunks and merges them with the existing graph in database in parallel.

When doing the merging, the flows are different depends on whether there is an overlap between new graph data and existing graph data. *Overlap* here means that the same key of a graph element may have different values in the new and existing graph data. For example, key `weight` of the vertex with ID 1 may have value 0.8 in the new graph data and value 0.5 in the existing graph data. In this case, you must specify whether the new value or the existing value should be used for the key.

The following options are available for graph data merging: JDB-based, external table-based, and SQL loader-based merging.

- JDBC-Based Graph Data Merging
- External Table-Based Data Merging
- SQL Loader-Based Data Merging

JDBC-Based Graph Data Merging

JDBC-based data merging uses Java Database Connectivity (JDBC) APIs to load the new graph data into Oracle Database and then merge the new graph data into an existing graph.

The following example merges the new graph data from vertex and edge files `szOPVFile` and `szOPEFile` in Oracle-defined Flat-file format with an existing graph named `opg`, using a JDBC-based data merging with a DOP (degree of parallelism) of 48, batch size of 1000, and specified data merging options.

```

String szOPVFile = "../data/connectionsNew.opv";
String szOPEFile = "../data/connectionsNew.ope";
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.mergeData(opg, szOPVFile, szOPEFile,
    48 /*DOP*/,
    1000 /*Batch Size*/,
    true /*Rebuild index*/,
    "pdml=t, pddl=t, no_dup=t, use_new_val_for_dup_key=t" /*Merge
options*/);

```

To optimize the performance of the data merging operations, a set of flags and hints can be specified in the merging options parameter when calling the JDBC-based data merging. These hints include:

- **DOP:** The degree of parallelism to use when merging the data. This parameter determines the number of chunks to generate when splitting the file, as well as the number of loader threads to use when merging the data into the property graph VT\$ and GE\$ tables.
- **Batch Size:** An integer specifying the batch size to use for Oracle JDBC statements in batching mode.
- **Rebuild index:** If set to true, the data loader will disable all the indexes and constraints defined over the property graph into which the data will be loaded. After all the data is merged into the property graph, all the original indexes and constraints will be rebuilt and enabled.
- **Merge options:** An option (or multiple options separated by commas) to optimize the data merging operations. These options include:
 - PDML=T: enables parallel execution for DML operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - PDDL=T: enables parallel execution for DDL operations for the database session used in the data loader. This hint is used to improve the performance of long-running batching jobs.
 - NO_DUP=T: assumes the input new graph data does not have invalid duplicates. In a valid property graph, each vertex (or edge) can at most have one value for a given property key. In an invalid property graph, a vertex (or edge) may have two or more values for a particular key. As an example, a vertex, v, has two key/value pairs: name/"John" and name/"Johnny", and they share the same key.
 - OVERLAP=F: assumes there is no overlap between new graph data and existing graph data. That is, there is no key with multiple distinct values in the new and existing graph data.
 - USE_NEW_VAL_FOR_DUP_KEY=T: if there is overlap between new graph data and existing graph data, use the value in the new graph data; otherwise, use the value in the existing graph data.

External Table-Based Data Merging

External table-based data merging uses an external table to load new graph data into Oracle Database and then merge the new graph data into an existing graph.

External-table based data merging requires a directory object, where the files read by the external tables will be stored. This directory can be created using the following SQL*Plus statements:

```
create or replace directory tmp_dir as '/tmppath/';  
grant read, write on directory tmp_dir to public;
```

The following example merges the new graph data from a vertex and edge files szOPVFile and szOPEFile in Oracle flat-file format with an existing graph opg using an

external table-based data merging, a DOP (degree of parallelism) of 48, and specified merging options.

```
String szOPVFile = "../../data/connectionsNew.opv";
String szOPEFile = "../../data/connectionsNew.ope";
String szExtDir = "tmp_dir";
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.mergeDataWithExtTab(opg, szOPVFile, szOPEFile,
    48 /*DOP*/,
    true /*Use Named Pipe for splitting*/,
    szExtDir /*database directory object*/,
    true /*Rebuild index*/,
    "pdml=t, pddl=t, no_dup=t, use_new_val_for_dup_key=t" /*Merge
options*/);
```

SQL Loader-Based Data Merging

SQL loader-based data merging uses Oracle SQL*Loader to load the new graph data into Oracle Database and then merge the new graph data into an existing graph.

The following example merges the new graph data from a vertex and edge files szOPVFile and szOPEFile in Oracle Flat-file format with an existing graph opg using an SQL loader - based data merging with a DOP (degree of parallelism) of 48 and the specified merging options. To use the APIs, the path to the SQL*Loader needs to be specified.

```
String szUser = "username";
String szPassword = "password";
String szDbId = "db18c"; /*service name of the database*/
String szOPVFile = "../../data/connectionsNew.opv";
String szOPEFile = "../../data/connectionsNew.ope";
String szSQLLoaderPath = "<YOUR_ORACLE_HOME>/bin/sqlldr";
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.mergeDataWithSqlLdr(opg, szUser, szPassword, szDbId, szOPVFile,
szOPEFile,
    48 /*DOP*/,
    true /*Use Named Pipe for splitting*/,
    szSQLLoaderPath /* SQL*Loader path: the path to bin/sqlldr */,
    true /*Rebuild index*/,
    "pdml=t, pddl=t, no_dup=t, use_new_val_for_dup_key=t" /*Merge
options*/);
```

5.4.9 Opening and Closing a Property Graph Instance

When describing a property graph, use these Oracle Property Graph classes to open and close the property graph instance properly:

- `OraclePropertyGraph.getInstance`: Opens an instance of an Oracle property graph. This method has two parameters, the connection information and the graph name. The format of the connection information depends on whether you use HBase or Oracle NoSQL Database as the backend database.
- `OraclePropertyGraph.clearRepository`: Removes all vertices and edges from the property graph instance.

- `OraclePropertyGraph.shutdown`: Closes the graph instance.

For Oracle Database, the `OraclePropertyGraph.getInstance` method uses an Oracle instance to manage the database connection. `OraclePropertyGraph` has a set of constructors that let you set the graph name, number of hash partitions, degree of parallelism, tablespace, and options for storage (such as compression). For example:

```
import oracle.pg.rdbms.*;
Oracle oracle = new Oracle(jdbcURL, username, password);

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(oracle,
graphName);
opg.clearRepository();
//      .
//      . Graph description
//      .
// Close the graph instance
opg.shutdown();
```

If the in-memory analyst functions are required for an application, you should use `GraphConfigBuilder` to create a graph for Oracle Database, and instantiate `OraclePropertyGraph` with that graph name as an argument. For example, the following code snippet constructs a graph config, gets an `OraclePropertyGraph` instance, loads some data into that graph, and gets an in-memory analyst.

```
import oracle.pgx.config.*;
import oracle.pgx.api.*;
import oracle.pgx.common.types.*;

...

PgNosqlGraphConfig cfg = GraphConfigBuilder.forPropertyGraphRdbms ()
    .setJdbcUrl("jdbc:oracle:thin:@<hostname>:1521:<sid>")
    .setUsername("<username>").setPassword("<password>")
    .setName(szGraphName)
    .setMaxNumConnections(8)
    .addEdgeProperty("lbl", PropertyType.STRING, "lbl")
    .addEdgeProperty("weight", PropertyType.DOUBLE, "1000000")
    .build();

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(cfg);

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// perform a parallel data load
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

...

PgxSession session = Pgx.createSession("session-id-1");
```

```
PgxGraph g = session.readGraphWithProperties(cfg);

Analyst analyst = session.createAnalyst();
...
```

5.4.10 Creating Vertices

To create a vertex, use these Oracle Property Graph methods:

- `OraclePropertyGraph.addVertex`: Adds a vertex instance to a graph.
- `OracleVertex.setProperty`: Assigns a key-value property to a vertex.
- `OraclePropertyGraph.commit`: Saves all changes to the property graph instance.

The following code fragment creates two vertices named `v1` and `v2`, with properties for age, name, weight, height, and sex in the `opg` property graph instance. The `v1` properties set the data types explicitly.

```
// Create vertex v1 and assign it properties as key-value pairs
Vertex v1 = opg.addVertex(1l);
    v1.setProperty("age", Integer.valueOf(31));
    v1.setProperty("name", "Alice");
    v1.setProperty("weight", Float.valueOf(135.0f));
    v1.setProperty("height", Double.valueOf(64.5d));
    v1.setProperty("female", Boolean.TRUE);

Vertex v2 = opg.addVertex(2l);
    v2.setProperty("age", 27);
    v2.setProperty("name", "Bob");
    v2.setProperty("weight", Float.valueOf(156.0f));
    v2.setProperty("height", Double.valueOf(69.5d));
    v2.setProperty("female", Boolean.FALSE);
```

5.4.11 Creating Edges

To create an edge, use these Oracle Property Graph methods:

- `OraclePropertyGraph.addEdge`: Adds an edge instance to a graph.
- `OracleEdge.setProperty`: Assigns a key-value property to an edge.

The following code fragment creates two vertices (`v1` and `v2`) and one edge (`e1`).

```
// Add vertices v1 and v2
Vertex v1 = opg.addVertex(1l);
v1.setProperty("name", "Alice");
v1.setProperty("age", 31);

Vertex v2 = opg.addVertex(2l);
v2.setProperty("name", "Bob");
v2.setProperty("age", 27);

// Add edge e1
Edge e1 = opg.addEdge(1l, v1, v2, "knows");
e1.setProperty("type", "friends");
```

5.4.12 Deleting Vertices and Edges

You can remove vertex and edge instances individually, or all of them simultaneously. Use these methods:

- `OraclePropertyGraph.removeEdge`: Removes the specified edge from the graph.
- `OraclePropertyGraph.removeVertex`: Removes the specified vertex from the graph.
- `OraclePropertyGraph.clearRepository`: Removes all vertices and edges from the property graph instance.

The following code fragment removes edge `e1` and vertex `v1` from the graph instance. The adjacent edges will also be deleted from the graph when removing a vertex. This is because every edge must have an beginning and ending vertex. After removing the beginning or ending vertex, the edge is no longer a valid edge.

```
// Remove edge e1
opg.removeEdge(e1);

// Remove vertex v1
opg.removeVertex(v1);
```

The `OraclePropertyGraph.clearRepository` method can be used to remove all contents from an `OraclePropertyGraph` instance. However, use it with care because this action cannot be reversed.

5.4.13 Reading a Graph from a Database into an Embedded In-Memory Analyst

You can read a graph from Oracle Database into an in-memory analyst that is embedded in the same client Java application (a single JVM). For the following example, a correct `java.io.tmpdir` setting is required.

```
int dop = 8; // need customization
Map<PgxCfg.Field, Object> confPgx = new HashMap<PgxCfg.Field, Object>();
confPgx.put(PgxCfg.Field.ENABLE_GM_COMPILER, false);
confPgx.put(PgxCfg.Field.NUM_WORKERS_IO, dop); //
confPgx.put(PgxCfg.Field.NUM_WORKERS_ANALYSIS, dop); // <= # of
physical cores
confPgx.put(PgxCfg.Field.NUM_WORKERS_FAST_TRACK_ANALYSIS, 2);
confPgx.put(PgxCfg.Field.SESSION_TASK_TIMEOUT_SECS, 0); // no
timeout set
confPgx.put(PgxCfg.Field.SESSION_IDLE_TIMEOUT_SECS, 0); // no
timeout set

PgRdbmsGraphConfig cfg =
GraphConfigBuilder.forPropertyGraphRdbms().setJdbcUrl("jdbc:oracle:thin:
@<your_db_host>:<db_port>:<db_sid>")
    .setUsername("<username>")
    .setPassword("<password>")
    .setName("<graph_name>")
    .setMaxNumConnections(8)
```

```
        .setLoadEdgeLabel(false)
        .build();
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(cfg);
ServerInstance localInstance = Pgx.getInstance();
localInstance.startEngine(confPgx);
PgxSession session = localInstance.createSession("session-id-1"); // Put
your session description here.

Analyst analyst = session.createAnalyst();

// The following call will trigger a read of graph data from the database
PgxGraph pgxGraph = session.readGraphWithProperties(opg.getConfig());

long triangles = analyst.countTriangles(pgxGraph, false);
System.out.println("triangles " + triangles);

// Remove edge e1
opg.removeEdge(e1);

// Remove vertex v1
opg.removeVertex(v1);
```

5.4.14 Specifying Labels for Vertices

The database and data access layer do not provide labels for vertices; however, you can treat the value of a designated vertex property as one or more labels. Such a transformation is relevant only to the in-memory analyst.

In the following example, a property "country" is specified in a call to `setUseVertexPropertyValueAsLabel()`, and the comma delimiter "," is specified in a call to `setPropertyValueDelimiter()`. These two together imply that values of the `country` vertex property will be treated as vertex labels separated by a comma. For example, if vertex X has a string value "US" for its country property, then its vertex label will be US; and if vertex Y has a string value "UK,CN", then it will have two labels: UK and CN.

```
GraphConfigBuilder.forPropertyGraph...
    .setName("<your_graph_name>")
    ...
    .setUseVertexPropertyValueAsLabel("country")
    .setPropertyValueDelimiter(",")
    .setLoadVertexLabels(true)
    .build();
```

Related Topics

- [What Are Property Graphs?](#)

5.4.15 Building an In-Memory Graph

In addition to [Store the Database Password in a Keystore](#), you can create an in-memory graph programmatically. This can simplify development when the size of graph is small or when the content of the graph is highly dynamic. The key Java class is `GraphBuilder`, which can accumulate a set of vertices and edges added with the `addVertex` and `addEdge` APIs.

After all changes are made, an in-memory graph instance (`PgxGraph`) can be created by the `GraphBuilder`.

The following Java code snippet illustrates a graph construction flow. Note that there are no explicit calls to `addVertex`, because any vertex that does not already exist will be added dynamically as its adjacent edges are created.

```
import oracle.pgx.api.*;

PgxSession session = Pgx.createSession("example");
GraphBuilder<Integer> builder = session.newGraphBuilder();

builder.addEdge(0, 1, 2);
builder.addEdge(1, 2, 3);
builder.addEdge(2, 2, 4);
builder.addEdge(3, 3, 4);
builder.addEdge(4, 4, 2);

PgxGraph graph = builder.build();
```

To construct a graph with vertex properties, you can use `setProperty` against the vertex objects created.

```
PgxSession session = Pgx.createSession("example");
GraphBuilder<Integer> builder = session.newGraphBuilder();

builder.addVertex(1).setProperty("double-prop", 0.1);
builder.addVertex(2).setProperty("double-prop", 2.0);
builder.addVertex(3).setProperty("double-prop", 0.3);
builder.addVertex(4).setProperty("double-prop", 4.56789);

builder.addEdge(0, 1, 2);
builder.addEdge(1, 2, 3);
builder.addEdge(2, 2, 4);
builder.addEdge(3, 3, 4);
builder.addEdge(4, 4, 2);

PgxGraph graph = builder.build();
```

To use long integers as vertex and edge identifiers, specify `IdType.LONG` when getting a new instance of `GraphBuilder`. For example:

```
import oracle.pgx.common.types.IdType;
GraphBuilder<Long> builder = session.newGraphBuilder(IdType.LONG);
```

During edge construction, you can directly use vertex objects that were previously created in a call to `addEdge`.

```
v1 = builder.addVertex(11).setProperty("double-prop", 0.5)
v2 = builder.addVertex(21).setProperty("double-prop", 2.0)

builder.addEdge(0, v1, v2)
```


As with vertices, edges can have properties. The following example sets the edge label by using `setLabel`:

```
builder.addEdge(4, v4, v2).setProperty("edge-prop",  
"edge_prop_4_2").setLabel("label")
```

5.4.16 Dropping a Property Graph

To drop a property graph from the database, use the `OraclePropertyGraphUtils.dropPropertyGraph` method. This method has two parameters, the connection information and the graph name. For example:

```
// Drop the graph  
Oracle oracle = new Oracle(jdbcUrl, username, password);  
OraclePropertyGraphUtils.dropPropertyGraph(oracle, graphName);
```

You can also drop a property graph using the PL/SQL API. For example:

```
EXECUTE opg_apis.drop_pg('my_graph_name');
```

5.4.17 Executing PGQL Queries

You can execute PGQL queries directly against Oracle Database with the `PgqlStatement` and `PgqlPreparedStatement` interfaces. See [Executing PGQL Queries Against Property Graph Schema Tables](#) for details.

5.5 Managing Text Indexing for Property Graph Data

Indexes in Oracle Spatial and Graph property graph support allow fast retrieval of elements by a particular key/value or key/text pair. These indexes are created based on an element type (vertices or edges), a set of keys (and values), and an index type.

Oracle Spatial and Graph supports the use of the Oracle Text indexing technology, which is a feature of Oracle Database.

Two types of indexing structures are supported.

- Automatic text indexes provide automatic indexing of vertices or edges by a set of property keys. Their main purpose is to enhance query performance on vertices and edges based on particular key/value pairs.
- Manual text indexes enable you to define multiple indexes over a designated set of vertices and edges of a property graph. You must specify what graph elements go into the index.

Oracle Spatial and Graph provides APIs to create manual and automatic text indexes over property graphs stored in Oracle Database. Indexes are managed using Oracle Text, a proprietary search and analysis engine. The rest of this section focuses on how to create text indexes using the property graph capabilities of the Data Access Layer.

- [Configuring a Text Index for Property Graph Data](#)
- [Using Automatic Indexes for Property Graph Data](#)
- [Using Manual Indexes for Property Graph Data](#)

- [Executing Search Queries Over a Property Graph's Text Indexes](#)
- [Handling Data Types](#)
- [Updating Configuration Settings on Text Indexes for Property Graph Data](#)
Oracle's property graph support manages manual and automatic text indexes through integration with Oracle Text.
- [Using Parallel Query on Text Indexes for Property Graph Data](#)

5.5.1 Configuring a Text Index for Property Graph Data

The configuration of a text index is defined using an `OracleIndexParameters` object. This object includes information about the index such as search engine, location, number of directories (or shards), and degree of parallelism.

By default, text indexes are configured based on the `OracleIndexParameters` associated with the property graph using the method `opg.setDefaultIndexParameters(indexParams)`. The initial creation of the automatic index delimits the configuration and text search engine for future indexed keys.

Indexes can also be created by specifying a different set of parameters. The following code fragment creates an automatic text index over an existing property graph using a Lucene engine with a physical directory.

```
// Create an OracleIndexParameters object to get Index configuration (search
engine, etc).
OracleIndexParameters indexParams = OracleIndexParameters.buildFS(args)

// Create auto indexing on above properties for all vertices
opg.createKeyIndex("name", Vertex.class, indexParams.getParameters());
```

Any index configuration operations cause updates to be made to the IT\$ table, which is explained in [Property Graph Tables \(Detailed Information\)](#).

- [Configuring Text Indexes Using Oracle Text](#)

5.5.1.1 Configuring Text Indexes Using Oracle Text

Oracle Spatial and Graph supports automatic text indexes using Oracle Text. Oracle Text uses standard SQL to index, search, and analyze text values stored in the V column of the vertices (or edges) table. Because Oracle Text indexes all the existing K/V pairs of the vertices (or edges) in the property graph, this option can be used **only** with automatic text indexes and must use a wildcard ("*") indexed key parameter during the index creation.

Because the property graph feature uses an NVARCHAR typed column for a better support of Unicode, it is highly recommended that UTF8 (AL32UTF8) be used as the database character set.

To create an Oracle Text index on the vertices table (or edges table), the ALTER SESSION privilege is required. The following example grants the privilege.

```
SQL> grant alter session to <YOUR_USER_SCHEMA_HERE>;
```

If customization is required, grant EXECUTE on CTX_DDL, as in the following example.

```
SQL> grant execute on ctx_ddl to <YOUR_USER_SCHEMA_HERE>;
```

A text index using Oracle Text uses an `OracleTextIndexParameters` object. The configuration parameters for indexes using a Oracle Text include:

- **Preference owner:** the owner of the preference.
- **Data store:** the datastore preference specifying how the text values are stored. A datastore preference can be created using `ctx_ddl.create_preference` API as follows:

```
SQL> -- The following requires access privilege to CTX_DDL
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_DATASTORE',
'DIRECT_DATASTORE');
```

If the value is set to NULL, then the index will be created with `CTXSYS.DEFAULT_DATASORE`. This preference uses a `DIRECT_DATASTORE` type.

- **Filter:** the filter preference determining how text is filtered for indexing. A filter preference can be created using `ctx_ddl.create_preference`, as follows:

```
SQL> -- The following requires access privilege to CTX_DDL
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_FILTER', 'AUTO_FILTER');
```

If the value is set to NULL, then the index will be created with `CTXSYS.NULL_FILTER`. This preference uses a `NULL_FILTER` type.

- **Storage:** the storage preference specifying table space and creation parameters for tables associated with a Text index. A storage preference can be created using `ctx_ddl.create_preference`, as follows:

```
SQL> -- The following requires access privilege to CTX_DDL
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_STORAGE', 'BASIC_STORAGE');
```

If the value is set to NULL, then the index will be created with `CTXSYS.DEFAULT_STORAGE`. This preference uses a `BASIC_STORAGE` type.

- **Word list:** the word list preference specifying the enabled query options. These query options may include stemming, fuzzy matching, substring, and prefix indexing. A data store preference can be created using `ctx_ddl.create_preference`, as follows:

```
SQL> -- The following example enables stemming and fuzzy matching for
English.
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_WORDLIST',
'BASIC_WORDLIST');
```

If the value is set to NULL, then the index will be created with `CTXSYS.DEFAULT_WORDLIST`. This preference uses the language stemmer for your database language.

- **Stop list:** the stop list preference specifying the list of words that are not meant to be indexed. A stop list preference can be created using `ctx_ddl.create_stoplist`.

If the value is set to NULL, then the index will be created with CTXSYS.DEFAULT_STOPLIST. This preference uses the stoplist of your database language.

- **Lexer:** the lexer preference specifying the language of the text to be indexed. A lexer preference can be created using `ctx_ddl.create_preference`, as follows:

```
SQL> -- The following requires access privilege to CTX_DDL
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_AUTO_LEXER',
'AUTO_LEXER');
```

If the value is set to NULL, then the index will be created with CTXSYS.DEFAULT_LEXER. This preference uses a BASIC_LEXER type with additional options based on the language used at installation time.

The following code fragment creates the configuration for a text index using Oracle Text with default options and OPG_AUTO_LEXER.

```
String prefOwner = "scott";
String datastore = (String) null;
String filter = (String) null;
String storage = (String) null;
String wordlist = (String) null;
String stoplist = (String) null;
String lexer = "OPG_AUTO_LEXER";
String options = (String) null;

OracleIndexParameters params
    =
OracleTextIndexParameters.buildOracleText(prefOwner,
                                           datastore,
                                           filter,
                                           storage,
                                           wordlist,
                                           stoplist,
                                           lexer,
                                           dop,
                                           options);
```

5.5.2 Using Automatic Indexes for Property Graph Data

An automatic text index provides automatic indexing of vertices or edges by a set of property keys. Its main purpose is to increase the speed of lookups over vertices and edges based on particular key/value pair. If an automatic index for the given key is enabled, then key/value pair lookups will be performed as a text search against the index instead of as a database lookup.

When specifying an automatic index over a property graph, use the following methods to create, remove, and manipulate an automatic index:

- `OraclePropertyGraph.createKeyIndex(String key, Class elementClass, Parameter[] parameters)`: Creates an automatic index for all elements of type `elementClass` by the given property key. The index is configured based on the specified parameters.
- `OraclePropertyGraph.createKeyIndex(String[] keys, Class elementClass, Parameter[] parameters)`: Creates an automatic index for all elements of type

`elementClass` by using a set of property keys. The index is configured based on the specified parameters.

- `OraclePropertyGraph.dropKeyIndex(String key, Class elementClass)`: Drops the automatic index for all elements of type `elementClass` for the given property key.
- `OraclePropertyGraph.dropKeyIndex(String[] keys, Class elementClass)`: Drops the automatic index for all elements of type `elementClass` for the given set of property keys.
- `OraclePropertyGraph.getAutoIndex(Class elementClass)`: Gets an index instance of the automatic index for type `elementClass`.
- `OraclePropertyGraph.getIndexedKeys(Class elementClass)`: Gets the set of indexed keys currently used in an automatic index for all elements of type `elementClass`.

By default, indexes are configured based on the `OracleIndexParameters` associated with the property graph using the method `opg.setDefaultIndexParameters(indexParams)`.

Indexes can also be created by specifying a different set of parameters. This is shown in the following code snippet.

```
// Create an OracleIndexParameters object to get Index configuration (search engine,
etc).
OracleIndexParameters indexParams = OracleIndexParameters.buildFS(args)

// Create auto indexing on above properties for all vertices
opg.createKeyIndex("name", Vertex.class, indexParams.getParameters());
```

The code fragment in the next example executes a query over all vertices to find all matching vertices with the key/value pair `name:Robert Smith`. This operation will execute a lookup into the text index.

Additionally, wildcard searches are supported by specifying the parameter `useWildCards` in the `getVertices` API call. Wildcard search is only supported when automatic indexes are enabled for the specified property key.

```
// Find all vertices with name Robert Smith.
Iterator<Vertices> vertices = opg.getVertices("name", "Robert Smith").iterator();
System.out.println("----- Vertices with name Robert Smith -----");
countV = 0;
while (vertices.hasNext()) {
    System.out.println(vertices.next());
    countV++;
}
System.out.println("Vertices found: " + countV);

// Find all vertices with name including keyword "Smith"
// Wildcard searching is supported.
boolean useWildcard = true;
Iterator<Vertices> vertices = opg.getVertices("name", "*Smith*").iterator();
System.out.println("----- Vertices with name *Smith* -----");
countV = 0;
while (vertices.hasNext()) {
    System.out.println(vertices.next());
    countV++;
}
System.out.println("Vertices found: " + countV);
```

The preceding code example produces output like the following:

```
----- Vertices with name Robert Smith-----
Vertex ID 1 {name:str:Robert Smith, role:str:political authority, occupation:str:CEO
```

```
of Example Corporation, country:str:United States, political
party:str:Bipartisan, religion:str:Unknown}
Vertices found: 1
```

```
----- Vertices with name *Smith* -----
Vertex ID 1 {name:str:Robert Smith, role:str:political authority,
occupation:str:CEO of Example Corporation, country:str:United States, political
party:str:Bipartisan, religion:str:Unknown}
Vertices found: 1
```

5.5.3 Using Manual Indexes for Property Graph Data

Manual indexes support the definition of multiple indexes over the vertices and edges of a property graph. A manual index requires that you manually put, get, and remove elements from the index.

When describing a manual index over a property graph, use the following methods to add, remove, and manipulate a manual index:

- `OraclePropertyGraph.createIndex(String name, Class elementClass, Parameter[] parameters)`: Creates a manual index with the specified name for all elements of type `elementClass`.
- `OraclePropertyGraph.dropIndex(String name)`: Drops the given manual index.
- `OraclePropertyGraph.getIndex(String name, Class elementClass)`: Gets an index instance of the given manual index for type `elementClass`.
- `OraclePropertyGraph.getIndices()`: Gets an array of index instances for all manual indexes created in the property graph.

5.5.4 Executing Search Queries Over a Property Graph's Text Indexes

Oracle Spatial and Graph provides a set of utilities to execute text search queries over automatic and manual text indexes. These utilities vary from querying based on a particular key/value pair, to executing a text search over a single or multiple keys (with extended query options such as wildcards, fuzzy searches, and range queries).

- [Executing Search Queries Over a Text Index Using Oracle Text](#)

5.5.4.1 Executing Search Queries Over a Text Index Using Oracle Text

Text search queries on Oracle Text are translated into SELECT SQL queries with a "*contains*" clause including a score range and ordering, and score ID. Oracle's property graph includes an utility called `OracleTextQueryObject`, which lets you execute text search queries over an Oracle Text index.

The following code fragment creates an automatic index using Oracle Text, and executes a query over the text index by specifying a particular key/value pair.

```
String prefOwner = "scott";
String datastore = (String) null;
String filter = (String) null;
String storage = (String) null;
String wordlist = (String) null;
String stoplist = (String) null;
String lexer = "OPG_AUTO_LEXER";
String options = (String) null;
```

```

OracleIndexParameters params
    = OracleTextIndexParameters.buildOracleText(prefOwner,
                                                datastore,
                                                filter,
                                                storage,
                                                wordlist,
                                                stoplist,
                                                lexer,
                                                dop,
                                                options);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on all existing properties, use wildcard for all
opg.createKeyIndex("*", Vertex.class);

// Get the auto index object
OracleIndex<Vertex> index = ((OracleIndex<Vertex>) opg.getAutoIndex(Vertex.class));

// Create the text query object for Oracle Text
OracleTextQueryObject otqo
    = OracleTextQueryObject.getInstance("Smith" /* query body */,
                                       1 /* score */,
                                       ScoreRange.POSITIVE /* Score range
*/,
                                       Direction.ASC /* order by
direction*/);

Iterator<Vertex> vertices = index.get("name", otqo).iterator();
System.out.println("----- Vertices with query: " + otqo.toString() + " -----");
countV = 0;
while (vertices.hasNext()) {
    System.out.println(vertices.next());
    countV++;
}
System.out.println("Vertices found: " + countV);

```

You can filter the date type of the matching key/value pairs by specifying the data type class to execute the query against. The following code fragment executes a query over the text index to retrieve all properties with a String value including the word *Smith*.

```

// Create the text query object for Oracle Text
OracleTextQueryObject otqo
    = OracleTextQueryObject.getInstance("Smith" /* query body */,
                                       1 /* score */,
                                       ScoreRange.POSITIVE
/* Score range */,
                                       Direction.ASC
/* order by direction*/,
                                       "name",
                                       String.class);

Iterator<Vertex> vertices = index.get("name", otqo).iterator();
System.out.println("----- Vertices with query: " + otqo.toString() + " -----");
countV = 0;
while (vertices.hasNext()) {
    System.out.println(vertices.next());
    countV++;
}
System.out.println("Vertices found: " + countV);

```

5.5.5 Handling Data Types

Oracle's property graph support indexes and stores an element's Key/Value pairs based on the value data type. The main purpose of handling data types is to provide extensive query support like numeric and date range queries.

By default, searches over a specific key/value pair are matched up to a query expression based on the value's data type. For example, to find vertices with the key/value pair `age:30`, a query is executed over all `age` fields with a data type integer. If the value is a query expression, you can also specify the data type class of the value to find by calling the API `get(String key, Object value, Class dtClass, Boolean useWildcards)`. If no data type is specified, the query expression will be matched to all possible data types.

When dealing with Boolean operators, each subsequent key/value pair must append the data type's prefix/suffix so the query can find proper matches.

- [Handling Data Types on Oracle Text](#)

5.5.5.1 Handling Data Types on Oracle Text

Text indexes using Oracle Text are created over the K and V text columns of the property graph tables. In order to provide text indexing capabilities on all available data types, Oracle populates the V column with a string representation of numeric, spatial, and date time key/value pairs.

To specify the date time and numeric formats used when populating the V column, you can use the methods `setNumberToCharSqlFormatString` and `setTimeToCharSqlFormatString`. The following code snippet shows how to set the date time and numeric formats in a property graph instance.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args,
                                                         szGraphName);
opg.setNumberToCharSqlFormatString("TM9");
opg.setTimeToCharSqlFormatString("SYYYY-MM-DD\"T\"HH24:MI:SS.FF9TZH:TZM");
```

When executing a text search query over a numeric or date time value, you should use a text expression using the format associated to the property graph.

OraclePropertyGraph includes a utility API `opg.parseValueToCharSQLFormatString` that lets you parse a numeric or date time object into format used in the V column storage. The following code snippet calls this function with a date value and creates a text query object out of the retrieved text.

```
Date d = new java.util.Date(1001);
String szDate = opg.parseValueToCharSQLFormatString(d);

// Create the text query object for Oracle Text
OracleTextQueryObject otqo
    = OracleTextQueryObject.getInstance(szDate /* query body */,
                                       1 /* score */,
                                       ScoreRange.POSITIVE /* Score
range */,
                                       Direction.ASC /* order by
direction);
```


5.5.6 Updating Configuration Settings on Text Indexes for Property Graph Data

Oracle's property graph support manages manual and automatic text indexes through integration with Oracle Text.

At creation time, you must create an `OracleIndexParameters` object specifying the search engine and other configuration settings to be used by the text index. After a text index for property graph is created, these configuration settings cannot be changed.

For automatic indexes, all vertex index keys are managed by a single text index, and all edge index keys are managed by a different text index using the configuration specified when the first vertex or edge key is indexed.

If you need to change the configuration settings, you must first disable the current index and create it again using a new `OracleIndexParameters` object.

5.5.7 Using Parallel Query on Text Indexes for Property Graph Data

Text indexes in Oracle Spatial and Graph allow executing text queries over millions of vertices and edges by a particular key/value or key/text pair using parallel query execution.

Parallel text query will produce an array where each element holds all the vertices (or edges) with an attribute matching the given K/V pair from a shard. The subset of shards queried will be delimited by the given start sub-directory ID and the size of the connections array provided. This way, the subset will consider shards in the range of [start, start - 1 + size of connections array]. Note that an integer ID (in the range of [0, N - 1]) is assigned to all the shards in index with N shards.

- [Parallel Text Search Using Oracle Text](#)

5.5.7.1 Parallel Text Search Using Oracle Text

You can use parallel text query using Oracle Text by calling the method `getPartitioned` in `OracleTextAutoIndex`, specifying an array of connections to Oracle Text (Connection objects), the key/value pair to search, and the starting partition ID.

The following code fragment generates an automatic text index using Oracle Text and executes a parallel text query. The number of calls to the `getPartitioned` method in the `OracleTextAutoIndex` class is controlled by the total number of partitions in the VT\$ (or GE\$ tables) and the number of connections used.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(...);
String prefOwner = "scott";
String datastore = (String) null;
String filter = (String) null;
String storage = (String) null;
String wordlist = (String) null;
String stoplist = (String) null;
String lexer = "OPG_AUTO_LEXER";
String options = (String) null;

OracleIndexParameters params
    = OracleTextIndexParameters.buildOracleText(prefOwner,
                                                datastore,
                                                filter,
```

```
storage,
wordlist,
stoplist,
lexer,
dop,
options);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on all existing properties, use wildcard for all
opg.createKeyIndex("*", Vertex.class);

// Create the text query object for Oracle Text
OracleTextQueryObject otqo
    = OracleTextQueryObject.getInstance("Smith" /* query body */,
    1 /* score */,
    ScoreRange.POSITIVE /* Score
range */,
    Direction.ASC /* order by
direction*/);

// Get the Connection object
Connection[] conns = new Connection[dop];
for (int idx = 0; idx < conns.length; idx++) {
    conns[idx] = opg.getOracle().clone().getConnection();
}

// Get the auto index object
OracleIndex<Vertex> index = ((OracleIndex<Vertex>)
opg.getAutoIndex(Vertex.class));

// Iterate to cover all the partitions in the index
long lCount = 0;
for (int split = 0; split < index.getTotalShards();
    split += conns.length) {
    // Gets elements from split to split + conns.length
    Iterable<Vertex>[] iterAr = index.getPartitioned(conns /* connections */,
    "name" /* key */,
    otqo,
    true /* wildcards */,
    split /* start split ID */);

    lCount = countFromIterables(iterAr); /* Consume iterables in parallel */
}

// Close the connections
for (int idx = 0; idx < conns.length; idx++) {
    conns[idx].dispose();
}

// Count results
System.out.println("Vertices found using parallel query: " + lCount);
```

5.6 Access Control for Property Graph Data (Graph-Level and OLS)

Oracle Graph supports two access control and security models: graph level access control, and fine-grained security through integration with Oracle Label Security (OLS).

- Graph-level access control relies on grant/revoke to allow/disallow users other than the owner to access a property graph.
- OLS for property graph data allows sensitivity labels to be associated with individual vertex or edge stored in a property graph.

The default control of access to property graph data stored in an Oracle Database is at the graph level: the owner of a graph can grant read, insert, delete, update and select privileges on the graph to other users.

However, for applications with stringent security requirements, you can enforce a fine-grained access control mechanism by using the Oracle Label Security option of Oracle Database. With OLS, for each query, access to specific elements (vertices or edges) is granted by comparing their labels with the user's labels. (For information about using OLS, see *Oracle Label Security Administrator's Guide* .)

With Oracle Label Security enabled, elements (vertices or edges) may not be inserted in the graph if the same elements exist in the database with a stronger sensitivity label. For example, assume that you have a vertex with a very sensitive label, such as: (Vertex ID 1 {name:str:v1} "SENSITIVE"). This actually prevents a low-privileged (PUBLIC) user from updating the vertex: (Vertex ID 1 {name:str:v1} "PUBLIC"). On the other hand, if a high-privileged user overwrites a vertex or an edge that had been created with a low-level security label, the newer label with higher security will be assigned to the vertex or edge, and the low-privileged user will not be able to see it anymore.

- [Applying Oracle Label Security \(OLS\) on Property Graph Data](#)
This topic presents an example illustrating how to apply OLS to property graph data.

5.6.1 Applying Oracle Label Security (OLS) on Property Graph Data

This topic presents an example illustrating how to apply OLS to property graph data.

Because the property graph is stored in regular relational tables, this example is no different from applying OLS on a regular relational table. The following shows how to configure and enable OLS, create a security policy with security labels, and apply it to a property graph. The code examples are very simplified, and do not necessarily reflect recommended practices regarding user names and passwords.

1. As SYSDBA, create database users named userP, userP2, userS, userTS, userTS2 and pgAdmin.

```
CONNECT / as sysdba;

CREATE USER userP IDENTIFIED BY userPpass;
GRANT connect, resource, create table, create view, create any index TO
userP;
GRANT unlimited TABLESPACE to userP;
```

```
CREATE USER userP2 IDENTIFIED BY userP2pass;
GRANT connect, resource, create table, create view, create any
index TO userP2;
GRANT unlimited TABLESPACE to userP2;
```

```
CREATE USER users IDENTIFIED BY userspass;
GRANT connect, resource, create table, create view, create any
index TO users;
GRANT unlimited TABLESPACE to users;
```

```
CREATE USER usersTS IDENTIFIED BY usersTSpas;
GRANT connect, resource, create table, create view, create any
index TO usersTS;
GRANT unlimited TABLESPACE to usersTS;
```

```
CREATE USER usersTS2 IDENTIFIED BY usersTS2pass;
GRANT connect, resource, create table, create view, create any
index TO usersTS2;
GRANT unlimited TABLESPACE to usersTS2;
```

```
CREATE USER pgAdmin IDENTIFIED BY pgAdminpass;
GRANT connect, resource, create table, create view, create any
index TO pgAdmin;
GRANT unlimited TABLESPACE to pgAdmin;
```

2. As SYSDBA, configure and enable Oracle Label Security.

```
ALTER USER lbacsys IDENTIFIED BY lbacsys ACCOUNT UNLOCK;
EXEC LBACSYS.CONFIGURE_OLS;
EXEC LBACSYS.OLS_ENFORCEMENT.ENABLE_OLS;
```

3. As SYSTEM, grant privileges to sec_admin and hr_sec.

```
CONNECT system/<system-password>
GRANT connect, create any index to sec_admin IDENTIFIED BY password;
GRANT connect, create user, drop user, create role, drop any role
TO hr_sec IDENTIFIED BY password;
```

4. As LBACSYS, create the security policy.

```
CONNECT lbacsys/<lbacsys-password>

BEGIN
SA_SYSDBA.CREATE_POLICY (
  policy_name => 'DEFENSE',
  column_name => 'SL',
  default_options => 'READ_CONTROL,LABEL_DEFAULT,HIDE');
END;
/
```

5. As LBACSYS , grant DEFENSE_DBA and execute to sec_admin and hr_sec users.

```
GRANT DEFENSE_DBA to sec_admin;
GRANT DEFENSE_DBA to hr_sec;

GRANT execute on SA_COMPONENTS to sec_admin;
GRANT execute on SA_USER_ADMIN to hr_sec;
```

6. As SEC_ADMIN, create three security levels (For simplicity, compartments and groups are omitted here.)

```
CONNECT sec_admin/<sec_admin-password>;

BEGIN
SA_COMPONENTS.CREATE_LEVEL (
  policy_name => 'DEFENSE',
  level_num => 1000,
  short_name => 'PUB',
  long_name => 'PUBLIC');
END;
/
EXECUTE SA_COMPONENTS.CREATE_LEVEL('DEFENSE',2000,'CONF','CONFIDENTIAL');
EXECUTE SA_COMPONENTS.CREATE_LEVEL('DEFENSE',3000,'SENS','SENSITIVE');
```

7. Create three labels.

```
EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('DEFENSE',1000,'PUB');
EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('DEFENSE',2000,'CONF');
EXECUTE SA_LABEL_ADMIN.CREATE_LABEL('DEFENSE',3000,'SENS');
```

8. As HR_SEC, assign labels and privileges.

```
CONNECT hr_sec/<hr_sec-password>;

BEGIN
SA_USER_ADMIN.SET_USER_LABELS (
  policy_name => 'DEFENSE',
  user_name => 'UT',
  max_read_label => 'SENS',
  max_write_label => 'SENS',
  min_write_label => 'CONF',
  def_label => 'SENS',
  row_label => 'SENS');
END;
/

EXECUTE SA_USER_ADMIN.SET_USER_LABELS('DEFENSE', 'userTS', 'SENS');
EXECUTE SA_USER_ADMIN.SET_USER_LABELS('DEFENSE','userTS2','SENS');
EXECUTE SA_USER_ADMIN.SET_USER_LABELS('DEFENSE', 'userS', 'CONF');
EXECUTE SA_USER_ADMIN.SET_USER_LABELS ('DEFENSE', userP, 'PUB', 'PUB',
'PUB', 'PUB', 'PUB');
EXECUTE SA_USER_ADMIN.SET_USER_LABELS ('DEFENSE', 'userP2', 'PUB', 'PUB',
'PUB', 'PUB', 'PUB');
EXECUTE SA_USER_ADMIN.SET_USER_PRIVS ('DEFENSE', 'pgAdmin', 'FULL');
```

9. As SEC_ADMIN, apply the security policies to the desired property graph. Assume a property graph with the name OLSEXAMPLE with userP as the graph owner. To apply OLS security, execute the following statements.

```
CONNECT sec_admin/<password>;

EXECUTE SA_POLICY_ADMIN.APPLY_TABLE_POLICY ('DEFENSE', 'userP',
'OLSEXAMPLEVT$');
EXECUTE SA_POLICY_ADMIN.APPLY_TABLE_POLICY ('DEFENSE', 'userP',
'OLSEXAMPLEGE$');
EXECUTE SA_POLICY_ADMIN.APPLY_TABLE_POLICY ('DEFENSE', 'userP',
'OLSEXAMPLEGT$');
EXECUTE SA_POLICY_ADMIN.APPLY_TABLE_POLICY ('DEFENSE', 'userP',
'OLSEXAMPLESS$');
```

Now Oracle Label Security has sensitivity labels to be associated with individual vertices or edges stored in the property graph.

The following example shows how to create a property graph with name OLSEXAMPLE, and an example flow to demonstrate the behavior when different users with different security labels create, read, and write graph elements.

```
// Create Oracle Property Graph
String graphName = "OLSEXAMPLE";
Oracle connPub = new Oracle("jdbc:oracle:thin:@host:port:SID",
"userP", "userPpass");
OraclePropertyGraph graphPub = OraclePropertyGraph.getInstance(connPub,
graphName, 48);

// Grant access to other users
graphPub.grantAccess("userP2", "RSIUD"); // Read, Select, Insert,
Update, Delete (RSIUD)
graphPub.grantAccess("userS", "RSIUD");
graphPub.grantAccess("userTS", "RSIUD");
graphPub.grantAccess("userTS2", "RSIUD");

// Load data
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
String vfile = "../data/connections.opv";
String efile = "../data/connections.ope";
graphPub.clearRepository();
opgdl.loadData(graphPub, vfile, efile, 48, 1000, true, null);
System.out.println("Vertices with user userP and PUBLIC LABEL: " +
graphPub.countVertices()); // 78
System.out.println("Vertices with user userP and PUBLIC LABEL: " +
graphPub.countEdges()); // 164

// Second user with a higher level
Oracle connTS = new Oracle("jdbc:oracle:thin:@host:port:SID", "userTS",
"userTpass");
OraclePropertyGraph graphTS = OraclePropertyGraph.getInstance(connTS,
"USERP", graphName, 8, 48, null, null);
System.out.println("Vertices with user userTS and SENSITIVE LABEL: " +
graphTS.countVertices()); // 78
```

```
System.out.println("Vertices with user userTS and SENSITIVE LABEL: " +
graphTS.countEdges()); // 164

// Add vertices and edges with the second user
long lMaxVertexID = graphTS.getMaxVertexID();
long lMaxEdgeID = graphTS.getMaxEdgeID();
long size = 10;
System.out.println("\nAdd " + size + " vertices and edges with user userTS
and SENSITIVE LABEL\n");
for (long idx = 1; idx <= size; idx++) {
    Vertex v = graphTS.addVertex(idx + lMaxVertexID);
    v.setProperty("name", "v_" + (idx + lMaxVertexID));
    Edge e = graphTS.addEdge(idx + lMaxEdgeID, v, graphTS.getVertex(idx),
"edge_" + (idx + lMaxEdgeID));
}
graphTS.commit();

// User userP with a lower level only sees the original vertices and edges,
user userTS can see more
System.out.println("Vertices with user userP and PUBLIC LABEL: " +
graphPub.countVertices()); // 78
System.out.println("Vertices with user userP and PUBLIC LABEL: " +
graphPub.countEdges()); // 164
System.out.println("Vertices with user userTS and SENSITIVE LABEL: " +
graphTS.countVertices()); // 88
System.out.println("Vertices with user userTS and SENSITIVE LABEL: " +
graphTS.countEdges()); // 174

// Third user with a higher level
Oracle connTS2 = new Oracle("jdbc:oracle:thin:@host:port:SID", "userTS2",
"userTS2pass");
OraclePropertyGraph graphTS2 = OraclePropertyGraph.getInstance(connTS2,
"USERP", graphName, 8, 48, null, null);
System.out.println("Vertices with user userTS2 and SENSITIVE LABEL: " +
graphTS2.countVertices()); // 88
System.out.println("Vertices with user userTS2 and SENSITIVE LABEL: " +
graphTS2.countEdges()); // 174

// Fourth user with a intermediate level
Oracle connS = new Oracle("jdbc:oracle:thin:@host:port:SID", "userS",
"userSpass");
OraclePropertyGraph graphS = OraclePropertyGraph.getInstance(connS, "USERP",
graphName, 8, 48, null, null);
System.out.println("Vertices with user userS and CONFIDENTIAL LABEL: " +
graphS.countVertices()); // 78
System.out.println("Vertices with user userS and CONFIDENTIAL LABEL: " +
graphS.countEdges()); // 164

// Modify vertices with the fourth user
System.out.println("\nModify " + size + " vertices with user userS and
CONFIDENTIAL LABEL\n");
for (long idx = 1; idx <= size; idx++) {
    Vertex v = graphS.getVertex(idx);
    v.setProperty("security_label", "CONFIDENTIAL");
}
}
```

```

graphS.commit();

// User userP with a lower level that userS cannot see the new vertices
// Users userS and userTS can see them
System.out.println("Vertices with user userP with property
security_label: " +
OraclePropertyGraphUtils.size(graphPub.getVertices("security_label",
"CONFIDENTIAL"))); // 0
System.out.println("Vertices with user userS with property
security_label: " +
OraclePropertyGraphUtils.size(graphS.getVertices("security_label",
"CONFIDENTIAL"))); // 10
System.out.println("Vertices with user userTS with property
security_label: " +
OraclePropertyGraphUtils.size(graphTS.getVertices("security_label",
"CONFIDENTIAL"))); // 10
System.out.println("Vertices with user userP and PUBLIC LABEL: " +
graphPub.countVertices()); // 68
System.out.println("Vertices with user userTS and SENSITIVE LABEL: " +
graphTS.countVertices()); // 88

```

The preceding example should produce the following output.

```

Vertices with user userP and PUBLIC LABEL: 78
Vertices with user userP and PUBLIC LABEL: 164
Vertices with user userTS and SENSITIVE LABEL: 78
Vertices with user userTS and SENSITIVE LABEL: 164

Add 10 vertices and edges with user userTS and SENSITIVE LABEL

Vertices with user userP and PUBLIC LABEL: 78
Vertices with user userP and PUBLIC LABEL: 164
Vertices with user userTS and SENSITIVE LABEL: 88
Vertices with user userTS and SENSITIVE LABEL: 174
Vertices with user userTS2 and SENSITIVE LABEL: 88
Vertices with user userTS2 and SENSITIVE LABEL: 174
Vertices with user userS and CONFIDENTIAL LABEL: 78
Vertices with user userS and CONFIDENTIAL LABEL: 164

Modify 10 vertices with user userS and CONFIDENTIAL LABEL

Vertices with user userP with property security_label: 0
Vertices with user userS with property security_label: 10
Vertices with user userTS with property security_label: 10
Vertices with user userP and PUBLIC LABEL: 68
Vertices with user userTS and SENSITIVE LABEL: 88

```

5.7 SQL-Based Property Graph Query and Analytics

You can use SQL to query property graph data in Oracle Spatial and Graph.

For the property graph support in Oracle Spatial and Graph, all the vertices and edges data are persisted in relational form in Oracle Database. For detailed information about the Oracle Spatial and Graph property graph schema objects, see [Property Graph Schema Objects for Oracle Database](#).

This chapter provides examples of typical graph queries implemented using SQL. The audience includes DBAs as well as application developers who understand SQL syntax and property graph schema objects.

The benefits of querying directly property graph using SQL include:

- There is no need to bring data outside Oracle Database.
- You can leverage the industry-proven SQL engine provided by Oracle Database.
- You can easily join or integrate property graph data with other data types (relational, JSON, XML, and so on).
- You can take advantage of existing Oracle SQL tuning and database management tools and user interface.

The examples assume that there is a property graph named `connections` in the current schema. The SQL queries and example output are for illustration purpose only, and your output may be different depending on the data in your `connections` graph. In some examples, the output is reformatted for readability.

- [Simple Property Graph Queries](#)
The examples in this topic query vertices, edges, and properties of the graph.
- [Text Queries on Property Graphs](#)
If values of a property (vertex property or edge property) contain free text, then it might help performance to create an Oracle Text index on the V column.
- [Navigation and Graph Pattern Matching](#)
A key benefit of using a graph data model is that you can easily navigate across entities (people, movies, products, services, events, and so on) that are modeled as vertices, following links and relationships modeled as edges. In addition, graph matching templates can be defined to do such things as detect patterns, aggregate individuals, and analyze trends.
- [Navigation Options: CONNECT BY and Parallel Recursion](#)
The CONNECT BY clause and parallel recursion provide options for advanced navigation and querying.
- [Pivot](#)
The PIVOT clause lets you dynamically add columns to a table to create a new table.
- [SQL-Based Property Graph Analytics](#)
In addition to the analytical functions offered by the in-memory analyst, the property graph feature in Oracle Spatial and Graph supports several native, SQL-based property graph analytics.

5.7.1 Simple Property Graph Queries

The examples in this topic query vertices, edges, and properties of the graph.

Example 5-1 Find a Vertex with a Specified Vertex ID

This example find the vertex with vertex ID 1 in the `connections` graph.

```
SQL> select vid, k, v, vn, vt
       from connectionsVT$
       where vid=1;
```

The output might be as follows:

```
1 country      United States
1 name         Robert Smith
1 occupation   CEO of Example Corporation
...
```

Example 5-2 Find an Edge with a Specified Edge ID

This example finds the edge with edge ID 100 in the `connections` graph.

```
SQL> select eid,svid,dvid,k,t,v,vn,vt
       from connectionsGE$
       where eid=1000;
```

The output might be as follows:

```
1000 1 2 weight 3 1 1
```

In the preceding output, the K of the edge property is "weight" and the type ID of the value is 3, indicating a float value.

Example 5-3 Perform Simple Counting

This example performs simple counting in the `connections` graph.

```
SQL> -- Get the total number of K/V pairs of all the vertices
SQL> select /*+ parallel */ count(1)
       from connectionsVT$;
```

```
299
```

```
SQL> -- Get the total number of K/V pairs of all the edges
SQL> select /*+ parallel(8) */ count(1)
       from connectionsGE$;
```

```
164
```

```
SQL> -- Get the total number of vertices
SQL> select /*+ parallel */ count(distinct vid)
       from connectionsVT$;
```

```
78
```

```
SQL> -- Get the total number of edges
SQL> select /*+ parallel */ count(distinct eid)
       from connectionsGE$;
```

```
164
```

Example 5-4 Get the Set of Property Keys Used

This example gets the set of property keys used for the vertices in the `connections` graph.

```
SQL> select /*+ parallel */ distinct k
       from connectionsVT$;
```

```
company
```

```
show
occupation
type
team
religion
criminal charge
music genre
genre
name
role
political party
country
```

13 rows selected.

```
SQL> -- get the set of property keys used for edges
SQL> select /*+ parallel */ distinct k
        from connectionsGE$;
```

```
weight
```

Example 5-5 Find Vertices with a Value

This example finds vertices with a value (of any property) that is of String type, and where and the value contains two adjacent occurrences of a, e, i, o, or u, regardless of case. The connections graph.

```
SQL> select vid, t, k, v
        from connectionsVT$
        where t=1
        and regexp_like(v, '([aeiou])\1', 'i');
```

```
6          1 name Jordan Peele
6          1 show Key and Peele
54         1 name John Green
...
```

It is usually hard to leverage a B-Tree index for the preceding kind of query because it is difficult to know beforehand what kind of regular expression is going to be used. For the above query, you might get the following execution plan. Note that full table scan is chosen by the optimizer.

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
Pstart	Pstop	TQ IN-OUT PQ Distrib				
0	SELECT STATEMENT		15	795	28 (0)	00:00:01
1	PX COORDINATOR					
2	PX SEND QC (RANDOM)	:TQ10000	15	795	28 (0)	00:00:01
3	PX BLOCK ITERATOR	Q1,00 P->S QC (RAND)	15	795	28 (0)	00:00:01
1	8	Q1,00 PCWC				

```
-----
```

```
|* 4 |      TABLE ACCESS FULL| CONNECTIONSVT$ |    15 |    795 |    28 | (0)|
00:00:01 |    1 |    8 | Q1,00 | PCWP |
```

```
-----
Predicate Information (identified by operation id):
-----
```

```
  4 - filter(INTERNAL_FUNCTION("V") AND REGEXP_LIKE ("V",U'([aeiou])
\005C1','i') AND "T "=1 AND INTERNAL_FUNCTION("K"))
```

```
Note
```

```
-----
- Degree of Parallelism is 2 because of table property
```

If the Oracle Database In-Memory option is available and memory is sufficient, it can help performance to place the table (full table or a set of relevant columns) in memory. One way to achieve that is as follows:

```
SQL> alter table connectionsVT$ inmemory;
Table altered.
```

Now, entering the same SQL containing the regular expression shows a plan that performs a "TABLE ACCESS INMEMORY FULL".

```
-----
| Id | Operation | Name | Rows | Bytes | Cost
(%CPU) | Time | Pstart| Pstop | TQ | IN-OUT| PQ Distrib |
-----
| 0 | SELECT STATEMENT | | 15 | 795 |
28 (0)| 00:00:01 | | | | |
| 1 | PX COORDINATOR | | | | |
| 2 | PX SEND QC (RANDOM) | :TQ10000 | 15 | 795 |
28 (0)| 00:00:01 | | Q1,00 | P->S | QC (RAND) |
| 3 | PX BLOCK ITERATOR | | 15 | 795 |
28 (0)| 00:00:01 | 1 | 8 | Q1,00 | PCWC |
|* 4 |      TABLE ACCESS INMEMORY FULL| CONNECTIONSVT$ |    15 |    795 |
28 (0)| 00:00:01 |    1 |    8 | Q1,00 | PCWP |
```

```
-----
Predicate Information (identified by operation id):
-----
```

```
  4 - filter(INTERNAL_FUNCTION("V") AND REGEXP_LIKE ("V",U'([aeiou])
\005C1','i') AND "T "=1 AND INTERNAL_FUNCTION("K"))
```

```
Note
```

```
-----
- Degree of Parallelism is 2 because of table property
```

5.7.2 Text Queries on Property Graphs

If values of a property (vertex property or edge property) contain free text, then it might help performance to create an Oracle Text index on the V column.

Oracle Text can process text that is directly stored in the database. The text can be short strings (such as names or addresses), or it can be full-length documents. These documents can be in a variety of textual format.

The text can also be in many different languages. Oracle Text can handle any space-separated languages (including character sets such as Greek or Cyrillic). In addition,

Oracle Text is able to handle the Chinese, Japanese and Korean pictographic languages)

Because the property graph feature uses NVARCHAR typed column for better support of Unicode, it is **highly recommended** that UTF8 (AL32UTF8) be used as the database character set.

To create an Oracle Text index on the vertices table (or edges table), the ALTER SESSION privilege is required. For example:

```
SQL> grant alter session to <YOUR_USER_SCHEMA_HERE>;
```

If customization is required, also grant the EXECUTE privilege on CTX_DDL:

```
SQL> grant execute on ctx_ddl to <YOUR_USER_SCHEMA_HERE>;
```

The following shows some example statements for granting these privileges to SCOTT.

```
SQL> conn / as sysdba
Connected.
SQL> -- This is a PDB setup --
SQL> alter session set container=orcl;
Session altered.

SQL> grant execute on ctx_ddl to scott;
Grant succeeded.

SQL> grant alter session to scott;
Grant succeeded.
```

Example 5-6 Create a Text Index

This example creates an Oracle Text index on the vertices table (V column) of the connections graph in the SCOTT schema. Note that the Oracle Text index created here is for all property keys, not just one or a subset of property keys. In addition, if a new property is added to the graph and the property value is of String data type, then it will automatically be included in the same text index.

The example uses the OPG_AUTO_LEXER lexer owned by MDSYS.

```
SQL> execute opg_apis.create_vertices_text_idx('scott', 'connections',
pref_owner=>'MDSYS', lexer=>'OPG_AUTO_LEXER', dop=>2);
```

If customization is desired, you can use the ctx_ddl.create_preference API. For example:

```
SQL> -- The following requires access privilege to CTX_DDL
SQL> exec ctx_ddl.create_preference('SCOTT.OPG_AUTO_LEXER', 'AUTO_LEXER');

PL/SQL procedure successfully completed.

SQL> execute opg_apis.create_vertices_text_idx('scott', 'connections',
pref_owner=>'scott', lexer=>'OPG_AUTO_LEXER', dop=>2);

PL/SQL procedure successfully completed.
```

You can now use a rich set of functions provided by Oracle Text to perform queries against graph elements.



Note:

If you no longer need an Oracle Text index, you can use the `drop_vertices_text_idx` or `opg_apis.drop_edges_text_idx` API to drop it. The following statements drop the text indexes on the vertices and edges of a graph named `connections` owned by `SCOTT`:

```
SQL> exec opg_apis.drop_vertices_text_idx('scott',
'connections');
SQL> exec opg_apis.drop_edges_text_idx('scott', 'connections');
```

Example 5-7 Find a Vertex that Has a Property Value

The following example find a vertex that has a property value (of string type) containing the keyword "Smith".

```
SQL> select vid, k, t, v
       from connectionsVT$
       where t=1
          and contains(v, 'Smith', 1) > 0
       order by score(1) desc
       ;
```

The output and SQL execution plan from the preceding statement may appear as follows. Note that **DOMAIN INDEX** appears as an operation in the execution plan.

```
1 name 1 Robert Smith
```

Execution Plan

Plan hash value: 1619508090

Id	Operation	Name	Rows	Bytes
Cost (%CPU)	Time	Pstart Pstop		
0	SELECT STATEMENT		1	56
5 (20)	(20) 00:00:01			
1	SORT ORDER BY		1	56
5 (20)	(20) 00:00:01			
* 2	TABLE ACCESS BY GLOBAL INDEX ROWID	CONNECTIONSVT\$	1	56
4	(0) 00:00:01 ROWID ROWID			
* 3	DOMAIN INDEX	CONNECTIONSXTV\$		
4	(0) 00:00:01			

Predicate Information (identified by operation id):

```
2 - filter("T"=1 AND INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
3 - access("CTXSYS"."CONTAINS"("V",'Smith',1)>0)
```

Example 5-8 Fuzzy Match

The following example finds a vertex that has a property value (of string type) containing variants of "ameriian" (a deliberate misspelling for this example) Fuzzy match is used.

```
SQL> select vid, k, t, v
      from connectionsVT$
      where contains(v, 'fuzzy(ameriian,,weight)', 1) > 0
      order by score(1) desc;
```

The output and SQL execution plan from the preceding statement may appear as follows.

```
8 role      1 american business man
9 role      1 american business man
4 role      1 american economist
6 role      1 american comedian actor
7 role      1 american comedian actor
1 occupation 1 44th president of United States of America
```

6 rows selected.

Execution Plan

Plan hash value: 1619508090

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	56	5
1	SORT ORDER BY		1	56	5
2	TABLE ACCESS BY GLOBAL INDEX ROWID	CONNECTIONSVT\$	1	56	
3	DOMAIN INDEX	CONNECTIONSXTV\$			

Predicate Information (identified by operation id):

```
2 - filter(INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
```

Example 5-9 Query Relaxation

The following example is a sophisticated Oracle Text query that implements **query relaxation**, which enables you to execute the most restrictive version of a query first, progressively relaxing the query until the required number of matches is obtained. Using query relaxation with queries that contain multiple strings, you can provide guidance for determining the “best” matches, so that these appear earlier in the results than other potential matches.

This example searches for "american actor" with a query relaxation sequence.

```
SQL> select vid, k, t, v
       from connectionsVT$
       where CONTAINS (v,
'<query>
  <textquery lang="ENGLISH" grammar="CONTEXT">
    <progression>
      <seq>{american} {actor}</seq>
      <seq>{american} NEAR {actor}</seq>
      <seq>{american} AND {actor}</seq>
      <seq>{american} ACCUM {actor}</seq>
    </progression>
  </textquery>
  <score datatype="INTEGER" algorithm="COUNT"/>
</query>') > 0;
```

The output and SQL execution plan from the preceding statement may appear as follows.

```
7 role      1 american comedian actor
6 role      1 american comedian actor
44 occupation 1 actor
8 role      1 american business man
53 occupation 1 actor film producer
52 occupation 1 actor
4 role      1 american economist
47 occupation 1 actor
9 role      1 american business man
```

9 rows selected.

Execution Plan

Plan hash value: 2158361449

```
-----
| Id | Operation | Name | Rows | Bytes | Cost
(%CPU)| Time | Pstart| Pstop |
-----
| 0 | SELECT STATEMENT | | | | 56
| 4 (0)| 00:00:01 | | | |
* | 1 | TABLE ACCESS BY GLOBAL INDEX ROWID | CONNECTIONSVT$ | 1 | 56
| 4 (0)| 00:00:01 | ROWID | ROWID |
* | 2 | DOMAIN INDEX | CONNECTIONSXTV$ | |
| 4 (0)| 00:00:01 | | |
-----
```

Predicate Information (identified by operation id):

```
-----
1 - filter(INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
2 - access("CTXSYS"."CONTAINS"("V",'<query>
grammar="CONTEXT">
  <progression>
    <seq>{american} {actor}</seq>
    <seq>{american}
```



```
NEAR {actor}</seq>
      <seq>{american} AND {actor}</seq>      <seq>{american} ACCUM {actor}</
seq>    </progression>
      </textquery>    <score datatype="INTEGER" algorithm="COUNT"/> </query>')>0)
```

Example 5-10 Find an Edge

Just as with vertices, you can create an Oracle Text index on the V column of the edges table (GE\$) of a property graph. The following example uses the OPG_AUTO_LEXER lexer owned by MDSYS.

```
SQL> exec opg_apis.create_edges_text_idx('scott', 'connections',
pref_owner=>'mdsys', lexer=>'OPG_AUTO_LEXER', dop=>4);
```

If customization is required, use the `ctx_ddl.create_preference` API.

5.7.3 Navigation and Graph Pattern Matching

A key benefit of using a graph data model is that you can easily navigate across entities (people, movies, products, services, events, and so on) that are modeled as vertices, following links and relationships modeled as edges. In addition, graph matching templates can be defined to do such things as detect patterns, aggregate individuals, and analyze trends.

This topic provides graph navigation and pattern matching examples using the example property graph named `connections`. Most of the SQL statements are relatively simple, but they can be used as building blocks to implement requirements that are more sophisticated. It is generally best to start from something simple, and progressively add complexity.

Example 5-11 Who Are a Person's Collaborators?

The following SQL statement finds all entities that a vertex with ID 1 collaborates with. For simplicity, it considers **only** outgoing relationships.

```
SQL> select dvid, el, k, vn, v
       from connectionsGE$
       where svid=1
          and el='collaborates';
```



Note:

To find the specific vertex ID of interest, you can perform a text query on the property graph using keywords or fuzzy matching. (For details and examples, see [Text Queries on Property Graphs](#).)

The preceding example's output and execution plan may be as follows.

```
2 collaborates weight 1 1
21 collaborates weight 1 1
22 collaborates weight 1 1
....
26 collaborates weight 1 1
```

10 rows selected.

```
-----
```

Id	Operation	Name	Rows
Bytes	Cost (%CPU) Time	TQ IN-OUT PQ Distrib	
0	SELECT STATEMENT		10
460	2 (0) 00:00:01		
1	PX COORDINATOR		
2	PX SEND QC (RANDOM)	:TQ10000	10
460	2 (0) 00:00:01	Q1,00 P->S QC (RAND)	
3	PX PARTITION HASH ALL		10
460	2 (0) 00:00:01	1 8 Q1,00 PCWC	
* 4	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	CONNECTIONSGE\$	10
460	2 (0) 00:00:01	1 8 Q1,00 PCWP	
* 5	INDEX RANGE SCAN	CONNECTIONSXSE\$	20
	1 (0) 00:00:01	1 8 Q1,00 PCWP	

```
-----
```

Predicate Information (identified by operation id):

```
-----
```

4 - filter(INTERNAL_FUNCTION("EL") AND "EL"=U'collaborates' AND
INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
5 - access("SVID"=1)

Example 5-12 Who Are a Person's Collaborators and What are Their Occupations?

The following SQL statement finds collaborators of the vertex with ID 1, and the occupation of each collaborator. A join with the vertices table (VT\$) is required.

```
SQL> select dvid, vertices.v
       from connectionsGE$, connectionsVT$ vertices
       where svid=1
          and el='collaborates'
          and dvid=vertices.vid
          and vertices.k='occupation';
```

The preceding example's output and execution plan may be as follows.

```
21 67th United States Secretary of State
22 68th United States Secretary of State
23 chancellor
28 7th president of Iran
19 junior United States Senator from New York
...
```

```
-----
```

Id	Operation	Name	Rows
Bytes	Cost (%CPU) Time	TQ IN-OUT PQ Distrib	

```
-----
```

```

-----
| 0 | SELECT STATEMENT | | | | | | | | | 7 |
525 | 7 (0) | 00:00:01 | | | | | | | | |
| 1 | PX COORDINATOR | | | | | | | | |
| 2 | PX SEND QC (RANDOM) | | | | | | | :TQ10000 | 7 |
525 | 7 (0) | 00:00:01 | | | Q1,00 | P->S | QC (RAND) | |
| 3 | NESTED LOOPS | | | | | | | | | 7 |
525 | 7 (0) | 00:00:01 | | | Q1,00 | PCWP | | |
| 4 | PX PARTITION HASH ALL | | | | | | | | | 10 |
250 | 2 (0) | 00:00:01 | 1 | 8 | Q1,00 | PCWC | | |
|* 5 | TABLE ACCESS BY LOCAL INDEX ROWID BATCHED | CONNECTIONSGE$ | | 10 |
250 | 2 (0) | 00:00:01 | 1 | 8 | Q1,00 | PCWP | | |
|* 6 | INDEX RANGE SCAN | CONNECTIONSXSE$ | | 20 | | | | | |
| 1 (0) | 00:00:01 | 1 | 8 | Q1,00 | PCWP | | |
| 7 | PARTITION HASH ITERATOR | | | | | | | | 1 |
| 0 (0) | 00:00:01 | KEY | KEY | Q1,00 | PCWP | | |
|* 8 | TABLE ACCESS BY LOCAL INDEX ROWID | CONNECTIONSVT$ | | |
| | | KEY | KEY | Q1,00 | PCWP | | |
|* 9 | INDEX UNIQUE SCAN | CONNECTIONSXQV$ | | 1 |
| 0 (0) | 00:00:01 | KEY | KEY | Q1,00 | PCWP | | |
-----

```

Predicate Information (identified by operation id):

- ```

5 - filter(INTERNAL_FUNCTION("EL") AND "EL"=U'collaborates')
6 - access("SVID"=1)
8 - filter(INTERNAL_FUNCTION("VERTICES"."V"))
9 - access("DVID"="VERTICES"."VID" AND "VERTICES"."K"=U'occupation')
 filter(INTERNAL_FUNCTION("VERTICES"."K"))

```

### Example 5-13 Find a Person's Enemies and Aggregate Them by Their Country

The following SQL statement finds enemies (that is, those with the feuds relationship) of the vertex with ID 1, and aggregates them by their countries. A join with the vertices table (VT\$) is required.

```

SQL> select vertices.v, count(1)
 from connectionsGE$, connectionsVT$ vertices
 where svid=1
 and el='feuds'
 and dvid=vertices.vid
 and vertices.k='country'
 group by vertices.v;

```

The example's output and execution plan may be as follows. In this case, the vertex with ID 1 has 3 enemies in the United States and 1 in Russia.

```

United States 3
Russia 1

```

```

| Id | Operation | Name | Rows |
Bytes | Cost (%CPU)| Time | Pstart| Pstop | TQ | IN-OUT| PQ Distrib |


```

|      |                                           |          |     |     |  |  |  |                          |  |    |
|------|-------------------------------------------|----------|-----|-----|--|--|--|--------------------------|--|----|
| 0    | SELECT STATEMENT                          |          |     |     |  |  |  |                          |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  |                          |  |    |
| 1    | PX COORDINATOR                            |          |     |     |  |  |  |                          |  |    |
| 2    | PX SEND QC (RANDOM)                       |          |     |     |  |  |  | :TQ10001                 |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  | Q1,01   P->S   QC (RAND) |  |    |
| 3    | HASH GROUP BY                             |          |     |     |  |  |  |                          |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  | Q1,01   PCWP             |  |    |
| 4    | PX RECEIVE                                |          |     |     |  |  |  |                          |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  | Q1,01   PCWP             |  |    |
| 5    | PX SEND HASH                              |          |     |     |  |  |  | :TQ10000                 |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  | Q1,00   P->P   HASH      |  |    |
| 6    | HASH GROUP BY                             |          |     |     |  |  |  |                          |  | 5  |
| 375  | 5 (20)                                    | 00:00:01 |     |     |  |  |  | Q1,00   PCWP             |  |    |
| 7    | NESTED LOOPS                              |          |     |     |  |  |  |                          |  | 5  |
| 375  | 4 (0)                                     | 00:00:01 |     |     |  |  |  | Q1,00   PCWP             |  |    |
| 8    | PX PARTITION HASH ALL                     |          |     |     |  |  |  |                          |  | 5  |
| 125  | 2 (0)                                     | 00:00:01 | 1   | 8   |  |  |  | Q1,00   PCWC             |  |    |
| * 9  | TABLE ACCESS BY LOCAL INDEX ROWID BATCHED |          | 1   | 8   |  |  |  | CONNECTIONSGE\$          |  | 5  |
| 125  | 2 (0)                                     | 00:00:01 |     |     |  |  |  | Q1,00   PCWP             |  |    |
| * 10 | INDEX RANGE SCAN                          |          |     |     |  |  |  | CONNECTIONSXSE\$         |  | 20 |
|      | 1 (0)                                     | 00:00:01 | 1   | 8   |  |  |  | Q1,00   PCWP             |  |    |
| 11   | PARTITION HASH ITERATOR                   |          |     |     |  |  |  |                          |  | 1  |
|      | 0 (0)                                     | 00:00:01 | KEY | KEY |  |  |  | Q1,00   PCWP             |  |    |
| * 12 | TABLE ACCESS BY LOCAL INDEX ROWID         |          |     |     |  |  |  | CONNECTIONSVT\$          |  |    |
|      |                                           |          | KEY | KEY |  |  |  | Q1,00   PCWP             |  |    |
| * 13 | INDEX UNIQUE SCAN                         |          |     |     |  |  |  | CONNECTIONSXQV\$         |  | 1  |
|      | 0 (0)                                     | 00:00:01 | KEY | KEY |  |  |  | Q1,00   PCWP             |  |    |

Predicate Information (identified by operation id):

```

9 - filter(INTERNAL_FUNCTION("EL") AND "EL"=U'feuds')
10 - access("SVID"=1)
12 - filter(INTERNAL_FUNCTION("VERTICES"."V"))
13 - access("DVID"="VERTICES"."VID" AND "VERTICES"."K"=U'country')
 filter(INTERNAL_FUNCTION("VERTICES"."K"))

```

### Example 5-14 Find a Person's Collaborators, and aggregate and sort them

The following SQL statement finds the collaborators of the vertex with ID 1, aggregates them by their country, and sorts them in ascending order.

```

SQL> select vertices.v, count(1)
 from connectionsGE$, connectionsVT$ vertices
 where svid=1
 and el='collaborates'
 and dvid=vertices.vid
 and vertices.k='country'
 group by vertices.v
 order by count(1) asc;

```

The example output and execution plan may be as follows. In this case, the vertex with ID 1 has the most collaborators in the United States.

```
Germany 1
Japan 1
Iran 1
United States 7
```

```

Id	Operation	Name	Rows					
Id	Operation	Name	Rows					
----	-----	-----	-----	-----	-----	-----		
0	SELECT STATEMENT		10					
750	9 (23)	00:00:01						
1	PX COORDINATOR							
2	PX SEND QC (ORDER)	:TQ10002	10					
750	9 (23)	00:00:01			Q1,02	P->S	QC (ORDER)	
3	SORT ORDER BY		10					
750	9 (23)	00:00:01			Q1,02	PCWP		
4	PX RECEIVE		10					
750	9 (23)	00:00:01			Q1,02	PCWP		
5	PX SEND RANGE	:TQ10001	10					
750	9 (23)	00:00:01			Q1,01	P->P	RANGE	
6	HASH GROUP BY		10					
750	9 (23)	00:00:01			Q1,01	PCWP		
7	PX RECEIVE		10					
750	9 (23)	00:00:01			Q1,01	PCWP		
8	PX SEND HASH	:TQ10000	10					
750	9 (23)	00:00:01			Q1,00	P->P	HASH	
9	HASH GROUP BY		10					
750	9 (23)	00:00:01			Q1,00	PCWP		
10	NESTED LOOPS		10					
750	7 (0)	00:00:01			Q1,00	PCWP		
11	PX PARTITION HASH ALL		10					
250	2 (0)	00:00:01		1	8	Q1,00	PCWC	
* 12	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	CONNECTIONSGE$	10					
250	2 (0)	00:00:01		1	8	Q1,00	PCWP	
* 13	INDEX RANGE SCAN	CONNECTIONSXSE$	20					
1	1 (0)	00:00:01		1	8	Q1,00	PCWP	
14	PARTITION HASH ITERATOR		1					
0	0 (0)	00:00:01	KEY	KEY	Q1,00	PCWP		
* 15	TABLE ACCESS BY LOCAL INDEX ROWID	CONNECTIONSVT$						
0	0 (0)	00:00:01	KEY	KEY	Q1,00	PCWP		
* 16	INDEX UNIQUE SCAN	CONNECTIONSXQV$	1					
0	0 (0)	00:00:01	KEY	KEY	Q1,00	PCWP		

```

Predicate Information (identified by operation id):

```

12 - filter(INTERNAL_FUNCTION("EL") AND "EL"=U'collaborates')
13 - access("SVID"=1)
15 - filter(INTERNAL_FUNCTION("VERTICES"."V"))
16 - access("DVID"="VERTICES"."VID" AND "VERTICES"."K"=U'country')
 filter(INTERNAL_FUNCTION("VERTICES"."K"))
```

## 5.7.4 Navigation Options: CONNECT BY and Parallel Recursion

The CONNECT BY clause and parallel recursion provide options for advanced navigation and querying.

- CONNECT BY lets you navigate and find matches in a hierarchical order. To follow outgoing edges, you can use prior dvid = svid to guide the navigation.
- Parallel recursion lets you perform navigation up to a specified number of hops away.

The examples use a property graph named connections.

### Example 5-15 CONNECT WITH

The following SQL statement follows the outgoing edges by 1 hop.

```
SQL> select G.dvid
 from connectionsGE$ G
 start with svid = 1
 connect by nocycle prior dvid = svid and level <= 1;
```

The preceding example's output and execution plan may be as follows.

```

2
3
4
5
6
7
8
9
10
...

| Id | Operation | Name | Rows | Bytes | Cost
(%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		7	273	3	
(67)	00:00:01					
* 1	CONNECT BY WITH FILTERING					
2	PX COORDINATOR					
3	PX SEND QC (RANDOM)	:TQ10000	2	12	0	
(0)	00:00:01			Q1,00	P->S	QC (RAND)
4	PX PARTITION HASH ALL		2	12	0	
(0)	00:00:01	1	8	Q1,00	PCWC	
* 5	INDEX RANGE SCAN	CONNECTIONSXSE$	2	12	0	
(0)	00:00:01	1	8	Q1,00	PCWP	
* 6	FILTER					
7	NESTED LOOPS		5	95	1	
(0)	00:00:01					
8	CONNECT BY PUMP					
9	PARTITION HASH ALL		2	12	0	

```

```
(0)| 00:00:01 | 1 | 8 | | | | |
|* 10 | INDEX RANGE SCAN | CONNECTIONSXSE$ | 2 | 12 | 0 (0)|
00:00:01 | 1 | 8 | | | | |
```

Predicate Information (identified by operation id):

- 1 - access("SVID"=PRIOR "DVID")
  - filter(LEVEL<=2)
- 5 - access("SVID"=1)
- 6 - filter(LEVEL<=2)
- 10 - access("connect\$\_by\$\_pump\$\_002"."prior dvid "="SVID")

To extend from 1 hop to multiple hops, change 1 in the preceding example to another integer. For example, to change it to 2 hops, specify: `level <= 2`

### Example 5-16 Parallel Recursion

The following SQL statement uses recursion within the WITH clause to perform navigation up to 4 hops away, a using recursively defined graph expansion: `g_exp` references `g_exp` in the query, and that defines the recursion. The example also uses the `PARALLEL` optimizer hint for parallel execution.

```
SQL> WITH g_exp(svid, dvid, depth) as
(
 select svid as svid, dvid as dvid, 0 as depth
 from connectionsGE$
 where svid=1
 union all
 select g2.svid, g1.dvid, g2.depth + 1
 from g_exp g2, connectionsGE$ g1
 where g2.dvid=g1.svid
 and g2.depth <= 3
)
select /*+ parallel(4) */ dvid, depth
 from g_exp
 where svid=1
;
```

The example's output and execution plan may be as follows. Note that `CURSOR DURATION MEMORY` is chosen in the execution, which indicates the graph expansion stores the intermediate data in memory.

```
22 4
25 4
24 4
1 4

23 4
33 4
22 4
22 4
... ...
```

Execution Plan

```


| Id | Operation
Name | Rows | Bytes | Cost (%CPU)| Time | Pstart|
Pstop | TQ | IN-OUT| PQ Distrib |

| 0 | SELECT STATEMENT
| | | | | | |
| | | | | | |
| 1 | TEMP TABLE TRANSFORMATION
| | | | | | |
| 2 | LOAD AS SELECT (CURSOR DURATION MEMORY)
SYS_TEMP_0FD9D6614_11CB2D2 | | | | | |
| 3 | UNION ALL (RECURSIVE WITH) BREADTH FIRST
| | | | | | |
| 4 | PX COORDINATOR
| | | | | | |
| 5 | PX SEND QC (RANDOM)
:TQ20000 | 2 | 12 | 0 (0)| 00:00:01 |
| Q2,00 | P->S | QC (RAND) |
| 6 | LOAD AS SELECT (CURSOR DURATION MEMORY)
SYS_TEMP_0FD9D6614_11CB2D2 | | | | | |
| Q2,00 | PCWP |
| 7 | PX PARTITION HASH ALL
| | 2 | 12 | 0 (0)| 00:00:01 | 1
| 8 | Q2,00 | PCWC |
|* 8 | INDEX RANGE SCAN
CONNECTIONSXSE$ | 2 | 12 | 0 (0)| 00:00:01 | 1 |
8 | Q2,00 | PCWP |
| 9 | PX COORDINATOR
| | | | | | |
| 10 | PX SEND QC (RANDOM)
:TQ10000 | 799 | 12M | 12 (0)| 00:00:01 |
| Q1,00 | P->S | QC (RAND) |
| 11 | LOAD AS SELECT (CURSOR DURATION MEMORY)
SYS_TEMP_0FD9D6614_11CB2D2 | | | | | |
| Q1,00 | PCWP |
|* 12 | HASH JOIN
| | 799 | 12M | 12 (0)| 00:00:01 |
| Q1,00 | PCWP |
| 13 | BUFFER SORT (REUSE)
| | | | | | |
| 14 | Q1,00 | PCWP |
| 14 | PARTITION HASH ALL
| | 164 | 984 | 2 (0)| 00:00:01 | 1
| 8 | Q1,00 | PCWC |
| 15 | INDEX FAST FULL SCAN
CONNECTIONSXDE$ | 164 | 984 | 2 (0)| 00:00:01 | 1 |
8 | Q1,00 | PCWP |
| 16 | PX BLOCK ITERATOR
| | | | | | |
| 17 | Q1,00 | PCWC |
|* 17 | TABLE ACCESS FULL

```



```

SYS_TEMP_0FD9D6614_11CB2D2 | | | | | | |
Q1,00 | PCWP | | | | | |
| 18 | PX COORDINATOR | | | | | |
| 19 | PX SEND QC (RANDOM) | | | | | :TQ30000 |
801 | 31239 | 135 (0) | 00:00:01 | | | Q3,00 | P->S | QC (RAND) |
| * 20 | VIEW | | | | | |
801 | 31239 | 135 (0) | 00:00:01 | | | Q3,00 | PCWP | |
| 21 | PX BLOCK ITERATOR | | | | | |
801 | 12M | 135 (0) | 00:00:01 | | | Q3,00 | PCWC | |
| 22 | TABLE ACCESS FULL | | | | | SYS_TEMP_0FD9D6614_11CB2D2 |
801 | 12M | 135 (0) | 00:00:01 | | | Q3,00 | PCWP | |

```

Predicate Information (identified by operation id):

- ```

-----
8 - access("SVID"=1)
12 - access("G2"."DVID"="G1"."SVID")
17 - filter("G2"."INTERNAL_ITERS$"=LEVEL AND "G2"."DEPTH"<=3)
20 - filter("SVID"=1)

```

5.7.5 Pivot

The PIVOT clause lets you dynamically add columns to a table to create a new table.

The schema design (VT\$ and GE\$) of the property graph is narrow ("skinny") rather than wide ("fat"). This means that if a vertex or edge has multiple properties, those property keys, values, data types, and so on will be stored using multiple rows instead of multiple columns. Such a design is very flexible in the sense that you can add properties dynamically without having to worry about adding too many columns or even reaching the physical maximum limit of number of columns a table may have. However, for some applications you may prefer to have a wide table if the properties are somewhat homogeneous.

Example 5-17 Pivot

The following CREATE TABLE ... AS SELECT statement uses PIVOT to add four columns: 'company', 'occupation', 'name', and 'religion'.

```

SQL> CREATE TABLE table pg_wide
as
with G AS (select vid, k, t, v
           from connectionsVT$
           )
select *
from G
pivot (
min(v) for k in ('company', 'occupation', 'name', 'religion')
);

```

Table created.

The following DESCRIBE statement shows the definition of the new table, including the four added columns. (The output is reformatted for readability.)

```
SQL> DESCRIBE pg_wide;
Name                                                    Null?    Type
-----
VID                                                    NOT NULL NUMBER
T
NUMBER(38)
'company'
NVARCHAR2(15000)
'occupation'
NVARCHAR2(15000)
'name'
NVARCHAR2(15000)
'religion'
NVARCHAR2(15000)
```

5.7.6 SQL-Based Property Graph Analytics

In addition to the analytical functions offered by the in-memory analyst, the property graph feature in Oracle Spatial and Graph supports several native, SQL-based property graph analytics.

The benefits of SQL-based analytics are:

- Easier analysis of larger graphs that do not fit in physical memory
- Cheaper analysis since no graph data is transferred outside the database
- Better analysis using the current state of a property graph database
- Simpler analysis by eliminating the step of synchronizing an in-memory graph with the latest updates from the graph database

However, when a graph (or a subgraph) fits in memory, then running analytics provided by the in-memory analyst usually provides better performance than using SQL-based analytics.

Because many of the analytics implementation require using intermediate data structures, most SQL- (and PL/SQL-) based analytics APIs have parameters for working tables (wt). A typical flow has the following steps:

1. Prepare the working table or tables.
2. Perform analytics (one or multiple calls).
3. Perform cleanup

The following subtopics provide SQL-based examples of some popular types of property graph analytics.

- [Shortest Path Examples](#)
- [Collaborative Filtering Overview and Examples](#)

5.7.6.1 Shortest Path Examples

The following examples demonstrate SQL-based shortest path analytics.

Example 5-18 Shortest Path Setup and Computation

Consider shortest path, for example. Internally, Oracle Database uses the bidirectional Dijkstra algorithm. The following code snippet shows an entire prepare, perform, and cleanup workflow.

```

set serveroutput on

DECLARE
  wt1 varchar2(100); -- intermediate working tables
  n number;
  path  varchar2(1000);
  weights varchar2(1000);
BEGIN
  -- prepare
  opg_apis.find_sp_prep('connectionsGE$', wt1);
  dbms_output.put_line('working table name   ' || wt1);

  -- compute
  opg_apis.find_sp(
    'connectionsGE$',
    1,                -- start vertex ID
    53,               -- destination vertex ID
    wt1,              -- working table (for Dijkstra expansion)
    dop => 1,         -- degree of parallelism
    stats_freq=>1000, -- frequency to collect statistics
    path_output => path, -- shortest path (a sequence of vertices)
    weights_output => weights, -- edge weights
    options => null
  );
  dbms_output.put_line('path   ' || path);
  dbms_output.put_line('weights ' || weights);

  -- cleanup (commented out here; see text after the example)
  -- opg_apis.find_sp_cleanup('connectionsGE$', wt1);
END;
/

```

This example may produce the following output. Note that if **no** working table name is provided, the preparation step will automatically generate a temporary table name and create it. Because the temporary working table name uses the session ID, your output will probably be different.

```

working table name   "CONNECTIONSGE$$TWFS12"
path   1 3   52 53
weights 4 3 1   1 1

```

PL/SQL procedure successfully completed.

If you want to know the definition of the working table or tables, then skip the cleanup phase (as shown in the preceding example that comments out the call to `find_sp_cleanup`). After the computation is done, you can describe the working table or tables.

```
SQL> describe "CONNECTIONSGE$$TWFS12"
Name                Null?    Type
-----
NID                  NUMBER
D2S                  NUMBER
P2S                  NUMBER
D2T                  NUMBER
P2T                  NUMBER
F                    NUMBER (38)
B                    NUMBER (38)
```

For advanced users who want to try different table creation options, such as using in-memory or advanced compression, you can pre-create the preceding working table and pass the name in.

Example 5-19 Shortest Path: Create Working Table and Perform Analytics

The following statements show some advanced options, first creating a working table with the same column structure and basic compression enabled, then passing it to the SQL-based computation. The code optimizes the intermediate table for computations with CREATE TABLE compression and in-memory options.

```
create table connections$MY_EXP(
  NID                NUMBER,
  D2S                NUMBER,
  P2S                NUMBER,
  D2T                NUMBER,
  P2T                NUMBER,
  F                  NUMBER (38),
  B                  NUMBER (38)
) compress nologging;

DECLARE
  wt1 varchar2(100) := 'connections$MY_EXP';
  n number;
  path varchar2(1000);
  weights varchar2(1000);
BEGIN
  dbms_output.put_line('working table name    ' || wt1);

  -- compute
  opg_apis.find_sp(
    'connectionsGE$',
    1,
    53,
    wt1,
    dop => 1,
    stats_freq=>1000,
    path_output => path,
```

```

        weights_output => weights,
        options => null
    );
    dbms_output.put_line('path      ' || path);
    dbms_output.put_line('weights  ' || weights);

    -- cleanup
    -- opg_apis.find_sp_cleanup('connectionsGE$', wt1);
END;
/

```

At the end of the computation, if the working table has not been dropped or truncated, you can check the content of the working table, as follows. Note that the working table structure may vary between releases.

```

SQL> select * from connections$MY_EXP;

```

NID	D2S	P2S	D2T	P2T	F	B
1	0	1.000E+100			1	-1
53	1.000E+100		0		-1	1
54	1.000E+100		1	53	-1	1
52	1.000E+100		1	53	-1	1
5	1	1.000E+100			0	-1
26	1	1.000E+100			0	-1
8	1000	1.000E+100			0	-1
3	1	1	2	52	0	0
15	1	1.000E+100			0	-1
21	1	1.000E+100			0	-1
19	1	1.000E+100			0	-1
...						

Example 5-20 Shortest Path: Perform Multiple Calls to Same Graph

To perform multiple calls to the same graph, only a *single call* to the preparation step is needed. The following shows an example of computing shortest path for multiple pairs of vertices in the same graph.

```

DECLARE
    wt1 varchar2(100); -- intermediate working tables
    n number;
    path varchar2(1000);
    weights varchar2(1000);
BEGIN
    -- prepare
    opg_apis.find_sp_prep('connectionsGE$', wt1);
    dbms_output.put_line('working table name      ' || wt1);

    -- find shortest path from vertex 1 to vertex 53
    opg_apis.find_sp( 'connectionsGE$', 1, 53,
        wt1, dop => 1, stats_freq=>1000, path_output => path, weights_output
=> weights, options => null);
    dbms_output.put_line('path      ' || path);
    dbms_output.put_line('weights  ' || weights);

```

```

-- find shortest path from vertex 2 to vertex 36
opg_apis.find_sp( 'connectionsGE$', 2, 36,
  wt1, dop => 1, stats_freq=>1000, path_output => path,
weights_output => weights, options => null);
dbms_output.put_line('path      ' || path);
dbms_output.put_line('weights  ' || weights);

-- find shortest path from vertex 30 to vertex 4
opg_apis.find_sp( 'connectionsGE$', 30, 4,
  wt1, dop => 1, stats_freq=>1000, path_output => path,
weights_output => weights, options => null);
dbms_output.put_line('path      ' || path);
dbms_output.put_line('weights  ' || weights);

-- cleanup
opg_apis.find_sp_cleanup('connectionsGE$', wt1);
END;
/

```

The example's output may be as follows: three shortest paths have been found for the multiple pairs of vertices provided.

```

working table name      "CONNECTIONSGE$TWFS12"
path      1 3      52 53
weights 4 3 1      1 1
path      2      36
weights 2 1      1
path      30 21      1 4
weights 4 3 1      1 1

```

PL/SQL procedure successfully completed.

5.7.6.2 Collaborative Filtering Overview and Examples

[Collaborative filtering](#), also referred to as social filtering, filters information by using the recommendations of other people. Collaborative filtering is widely used in systems that recommend purchases based on purchases by others with similar preferences.

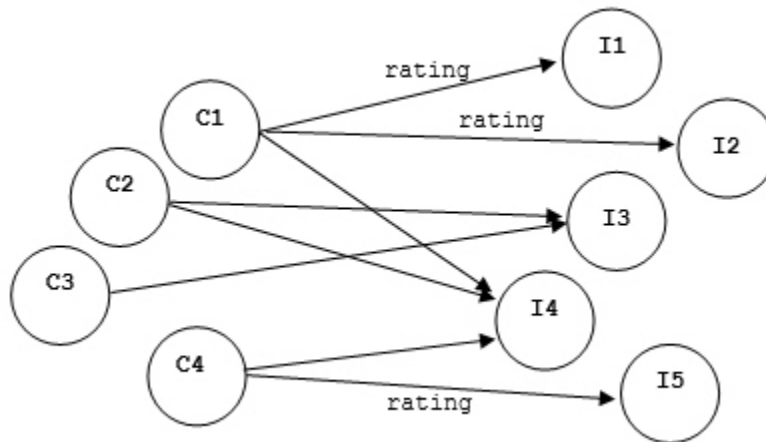
The following examples demonstrate SQL-based collaborative filtering analytics.

Example 5-21 Collaborative Filtering Setup and Computation

This example shows how to use SQL-based collaborative filtering, specifically using matrix factorization to recommend telephone brands to customers. This example assumes there exists a graph called "PHONES" in the database. This example graph contains customer and item vertices, and edges with a 'rating' label linking some customer vertices to other some item vertices. The rating labels have a numeric value corresponding to the rating that a specific customer (edge OUT vertex) assigned to the specified product (edge IN vertex).

The following figure shows this graph.

Figure 5-1 Phones Graph for Collaborative Filtering



```
set serveroutput on
```

```

DECLARE
  wt_l varchar2(32); -- working tables
  wt_r varchar2(32);
  wt_l1 varchar2(32);
  wt_r1 varchar2(32);
  wt_i varchar2(32);
  wt_ld varchar2(32);
  wt_rd varchar2(32);
  edge_tab_name  varchar2(32) := 'phonesge$';
  edge_label     varchar2(32) := 'rating';
  rating_property varchar2(32) := '';
  iterations     integer      := 100;
  min_error      number       := 0.001;
  k              integer      := 5;
  learning_rate  number       := 0.001;
  decrease_rate  number       := 0.95;
  regularization number       := 0.02;
  dop            number       := 2;
  tablespace     varchar2(32) := null;
  options        varchar2(32) := null;
BEGIN

  -- prepare
  opg_apis.cf_prep(edge_tab_name,wt_l,wt_r,wt_l1,wt_r1,wt_i,wt_ld,wt_rd);
  dbms_output.put_line('working table wt_l ' || wt_l);
  dbms_output.put_line('working table wt_r ' || wt_r);
  dbms_output.put_line('working table wt_l1 ' || wt_l1);
  dbms_output.put_line('working table wt_r1 ' || wt_r1);
  dbms_output.put_line('working table wt_i ' || wt_i);
  dbms_output.put_line('working table wt_ld ' || wt_ld);
  dbms_output.put_line('working table wt_rd ' || wt_rd);

  -- compute
  opg_apis.cf(edge_tab_name,edge_label,rating_property,iterations,

```

```

min_error,k,learning_rate,decrease_rate,regularization,dop,
wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd,tablespace,options);
END;
/

```

no

```

working table wt_l      "PHONESGE$$CFL57"
working table wt_r      "PHONESGE$$CFR57"
working table wt_ll     "PHONESGE$$CFL157"
working table wt_rl     "PHONESGE$$CFR157"
working table wt_i      "PHONESGE$$CFI57"
working table wt_ld     "PHONESGE$$CFLD57"
working table wt_rd     "PHONESGE$$CFRD57"

```

PL/SQL procedure successfully completed.

Example 5-22 Collaborative Filtering: Validating the Intermediate Error

At the end of every computation, you can check the current error of the algorithm with the following query as long as the data in the working tables has not been already deleted. The following SQL query illustrates how to get the intermediate error of a current run of the collaborative filtering algorithm.

```

SELECT /*+ parallel(48) */ SQRT(SUM((w1-w2)*(w1-w2) +
      <regularization>/2 * (err_reg_l+err_reg_r))) AS err
FROM <wt_i>;

```

Note that the regularization parameter and the working table name (parameter `wt_i`) should be replaced according to the values used when running the [OPG_APIS.CF](#) algorithm. In the preceding previous example, replace `<regularization>` with 0.02 and `<wt_i>` with "PHONESGE\$\$CFI149" as follows:

```

SELECT /*+ parallel(48) */ SQRT(SUM((w1-w2)*(w1-w2) + 0.02/2 *
      (err_reg_l+err_reg_r))) AS err
FROM "PHONESGE$$CFI149";

```

This query may produce the following output.

```

      ERR
-----
4.82163662

```

If the value of the current error is too high or if the predictions obtained from the matrix factorization results of the collaborative filtering are not yet useful, you can run more iterations of the algorithm, by reusing the working tables and the progress made so far. The following example shows how to make predictions using the SQL-based collaborative filtering.

Example 5-23 Collaborative Filtering: Making Predictions

The result of the collaborative filtering algorithm is stored in the tables `wt_l` and `wt_r`, which are the two factors of a matrix product. These matrix factors should be used when making the predictions of the collaborative filtering.

In a typical flow of the algorithm, the two matrix factors can be used to make the predictions before calling the `OPG_APIS.CF_CLEANUP` procedure, or they can be copied and persisted into other tables for later use. The following example demonstrates the latter case:

```

DECLARE
  wt_l varchar2(32); -- working tables
  wt_r varchar2(32);
  wt_ll varchar2(32);
  wt_rl varchar2(32);
  wt_i varchar2(32);
  wt_ld varchar2(32);
  wt_rd varchar2(32);
  edge_tab_name  varchar2(32) := 'phonesge$';
  edge_label     varchar2(32) := 'rating';
  rating_property varchar2(32) := '';
  iterations     integer      := 100;
  min_error      number       := 0.001;
  k              integer      := 5;
  learning_rate  number       := 0.001;
  decrease_rate  number       := 0.95;
  regularization number       := 0.02;
  dop           number       := 2;
  tablespace    varchar2(32) := null;
  options       varchar2(32) := null;
BEGIN

  -- prepare
  opg_apis.cf_prep(edge_tab_name,wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd);

  -- compute
  opg_apis.cf(edge_tab_name,edge_label,rating_property,iterations,
             min_error,k,learning_rate,decrease_rate,regularization,dop,
             wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd,tablespace,options);

  -- save only these two tables for later predictions
  EXECUTE IMMEDIATE 'CREATE TABLE customer_mat AS SELECT * FROM ' || wt_l;
  EXECUTE IMMEDIATE 'CREATE TABLE item_mat AS SELECT * FROM ' || wt_r;

  -- cleanup
  opg_apis.cf_cleanup('phonesge$',wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd);
END;
/

```

This example will produce the only the following output.

```
PL/SQL procedure successfully completed.
```

Now that the matrix factors are saved in the tables `customer_mat` and `item_mat`, you can use the following query to check the "error" (difference) between the real values (those values that previously existed in the graph as 'ratings') and the estimated predictions (the result of the matrix multiplication in a certain customer row and item column).

Note that the following query is customized with a join on the vertex table in order return an NVARCHAR property of the vertices (for example, the name property) instead of a numeric

ID. This query will return all the predictions for every single customer vertex to every item vertex in the graph.

```
SELECT /*+ parallel(48) */ MIN(vertex1.v) AS customer,
                           MIN(vertex2.v) AS item,
                           MIN(edges.vn) AS real,
                           SUM(l.v * r.v) AS predicted
FROM PHONESGE$ edges,
     CUSTOMER_MAT l,
     ITEM_MAT r,
     PHONESVT$ vertex1,
     PHONESVT$ vertex2
WHERE l.k = r.k
     AND l.c = edges.svid(+)
     AND r.p = edges.dvid(+)
     AND l.c = vertex1.vid
     AND r.p = vertex2.vid
GROUP BY l.c, r.p
ORDER BY l.c, r.p -- This order by clause is optional
;
```

This query may produce an output similar to the following (some rows are omitted for brevity).

CUSTOMER	ITEM	REAL	PREDICTED
Adam	Apple	5	3.67375703
Adam	Blackberry		3.66079652
Adam	Danger		2.77049596
Adam	Ericsson	4	2.1764858
Adam	Figo		3.10631337
Adam	Google	4	4.42429022
Adam	Huawei	3	3.4289115
Ben	Apple		2.82127589
Ben	Blackberry	2	2.81132282
Ben	Danger	3	2.12761307
Ben	Ericsson	3	3.2389595
Ben	Figo		2.38550534
Ben	Google		3.39765075
Ben	Huawei		2.63324582
...			
Don	Apple		1.3777496
Don	Blackberry	1	1.37288909
Don	Danger	1	1.03900439
Don	Ericsson		1.58172236
Don	Figo	1	1.16494421
Don	Google		1.65921807
Don	Huawei	1	1.28592648
Erik	Apple	3	2.80809351
Erik	Blackberry	3	2.79818695
Erik	Danger		2.11767182
Erik	Ericsson	3	3.2238255
Erik	Figo		2.3743591
Erik	Google	3	3.38177526
Erik	Huawei	3	2.62094201

If you want to check only some rows to decide whether the prediction results are ready or more iterations of the algorithm should be run, the previous query can be wrapped in an outer query. The following example will select only the first 11 results.

```
SELECT /*+ parallel(48) */ * FROM (
SELECT /*+ parallel(48) */ MIN(vertex1.v) AS customer,
                           MIN(vertex2.v) AS item,
                           MIN(edges.vn) AS real,
                           SUM(l.v * r.v) AS predicted

FROM PHONESGE$ edges,
     CUSTOMER_MAT l,
     ITEM_MAT r,
     PHONESVT$ vertex1,
     PHONESVT$ vertex2
WHERE l.k = r.k
     AND l.c = edges.svid(+)
     AND r.p = edges.dvid(+)
     AND l.c = vertex1.vid
     AND r.p = vertex2.vid
GROUP BY l.c, r.p
ORDER BY l.c, r.p
) WHERE rownum <= 11;
```

This query may produce an output similar to the following.

CUSTOMER	ITEM	REAL	PREDICTED
Adam	Apple	5	3.67375703
Adam	Blackberry		3.66079652
Adam	Danger		2.77049596
Adam	Ericsson	4	2.1764858
Adam	Figo		3.10631337
Adam	Google	4	4.42429022
Adam	Huawei	3	3.4289115
Ben	Apple		2.82127589
Ben	Blackberry	2	2.81132282
Ben	Danger	3	2.12761307
Ben	Ericsson	3	3.2389595

To get a prediction for a specific vertex (customer, item, or both) the query can be restricted with the desired ID values. For example, to get the predicted value of vertex 1 (customer) and vertex 105 (item), you can use the following query.

```
SELECT /*+ parallel(48) */ MIN(vertex1.v) AS customer,
                           MIN(vertex2.v) AS item,
                           MIN(edges.vn) AS real,
                           SUM(l.v * r.v) AS predicted

FROM PHONESGE$ edges,
     CUSTOMER_MAT l,
     ITEM_MAT r,
     PHONESVT$ vertex1,
     PHONESVT$ vertex2
WHERE l.k = r.k
     AND l.c = edges.svid(+)
     AND r.p = edges.dvid(+)
     AND l.c = vertex1.vid
```

```

AND vertex1.vid = 1 /* Remove to get all predictions for item 105 */
AND r.p = vertex2.vid
AND vertex2.vid = 105 /* Remove to get all predictions for customer 1
*/
                                /* Remove both lines to get all predictions */
GROUP BY l.c, r.p
ORDER BY l.c, r.p;

```

This query may produce an output similar to the following.

CUSTOMER	ITEM	REAL	PREDICTED
Adam	Ericsson	4.21764858	

5.8 Creating Property Graph Views on an RDF Graph

With Oracle Graph, you can view RDF data as a property graph to execute graph analytics operations by creating property graph views over an RDF graph stored in Oracle Database.

Given an RDF model (or a virtual model), the property graph feature creates two views, a `<graph_name>VT$` view for vertices and a `<graph_name>GE$` view for edges.

The `PGUtils.createPropertyGraphViewOnRDF` method lets you customize a property graph view over RDF data:

```

public static void createPropertyGraphViewOnRDF( Connection conn /* a Connection
instance to Oracle database */,
        String pgGraphName /* the name of the property graph to be created */,
        String rdfModelName /* the name of the RDF model */,
        boolean virtualModel /* a flag represents if the RDF model
is virtual model or not;
true - virtual mode, false - normal model*/,
        RDFPredicate[] predListForVertexAttrs /* an array of RDFPredicate objects
specifying how to create vertex view using these predicates; each RDFPredicate
includes two fields: an URL of the RDF predicate, the corresponding name of
vertex key in the Property Graph. The mapping from RDF predicates to vertex keys
will be created based on this parameter. */,
        RDFPredicate[] predListForEdges /* an array of RDFPredicate specifying how
to create edge view using these predicates; each RDFPredicate includes two (or
three) fields: an URL of the RDF predicate, the edge label in the Property
Graph, the weight of the edge (optional). The mapping from RDF predicates to
edges will be created based on this parameter. */)

```

This operation requires the name of the property graph, the name of the RDF Model used to generate the Property Graph view, and a set of mappings determining how triples will be parsed into vertices or edges. The `createPropertyGraphViewOnRDF` method requires a *key/value mapping* array specifying how RDF predicates are mapped to Key/Value properties for vertices, and an *edge mapping* array specifying how RDF predicates are mapped to edges. The `PGUtils.RDFPredicate` API lets you create a map from RDF assertions to vertices/edges.

Vertices are created based on the triples matching at least one of the RDF predicates in the key/value mappings. Each triple satisfying one of the RDF predicates defined in the mapping array is parsed into a vertex with ID based on the internal RDF resource

ID of the subject of the triple, and a key/value pair whose key is defined by the mapping itself and whose value is obtained from the object of the triple.

The following example defines a key/value mapping of the RDF predicate URI `http://purl.org/dc/elements/1.1/title` to the key/value property with property name `title`.

```
String titleURL = "http://purl.org/dc/elements/1.1/title";
// create an RDFPredicate to specify how to map the RDF predicate to vertex keys
RDFPredicate titleRDFPredicate
    = RDFPredicate.getInstance(titleURL /* RDF Predicate URI */,
                              "title" /* property name */);
```

Edges are created based on the triples matching at least one of the RDF predicates in the edge mapping array. Each triple satisfying the RDF predicate defined in the mapping array is parsed into an edge with ID based on the row number, an edge label defined by the mapping itself, a source vertex obtained from the RDF Resource ID of the subject of the triple, and a destination vertex obtained from the RDF Resource ID of the object of the triple. For each triple parsed here, two vertices will be created if they were not generated from the key/value mapping.

The following example defines an edge mapping of the RDF predicate URI `http://purl.org/dc/elements/1.1/reference` to an edge with a label `references` and a weight of `0.5d`.

```
String referencesURL = "http://purl.org/dc/terms/references";
// create an RDFPredicate to specify how to map the RDF predicate to edges
RDFPredicate referencesRDFPredicate
    = RDFPredicate.getInstance(referencesURL, "references", 0.5d);
```

The following example creates a property graph view over the RDF model `articles` describing different publications, their authors, and references. The generated property graph will include vertices with some key/value properties that may include `title` and `creator`. The edges in the property graph will be determined by the references among publications.

```
Oracle oracle = null;
Connection conn = null;
OraclePropertyGraph pggraph = null;
try {
    // create the connection instance to Oracle database
    OracleDataSource ds = new oracle.jdbc.pool.OracleDataSource();
    ds.setURL(jdbcUrl);
    conn = (OracleConnection) ds.getConnection(user, password);

    // define some string variables for RDF predicates
    String titleURL = "http://purl.org/dc/elements/1.1/title";
    String creatorURL = "http://purl.org/dc/elements/1.1/creator";
    String serialnumberURL = "http://purl.org/dc/elements/1.1/serialnumber";
    String widthURL = "http://purl.org/dc/elements/1.1/width";
    String weightURL = "http://purl.org/dc/elements/1.1/weight";
    String onsaleURL = "http://purl.org/dc/elements/1.1/onsale";
    String publicationDateURL = "http://purl.org/dc/elements/1.1/publicationDate";
    String publicationTimeURL = "http://purl.org/dc/elements/1.1/publicationTime";
    String referencesURL = "http://purl.org/dc/terms/references";

    // create RDFPredicate[] predsForVertexAttrs to specify how to map
    // RDF predicate to vertex keys
    RDFPredicate[] predsForVertexAttrs = new RDFPredicate[8];
    predsForVertexAttrs[0] = RDFPredicate.getInstance(titleURL, "title");
    predsForVertexAttrs[1] = RDFPredicate.getInstance(creatorURL, "creator");
    predsForVertexAttrs[2] = RDFPredicate.getInstance(serialnumberURL,
```

```

        "serialnumber");
predsForVertexAttrs[3] = RDFPredicate.getInstance(widthURL, "width");
predsForVertexAttrs[4] = RDFPredicate.getInstance(weightURL, "weight");
predsForVertexAttrs[5] = RDFPredicate.getInstance(onsaleURL, "onsale");
predsForVertexAttrs[6] = RDFPredicate.getInstance(publicationDateURL,
        "publicationDate");
predsForVertexAttrs[7] = RDFPredicate.getInstance(publicationTimeURL,
        "publicationTime");

// create RDFPredicate[] predsForEdges to specify how to map RDF predicates to
// edges
RDFPredicate[] predsForEdges = new RDFPredicate[1];
predsForEdges[0] = RDFPredicate.getInstance(referencesURL, "references", 0.5d);

// create PG view on RDF model
PGUtils.createPropertyGraphViewOnRDF(conn, "articles", "articles", false,
        predsForVertexAttrs, predsForEdges);

// get the Property Graph instance
oracle = new Oracle(jdbcUrl, user, password);
pggraph = OraclePropertyGraph.getInstance(oracle, "articles", 24);

System.err.println("----- Vertices from property graph view -----");
pggraph.getVertices();
System.err.println("----- Edges from property graph view -----");
pggraph.getEdges();
}
finally {
    pggraph.shutdown();
    oracle.dispose();
    conn.close();
}

```

Given the following triples in the `articles` RDF model (11 triples), the output property graph will include two vertices, one for `<http://nature.example.com/Article1>` (`v1`) and another one for `<http://nature.example.com/Article2>` (`v2`). For vertex `v1`, it has eight properties, whose values are the same as their RDF predicates. For example, `v1`'s title is *"All about XYZ"*. Similarly for vertex `v2`, it has two properties: title and creator. The output property graph will include a single edge (`eid:1`) from vertex `v1` to vertex `v2` with an edge label *"references"* and a weight of `0.5d`.

```

<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/title>
"All about XYZ"^^xsd:string.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/creator>
"Jane Smith"^^xsd:string.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/
serialnumber> "123456"^^xsd:integer.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/width>
"10.5"^^xsd:float.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/weight>
"1.08"^^xsd:double.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/onsale>
"false"^^xsd:boolean.
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/
publicationDate> "2016-03-08"^^xsd:date)
<http://nature.example.com/Article1> <http://purl.org/dc/elements/1.1/
publicationTime> "2016-03-08T10:10:10"^^xsd:dateTime)
<http://nature.example.com/Article2> <http://purl.org/dc/elements/1.1/title> "A
review of ABC"^^xsd:string.
<http://nature.example.com/Article2> <http://purl.org/dc/elements/1.1/creator>
"Joe Bloggs"^^xsd:string.

```

```
<http://nature.example.com/Article1> <http://purl.org/dc/terms/references> <http://  
nature.example.com/Article2>.
```

The preceding code will produce an output similar as the following. Note that the internal RDF resource ID values may vary across different Oracle databases.

```
----- Vertices from property graph view -----  
Vertex ID 7299961478807817799 {creator:str:Jane Smith, onsale:bol:false,  
publicationDate:dat:Mon Mar 07 16:00:00 PST 2016, publicationTime:dat:Tue Mar 08  
02:10:10 PST 2016, serialnumber:dbl:123456.0, title:str:All about XYZ,  
weight:dbl:1.08, width:flo:10.5}  
Vertex ID 7074365724528867041 {creator:str:Joe Bloggs, title:str:A review of ABC}  
----- Edges from property graph view -----  
Edge ID 1 from Vertex ID 7299961478807817799 {creator:str:Jane Smith,  
onsale:bol:false, publicationDate:dat:Mon Mar 07 16:00:00 PST 2016,  
publicationTime:dat:Tue Mar 08 02:10:10 PST 2016, serialnumber:dbl:123456.0,  
title:str:All about XYZ, weight:dbl:1.08, width:flo:10.5} =[references]=> Vertex ID  
7074365724528867041 {creator:str:Joe Bloggs, title:str:A review of ABC}  
edgeKV[{weight:dbl:0.5}]
```

5.9 Oracle Flat File Format Definition

A property graph can be defined in two flat files, specifically description files for the vertices and edges.

- [About the Property Graph Description Files](#)
- [Edge File](#)
- [Vertex File](#)
- [Encoding Special Characters](#)
- [Example Property Graph in Oracle Flat File Format](#)
- [Converting an Oracle Database Table to an Oracle-Defined Property Graph Flat File](#)
- [Converting CSV Files for Vertices and Edges to Oracle-Defined Property Graph Flat Files](#)

5.9.1 About the Property Graph Description Files

A pair of files describe a property graph:

- **Vertex file:** Describes the vertices of the property graph. This file has an `.opv` file name extension.
- **Edge file:** Describes the edges of the property graph. This file has an `.ope` file name extension.

It is recommended that these two files share the same base name. For example, `simple.opv` and `simple.ope` define a property graph.

5.9.2 Edge File

Each line in an edge file is a record that describes an edge of the property graph. A record can describe one key-value property of an edge, thus multiple records are used to describe an edge with multiple properties.

A record contains nine fields separated by commas. Each record must contain eight commas to delimit all fields, whether or not they have values:

edge_ID, *source_vertex_ID*, *destination_vertex_ID*, *edge_label*, *key_name*,
value_type, *value*, *value*, *value*

The following table describes the fields composing an edge file record.

Table 5-1 Edge File Record Format

Field Number	Name	Description
1	<i>edge_ID</i>	An integer that uniquely identifies the edge
2	<i>source_vertex_ID</i>	The <i>vertex_ID</i> of the outgoing tail of the edge.
3	<i>destination_vertex_ID</i>	The <i>vertex_ID</i> of the incoming head of the edge.
4	<i>edge_label</i>	The encoded label of the edge, which describes the relationship between the two vertices
5	<i>key_name</i>	The encoded name of the key in a key-value pair If the edge has no properties, then enter a space (%20). This example describes edge 100 with no properties: 100,1,2,likes,%20,,,,
6	<i>value_type</i>	An integer that represents the data type of the value in the key-value pair: 1 String 2 Integer 3 Float 4 Double 5 Timestamp (date) 6 Boolean 7 Long integer 8 Short integer 9 Byte 10 Char 20 Spatial 101 Serializable Java object
7	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is neither numeric nor timestamp (date)
8	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is numeric
9	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is a timestamp (date) Use the Java SimpleDateFormat class to identify the format of the date. This example describes the date format of 2015-03-26Th00:00:00.000-05:00: SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM- dd'T'HH:mm:ss.SSSXXX"); encode(sdf.format((java.util.Date) value));

Required Grouping of Edges: An edge can have multiple properties, and the edge file includes a record (represented by a single line of text in the flat file) for each combination of an edge ID and a property for that edge. In the edge file, all records for each edge must be grouped together (that is, not have any intervening records for other edges. You can accomplish this any way you want, but a convenient way is to sort the edge file records in ascending (or descending) order by edge ID. (Note, however, an edge file is not required to have all records sorted by edge ID; this is merely one way to achieve the grouping requirement.)

When building an edge file in Oracle flat file format, it is important to verify that the edge property name and value fields are correctly encoded (see especially [Encoding Special Characters](#)). To simplify the encoding, you can use the `OraclePropertyGraphUtils.escape` Java API.

You can use the `OraclePropertyGraphUtils.outputEdgeRecord(os, eid, svid, dvid, label, key, value)` utility method to serialize an edge record directly in Oracle flat file format. With this method, you no longer need to worry about encoding of special characters. The method writes a new line of text in the given output stream describing the key/value property of the given edge identified by `eid`.

Example 5-24 Using `OraclePropertyGraphUtils.outputEdgeRecord`

This example uses `OraclePropertyGraphUtils.outputEdgeRecord` to write two new lines for edge 100 between vertices 1 and 2 with label `friendOf`.

```
OutputStream os = new FileOutputStream("./example.ope");
int sinceYear = 2009;
long eid = 100;
long svid = 1;
long dvid = 2;
OraclePropertyGraphUtils.outputEdgeRecord(os, eid, svid, dvid, "friendOf",
"since (year)", sinceYear);
OraclePropertyGraphUtils.outputEdgeRecord(os, eid, svid, dvid, "friendOf",
"weight", 1);
os.flush();
os.close();
```

The first line in the generated output file describes the property “since (year)” with value 2009, and the second line and the next line sets the edge weight to 1.

```
% cat example.ope
100,1,2,friendOf,since%20(year),2,,2009,
100,1,2,friendOf,weight,2,,1,
```

5.9.3 Vertex File

Each line in a vertex file is a record that describes a vertex of the property graph. A record can describe one key-value property of a vertex, thus multiple records/lines are used to describe a vertex with multiple properties.

A record contains fields separated by commas. Each record must contain five commas to delimit first six fields, whether or not they have values. An optional seventh field can be added (delimited from the sixth field by a comma) to define a vertex label:

vertex_ID, key_name, value_type, value, value, value, vertex_label

The following table describes the fields composing a vertex file record.

Table 5-2 Vertex File Record Format

Field Number	Name	Description
1	<i>vertex_ID</i>	An integer that uniquely identifies the vertex
2	<i>key_name</i>	The name of the key in the key-value pair If the vertex has no properties, then enter a space (%20). This example describes vertex 1 with no properties: 1,%20,,,,
3	<i>value_type</i>	An integer that represents the data type of the value in the key-value pair: 1 String 2 Integer 3 Float 4 Double 5 Timestamp (date) 6 Boolean 7 Long integer 8 Short integer 9 Byte 10 Char 20 Spatial data, which can be geospatial coordinates, lines, polygons, or Well-Known Text (WKT) literals 101 Serializable Java object
4	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is neither numeric nor date
5	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is numeric
6	<i>value</i>	The encoded, nonnull value of <i>key_name</i> when it is a timestamp (date) Use the Java <code>SimpleDateFormat</code> class to identify the format of the date. This example describes the date format of 2015-03-26T00:00:00.000-05:00: <pre>SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM- dd'T'HH:mm:ss.SSSXXX"); encode(sdf.format((java.util.Date) value));</pre>
7	<i>vertex_label</i>	The optional encoded label of the vertex, which can be used to describe the type or category of the vertex.

Required Grouping of Vertices: A vertex can have multiple properties, and the vertex file includes a record (represented by a single line of text in the flat file) for each combination of a vertex ID and a property for that vertex. In the vertex file, all records for each vertex must be grouped together (that is, not have any intervening records for other vertices. You can accomplish this any way you want, but a convenient way is to sort the vertex file records in ascending (or descending) order by vertex ID. (Note,

however, a vertex file is not required to have all records sorted by vertex ID; this is merely one way to achieve the grouping requirement.)

When building an edge file in Oracle flat file format, it is important to verify that the vertex property name and value fields are correctly encoded (see especially [Encoding Special Characters](#)). To simplify the encoding, you can use the `OraclePropertyGraphUtils.escape` Java API.

You can use the `OraclePropertyGraphUtils.outputVertexRecord(os, vid, key, value)` utility method to serialize a vertex record directly in Oracle flat file format. With this method, you no longer need to worry about encoding of special characters. The method writes a new line of text in the given output stream describing the key/value property of the given vertex identified by `vid`.

Example 5-25 Using `OraclePropertyGraphUtils.outputVertexRecord`

This example uses `OraclePropertyGraphUtils.outputVertexRecord` to write two new lines for vertex 1.

```
OutputStream os = new FileOutputStream("./example.opv");
long vid = 1;
String label = "person";
OraclePropertyGraphUtils.outputVertexRecord(os, vid, label, "name", "Robert
Smith");
OraclePropertyGraphUtils.outputVertexRecord(os, vid, label, "birth year",
1961);
os.flush();
os.close();
```

The first line in the generated output file describes the property name with value "Robert Smith", and the second line describes his birth year of 1961.

```
% cat example.opv
1,name,1,Robert%20Smith,,person
1,birth%20year,2,,1961,,person
```

5.9.4 Encoding Special Characters

The encoding is UTF-8 for the vertex and edge files. The following table lists the special characters that must be encoded as strings when they appear in a vertex or edge property (key-value pair) or an edge label. No other characters require encoding.

Table 5-3 Special Character Codes in the Oracle Flat File Format

Special Character	String Encoding	Description
%	%25	Percent
\t	%09	Tab
(space)	%20	Space
\n	%0A	New line
\r	%0D	Return
,	%2C	Comma

5.9.5 Example Property Graph in Oracle Flat File Format

An example property graph in Oracle flat file format is as follows. In this example, there are two vertices (John and Mary), and a single edge denoting that John is a friend of Mary.

```
%cat simple.opv
1,age,2,,10,
1,name,1,John,,
2,name,1,Mary,,
2,hobby,1,soccer,,

%cat simple.ope
100,1,2,friendOf,%20,,,,
```

5.9.6 Converting an Oracle Database Table to an Oracle-Defined Property Graph Flat File

You can convert Oracle Database tables that represent the vertices and edges of a graph into an Oracle-defined flat file format (.opv and .ope file extensions).

If you have graph data stored in Oracle Database tables, you can use Java API methods to convert that data into flat files, and later load the tables into Oracle Database as a property graph. This eliminates the need to take some other manual approach to generating the flat files from existing Oracle Database tables.

Converting a Table Storing Graph Vertices to an .opv File

You can convert an Oracle Database table that contains entities (that can be represented as vertices of a graph) to a property graph flat file in .opv format.

For example, assume the following relational table: EmployeeTab (empID integer not null, hasName varchar(255), hasAge integer, hasSalary number)

Assume that this table has the following data:

```
101, Jean, 20, 120.0
102, Mary, 21, 50.0
103, Jack, 22, 110.0
.....
```

Each employee can be viewed as a vertex in the graph. The vertex ID could be the value of employeeID or an ID generated using some heuristics like hashing. The columns hasName, hasAge, and hasSalary can be viewed as attributes.

The Java method `OraclePropertyGraphUtils.convertRDBMSTable2OPV` and its Javadoc information are as follows:

```
/**
 * conn: is an connect instance to the Oracle relational database
 * rdbmsTableName: name of the RDBMS table to be converted
 * vidColName is the name of an column in RDBMS table to be treated as vertex ID
 * lVIDOffset is the offset will be applied to the vertex ID
 * ctams defines how to map columns in the RDBMS table to the attributes
 * dop degree of parallelism
 * dcl an instance of DataConverterListener to report the progress and control
 the behavior when errors happen
```

```

*/
OraclePropertyGraphUtils.convertRDBMSTable2OPV(
    Connection conn,
    String rdbmsTableName,
    String vidColName,
    long lVIDOffset,
    ColumnToAttrMapping[] ctams,
    int dop,
    OutputStream opvOS,
    DataConverterListener dcl);

```

The following code snippet converts this table into an Oracle-defined vertex file (.opv):

```

// location of the output file
String opv = "./EmployeeTab.opv";
OutputStream opvOS = new FileOutputStream(opv);
// an array of ColumnToAttrMapping objects; each object defines how to map a
column in the RDBMS table to an attribute of the vertex in an Oracle
Property Graph.
ColumnToAttrMapping[] ctams = new ColumnToAttrMapping[3];
// map column "hasName" to attribute "name" of type String
ctams[0] = ColumnToAttrMapping.getInstance("hasName", "name", String.class);
// map column "hasAge" to attribute "age" of type Integer
ctams[1] = ColumnToAttrMapping.getInstance("hasAge", "age", Integer.class);
// map column "hasSalary" to attribute "salary" of type Double
ctams[2] = ColumnToAttrMapping.getInstance("hasSalary",
"salary",Double.class);
// convert RDBMS table "EmployeeTab" into opv file "./EmployeeTab.opv",
column "empID" is the vertex ID column, offset 10001 will be applied to
vertex ID, use ctams to map RDBMS columns to attributes, set DOP to 8
OraclePropertyGraphUtils.convertRDBMSTable2OPV(conn, "EmployeeTab", "empID",
10001, ctams, 8, opvOS, (DataConverterListener) null);

```



Note:

The lowercase letter "l" as the last character in the offset value 10001 denotes that the value before it is a long integer.

The conversion result is as follows:

```

1101,name,1,Jean,,
1101,age,2,,20,
1101,salary,4,,120.0,
1102,name,1,Mary,,
1102,age,2,,21,
1102,salary,4,,50.0,
1103,name,1,Jack,,
1103,age,2,,22,
1103,salary,4,,110.0,

```

In this case, each row in table EmployeeTab is converted to one vertex with three attributes. For example, the row with data "101, Jean, 20, 120.0" is converted to a vertex with ID 1101 with attributes name/"Jean", age/20, salary/120.0. There is an offset between original empID

101 and vertex ID 1101 because an offset 1000 is applied. An offset is useful to avoid collision in ID values of graph elements.

Converting a Table Storing Graph Edges to an .ope File

You can convert an Oracle Database table that contains entity relationships (that can be represented as edges of a graph) to a property graph flat file in .ope format.

For example, assume the following relational table: `EmpRelationTab` (`relationID` integer not null, `source` integer not null, `destination` integer not null, `relationType` varchar(255), `startDate` date)

Assume that this table has the following data:

```
90001, 101, 102, manage, 10-May-2015
90002, 101, 103, manage, 11-Jan-2015
90003, 102, 103, colleague, 11-Jan-2015
.....
```

Each relation (row) can be viewed as an edge in a graph. Specifically, edge ID could be the same as `relationID` or an ID generated using some heuristics like hashing. The column `relationType` can be used to define edge labels, and the column `startDate` can be treated as an edge attribute.

The Java method `OraclePropertyGraphUtils.convertRDBMSTable2OPE` and its Javadoc information are as follows:

```
/**
 * conn: is an connect instance to the Oracle relational database
 * rdbmsTableName: name of the RDBMS table to be converted
 * eidColName is the name of an column in RDBMS table to be treated as edge ID
 * lEIDOffset is the offset will be applied to the edge ID
 * svidColName is the name of an column in RDBMS table to be treated as source
 vertex ID of the edge
 * dvidColName is the name of an column in RDBMS table to be treated as
 destination vertex ID of the edge
 * lVIDOffset is the offset will be applied to the vertex ID
 * bHasEdgeLabelCol a Boolean flag represents if the given RDBMS table has a
 column for edge labels; if true, use value of column elColName as the edge
 label; otherwise, use the constant string elColName as the edge label
 * elColName is the name of an column in RDBMS table to be treated as edge labels
 * ctams defines how to map columns in the RDBMS table to the attributes
 * dop degree of parallelism
 * dcl an instance of DataConverterListener to report the progress and control
 the behavior when errors happen
 */
OraclePropertyGraphUtils.convertRDBMSTable2OPE(
    Connection conn,
    String rdbmsTableName,
    String eidColName,
    long lEIDOffset,
    String svidColName,
    String dvidColName,
    long lVIDOffset,
    boolean bHasEdgeLabelCol,
    String elColName,
    ColumnToAttrMapping[] ctams,
    int dop,
    OutputStream opeOS,
    DataConverterListener dcl);
```

The following code snippet converts this table into an Oracle-defined edge file (.ope):

```
// location of the output file
String ope = "./EmpRelationTab.ope";
OutputStream opeOS = new FileOutputStream(ope);
// an array of ColumnToAttrMapping objects; each object defines how to map a
column in the RDBMS table to an attribute of the edge in an Oracle Property
Graph.
ColumnToAttrMapping[] ctams = new ColumnToAttrMapping[1];
// map column "startDate" to attribute "since" of type Date
ctams[0] = ColumnToAttrMapping.getInstance("startDate", "since", Date.class);
// convert RDBMS table "EmpRelationTab" into ope file "./
EmpRelationTab.opv", column "relationID" is the edge ID column, offset
100001 will be applied to edge ID, the source and destination vertices of
the edge are defined by columns "source" and "destination", offset 10001
will be applied to vertex ID, the RDBMS table has an column "relationType"
to be treated as edge labels, use ctams to map RDBMS columns to edge
attributes, set DOP to 8
OraclePropertyGraphUtils.convertRDBMSTable2OPE(conn, "EmpRelationTab",
"relationID", 100001, "source", "destination", 10001, true, "relationType",
ctams, 8, opeOS, (DataConverterListener) null);
```



Note:

The lowercase letter "l" as the last character in the offset value 100001 denotes that the value before it is a long integer.

The conversion result is as follows:

```
100001,1101,1102,manage,since,5,,2015-05-10T00:00:00.000-07:00
100002,1101,1103,manage,since,5,,2015-01-11T00:00:00.000-07:00
100003,1102,1103,colleague,since,5,,2015-01-11T00:00:00.000-07:00
```

In this case, each row in table EmpRelationTab is converted to a distinct edge with the attribute since. For example, the row with data "90001, 101, 102, manage, 10-May-2015" is converted to an edge with ID 100001 linking vertex 1101 to vertex 1102. This edge has attribute since/"2015-05-10T00:00:00.000-07:00". There is an offset between original relationID "90001" and edge ID "100001" because an offset 100001 is applied. Similarly, an offset 10001 is applied to the source and destination vertex IDs.

5.9.7 Converting CSV Files for Vertices and Edges to Oracle-Defined Property Graph Flat Files

Some applications use CSV (comma-separated value) format to encode vertices and edges of a graph. In this format, each record of the CSV file represents a single vertex or edge, with all its properties. You can convert a CSV file representing the vertices of a graph to Oracle-defined flat file format definition (.opv for vertices, .ope for edges).

The CSV file to be converted may include a header line specifying the column name and the type of the attribute that the column represents. If the header includes only the attribute names, then the converter will assume that the data type of the values will be String.

The Java APIs to convert CSV to OPV or OPE receive an `InputStream` from which they read the vertices or edges (from CSV), and write them in the `.opv` or `.ope` format to an `OutputStream`. The converter APIs also allow customization of the conversion process.

The following subtopics provide instructions for converting vertices and edges:

- Vertices: Converting a CSV File to Oracle-Defined Flat File Format (`.opv`)
- Edges: Converting a CSV File to Oracle-Defined Flat File Format (`.ope`)

The instructions for both are very similar, but with differences specific to vertices and edges.

Vertices: Converting a CSV File to Oracle-Defined Flat File Format (`.opv`)

If the CSV file does not include a header, you must specify a `ColumnToAttrMapping` array describing all the attribute names (mapped to its values data types) in the same order in which they appear in the CSV file. Additionally, the entire columns from the CSV file must be described in the array, including special columns such as the ID for the vertices. If you want to specify the headers for the column in the first line of the same CSV file, then this parameter must be set to null.

To convert a CSV file representing vertices, you can use one of the `convertCSV2OPV` APIs. The simplest of these APIs requires:

- An `InputStream` to read vertices from a CSV file
- The name of the column that is representing the vertex ID (this column must appear in the CSV file)
- An integer offset to add to the VID (an offset is useful to avoid collision in ID values of graph elements)
- A `ColumnToAttrMapping` array (which must be null if the headers are specified in the file)
- Degree of parallelism (DOP)
- An integer denoting offset (number of vertex records to skip) before converting
- An `OutputStream` in which the vertex flat file (`.opv`) will be written
- An optional `DataConverterListener` that can be used to keep track of the conversion progress and decide what to do if an error occurs

Additional parameters can be used to specify a different format of the CSV file:

- The delimiter character, which is used to separate tokens in a record. The default is the comma character `,`.
- The quotation character, which is used to quote String values so they can contain special characters, for example, commas. If a quotation character appears in the value of the String itself, it must be escaped either by duplication or by placing a backslash character `\` before it. Some examples are:
 - `""Hello, world""`, the screen showed...
 - `"But Vader replied: \"No, I am your father.\""`
- The Date format, which will be used to parse the date values. For the CSV conversion, this parameter can be null, but it is recommended to be specified if the CSV has a specific date format. Providing a specific date format helps

performance, because that format will be used as the first option when trying to parse date values. Some example date formats are:

- "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
 - "MM/dd/yyyy HH:mm:ss"
 - "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'"
 - "dddd, dd MMMM yyyy hh:mm:ss"
 - "yyyy-MM-dd"
 - "MM/dd/yyyy"
- A flag indicating if the CSV file contains String values with new line characters. If this parameter is set to true, all the Strings in the file that contain new lines or quotation characters as values must be quoted.
 - "The first lines of Don Quixote are:""In a village of La Mancha, the name of which I have no desire to call to mind""."

The following code fragment shows how to create a `ColumnToAttrMapping` array and use the API to convert a CSV file into an `.opv` file.

```
String inputCSV           = "/path/mygraph-vertices.csv";
String outputOPV         = "/path/mygraph.opv";
ColumnToAttrMapping[] ctams = new ColumnToAttrMapping[4];
ctams[0]                 = ColumnToAttrMapping.getInstance("VID",
Long.class);
ctams[1]                 = ColumnToAttrMapping.getInstance("name",
String.class);
ctams[2]                 = ColumnToAttrMapping.getInstance("score",
Double.class);
ctams[3]                 = ColumnToAttrMapping.getInstance("age",
Integer.class);
String vidColumn         = "VID";

isCSV = new FileInputStream(inputCSV);
osOPV = new FileOutputStream(new File(outputOPV));

// Convert Vertices
OraclePropertyGraphUtilsBase.convertCSV2OPV(isCSV, vidColumn, 0, ctams,
1, 0, osOPV, null);
isOPV.close();
osOPV.close();
```

In this example, the CSV file to be converted must not include the header and contain four columns (the vertex ID, name, score, and age). An example CVS is as follows:

```
1,John,4.2,30
2,Mary,4.3,32
3,"Skywalker, Anakin",5.0,46
4,"Darth Vader",5.0,46
5,"Skywalker, Luke",5.0,53
```

The resulting `.opv` file is as follows:

```
1,name,1,John,,
1,score,4,,4.2,
```

```
1,age,2,,30,
2,name,1,Mary,,
2,score,4,,4.3,
2,age,2,,32,
3,name,1,Skywalker%2C%20Anakin,,
3,score,4,,5.0,
3,age,2,,46,
4,name,1,Darth%20Vader,,
4,score,4,,5.0,
4,age,2,,46,
5,name,1,Skywalker%2C%20Luke,,
5,score,4,,5.0,
5,age,2,,53,
```

Edges: Converting a CSV File to Oracle-Defined Flat File Format (.ope)

If the CSV file does not include a header, you must specify a `ColumnToAttrMapping` array describing all the attribute names (mapped to its values data types) in the same order in which they appear in the CSV file. Additionally, the entire columns from the CSV file must be described in the array, including special columns such as the ID for the edges if it applies, and the `START_ID`, `END_ID`, and `TYPE`, which are required. If you want to specify the headers for the column in the first line of the same CSV file, then this parameter must be set to null.

To convert a CSV file representing vertices, you can use one of the `convertCSV2OPE` APIs. The simplest of these APIs requires:

- An `InputStream` to read vertices from a CSV file
- The name of the column that is representing the edge ID (this is optional in the CSV file; if it is not present, the line number will be used as the ID)
- An integer offset to add to the EID (an offset is useful to avoid collision in ID values of graph elements)
- Name of the column that is representing the source vertex ID (this column must appear in the CSV file)
- Name of the column that is representing the destination vertex ID (this column must appear in the CSV file)
- Offset to the VID (`lOffsetVID`). This offset will be added on top of the original SVID and DVID values. (A variation of this API takes in two arguments (`lOffsetSVID` and `lOffsetDVID`): one offset for SVID, the other offset for DVID.)
- A boolean flag indicating if the edge label column is present in the CSV file.
- Name of the column that is representing the edge label (if this column is not present in the CSV file, then this parameter will be used as a constant for all edge labels)
- A `ColumnToAttrMapping` array (which must be null if the headers are specified in the file)
- Degree of parallelism (DOP)
- An integer denoting offset (number of edge records to skip) before converting
- An `OutputStream` in which the edge flat file (.ope) will be written
- An optional `DataConverterListener` that can be used to keep track of the conversion progress and decide what to do if an error occurs.

Additional parameters can be used to specify a different format of the CSV file:

- The delimiter character, which is used to separate tokens in a record. The default is the comma character ','.
- The quotation character, which is used to quote String values so they can contain special characters, for example, commas. If a quotation character appears in the value of the String itself, it must be escaped either by duplication or by placing a backslash character '\' before it. Some examples are:
 - ""Hello, world"", the screen showed..."
 - "But Vader replied: \"No, I am your father.\""
- The Date format, which will be used to parse the date values. For the CSV conversion, this parameter can be null, but it is recommended to be specified if the CSV has a specific date format. Providing a specific date format helps performance, because that format will be used as the first option when trying to parse date values. Some example date formats are:
 - "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
 - "MM/dd/yyyy HH:mm:ss"
 - "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'"
 - "dddd, dd MMMM yyyy hh:mm:ss"
 - "yyyy-MM-dd"
 - "MM/dd/yyyy"
- A flag indicating if the CSV file contains String values with new line characters. If this parameter is set to true, all the Strings in the file that contain new lines or quotation characters as values must be quoted.
 - "The first lines of Don Quixote are: ""In a village of La Mancha, the name of which I have no desire to call to mind""."

The following code fragment shows how to use the API to convert a CSV file into an .ope file with a null `ColumnToAttrMapping` array.

```
String inputOPE    = "/path/mygraph-edges.csv";
String outputOPE  = "/path/mygraph.ope";
String eidColumn  = null;                // null implies that an integer
sequence will be used
String svidColumn = "START_ID";
String dvidColumn = "END_ID";
boolean hasLabel  = true;
String labelColumn = "TYPE";

isOPE = new FileInputStream(inputOPE);
osOPE = new FileOutputStream(new File(outputOPE));

// Convert Edges
OraclePropertyGraphUtilsBase.convertCSV2OPE(isOPE, eidColumn, 0,
svidColumn, dvidColumn, hasLabel, labelColumn, null, 1, 0, osOPE, null);
```

An input CSV that uses the former example to be converted should include the header specifying the columns name and their type. An example CSV file is as follows.

```
START_ID:long,weight:float,END_ID:long,:TYPE
1,1.0,2,loves
```

```
1,1.0,5,admires
2,0.9,1,loves
1,0.5,3,likes
2,0.0,4,likes
4,1.0,5,is the dad of
3,1.0,4,turns to
5,1.0,3,saves from the dark side
```

The resulting .ope file is as follows.

```
1,1,2,loves,weight,3,,1.0,
2,1,5,admires,weight,3,,1.0,
3,2,1,loves,weight,3,,0.9,
4,1,3,likes,weight,3,,0.5,
5,2,4,likes,weight,3,,0.0,
6,4,5,is%20the%20dad%20of,weight,3,,1.0,
7,3,4,turns%20to,weight,3,,1.0,
8,5,3,saves%20from%20the%20dark%20side,weight,3,,1.0,
```

6

Property Graph Query Language (PGQL)

PGQL is a SQL-like query language for property graph data structures that consist of *vertices* that are connected to other vertices by *edges*, each of which can have key-value pairs (properties) associated with them.

The language is based on the concept of *graph pattern matching*, which allows you to specify patterns that are matched against vertices and edges in a data graph.

Note:

The graph server (PGX) 21.3.0 supports [PGQL 1.4](#) and earlier versions.

The property graph support provides two ways to execute Property Graph Query Language (PGQL) queries through Java APIs:

- Use the `oracle.pgx.api` Java package to query an in-memory snapshot of a graph that has been loaded into the in-memory analyst (PGX), as described in [Using the In-Memory Graph Server \(PGX\)](#).
- Use the `oracle.pg.rdbms.pgql` Java package to directly query graph data stored in Oracle Database, as described in [Executing PGQL Queries Against Property Graph Schema Tables](#).

For more information about PGQL, see <https://pgql-lang.org>.

- [Creating a Property Graph using PGQL](#)
- [Pattern Matching with PGQL](#)
- [Edge Patterns Have a Direction with PGQL](#)
- [Vertex and Edge Labels with PGQL](#)
- [Variable-Length Paths with PGQL](#)
- [Aggregation and Sorting with PGQL](#)
- [Executing PGQL Queries Against the In-Memory Graph Server \(PGX\)](#)
This section describes the Java APIs that are used to execute PGQL queries in the In-Memory graph server (PGX).
- [Executing PGQL Queries Directly Against Oracle Database](#)
This topic explains how you can execute PGQL queries directly against the graph in Oracle Database (as opposed to in-memory).

6.1 Creating a Property Graph using PGQL

`CREATE PROPERTY GRAPH` is a PGQL DDL statement to create a graph from database tables. The graph is stored in the property graph schema.

The CREATE PROPERTY GRAPH statement starts with the name you give the graph, followed by a set of vertex tables and edge tables. The graph can have no vertex tables or edge tables (an empty graph), or vertex tables and no edge tables (a graph with only vertices and no edges), or both vertex tables and edge tables (a graph with vertices and edges). However, a graph cannot be specified with only edge tables and no vertex tables.

Consider the following example:

- **PERSONS** is a table with columns ID, NAME, and ACCOUNT_NUMBER. A row is added to this table for every person who has an account.
- **TRANSACTIONS** is a table with columns FROM_ACCOUNT, TO_ACCOUNT, DATE, and AMOUNT. A row is added into this table in the database every time money is transferred from a FROM_ACCOUNT to a TO_ACCOUNT.

A straightforward mapping of tables to graphs is as follows. The graph concepts mapped are: vertices, edges, labels, properties.

- **Vertex tables:** A table that contains data entities is a vertex table.
 - Each row in the vertex table is a vertex.
 - The columns in the vertex table are properties of the vertex.
 - The name of the vertex table is the default label for this set of vertices. Alternatively, you can specify a label name as part of the CREATE PROPERTY GRAPH statement.
- **Edge tables:** An edge table can be any table that links two vertex tables, or a table that has data that indicates an action from a source entity to a target entity. For example, a transfer of money from FROM_ACCOUNT to TO_ACCOUNT is a natural edge.
 - Foreign key relationships can give guidance on what links are relevant in your data. CREATE PROPERTY GRAPH will default to using foreign key relationships to identify edges.
 - Some of the properties of an edge table can be the properties of the edge. For example, an edge from FROM_ACCOUNT to TO_ACCOUNT can have properties DATE and AMOUNT.
 - The name of an edge table is the default label for this set of edges. Alternatively, you can specify a label name as part of the CREATE PROPERTY GRAPH statement.
- **Keys:**
 - **Keys in a vertex table:** The key of a vertex table identifies a unique vertex in the graph. The key can be specified in the CREATE PROPERTY GRAPH statement; otherwise, it defaults to the primary key of the table. If there are duplicate rows in the table, the CREATE PROPERTY GRAPH statement will return an error.
 - **Key in an edge table:** The key of an edge table uniquely identifies an edge in the graph. The KEY clause when specifying source and destination vertices uniquely identifies the source and destination vertices.

The following is an example CREATE PROPERTY GRAPH statement for the tables PERSONS and TRANSACTIONS.

```
CREATE PROPERTY GRAPH bank_transfers
  VERTEX TABLES (persons KEY(account_number))
  EDGE TABLES(
    transactions KEY (from_acct, to_acct, date, amount)
    SOURCE KEY (from_account) REFERENCES persons
    DESTINATION KEY (to_account) REFERENCES persons
    PROPERTIES (date, amount)
  )
```

- **Table aliases:** Vertex and edge tables must have unique names. If you need to identify multiple vertex tables from the same relational table, or multiple edge tables from the same relational table, you must use aliases. For example, you can create two vertex tables PERSONS and PERSONS_ID from one table PERSONS, as in the following example.

```
CREATE PROPERTY GRAPH bank_transfers
  VERTEX TABLES (persons KEY(account_number)
    persons_id AS persons KEY(id))
```

- **REFERENCES clause:** This connects the source and destination vertices of an edge to the corresponding vertex tables.

For more details, see: <https://pgql-lang.org/spec/latest/#creating-a-property-graph>.

The following table lists the different ways you can create a property graph using the CREATE PROPERTY GRAPH statement:

Table 6-1 CREATE PROPERTY GRAPH Statement Support

Method	More Information
Create a property graph in the in-memory graph server (PGX) using the <code>oracle.pgx.api</code> Java package	Java APIs for Executing CREATE PROPERTY GRAPH Statements
Create a property graph in the in-memory graph server (PGX) using the <code>pypgx.api</code> Python package	Python APIs for Executing CREATE PROPERTY GRAPH Statements
Create a property graph in Oracle Database (Property Graph Schema) using the <code>oracle.pg.rdbms.pgql</code> Java package	Creating Property Graphs through CREATE PROPERTY GRAPH Statements
Create a property graph in Oracle Database (Property Graph Schema) using the <code>opgpy.pgql</code> Python package	Creating a Property Graph Using the Python Client
Create a property graph view on Oracle Database tables	Creating a Property Graph View

6.2 Pattern Matching with PGQL

Pattern matching is done by specifying one or more path patterns in the MATCH clause. A single path pattern matches a linear path of vertices and edges, while more complex patterns can be matched by combining multiple path patterns, separated by comma. Value

expressions (similar to their SQL equivalents) are specified in the WHERE clause and let you filter out matches, typically by specifying constraints on the properties of the vertices and edges

For example, assume a graph of TCP/IP connections on a computer network, and you want to detect cases where someone logged into one machine, from there into another, and from there into yet another. You would query for that pattern like this:

```
SELECT id(host1) AS id1, id(host2) AS id2, id(host3) AS id3      /*
choose what to return */
FROM MATCH
  (host1) -[connection1]-> (host2) -[connection2]-> (host3)    /*
single linear path pattern to match */
WHERE
  connection1.toPort = 22 AND connection1.opened = true AND
  connection2.toPort = 22 AND connection2.opened = true AND
  connection1.bytes > 300 AND                                  /*
meaningful amount of data was exchanged */
  connection2.bytes > 300 AND
  connection1.start < connection2.start AND                  /*
second connection within time-frame of first */
  connection2.start + connection2.duration < connection1.start +
  connection1.duration
GROUP BY id1, id2, id3                                       /*
aggregate multiple matching connections */
```

For more examples of pattern matching, see the [relevant section of the PGQL specification](#).

6.3 Edge Patterns Have a Direction with PGQL

An edge pattern has a direction, as edges in graphs do. Thus, (a) `<-[]-` (b) specifies a case where *b* has an edge pointing at *a*, whereas (a) `-[]->` (b) looks for an edge in the opposite direction.

The following example finds common friends of April and Chris who are older than both of them.

```
SELECT friend.name, friend.dob
FROM MATCH                                /* note the arrow directions below */
  (p1:person) -[:likes]-> (friend) <-[:likes]- (p2:person)
WHERE
  p1.name = 'April' AND p2.name = 'Chris' AND
  friend.dob > p1.dob AND friend.dob > p2.dob
ORDER BY friend.dob DESC
```

For more examples of edge patterns, see the relevant section of the PGQL specification [here](#).

6.4 Vertex and Edge Labels with PGQL

Labels are a way of attaching type information to edges and nodes in a graph, and can be used in constraints in graphs where not all nodes represent the same thing. For example:

```
SELECT p.name
FROM MATCH (p:person) -[e1:likes]-> (m1:movie),
      MATCH (p) -[e2:likes]-> (m2:movie)
WHERE m1.title = 'Star Wars'
      AND m2.title = 'Avatar'
```

For more examples of label expressions, see the relevant section of the PGQL specification [here](#).

6.5 Variable-Length Paths with PGQL

Variable-length path patterns have a quantifier like `*` to match a variable number of vertices and edges. Using a `PATH` macro, you can specify a named path pattern at the start of a query that can be embedded into the `MATCH` clause any number of times, by referencing its name. The following example finds all of the common ancestors of Mario and Luigi.

```
PATH has_parent AS () -[:has_father|has_mother]-> ()
SELECT ancestor.name
FROM MATCH (p1:Person) -/:has_parent*/-> (ancestor:Person)
      , MATCH (p2:Person) -/:has_parent*/-> (ancestor)
WHERE
  p1.name = 'Mario' AND
  p2.name = 'Luigi'
```

The preceding path specification also shows the use of anonymous constraints, because there is no need to define names for intermediate edges or nodes that will not be used in additional constraints or query results. Anonymous elements can have constraints, such as `[:has_father|has_mother]` -- the edge does not get a variable name (because it will not be referenced elsewhere), but it is constrained.

For more examples of variable-length path pattern matching, see the relevant section of the PGQL specification [here](#).

6.6 Aggregation and Sorting with PGQL

Like SQL, PGQL has support for the following:

- `GROUP BY` to create groups of solutions
- `MIN`, `MAX`, `SUM`, and `AVG` aggregations
- `ORDER BY` to sort results

And for many other familiar SQL constructs.

For `GROUP BY` and aggregation, see the relevant section of the PGQL specification [here](#).

For `ORDER BY`, see the relevant section of the PGQL specification [here](#).

6.7 Executing PGQL Queries Against the In-Memory Graph Server (PGX)

This section describes the Java APIs that are used to execute PGQL queries in the In-Memory graph server (PGX).

- [Getting Started with PGQL](#)
- [Supported PGQL Features](#)
The In-Memory graph server (PGX) supports all PGQL features except `DROP PROPERTY GRAPH`.
- [Java APIs for Executing CREATE PROPERTY GRAPH Statements](#)
- [Python APIs for Executing CREATE PROPERTY GRAPH Statements](#)
- [Java APIs for Executing SELECT Queries](#)
This section describes the APIs to execute `SELECT` queries in the In-Memory graph server (PGX).
- [Java APIs for Executing UPDATE Queries](#)
The `UPDATE` queries make changes to existing graphs using the `INSERT`, `UPDATE`, and `DELETE` operations as detailed in the section Graph Modification of the PGQL 1.3 specification.
- [Security Tools for Executing PGQL Queries](#)
To safeguard against query injection, bind variables can be used in place of literals while `printIdentifier(String identifier)` can be used in place of identifiers like graph names, labels, and property names.
- [Best Practices for Tuning PGQL Queries](#)
This section describes best practices regarding memory allocation, parallelism, and query planning.

6.7.1 Getting Started with PGQL

This section provides an example on how to get started with PGQL. It assumes a database realm that has been previously set up (follow the steps in [Prepare the Graph Server for Database Authentication](#)). It also assumes that the user has `read` access to the HR schema.

First, create a graph with employees, departments, and `employee` works at department, by executing a `CREATE PROPERTY GRAPH` statement.

Example 6-1 Creating a graph in the in-memory graph server (PGX)

The following statement creates a graph in the in-memory graph server (PGX)

```
String statement =
    "CREATE PROPERTY GRAPH hr_simplified "
    + " VERTEX TABLES ( "
    + "   hr.employees LABEL employee "
    + "   PROPERTIES ARE ALL COLUMNS EXCEPT ( job_id, manager_id,
department_id ), "
    + "   hr.departments LABEL department "
    + "   PROPERTIES ( department_id, department_name ) "
```

```

+ " ) "
+ " EDGE TABLES ( "
+ "     hr.employees AS works_at "
+ "     SOURCE KEY ( employee_id ) REFERENCES employees "
+ "     DESTINATION departments "
+ "     PROPERTIES ( employee_id ) "
+ " );";
session.executePgql(statement);

/**
 * To get a handle to the graph, execute:
 */
PgxGraph g = session.getGraph("HR_SIMPLIFIED");

/**
 * You can use this handle to run PGQL queries on this graph.
 * For example, to find the department that "Nandita Sarchand" works for,
 * execute:
 */
String query =
    "SELECT dep.department_name "
    + "FROM MATCH (emp:Employee) -[:works_at]-> (dep:Department) "
    + "WHERE emp.first_name = 'Nandita' AND emp.last_name = 'Sarchand' "
    + "ORDER BY 1";
PgqlResultSet resultSet = g.queryPgql(query);
resultSet.print();
+-----+
| department_name |
+-----+
| Shipping        |
+-----+

/**
 * To get an overview of the types of vertices and their frequencies,
 * execute:
 */
String query =
    "SELECT label(n), COUNT(*) "
    + "FROM MATCH (n) "
    + "GROUP BY label(n) "
    + "ORDER BY COUNT(*) DESC";
PgqlResultSet resultSet = g.queryPgql(query);
resultSet.print();

+-----+
| label(n)      | COUNT(*) |
+-----+
| EMPLOYEE     | 107      |
| DEPARTMENT   | 27       |
+-----+

/**
 *To get an overview of the types of edges and their frequencies, execute:
 */
String query =

```

```

"SELECT label(n) AS srcLbl, label(e) AS edgeLbl, label(m) AS
dstLbl, COUNT(*) "
+ "FROM MATCH (n) -[e]-> (m) "
+ "GROUP BY srcLbl, edgeLbl, dstLbl "
+ "ORDER BY COUNT(*) DESC";
PgqlResultSet resultSet = g.queryPgql(query);
resultSet.print();

```

```

+-----+
| srcLbl | edgeLbl | dstLbl | COUNT(*) |
+-----+
| EMPLOYEE | WORKS_AT | DEPARTMENT | 106 |
+-----+

```

6.7.2 Supported PGQL Features

The In-Memory graph server (PGX) supports all PGQL features except `DROP PROPERTY GRAPH`.

Few features have certain limitations that are described below.

- [Limitations on Quantifiers](#)
Although all quantifiers such as `*`, `+`, and `{1,4}` are supported for reachability and shortest path patterns, the only quantifier that is supported for cheapest path patterns is `*` (zero or more).
- [Limitations on WHERE and COST Clauses in Quantified Patterns](#)

6.7.2.1 Limitations on Quantifiers

Although all quantifiers such as `*`, `+`, and `{1,4}` are supported for reachability and shortest path patterns, the only quantifier that is supported for cheapest path patterns is `*` (zero or more).

6.7.2.2 Limitations on WHERE and COST Clauses in Quantified Patterns

The `WHERE` and `COST` clauses in quantified patterns, such as reachability patterns or shortest and cheapest path patterns, are limited to referencing a single variable only.

The following are examples of queries that are not supported because the `WHERE` or `COST` clauses reference two variables `e` and `x` instead of zero or one:

```

... PATH p AS (n) -[e]-> (m) WHERE e.prop > m.prop ...
... SHORTEST ( (n) (-[e]-> (x) WHERE e.prop + x.prop > 10)* (m) ) ...
... CHEAPEST ( (n) (-[e]-> (x) COST e.prop + x.prop )* (m) ) ...

```

The following query is supported because the subquery only references a single variable `a` from the outer scope, while the variable `c` does not count since it is newly introduced in the subquery:

```

... PATH p AS (a) -> (b)
      WHERE EXISTS ( SELECT * FROM MATCH (a) -> (c) ) ...

```

6.7.3 Java APIs for Executing CREATE PROPERTY GRAPH Statements

The easiest way to execute a CREATE PROPERTY GRAPH statement is through the `PgxSession.executePgql(String statement)` method.

Example 6-2 Executing a CREATE PROPERTY GRAPH statement

```
String statement =
    "CREATE PROPERTY GRAPH hr_simplified "
    + " VERTEX TABLES ( "
    + "   hr.employees LABEL employee "
    + "   PROPERTIES ARE ALL COLUMNS EXCEPT ( job_id, manager_id,
department_id ), "
    + "   hr.departments LABEL department "
    + "   PROPERTIES ( department_id, department_name ) "
    + " ) "
    + " EDGE TABLES ( "
    + "   hr.employees AS works_at "
    + "   SOURCE KEY ( employee_id ) REFERENCES employees "
    + "   DESTINATION departments "
    + "   PROPERTIES ( employee_id ) "
    + " )";
session.executePgql(statement);
PgxGraph g = session.getGraph("HR_SIMPLIFIED");

/**
 * Alternatively, one can use the prepared statement API, for example:
 */

PgxPreparedStatement stmt = session.preparePgql(statement);
stmt.execute();
stmt.close();
PgxGraph g = session.getGraph("HR_SIMPLIFIED");
```

6.7.4 Python APIs for Executing CREATE PROPERTY GRAPH Statements

You can create a property graph by executing the CREATE PROPERTY GRAPH statement through the Python API.

Creating a Property Graph Using the Python Client

- Launch the Python client:

```
./bin/opg4py --base_url https://localhost:7007 --user customer_360
```

- Define and execute the CREATE PROPERTY GRAPH statement as shown:

```
statement = (
    "CREATE PROPERTY GRAPH "+ "<graph_name>" + " " +
    "VERTEX TABLES ( " +
    "bank_accounts " +
    "KEY(acct_id) " +
    "LABEL Account PROPERTIES (acct_id) " +
```

```

)" +
"EDGE TABLES ( " +
"bank_txns " +
"KEY (txn_id) " +
"SOURCE KEY (from_acct_id) REFERENCES bank_accounts " +
"DESTINATION KEY (to_acct_id) REFERENCES bank_accounts " +
"LABEL Transfer PROPERTIES(amount) " +
")" )
>>> session.prepare_pgql(statement).execute()

```

where `<graph_name>` is the name of the graph.

The graph gets created and you can verify through the `get_graph` method:

```

>>> graph = session.get_graph("<graph_name>")
>>> graph
PgxGraph(name:<graph_variable>, v: 1000, e: 5001, directed: True,
memory(Mb): 0)

```

6.7.5 Java APIs for Executing SELECT Queries

This section describes the APIs to execute `SELECT` queries in the In-Memory graph server (PGX).

- [Executing SELECT Queries Against a Graph in the In-memory Graph Server \(PGX\)](#)
The `PgxGraph.queryPgql(String query)` method executes the query in the current session. The method returns a `PgqlResultSet`.
- [Executing SELECT Queries Against a PGX Session](#)
The `PgxSession.queryPgql(String query)` method executes the given query in the session and returns a `PgqlResultSet`.
- [Iterating Through a Result Set](#)
There are two ways to iterate through a result set: in a JDBC-like manner or using the Java Iterator interface.
- [Printing a Result Set](#)
The following methods of `PgqlResultSet` (package `oracle.pgx.api`) are used to print a result set:

6.7.5.1 Executing SELECT Queries Against a Graph in the In-memory Graph Server (PGX)

The `PgxGraph.queryPgql(String query)` method executes the query in the current session. The method returns a `PgqlResultSet`.

The `ON` clauses inside the `MATCH` clauses can be omitted since the query is executed directly against a PGX graph. For the same reason, the `INTO` clauses inside the `INSERT` clauses can be omitted. However, if you want to explicitly specify graph names in the `ON` and `INTO` clauses, then those graph names have to match the actual name of the graph (`PgxGraph.getName()`).

6.7.5.2 Executing SELECT Queries Against a PGX Session

The `PgxSession.queryPgql(String query)` method executes the given query in the session and returns a `PgqlResultSet`.

The `ON` clauses inside the `MATCH` clauses, and the `INTO` clauses inside the `INSERT` clauses, must be specified and cannot be omitted. At this moment, all the `ON` and `INTO` clauses of a query need to reference the same graph since joining data from multiple graphs in a single query is not yet supported.

6.7.5.3 Iterating Through a Result Set

There are two ways to iterate through a result set: in a JDBC-like manner or using the Java Iterator interface.

For JDBC-like iterations, the methods in `PgqlResultSet` (package `oracle.pgx.api`) are similar to the ones in `java.sql.ResultSet`. A noteworthy difference is that PGQL's result set interface is based on the new date and time library that was introduced in Java 8, while `java.sql.ResultSet` is based on the legacy `java.util.Date`. To bridge the gap, PGQL's result set provides `getLegacyDate(...)` for applications that still use `java.util.Date`.

A `PgqlResultSet` has a cursor that is initially set before the first row. Then, the following methods are available to reposition the cursor:

- `next() : boolean`
- `previous() : boolean`
- `beforeFirst()`
- `afterLast()`
- `first() : boolean`
- `last() : boolean`
- `absolute(long row) : boolean`
- `relative(long rows) : boolean`

Above methods are documented in more detail [here](#).

After the cursor is positioned at the desired row, the following getters are used to obtain values:

- `getObject(int columnIndex) : Object`
- `getObject(String columnName) : Object`
- `getString(int columnIndex) : String`
- `getString(String columnName) : String`
- `getInteger(int columnIndex) : Integer`
- `getInteger(String columnName) : Integer`
- `getLong(int columnIndex) : Long`
- `getLong(String columnName) : Long`
- `getFloat(int columnIndex) : Float`

- `getFloat(String columnName) : Float`
- `getDouble(int columnIdx) : Double`
- `getDouble(String columnName) : Double`
- `getBoolean(int columnIdx) : Boolean`
- `getBoolean(String columnName) : Boolean`
- `getVertexLabels(int columnIdx) : Set<String>`
- `getVertexLabels(String columnName) : Set<String>`
- `getDate(int columnIdx) : LocalDate`
- `getDate(String columnName) : LocalDate`
- `getTime(int columnIdx) : LocalTime`
- `getTime(String columnName) : LocalTime`
- `getTimestamp(int columnIdx) : LocalDateTime`
- `getTimestamp(String columnName) : LocalDateTime`
- `getTimeWithTimezone(int columnIdx) : OffsetTime`
- `getTimeWithTimezone(String columnName) : OffsetTime`
- `getTimestampWithTimezone(int columnIdx) : OffsetDateTime`
- `getTimestampWithTimezone(String columnName) : OffsetDateTime`
- `getLegacyDate(int columnIdx) : java.util.Date`
- `getLegacyDate(String columnName) : java.util.Date`
- `getVertex(int columnIdx) : PgxVertex<ID>`
- `getVertex(String columnName) : PgxVertex<ID>`
- `getEdge(int columnIdx) : PgxEdge`
- `getEdge(String columnName) : PgxEdge`

Above methods are documented in more detail [here](#).

Finally, there is a `PgqlResultSet.close()` which releases the result set's resources, and there is a `PgqlResultSet.getMetaData()` through which the column names and column count can be retrieved.

An example for result set iteration is as follows:

```
PgqlResultSet resultSet = g.queryPgql(
    " SELECT owner.name AS account_holder, SUM(t.amount) AS
total_transacted_with_Nikita "
    + " FROM MATCH (p:Person) -[:ownerOf]-> (account1:Account) "
    + " , MATCH (account1) -[t:transaction]- (account2) "
    + " , MATCH (account2:Account) <-[:ownerOf]- (owner:Person|
Company) "
    + " WHERE p.name = 'Nikita' "
    + " GROUP BY owner");

while (resultSet.next()) {
    String accountHolder = resultSet.getString(1);
```



```

    long totalTransacted = resultSet.getLong(2);
    System.out.println(accountHolder + ": " + totalTransacted);
}

resultSet.close();

```

The output of the above example will look like:

```

Oracle: 4501
Camille: 1000

```

In addition, the `PgqlResultSet` is also iterable via the Java Iterator interface. An example of a “for each loop” over the result set is as follows:

```

for (PgxResult result : resultSet) {
    String accountHolder = result.getString(1);
    long totalTransacted = result.getLong(2);
    System.out.println(accountHolder + ": " + totalTransacted);
}

```

The output of the above example will look like:

```

Oracle: 4501
Camille: 1000

```

Note that the same getters that are available for `PgqlResultSet` are also available for `PgxResult`.

6.7.5.4 Printing a Result Set

The following methods of `PgqlResultSet` (package `oracle.pgx.api`) are used to print a result set:

- `print() : PgqlResultSet`
- `print(long numResults) : PgqlResultSet`
- `print(long numResults, int from) : PgqlResultSet`
- `print(PrintStream printStream, long numResults, int from) : PgqlResultSet`

For example:

```

g.queryPgql("SELECT COUNT(*) AS numPersons FROM MATCH
(n:Person)").print().close()
+-----+
| numPersons |
+-----+
| 3          |
+-----+

```

Another example:

```
PgqlResultSet resultSet = g.queryPgql(
    " SELECT owner.name AS account_holder, SUM(t.amount) AS
total_transacted_with_Nikita "
    + " FROM MATCH (p:Person) -[:ownerOf]-> (account1:Account) "
    + " , MATCH (account1) -[t:transaction]- (account2) "
    + " , MATCH (account2:Account) <-[:ownerOf]- (owner:Person|
Company) "
    + " WHERE p.name = 'Nikita' "
    + " GROUP BY owner")
```

```
resultSet.print().close()
```

```
+-----+
| account_holder | total_transacted_with_Nikita |
+-----+
| Camille       | 1000.0                       |
| Oracle        | 4501.0                       |
+-----+
```

6.7.6 Java APIs for Executing UPDATE Queries

The `UPDATE` queries make changes to existing graphs using the `INSERT`, `UPDATE`, and `DELETE` operations as detailed in the section Graph Modification of the PGQL 1.3 specification.

Note that `INSERT` allows you to insert new vertices and edges into a graph, `UPDATE` allows you to update existing vertices and edges by setting their properties to new values, and `DELETE` allows you to delete vertices and edges from a graph.

- [Updatability of Graphs Through PGQL](#)
Graph data that is loaded from the Oracle RDBMS or from CSV files into the PGX is not updatable through PGQL right away.
- [Executing UPDATE Queries against a Graph in the in-memory Graph Server \(PGX\)](#)
To execute `UPDATE` queries against a graph, use the `PgxGraph.executePgql(String query)` method.
- [Executing UPDATE Queries Against a PGX Session](#)
For now, there is no support for executing `UPDATE` queries against a `PgxSession` and therefore, updates always have to be executed against a `PgxGraph`. To obtain a graph from a session, use the `PgxSession.getGraph(String graphName)` method.
- [Altering the Underlying Schema of a Graph](#)
The `INSERT` operations can only insert vertices and edges with known labels and properties. Similarly, `UPDATE` operations can only set values of known properties. Thus, new data must always conform to the existing schema of the graph.

6.7.6.1 Updatability of Graphs Through PGQL

Graph data that is loaded from the Oracle RDBMS or from CSV files into the PGX is not updatable through PGQL right away.

First, you need to create a copy of the data through the `PgxGraph.clone()` method. The resulting graph is fully updatable.

Consider the following example:

```
// load a graph from the RDBMS or from CSV
PgxGraph g1 = session.readGraphWithProperties("path/to/graph_config.json");

// create an updatable copy of the graph
PgxGraph g2 = g1.clone("new_graph_name");

// insert an additional vertex into the graph
g2.executePgql("INSERT VERTEX v " +
               "          LABELS ( Person ) " +
               "          PROPERTIES ( v.firstName = 'Camille', " +
               "          v.lastName = ' Mullins' )");
```

Additionally, there is also a `PgxGraph.cloneAndExecutePgql(String query, String graphName)` method that combines the last two steps from above example into a single step:

```
// create an updatable copy of the graph while inserting a new vertex
PgxGraph g2_copy = g1.cloneAndExecutePgql(
    "INSERT VERTEX v " +
    "          LABELS ( Person ) " +
    "          PROPERTIES ( v.firstName = 'Camille', " +
    "          v.lastName = ' Mullins' ) "
    , "new_graph_name");
```

Note that graphs that are created through `PgxGraph.clone()` are local to the session. However, they can be shared with other sessions through the `PgxGraph.publish(..)` methods but then they are no longer updatable through PGQL. Only session-local graphs are updatable but persistent graphs are not.

6.7.6.2 Executing UPDATE Queries against a Graph in the in-memory Graph Server (PGX)

To execute UPDATE queries against a graph, use the `PgxGraph.executePgql(String query)` method.

The following is an example of INSERT query:

```
g.executePgql("INSERT VERTEX v " +
              "          LABELS ( Person ) " +
              "          PROPERTIES ( v.firstName = 'Camille', " +
              "          v.lastName = ' Mullins' ) ");
```

Note that the `INTO` clause of the `INSERT` can be omitted. If you use an `INTO` clause, the graph name in the `INTO` clause must correspond to the name of the PGX graph (`PgxGraph.getName()`) that the query is executed against.

The following is an example of UPDATE query:

```
// set the date of birth of Camille to 2014-11-15
g.executePgql("UPDATE v SET ( v.dob = DATE '2014-11-14' ) " +
             "FROM MATCH (v:Person) " +
             "WHERE v.firstName = 'Camille' AND v.lastName = 'Mullins' ");
```

The following is an example of DELETE query:

```
// delete Camille from the graph
g.executePgql("DELETE v " +
             "FROM MATCH (v:Person) " +
             "WHERE v.firstName = 'Camille' AND v.lastName = 'Mullins' ");
```

6.7.6.3 Executing UPDATE Queries Against a PGX Session

For now, there is no support for executing UPDATE queries against a `PgxSession` and therefore, updates always have to be executed against a `PgxGraph`. To obtain a graph from a session, use the `PgxSession.getGraph(String graphName)` method.

6.7.6.4 Altering the Underlying Schema of a Graph

The INSERT operations can only insert vertices and edges with known labels and properties. Similarly, UPDATE operations can only set values of known properties. Thus, new data must always conform to the existing schema of the graph.

However, some PGX APIs exist for updating the schema of a graph: while no APIs exist for adding new labels, new properties can be added through the `PgxGraph.createVertexProperty(PropertyType type, String name)` and `PgxGraph.createEdgeProperty(PropertyType type, String name)` methods. The new properties are attached to each vertex/edge in the graph, irrespective of their labels. Initially the properties are assigned a default value but then the values can be updated through the UPDATE statements.

Consider the following example:

```
// load a graph from the RDBMS or from CSV
PgxGraph g = session.readGraphWithProperties("path/to/graph_config.json");

// add a new property to the graph
g.createVertexProperty(PropertyType.LOCAL_DATE, "dob");

// set the date of birth of Camille to 2014-11-15
g.executePgql("UPDATE v SET ( v.dob = DATE '2014-11-14' ) " +
             "FROM MATCH (v:Person) " +
             "WHERE v.firstName = 'Camille' AND v.lastName = 'Mullins' ");
```

6.7.7 Security Tools for Executing PGQL Queries

To safeguard against query injection, bind variables can be used in place of literals while `printIdentifier(String identifier)` can be used in place of identifiers like graph names, labels, and property names.

- [Using Bind Variables](#)

There are two reasons for using bind variables:

- [Using Identifiers in a Safe Manner](#)

When you create a query through string concatenation, not only literals in queries pose a security risk, but also identifiers like graph names, labels, and property names do. The only problem is that bind variables are not supported for such identifier. Therefore, if these identifiers are variable from the application's perspective, then it is recommended to protect against query injection by passing the identifier through the `oracle.pgql.lang.ir.PgqlUtils.printIdentifier(String identifier)` method.

6.7.7.1 Using Bind Variables

There are two reasons for using bind variables:

- It protects against query injection.
- It speeds up query execution because the same bind variables can be set multiple times without requiring recompilation of the query.

To create a prepared statement, use one of the following two methods:

- `PgxGraph.preparePgql(String query) : PgxPreparedStatement`
- `PgxSession.preparePgql(String query) : PgxPreparedStatement`

The `PgxPreparedStatement` (package `oracle.pgx.api`) returned from these methods have setter methods for binding the bind variables to values of the designated data type.

```
PreparedStatement stmt = g.preparePgql(
    "SELECT v.id, v.dob " +
    "FROM MATCH (v) " +
    "WHERE v.firstName = ? AND v.lastName = ?");
stmt.setString(1, "Camille");
stmt.setString(2, "Mullins");
ResultSet rs = stmt.executeQuery();
```

Each bind variable in the query needs to be set to a value using one of the following setters of `PgxPreparedStatement`:

- `setBoolean(int parameterIndex, boolean x)`
- `setDouble(int parameterIndex, double x)`
- `setFloat(int parameterIndex, float x)`
- `setInt(int parameterIndex, int x)`
- `setLong(int parameterIndex, long x)`
- `setDate(int parameterIndex, LocalDate x)`

- setTime(int parameterIndex, LocalTime x)
- setTimestamp(int parameterIndex, LocalDateTime x)
- setTimeWithTimezone(int parameterIndex, OffsetTime x)
- setTimestampWithTimezone(int parameterIndex, OffsetDateTime x)
- setArray(int parameterIndex, List<?> x)

Once all the bind variables are set, the statement can be executed through:

- PgxPreparedStatement.executeQuery()
 - For SELECT queries only
 - Returns a ResultSet
- PgxPreparedStatement.execute()
 - For any type of statement
 - Returns a Boolean to indicate the form of the result: true in case of a SELECT query, false otherwise
 - In case of SELECT, the ResultSet can afterwards be accessed through PgxPreparedStatement.getResultSet()

In PGQL, bind variables can be used in place of literals of any data type, including array literals. An example query with a bind variable to is set to an instance of a String array is:

```
List<String> countryNames = new ArrayList<String>();
countryNames.add("Scotland");
countryNames.add("Tanzania");
countryNames.add("Serbia");
```

```
PreparedStatement stmtnt = g.preparePgql(
    "SELECT n.name, n.population " +
    "FROM MATCH (c:Country) " +
    "WHERE c.name IN ?");
```

```
ResultSet rs = stmtnt.executeQuery();
```

Finally, if a prepared statement is no longer needed, it is closed through PgxPreparedStatement.close() to free up resources.

6.7.7.2 Using Identifiers in a Safe Manner

When you create a query through string concatenation, not only literals in queries pose a security risk, but also identifiers like graph names, labels, and property names do. The only problem is that bind variables are not supported for such identifier. Therefore, if these identifiers are variable from the application's perspective, then it is recommended to protect against query injection by passing the identifier through the oracle.pgql.lang.ir.PgqlUtils.printIdentifier(String identifier) method.

Given an identifier string, the method automatically adds double quotes to the start and end of the identifier and escapes the characters in the identifier appropriately.

Consider the following example:

```
String graphNamePrinted = printIdentifier("my graph name with \" special %  
characters ");  
PreparedStatement stmtnt = g.preparePgql(  
    "SELECT COUNT(*) AS numVertices FROM MATCH (v) ON " + graphNamePrinted);
```

6.7.8 Best Practices for Tuning PGQL Queries

This section describes best practices regarding memory allocation, parallelism, and query planning.

- **Memory Allocation**
The In-Memory Analyst (PGX) has on-heap and off-heap memory, the earlier being the standard JVM heap while the latter being a separate heap that is managed by PGX. Just like graph data, intermediate and final results of PGQL queries are partially stored on-heap and partially off-heap. Therefore, both heaps are needed.
- **Parallelism**
By default, all available processor threads are used to process PGQL queries. However, if needed, the number of threads can be limited by setting the `parallelism` option of the In-Memory Analyst (PGX).
- **Query Plan Explaining**
The `PgxGraph.explainPgql(String query)` method is used to get insight into the query plan of the query. The method returns an instance of `Operation` (package `oracle.pgx.api`) which has the following methods:

6.7.8.1 Memory Allocation

The In-Memory Analyst (PGX) has on-heap and off-heap memory, the earlier being the standard JVM heap while the latter being a separate heap that is managed by PGX. Just like graph data, intermediate and final results of PGQL queries are partially stored on-heap and partially off-heap. Therefore, both heaps are needed.

In case of the on-heap memory, the default maximum is chosen upon startup of the JVM, but it can be overwritten through the `-Xmx` option.

In case of the off-heap, there is no maximum set by default and the off-heap memory usage, therefore, keeps increasing automatically until it depletes the system resources, in which case the operation is cancelled, it's memory is released, and an appropriate exception is passed to the user. If needed, a maximum off-heap size can be configured through the `max_off_heap_size` option of PGX.

A ratio of 1:1 for on-heap vs. off-heap is recommended as a good starting point to allow for the largest possible graphs to be loaded and queried. For example, if you have 256 GB of memory available on your machine, then setting the maximum on-heap to 125 GB will make sure that there is a similar amount of memory available for off-heap:

```
export JAVA_OPTS="-Xmx125g"
```

6.7.8.2 Parallelism

By default, all available processor threads are used to process PGQL queries. However, if needed, the number of threads can be limited by setting the `parallelism` option of the In-Memory Analyst (PGX).

See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information on the graph server configuration parameters.

6.7.8.3 Query Plan Explaining

The `PgxGraph.explainPgql(String query)` method is used to get insight into the query plan of the query. The method returns an instance of `Operation` (package `oracle.pgx.api`) which has the following methods:

- `print()`: for printing the operation and its child operations
- `getOperationType()`: for getting the type of the operation
- `getPatternInfo()`: for getting a string representation of the operation
- `getCostEstimate()`: for getting the cost of the operation
- `getTotalCostEstimate()`: for getting the cost of the operations and its child operations
- `getCardinalityEstimate()`: for getting the expected number of result rows
- `getChildren()`: for accessing the child operations

Consider the following example:

```
g.explainPgql("SELECT COUNT(*) FROM MATCH (n) -[e1]-> (m) -[e2]->
(o)").print()
\--- GROUP BY GroupBy {"cardinality":"42", "cost":"42",
"accumulatedCost":"58.1"}
  \--- (m) -[e2]-> (o) NeighborMatch {"cardinality":"3.12",
"cost":"3.12", "accumulatedCost":"16.1"}
    \--- (n) -[e1]-> (m) NeighborMatch {"cardinality":"5",
"cost":"5", "accumulatedCost":"13"}
      \--- (n) RootVertexMatch {"cardinality":"8", "cost":"8",
"accumulatedCost":"8" }
```

In the above example, the `print()` method is used to print the query plan.

If a query plan is not optimal, it is often possible to rewrite the query to improve its performance. For example, a `SELECT` query may be split into an `UPDATE` and a `SELECT` query as a way to improve the total runtime.

Note that the In-Memory Analyst (PGX) does not provide a hint mechanism.

6.8 Executing PGQL Queries Directly Against Oracle Database

This topic explains how you can execute PGQL queries directly against the graph in Oracle Database (as opposed to in-memory).

Property Graph Query Language (PGQL) queries can be executed against disk-resident property graph data stored in Oracle Database. PGQL on Oracle Database (RDBMS) provides a Java API for executing PGQL queries. Logic in PGQL on RDBMS translates a submitted PGQL query into an equivalent SQL query, and the

resulting SQL is executed on the database server. PGQL on RDBMS then wraps the SQL query results with a convenient PGQL result set API.

Property graph data in RDBMS can exist either in the property graph schema tables or as a property graph view on the Oracle Database tables.

The following topics explain in detail how you can execute PGQL queries against the graph in the Oracle Database:

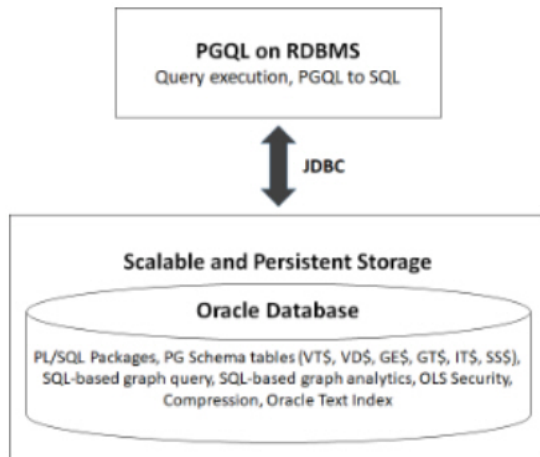
- [Executing PGQL Queries Against Property Graph Schema Tables](#)
This topic explains how you can execute PGQL queries directly against the graph stored in property graph schema tables.
- [Executing PGQL Queries Against Property Graph Views](#)
This topic explains how you can execute PGQL queries directly against the property graph views on Oracle Database tables.

6.8.1 Executing PGQL Queries Against Property Graph Schema Tables

This topic explains how you can execute PGQL queries directly against the graph stored in property graph schema tables.

The PGQL query execution flow is shown in the following figure.

Figure 6-1 PGQL on Property Graph Schema Tables in Oracle Database (RDBMS)



The basic execution flow is:

1. The PGQL query is submitted to PGQL on RDBMS through a Java API.
2. The PGQL query is translated to SQL.
3. The translated SQL is submitted to Oracle Database by JDBC.
4. The SQL result set is wrapped as a PGQL result set and returned to the caller.

The ability to execute PGQL queries directly against property graph data stored in Oracle Database provides several benefits.

- PGQL provides a more natural way to express graph queries than SQL manually written to query schema tables, including VT\$, VD\$, GE\$, and GT\$.

- PGQL queries can be executed without the need to load a snapshot of your graph data into PGX, so there is no need to worry about staleness of frequently updated graph data.
- PGQL queries can be executed against graph data that is too large to fit in memory.
- The robust and scalable Oracle SQL engine can be used to execute PGQL queries.
- Mature tools for management, monitoring and tuning of Oracle Database can be used to tune and monitor PGQL queries.
- [PGQL Features Supported](#)
- [Creating Property Graphs through CREATE PROPERTY GRAPH Statements](#)
- [Dropping Property Graphs through DROP PROPERTY GRAPH Statements](#)
- [Using the oracle.pg.rdbms.pgql Java Package to Execute PGQL Queries](#)
- [Using the Python Client to Execute PGQL Queries](#)
- [Performance Considerations for PGQL Queries](#)

6.8.1.1 PGQL Features Supported

[PGQL](#) is a SQL-like query language for querying property graph data. It is based on the concept of graph pattern matching and allows you to specify, among other things, topology constraints, paths, filters, sorting and aggregation.

The Java API for PGQL defined in the `oracle.pg.rdbms.pgql` package supports the PGQL specification with a few exceptions. (The PGQL specification can be found at <https://pgql-lang.org>).

The following features of PGQL are not supported.

- Shortest path
- `ARRAY_AGG` aggregation
- Single `CHEAPEST` path and `TOP-K CHEAPEST` path using `COST` functions
- Case-insensitive matching of uppercased references to labels and properties

In addition, the following features of PGQL require special consideration.

- [Temporal Types](#)
- [Type Casting](#)
- [CONTAINS Built-in Function](#)

6.8.1.1.1 Temporal Types

The temporal types `DATE`, `TIMESTAMP` and `TIMESTAMP WITH TIMEZONE` are supported in PGQL queries.

All of these value types are represented internally using the Oracle SQL `TIMESTAMP WITH TIME ZONE` type. `DATE` values are automatically converted to `TIMESTAMP WITH TIME ZONE` by assuming the earliest time in UTC+0 timezone (for example, 2000-01-01 becomes 2000-01-01 00:00:00.00+00:00). `TIMESTAMP` values are automatically converted to `TIMESTAMP WITH TIME ZONE` by assuming UTC+0

timezone (for example, 2000-01-01 12:00:00.00 becomes 2000-01-01 12:00:00.00+00:00).

Temporal constants are written in PGQL queries as follows.

- DATE 'YYYY-MM-DD'
- TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.FF'
- TIMESTAMP WITH TIMEZONE 'YYYY-MM-DD HH24:MI:SS.FFTZH:TZM'

Some examples are DATE '2000-01-01', TIMESTAMP '2000-01-01 14:01:45.23',
 TIMESTAMP WITH TIMEZONE '2000-01-01 13:00:00.00-05:00', and TIMESTAMP WITH
 TIMEZONE '2000-01-01 13:00:00.00+01:00'.

In addition, temporal values can be obtained by casting string values to a temporal type. The supported string formats are:

- DATE 'YYYY-MM-DD'
- TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.FF' and 'YYYY-MM-DD"T"HH24:MI:SS.FF'
- TIMESTAMP WITH TIMEZONE 'YYYY-MM-DD HH24:MI:SS.FFTZH:TZM' and 'YYYY-MM-DD"T"HH24:MI:SS.FFTZH:TZM'.

Some examples are CAST ('2005-02-04' AS DATE), CAST ('1990-01-01 12:00:00.00' AS
 TIMESTAMP), CAST ('1985-01-01T14:05:05.00-08:00' AS TIMESTAMP WITH TIMEZONE).

When consuming results from a `PgqlResultSet` object, `getObject` returns a
`java.sql.Timestamp` object for temporal types.

Bind variables can only be used for the TIMESTAMP WITH TIMEZONE temporal type in
 PGQL, and a `setTimestamp` method that takes a `java.sql.Timestamp` object as input is used
 to set the bind value. As a simpler alternative, you can use a string bind variable in a CAST
 statement to bind temporal values (for example, CAST (? AS TIMESTAMP WITH TIMEZONE)
 followed by `setString(1, "1985-01-01T14:05:05.00-08:00")`). See also [Using Bind
 Variables in PGQL Queries](#) for more information about bind variables.

6.8.1.1.2 Type Casting

Type casting is supported in PGQL with a SQL-style CAST (VALUE AS DATATYPE) syntax,
 for example CAST('25' AS INT), CAST(10 AS STRING), CAST('2005-02-04' AS DATE),
 CAST(e.weight AS STRING). Supported casting operations are summarized in the following
 table. Y indicates that the conversion is supported, and N indicates that it is not supported.
 Casting operations on invalid values (for example, CAST('xyz' AS INT)) or unsupported
 conversions (for example, CAST(10 AS TIMESTAMP)) return NULL instead of raising a SQL
 exception.

Table 6-2 Type Casting Support in PGQL (From and To Types)

"to" type	from STRIN G	from INT	from LONG	from FLOA T	from DOUBL E	from BOOLEA N	from DATE	from TIMESTAM P	from TIMESTAM P WITH TIMEZONE
to STRING	Y	Y	Y	Y	Y	Y	Y	Y	Y
to INT	Y	Y	Y	Y	Y	Y	N	N	N
to LONG	Y	Y	Y	Y	Y	Y	N	N	N
to FLOAT	Y	Y	Y	Y	Y	Y	N	N	N
to DOUBLE	Y	Y	Y	Y	Y	Y	N	N	N

Table 6-2 (Cont.) Type Casting Support in PGQL (From and To Types)

“to” type	from STRIN G	from INT	from LONG	from FLOA T	from DOUBL E	from BOOLEA N	from DATE	from TIMESTAM P	from TIMESTAM P WITH TIMEZONE
to BOOLEAN	Y	Y	Y	Y	Y	Y	N	N	N
to DATE	Y	N	N	N	N	N	Y	Y	Y
to TIMESTAM P	Y	N	N	N	N	N	Y	Y	Y
to TIMESTAM P WITH TIMEZONE	Y	N	N	N	N	N	Y	Y	Y

An example query that uses type casting is:

```
SELECT e.name, CAST (e.birthDate AS STRING) AS dob
FROM MATCH (e)
WHERE e.birthDate < CAST ('1980-01-01' AS DATE)
```

6.8.1.1.3 CONTAINS Built-in Function

A CONTAINS built-in function is supported. It is used in conjunction with an Oracle Text index on vertex and edge properties. CONTAINS returns `true` if a value matches an Oracle Text search string and `false` if it does not match.

An example query is:

```
SELECT v.name
FROM MATCH (v)
WHERE CONTAINS(v.abstract, 'Oracle')
```

See also [Using a Text Index with PGQL Queries](#) for more information about using full text indexes with PGQL.

6.8.1.2 Creating Property Graphs through CREATE PROPERTY GRAPH Statements

You can use PGQL to create property graphs from relational database tables. A CREATE PROPERTY GRAPH statement defines a set of vertex tables that are transformed into vertices and a set of edge tables that are transformed into edges. For each table a key, a label and a set of column properties can be specified. The column types CHAR, NCHAR, VARCHAR, VARCHAR2, NVARCHAR2, NUMBER, LONG, FLOAT, DATE, TIMESTAMP and TIMESTAMP WITH TIMEZONE are supported for CREATE PROPERTY GRAPH column properties.

When a CREATE PROPERTY GRAPH statement is called, a property graph schema for the graph is created, and the data is copied from the source tables into the property

graph schema tables. The graph is created as a one-time copy and is not automatically kept in sync with the source data.

Example 6-3 PgqlCreateExample1.java

This example shows how to create a property graph from a set of relational tables. Notice that the example creates tables Person, Hobby, and Hobbies, so they should not exist before running the example. The example also shows how to execute a query against a property graph.

```
import java.sql.Connection;
import java.sql.Statement;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to create a Property Graph from relational
 * data stored in Oracle Database executing a PGQL create statement.
 */
public class PgqlCreateExample1
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        Statement stmt = null;
        PgqlStatement pgqlStmt = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+"."+port+"."+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Create relational data
            stmt = conn.createStatement();
```

```

//Table Person
stmt.executeUpdate(
    "create table Person( " +
    " id NUMBER, " +
    " name VARCHAR2(20), " +
    " dob TIMESTAMP " +
    ")");

// Insert some data
stmt.executeUpdate("insert into Person values(1,'Alan', DATE
'1995-05-26')");
stmt.executeUpdate("insert into Person values(2,'Ben', DATE
'2007-02-15')");
stmt.executeUpdate("insert into Person values(3,'Claire', DATE
'1967-11-30')");

// Table Hobby
stmt.executeUpdate(
    "create table Hobby( " +
    " id NUMBER, " +
    " name VARCHAR2(20) " +
    ")");

// Insert some data
stmt.executeUpdate("insert into Hobby values(1, 'Sports')");
stmt.executeUpdate("insert into Hobby values(2, 'Music')");

// Table Hobbies
stmt.executeUpdate(
    "create table Hobbies( "+
    " person NUMBER, "+
    " hobby NUMBER, "+
    " strength NUMBER "+
    ")");

// Insert some data
stmt.executeUpdate("insert into Hobbies values(1, 1, 20)");
stmt.executeUpdate("insert into Hobbies values(1, 2, 30)");
stmt.executeUpdate("insert into Hobbies values(2, 1, 10)");
stmt.executeUpdate("insert into Hobbies values(3, 2, 20)");

//Commit changes
conn.commit();

// Get a PGQL connection
PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);

// Create a PgqlStatement
pgqlStmt = pgqlConn.createStatement();

// Execute PGQL to create property graph
String pgql =
    "Create Property Graph " + graph + " " +
    "VERTEX TABLES ( " +
    " Person " +

```


Example 6-4 PgqlCreateExample2.java

This example shows how a create property graph statement without specifying any keys. Notice that the example creates tables Person, Hobby, and Hobbies, so they should not exist before running the example.

```
import java.sql.Connection;
import java.sql.Statement;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to create a Property Graph from relational
 * data stored in Oracle Database executing a PGQL create statement.
 */
public class PgqlCreateExample2
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        Statement stmt = null;
        PgqlStatement pgqlStmt = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Create relational data
            stmt = conn.createStatement();

            //Table Person
            stmt.executeUpdate(
```



```
"create table Person( " +
" id NUMBER, " +
" name VARCHAR2(20), " +
" dob TIMESTAMP, " +
" CONSTRAINT pk_person PRIMARY KEY(id)" +
");

// Insert some data
stmt.executeUpdate("insert into Person values(1,'Alan', DATE
'1995-05-26')");
stmt.executeUpdate("insert into Person values(2,'Ben', DATE
'2007-02-15')");
stmt.executeUpdate("insert into Person values(3,'Claire', DATE
'1967-11-30')");

// Table Hobby
stmt.executeUpdate(
"create table Hobby( " +
" id NUMBER, " +
" name VARCHAR2(20), " +
" CONSTRAINT pk_hobby PRIMARY KEY(id)" +
");

// Insert some data
stmt.executeUpdate("insert into Hobby values(1, 'Sports')");
stmt.executeUpdate("insert into Hobby values(2, 'Music')");

// Table Hobbies
stmt.executeUpdate(
"create table Hobbies( "+
" person NUMBER, "+
" hobby NUMBER, "+
" strength NUMBER, "+
" CONSTRAINT fk_hobbies1 FOREIGN KEY (person) REFERENCES
Person(id), "+
" CONSTRAINT fk_hobbies2 FOREIGN KEY (hobby) REFERENCES Hobby(id)" +
");

// Insert some data
stmt.executeUpdate("insert into Hobbies values(1, 1, 20)");
stmt.executeUpdate("insert into Hobbies values(1, 2, 30)");
stmt.executeUpdate("insert into Hobbies values(2, 1, 10)");
stmt.executeUpdate("insert into Hobbies values(3, 2, 20)");

//Commit changes
conn.commit();

// Get a PGQL connection
PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);

// Create a PgqlStatement
pgqlStmt = pgqlConn.createStatement();

// Execute PGQL to create property graph
String pgql =
```

```

"Create Property Graph " + graph + " " +
"VERTEX TABLES ( " +
"  Person " +
"    Label people +
"    PROPERTIES ALL COLUMNS," +
"  Hobby " +
"    Label hobby PROPERTIES ALL COLUMNS EXCEPT(id)" +
")" +
"EDGE TABLES ( " +
"  Hobbies" +
"    SOURCE Person DESTINATION Hobby " +
"    LABEL likes NO PROPERTIES" +
")";
pgqlStmt.execute(pgql);

// Execute a PGQL query to verify Graph creation
pgql =
  "SELECT p.NAME AS person, p.DOB, h.NAME AS hobby " +
  "FROM MATCH (p:people)-[e:likes]->(h:hobby) ON " + graph;
rs = pgqlStmt.executeQuery(pgql, "");

// Print the results
rs.print();
}
finally {
  // close the sql statement
  if (stmt != null) {
    stmt.close();
  }
  // close the result set
  if (rs != null) {
    rs.close();
  }
  // close the statement
  if (pgqlStmt != null) {
    pgqlStmt.close();
  }
  // close the connection
  if (conn != null) {
    conn.close();
  }
}
}
}
}

```

The output for PgqlCreateExample2.java is:

```

+-----+
| PERSON | DOB | HOBBY |
+-----+
| Alan | 1995-05-25 17:00:00.0 | Music |
| Claire | 1967-11-29 16:00:00.0 | Music |
| Ben | 2007-02-14 16:00:00.0 | Sports |
| Alan | 1995-05-25 17:00:00.0 | Sports |
+-----+

```

6.8.1.3 Dropping Property Graphs through DROP PROPERTY GRAPH Statements

You can use PGQL to drop property graphs. When a `DROP PROPERTY GRAPH` statement is called, all the property graph schema tables of the graph are dropped.

Example 6-5 PqqlDropExample1.java

This example shows how to drop a property graph.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to drop a Property executing a PGQL drop statement.
 */
public class PqqlDropExample1
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement pqqlStmt = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Get a PGQL connection
            PgqlConnection pqqlConn = PgqlConnection.getConnection(conn);

            // Create a PgqlStatement
            pqqlStmt = pqqlConn.createStatement();

            // Execute PGQL to drop property graph
```

```
String pgql = "Drop Property Graph " + graph;
pgqlStmt.execute(pgql);

}
finally {
    // close the statement
    if (pgqlStmt != null) {
        pgqlStmt.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}
```

6.8.1.4 Using the oracle.pg.rdbms.pgql Java Package to Execute PGQL Queries

The Java API in the `oracle.pg.rdbms.pgql` package provides support for executing PGQL queries against Oracle Database. This topic explains how to use the Java API through a series of examples.

 **Note:**

Effective with Release 21c, the following classes in the `oracle.pg.rdbms` package are deprecated:

```
oracle.pg.rdbms.OraclePgqlColumnDescriptorImpl
oracle.pg.rdbms.OraclePgqlColumnDescriptor
oracle.pg.rdbms.OraclePgqlExecutionFactory
oracle.pg.rdbms.OraclePgqlExecution
oracle.pg.rdbms.PgqlPreparedStatement
oracle.pg.rdbms.OraclePgqlResultElementImpl
oracle.pg.rdbms.OraclePgqlResultElement
oracle.pg.rdbms.OraclePgqlResultImpl
oracle.pg.rdbms.OraclePgqlResultIterable
oracle.pg.rdbms.OraclePgqlResultIteratorImpl
oracle.pg.rdbms.OraclePgqlResult
oracle.pg.rdbms.OraclePgqlResultSetImpl
oracle.pg.rdbms.OraclePgqlResultSet
oracle.pg.rdbms.OraclePgqlResultSetMetaDataImpl
oracle.pg.rdbms.OraclePgqlResultSetMetaData
oracle.pg.rdbms.PgqlSqlQueryTransImpl
oracle.pg.rdbms.PgqlSqlQueryTrans
oracle.pg.rdbms.PgqlStatement
```

You should instead use equivalent classes in `oracle.pg.rdbms.pgql`:

```
oracle.pg.rdbms.pgql.PgqlColumnDescriptorImpl
oracle.pg.rdbms.pgql.PgqlColumnDescriptor
oracle.pg.rdbms.pgql.PgqlConnection
oracle.pg.rdbms.pgql.PgqlExecution
oracle.pg.rdbms.pgql.PgqlPreparedStatement
oracle.pg.rdbms.pgql.PgqlResultElementImpl
oracle.pg.rdbms.pgql.PgqlResultElement
oracle.pg.rdbms.pgql.PgqlResultSetImpl
oracle.pg.rdbms.pgql.PgqlResultSet
oracle.pg.rdbms.pgql.PgqlResultSetMetaDataImpl
oracle.pg.rdbms.pgql.PgqlSqlTransImpl
oracle.pg.rdbms.pgql.PgqlSqlTrans
oracle.pg.rdbms.pgql.PgqlStatement
```

One difference between `oracle.pg.rdbms.OraclePgqlResultSet` and `oracle.pg.rdbms.pgql.PgqlResultSet` is that `oracle.pg.rdbms.pgql.PgqlResultSet` does not provide APIs to retrieve vertex and edge objects. Existing code using those interfaces should be changed to project IDs rather than `OracleVertex` and `OracleEdge` objects. You can obtain an `OracleVertex` or `OracleEdge` object from the projected ID values by calling `OracleVertex.getInstance()` or `OracleEdge.getInstance()`. (For an example, see [Example 6-21](#).)

See [Oracle Graph Property Graph Java APIs](#) for more details on setting the classpath for compiling and executing your Java applications.

The following `test_graph` data set in Oracle flat file format will be used in the examples in subtopics that follow. The data set includes a vertex file (`test_graph.opv`) and an edge file (`test_graph.ope`).

`test_graph.opv`:

```
2, fname, 1, Ray, , , person
2, lname, 1, Green, , , person
2, mval, 5, , , 1985-01-01T12:00:00.000Z, person
2, age, 2, , 41, , person
0, bval, 6, Y, , , person
0, fname, 1, Bill, , , person
0, lname, 1, Brown, , , person
0, mval, 1, Y, , , person
0, age, 2, , 40, , person
1, bval, 6, Y, , , person
1, fname, 1, John, , , person
1, lname, 1, Black, , , person
1, mval, 2, , 27, , person
1, age, 2, , 30, , person
3, bval, 6, N, , , person
3, fname, 1, Susan, , , person
3, lname, 1, Blue, , , person
3, mval, 6, N, , , person
3, age, 2, , 35, , person
```

test_graph.ope:

```
4, 0, 1, knows, mval, 1, Y, ,
4, 0, 1, knows, firstMetIn, 1, MI, ,
4, 0, 1, knows, since, 5, , , 1990-01-01T12:00:00.000Z
16, 0, 1, friendOf, strength, 2, , 6,
7, 1, 0, knows, mval, 5, , , 2003-01-01T12:00:00.000Z
7, 1, 0, knows, firstMetIn, 1, GA, ,
7, 1, 0, knows, since, 5, , , 2000-01-01T12:00:00.000Z
17, 1, 0, friendOf, strength, 2, , 7,
9, 1, 3, knows, mval, 6, N, ,
9, 1, 3, knows, firstMetIn, 1, SC, ,
9, 1, 3, knows, since, 5, , , 2005-01-01T12:00:00.000Z
10, 2, 0, knows, mval, 1, N, ,
10, 2, 0, knows, firstMetIn, 1, TX, ,
10, 2, 0, knows, since, 5, , , 1997-01-01T12:00:00.000Z
12, 2, 3, knows, mval, 3, , 342.5,
12, 2, 3, knows, firstMetIn, 1, TX, ,
12, 2, 3, knows, since, 5, , , 2011-01-01T12:00:00.000Z
19, 2, 3, friendOf, strength, 2, , 4,
14, 3, 1, knows, mval, 1, a, ,
14, 3, 1, knows, firstMetIn, 1, CA, ,
14, 3, 1, knows, since, 5, , , 2010-01-01T12:00:00.000Z
15, 3, 2, knows, mval, 1, z, ,
15, 3, 2, knows, firstMetIn, 1, CA, ,
15, 3, 2, knows, since, 5, , , 2004-01-01T12:00:00.000Z
5, 0, 2, knows, mval, 2, , 23,
5, 0, 2, knows, firstMetIn, 1, OH, ,
5, 0, 2, knows, since, 5, , , 2002-01-01T12:00:00.000Z
6, 0, 3, knows, mval, 3, , 159.7,
6, 0, 3, knows, firstMetIn, 1, IN, ,
6, 0, 3, knows, since, 5, , , 1994-01-01T12:00:00.000Z
8, 1, 2, knows, mval, 6, Y, ,
8, 1, 2, knows, firstMetIn, 1, FL, ,
8, 1, 2, knows, since, 5, , , 1999-01-01T12:00:00.000Z
18, 1, 3, friendOf, strength, 2, , 5,
11, 2, 1, knows, mval, 2, , 1001,
11, 2, 1, knows, firstMetIn, 1, OK, ,
11, 2, 1, knows, since, 5, , , 2003-01-01T12:00:00.000Z
13, 3, 0, knows, mval, 5, , , 2001-01-01T12:00:00.000Z
13, 3, 0, knows, firstMetIn, 1, CA, ,
```

```
13,3,0, knows, since, 5, , , 2006-01-01T12:00:00.000Z
20,3,1, friendOf, strength, 2, , 3,
```

- [Basic Query Execution](#)
- [Executing PGQL Queries Using JDBC Driver](#)
- [Security Techniques for PGQL Queries](#)
- [Using a Text Index with PGQL Queries](#)
- [Obtaining the SQL Translation for a PGQL Query](#)
- [Additional Options for PGQL Translation and Execution](#)
- [Querying Another User's Property Graph](#)
- [Using Query Optimizer Hints with PGQL](#)
- [Modifying Property Graphs through INSERT, UPDATE, and DELETE Statements](#)

6.8.1.4.1 Basic Query Execution

Two main Java Interfaces, `PgqlStatement` and `PgqlResultSet`, are used for PGQL execution. This topic includes several examples of basic query execution.

Example 6-6 `GraphLoaderExample.java`

`GraphLoaderExample.java` loads some Oracle property graph data that will be used in subsequent examples in this topic.

```
import oracle.pg.rdbms.Oracle;
import oracle.pg.rdbms.OraclePropertyGraph;
import oracle.pg.rdbms.OraclePropertyGraphDataLoader;

/**
 * This example shows how to create an Oracle Property Graph
 * and load data into it from vertex and edge flat files.
 */
public class GraphLoaderExample
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];
        String vertexFile    = args[idx++];
        String edgeFile      = args[idx++];

        Oracle oracle = null;
        OraclePropertyGraph opg = null;

        try {
            // Create a connection to Oracle
            oracle = new Oracle("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid, user,
```



```

* This example shows how to execute a basic PGQL query against disk-
resident
* PG data stored in Oracle Database and iterate through the result.
*/
public class PgqlExample1
{

public static void main(String[] args) throws Exception
{
    int idx=0;
    String host          = args[idx++];
    String port         = args[idx++];
    String sid          = args[idx++];
    String user         = args[idx++];
    String password     = args[idx++];
    String graph        = args[idx++];

    Connection conn = null;
    PgqlStatement ps = null;
    PgqlResultSet rs = null;

    try {

        //Get a jdbc connection
        PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
        pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
        pds.setURL("jdbc:oracle:thin:@"+host+": "+port +":"+sid);
        pds.setUser(user);
        pds.setPassword(password);
        conn = pds.getConnection();

        // Get a PGQL connection
        PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
        pgqlConn.setGraph(graph);

        // Create a PgqlStatement
        ps = pgqlConn.createStatement();

        // Execute query to get a PgqlResultSet object
        String pgql =
mval "+
        "SELECT v.\"fname\" AS fname, v.\"lname\" AS lname, v.\"mval\" AS
        "FROM MATCH (v)";
        rs = ps.executeQuery(pgql, /* query string */
                            " /* options */");

        // Print the results
        rs.print();
    }
    finally {
        // close the result set
        if (rs != null) {
            rs.close();
        }
        // close the statement
    }
}
}

```

```

        if (ps != null) {
            ps.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}
}
}

```

PgqlExample1.java gives the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```

+-----+
| FNAME | LNAME | MVAL |
+-----+
| Susan | Blue  | false |
| Bill  | Brown | Y      |
| Ray   | Green | 1985-01-01 04:00:00.0 |
| John  | Black | 27     |
+-----+

```

Example 6-8 PgqlExample2.java

PgqlExample2.java shows a PGQL query with a temporal filter on an edge property.

- PgqlResultSet provides an interface for consuming the query result that is very similar to the java.sql.ResultSet interface.
- A next() method allows moving through the query result, and a close() method allows releasing resources after the application is finished reading the query result.
- In addition, PgqlResultSet provides getters for String, Integer, Long, Float, Double, Boolean, LocalDateTime, and OffsetDateTime, and it provides a generic getObject() method for values of any type.

```

import java.sql.Connection;

import java.text.SimpleDateFormat;

import java.util.Date;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.pgql.lang.ResultSet;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL query with a temporal edge
 * property filter against disk-resident PG data stored in Oracle
 * Database
 * and iterate through the result.
 */

```

```
public class PgqlExample2
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        ResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+": "+port +":"+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute query to get a ResultSet object
            String pgql =
                "SELECT v.\"fname\" AS n1, v2.\"fname\" AS n2, e.\"firstMetIn\" AS
loc "+
                "FROM MATCH (v)-[e:\"knows\"]->(v2) "+
                "WHERE e.\"since\" > TIMESTAMP '2000-01-01 00:00:00.00+00:00'";
            rs = ps.executeQuery(pgql, "");

            // Print results
            printResults(rs);
        }
        finally {
            // close the result set
            if (rs != null) {
                rs.close();
            }
            // close the statement
            if (ps != null) {
                ps.close();
            }
            // close the connection
        }
    }
}
```

```
        if (conn != null) {
            conn.close();
        }
    }
}

/**
 * Prints a PGQL ResultSet
 */
static void printResults(ResultSet rs) throws Exception
{
    StringBuffer buff = new StringBuffer("");
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX");
    while (rs.next()) {
        buff.append("[");
        for (int i = 1; i <= rs.getMetaData().getColumnCount(); i++) {
            // use generic getObject to handle all types
            Object mval = rs.getObject(i);
            String mStr = "";
            if (mval instanceof java.lang.String) {
                mStr = "STRING: "+mval.toString();
            }
            else if (mval instanceof java.lang.Integer) {
                mStr = "INTEGER: "+mval.toString();
            }
            else if (mval instanceof java.lang.Long) {
                mStr = "LONG: "+mval.toString();
            }
            else if (mval instanceof java.lang.Float) {
                mStr = "FLOAT: "+mval.toString();
            }
            else if (mval instanceof java.lang.Double) {
                mStr = "DOUBLE: "+mval.toString();
            }
            else if (mval instanceof java.sql.Timestamp) {
                mStr = "DATE: "+sdf.format((Date)mval);
            }
            else if (mval instanceof java.lang.Boolean) {
                mStr = "BOOLEAN: "+mval.toString();
            }
            if (i > 1) {
                buff.append(",\t");
            }
            buff.append(mStr);
        }
        buff.append("]\n");
    }
    System.out.println(buff.toString());
}
}
```

PgqlExample2.java gives the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```
[STRING: Susan, STRING: Bill,   STRING: CA]
[STRING: Susan, STRING: John,   STRING: CA]
[STRING: Susan, STRING: Ray,    STRING: CA]
[STRING: Bill,   STRING: Ray,    STRING: OH]
[STRING: Ray,    STRING: John,   STRING: OK]
[STRING: Ray,    STRING: Susan,  STRING: TX]
[STRING: John,   STRING: Susan,  STRING: SC]
[STRING: John,   STRING: Bill,   STRING: GA]
```

Example 6-9 PgqlExample3.java

PgqlExample3.java shows a PGQL query with grouping and aggregation.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL query with aggregation
 * against disk-resident PG data stored in Oracle Database and iterate
 * through the result.
 */
public class PgqlExample3
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];
        String user           = args[idx++];
        String password       = args[idx++];
        String graph          = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {
            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+": "+port +":"+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);
```



```

/**
 * This example shows how to execute a path query in PGQL against
 * disk-resident PG data stored in Oracle Database and iterate
 * through the result.
 */
public class PgqlExample4
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute query to get a ResultSet object
            String pgql =
                "PATH fof AS ()-[:\"friendOf\"|\"knows\"]->() "+
                "SELECT v2.\"fname\" AS friend "+
                "FROM MATCH (v)-/:fof*/->(v2) "+
                "WHERE v.\"fname\" = 'John' AND v != v2";
            rs = ps.executeQuery(pgql, "");

            // Print results
            rs.print();
        }
        finally {
            // close the result set
            if (rs != null) {
                rs.close();
            }
        }
    }
}

```



```
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import oracle.pg.rdbms.pgql.jdbc.PgqlJdbcRdbmsDriver;

public class PgqlJdbcTest {

    public static void main(String[] args) throws Exception {

        DriverManager.registerDriver(new PgqlJdbcRdbmsDriver());
        String jdbcUrl = "jdbc:oracle:pgql:@<DB Host>:<DB Port>/<DB SID>";
        String username = "<DB Username>";
        String password = "<DB Password>";

        try (Connection conn = DriverManager.getConnection(jdbcUrl, username,
password)) {
            String query = "SELECT n.name FROM MATCH(n) ON test_graph WHERE id(n)
= ?";
            PreparedStatement pstmt = conn.prepareStatement(query);
            pstmt.setLong(1, 10L);
            pstmt.execute();
            ResultSet rs = pstmt.getResultSet();
            while(rs.next()){
                System.out.println("NAME = " + rs.getString("name"));
            }
        }
    }
}
```

Save the preceding code in a file `PgqlJdbcTest.java` and compile using:

```
javac -cp "<graph-client>/lib/*" PgqlJdbcTest.java
```

The driver is also included in a regular graph server (RPM) install. For example:

```
javac -cp "/opt/oracle/graph/lib/*" PgqlJdbcTest.java
```

6.8.1.4.3 Security Techniques for PGQL Queries

Programs executing dynamic queries might be subject to injection attacks that could compromise integrity and functioning of the applications.

This topic presents some techniques that can be used to prevent injection attacks when building PGQL queries using string concatenation.

- [Using Bind Variables in PGQL Queries](#)
- [Verifying PGQL Identifiers](#)

6.8.1.4.3.1 Using Bind Variables in PGQL Queries

Bind variables can be used in PGQL queries for better performance and increased security. Constant scalar values in PGQL queries can be replaced with bind variables. Bind variables

are denoted by a '?' (question mark). Consider the following two queries that select people who are older than a constant age value.

```
// people older than 30
SELECT v.fname AS fname, v.lname AS lname, v.age AS age
FROM MATCH (v)
WHERE v.age > 30
```

```
// people older than 40
SELECT v.fname AS fname, v.lname AS lname, v.age AS age
FROM MATCH (v)
WHERE v.age > 40
```

The SQL translations for these queries would use the constants 30 and 40 in a similar way for the age filter. The database would perform a hard parse for each of these queries. This hard parse time can often exceed the execution time for simple queries.

You could replace the constant in each query with a bind variable as follows.

```
SELECT v.fname AS fname, v.lname AS lname, v.age AS age
FROM MATCH (v)
WHERE v.age > ?
```

This will allow the SQL engine to create a generic cursor for this query, which can be reused for different age values. As a result, a hard parse is no longer required to execute this query for different age values, and the parse time for each query will be drastically reduced.

In addition, applications that use bind variables in PGQL queries are less vulnerable to injection attacks than those that use string concatenation to embed constant values in PGQL queries.

See also *Oracle Database SQL Tuning Guide* for more information on cursor sharing and bind variables.

The `PgqlPreparedStatement` interface can be used to execute queries with bind variables as shown in `PgqlExample5.java`. `PgqlPreparedStatement` provides several set methods for different value types that can be used to set values for query execution.

There are a few limitations with bind variables in PGQL. Bind variables can only be used for constant property values. That is, vertices and edges cannot be replaced with bind variables. Also, once a particular bind variable has been set to a type, it cannot be set to a different type. For example, if `setInt(1, 30)` is executed for an `PgqlPreparedStatement`, you cannot call `setString(1, "abc")` on that same `PgqlPreparedStatement`.

Example 6-12 `PgqlExample5.java`

`PgqlExample5.java` shows how to use bind variables with a PGQL query.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlPreparedStatement;
import oracle.pg.rdbms.pgql.PgqlResultSet;
```

```

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use bind variables with a PGQL query.
 */
public class PgqlExample5
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlPreparedStatement pps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+": "+port +"@"+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Query string with a bind variable (denoted by ?)
            String pgql =
                "SELECT v.\"fname\" AS fname, v.\"lname\" AS lname, v.\"age\" AS age
"+
                "FROM MATCH (v) "+
                "WHERE v.\"age\" > ?";

            // Create a PgqlPreparedStatement
            pps = pgqlConn.prepareStatement(pgql);

            // Set filter value to 30
            pps.setInt(1, 30);

            // execute query
            rs = pps.executeQuery();

            // Print query results

```

```
System.out.println("-- Values for v.\"age\" > 30 --");
rs.print();
// close result set
rs.close();

// set filter value to 40
pps.setInt(1, 40);

// execute query
rs = pps.executeQuery();

// Print query results
System.out.println("-- Values for v.\"age\" > 40 --");
rs.print();
// close result set
rs.close();
}
finally {
    // close the result set
    if (rs != null) {
        rs.close();
    }
    // close the statement
    if (pps != null) {
        pps.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}
```

PgqlExample5.java has the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```
-- Values for v.age > 30 --
+-----+
| fname | lname | age |
+-----+
| Susan | Blue  | 35  |
| Bill  | Brown | 40  |
| Ray   | Green | 41  |
+-----+
-- Values for v.age > 40 --
+-----+
| fname | lname | age |
+-----+
| Ray   | Green | 41  |
+-----+
```

Example 6-13 PgqlExample6.java

PgqlExample6.java shows a query with two bind variables: one String variable and one Timestamp variable.

```
import java.sql.Connection;
import java.sql.Timestamp;

import java.time.OffsetDateTime;
import java.time.ZoneOffset;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlPreparedStatement;
import oracle.pg.rdbms.pgql.PgqlResultSet;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use multiple bind variables with a PGQL query.
 */
public class PgqlExample6
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];
        String user           = args[idx++];
        String password       = args[idx++];
        String graph          = args[idx++];

        Connection conn = null;
        PgqlPreparedStatement pps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+" "+"port "+" "+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Query string with multiple bind variables
            String pgql =
                "SELECT v1.\"fname\" AS fname1, v2.\"fname\" AS fname2 "+
```

```

"FROM MATCH (v1)-[e:\"knows\"]->(v2) "+
"WHERE e.\"since\" < ? AND e.\"firstMetIn\" = ?";

// Create a PgqlPreparedStatement
pps = pgqlConn.prepareStatement(pgql);

// Set e.since < 2006-01-01T12:00:00.00Z
Timestamp t =
Timestamp.valueOf(OffsetDateTime.parse("2006-01-01T12:00:01.00Z").atZone
SameInstant(ZoneOffset.UTC).toLocalDateTime());
pps.setTimestamp(1, t);
// Set e.firstMetIn = 'CA'
pps.setString(2, "CA");

// execute query
rs = pps.executeQuery();

// Print query results
System.out.println("-- Values for e.\"since\" <
2006-01-01T12:00:01.00Z AND e.\"firstMetIn\" = 'CA' --");
rs.print();
// close result set
rs.close();

// Set e.since < 2000-01-01T12:00:00.00Z
t =
Timestamp.valueOf(OffsetDateTime.parse("2000-01-01T12:00:00.00Z").atZone
SameInstant(ZoneOffset.UTC).toLocalDateTime());
pps.setTimestamp(1, t);
// Set e.firstMetIn = 'TX'
pps.setString(2, "TX");

// execute query
rs = pps.executeQuery();

// Print query results
System.out.println("-- Values for e.\"since\" <
2000-01-01T12:00:00.00Z AND e.\"firstMetIn\" = 'TX' --");
rs.print();
// close result set
rs.close();
}
finally {
// close the result set
if (rs != null) {
rs.close();
}
// close the statement
if (pps != null) {
pps.close();
}
// close the connection
if (conn != null) {
conn.close();
}
}

```

```

    }
  }
}

```

PgqlExample6.java gives the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```

-- Values for e."since" < 2006-01-01T12:00:01.00Z AND e."firstMetIn" = 'CA' --
+-----+
| FNAME1 | FNAME2 |
+-----+
| Susan  | Bill   |
| Susan  | Ray    |
+-----+
-- Values for e."since" < 2000-01-01T12:00:00.00Z AND e."firstMetIn" = 'TX' --
+-----+
| FNAME1 | FNAME2 |
+-----+
| Ray    | Bill   |
+-----+

```

6.8.1.4.3.2 Verifying PGQL Identifiers

For some parts of a PGQL query the parser does not allow use of bind variables. In such cases, the input can be verified using the printIdentifier method in package oracle.pgql.lang.ir.PgqlUtils.

Consider the following query execution that concatenates the graph against which the graph pattern will be matched:

```
stmt.executeQuery("SELECT n.name FROM MATCH (n) ON " + graphName, "");
```

In order to avoid injection, the identifier graphName should be verified as follows:

```
stmt.executeQuery("SELECT n.name FROM MATCH (n) ON " +
PgqlUtils.printIdentifier(graphName), "");
```

6.8.1.4.4 Using a Text Index with PGQL Queries

PGQL queries executed against Oracle Database can use Oracle Text indexes created for vertex and edge properties. After creating a text index, you can use the CONTAINS operator to perform a full text search. CONTAINS has two arguments: a vertex or edge property, and an Oracle Text search string. Any valid Oracle Text search string can be used, including advanced features such as wildcards, stemming, and soundex.

Example 6-14 PgqlExample7.java

PgqlExample7.java shows how to execute a CONTAINS query.

```
import java.sql.CallableStatement;
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;
```

```

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use an Oracle Text index with a PGQL query.
 */
public class PgqlExample7
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create text index with SQL API
            CallableStatement cs = null;
            // text index on vertices
            cs = conn.prepareCall(
                "begin opg_apis.create_vertices_text_idx(:1,:2); end;"
            );
            cs.setString(1,user);
            cs.setString(2,graph);
            cs.execute();
            cs.close();
            // text index on edges
            cs = conn.prepareCall(
                "begin opg_apis.create_edges_text_idx(:1,:2); end;"
            );
            cs.setString(1,user);
            cs.setString(2,graph);
            cs.execute();
            cs.close();

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);

```



```
pgqlConn.setGraph(graph);

// Create a PgqlStatement
ps = pgqlConn.createStatement();

// Query using CONTAINS text search operator on vertex property
// Find all vertices with an lname property value that starts with 'B'
String pgql =
    "SELECT v.\"fname\" AS fname, v.\"lname\" AS lname "+
    "FROM MATCH (v) "+
    "WHERE CONTAINS(v.\"lname\", 'B%')";

// execute query
rs = ps.executeQuery(pgql, "");

// print results
System.out.println("-- Vertex Property Query --");
rs.print();

// close result set
rs.close();

// Query using CONTAINS text search operator on edge property
// Find all knows edges with a firstMetIn property value that ends
with 'A'
pgql =
    "SELECT v1.\"fname\" AS fname1, v2.\"fname\" AS fname2,
e.\"firstMetIn\" AS loc "+
    "FROM MATCH (v1)-[e:\"knows\"]->(v2) "+
    "WHERE CONTAINS(e.\"firstMetIn\", '%A')";

// execute query
rs = ps.executeQuery(pgql, "");

// print results
System.out.println("-- Edge Property Query --");
rs.print();

}
finally {
    // close the result set
    if (rs != null) {
        rs.close();
    }
    // close the statement
    if (ps != null) {
        ps.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}
```

PgqlExample7.java has the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```
-- Vertex Property Query --
+-----+
| FNAME | LNAME |
+-----+
| Susan | Blue  |
| Bill  | Brown |
| John  | Black |
+-----+
-- Edge Property Query --
+-----+
| FNAME1 | FNAME1 | LOC |
+-----+
| Susan  | Bill   | CA  |
| John   | Bill   | GA  |
| Susan  | John   | CA  |
| Susan  | Ray    | CA  |
+-----+
```

6.8.1.4.5 Obtaining the SQL Translation for a PGQL Query

You can obtain the SQL translation for a PGQL query through methods in `PgqlStatement` and `PgqlPreparedStatement`. The raw SQL for a PGQL query can be useful for several reasons:

- You can execute the SQL directly against the database with other SQL-based tools or interfaces (for example, SQL*Plus or SQL Developer).
- You can customize and tune the generated SQL to optimize performance or to satisfy a particular requirement of your application.
- You can build a larger SQL query that joins a PGQL subquery with other data stored in Oracle Database (such as relational tables, spatial data, and JSON data).

Example 6-15 PgqlExample8.java

`PgqlExample8.java` shows how to obtain the raw SQL translation for a PGQL query. The `translateQuery` method of `PgqlStatement` returns an `PgqlSqlQueryTrans` object that contains information about return columns from the query and the SQL translation itself.

The translated SQL returns different columns depending on the type of "logical" object or value projected from the PGQL query. A vertex or edge projected in PGQL has two corresponding columns projected in the translated SQL:

- `$IT` : id type – `NVARCHAR(1)`: 'V' for vertex or 'E' for edge
- `$ID` : vertex or edge identifier – `NUMBER`: same content as `VID` or `EID` columns in `VT$` and `GE$` tables

A property value or constant scalar value projected in PGQL has four corresponding columns projected in the translated SQL:

- `$T` : value type – `NUMBER`: same content as `T` column in `VT$` and `GE$` tables
- `$V`: value – `NVARCHAR2(15000)`: same content as `V` column in `VT$` and `GE$` tables

- \$VN: number value – NUMBER: same content as VN column in VT\$ and GE\$ tables
- \$VT: temporal value – TIMESTAMP WITH TIME ZONE: same content as VT column in VT\$ and GE\$ tables

```

import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlColumnDescriptor;
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.pg.rdbms.pgql.PgqlSqlQueryTrans;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to obtain the SQL translation for a PGQL query.
 */
public class PgqlExample8
{
    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // PGQL query to be translated
            String pgql =
                "SELECT v1, v1.\"fname\" AS fname1, e, e.\"since\" AS since "+
                "FROM MATCH (v1)-[e:\"knows\"]->(v2)";

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

```

```

// Get the SQL translation
PgqlSqlQueryTrans sqlTrans = ps.translateQuery(pgql,"");

// Get the return column descriptions
PgqlColumnDescriptor[] cols = sqlTrans.getReturnTypes();

// Print column descriptions
System.out.println("-- Return Columns -----");
printReturnCols(cols);

// Print SQL translation
System.out.println("-- SQL Translation -----");
System.out.println(sqlTrans.getSqlTranslation());
}
finally {
    // close the statement
    if (ps != null) {
        ps.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}

/**
 * Prints return columns for a SQL translation
 */
static void printReturnCols(PgqlColumnDescriptor[] cols) throws
Exception
{
    StringBuffer buff = new StringBuffer("");

    for (int i = 0; i < cols.length; i++) {

        String colName = cols[i].getColName();
        PgqlColumnDescriptor.Type colType = cols[i].getColType();
        int offset = cols[i].getSqlOffset();

        String readableType = "";
        switch(colType) {
            case VERTEX:
                readableType = "VERTEX";
                break;
            case EDGE:
                readableType = "EDGE";
                break;
            case VALUE:
                readableType = "VALUE";
                break;
        }

        buff.append("colName=["+colName+"] colType=["+readableType+"]

```

```

offset=["+offset+" ]\n");
    }
    System.out.println(buff.toString());
}
}
}

```

PgqlExample8.java has the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```

-- Return Columns -----
colName=[v1] colType=[VERTEX] offset=[1]
colName=[fname1] colType=[VALUE] offset=[3]
colName=[e] colType=[EDGE] offset=[7]
colName=[since] colType=[VALUE] offset=[9]
-- SQL Translation -----
SELECT n'V' AS "V1$IT",
TO$0.SVID AS "V1$ID",
TO$1.T AS "FNAME1$T",
TO$1.V AS "FNAME1$V",
TO$1.VN AS "FNAME1$VN",
TO$1.VT AS "FNAME1$VT",
n'E' AS "E$IT",
TO$0.EID AS "E$ID",
TO$0.T AS "SINCE$T",
TO$0.V AS "SINCE$V",
TO$0.VN AS "SINCE$VN",
TO$0.VT AS "SINCE$VT"
FROM ( SELECT L.EID, L.SVID, L.DVID, L.EL, R.K, R.T, R.V, R.VN, R.VT
      FROM "SCOTT".TEST_GRAPHGT$ L,
           (SELECT * FROM "SCOTT".TEST_GRAPHGE$ WHERE K=n'since' ) R
      WHERE L.EID = R.EID(+)
    ) TO$0,
( SELECT L.VID, L.VL, R.K, R.T, R.V, R.VN, R.VT
  FROM "SCOTT".TEST_GRAPHVD$ L,
       (SELECT * FROM "SCOTT".TEST_GRAPHVT$ WHERE K=n'fname' ) R
  WHERE L.VID = R.VID(+)
) TO$1
WHERE TO$0.SVID=TO$1.VID AND
(TO$0.EL = n'knows' AND TO$0.EL IS NOT NULL)

```

Example 6-16 PgqlExample9.java

You can also obtain the SQL translation for PGQL queries with bind variables. In this case, the corresponding SQL translation will also contain bind variables. The PgqlSqlQueryTrans interface has a getSqlBvList method that returns an ordered List of Java Objects that should be bound to the SQL query (the first Object on the list should be set at position 1, and the second should be set at position 2, and so on).

PgqlExample9.java shows how to get and execute the SQL for a PGQL query with bind variables.

```

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Timestamp;

import java.util.List;

```

```

import oracle.pg.rdbms.pgql.PgqlColumnDescriptor;
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlPreparedStatement;
import oracle.pg.rdbms.pgql.PgqlSqlQueryTrans;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to obtain and execute the SQL translation for
 a
 * PGQL query that uses bind variables.
 */
public class PgqlExample9
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];
        String user           = args[idx++];
        String password       = args[idx++];
        String graph          = args[idx++];

        Connection conn = null;
        PgqlPreparedStatement pgqlPs = null;

        PreparedStatement sqlPs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+":"+port +":"+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Create a Pgql connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Execute query to get a ResultSet object
            String pgql =
                "SELECT v1, v1.\"fname\" AS fname1, v1.\"age\" AS age, ? as
constVal "+
                "FROM MATCH (v1) "+
                "WHERE v1.\"fname\" = ? OR v1.\"age\" < ?";

            // Create a PgqlStatement

```

```

pgqlPs = pgqlConn.prepareStatement(pgql);

// set bind values
pgqlPs.setDouble(1, 2.05d);
pgqlPs.setString(2, "Bill");
pgqlPs.setInt(3, 35);

// Get the SQL translation
PgqlSqlQueryTrans sqlTrans = pgqlPs.translateQuery("");

// Get the SQL String
String sqlStr = sqlTrans.getSqlTranslation();

// Get the return column descriptions
PgqlColumnDescriptor[] cols = sqlTrans.getReturnTypes();

// Get the bind values
List<Object> bindVals = sqlTrans.getSqlBvList();

// Print column descriptions
System.out.println("-- Return Columns -----");
printReturnCols(cols);

// Print SQL translation
System.out.println("-- SQL Translation -----");
System.out.println(sqlStr);

// Print Bind Values
System.out.println("\n-- Bind Values -----");
for (Object obj : bindVals) {
    System.out.println(obj.toString());
}

// Execute Query
// Get PreparedStatement
sqlPs = conn.prepareStatement("SELECT COUNT(*) FROM (" + sqlStr + ")");
// Set bind values and execute the PreparedStatement
executePs(sqlPs, bindVals);

// Set new bind values in the PGQL PreparedStatement
pgqlPs.setDouble(1, 3.02d);
pgqlPs.setString(2, "Ray");
pgqlPs.setInt(3, 30);

// Print Bind Values
bindVals = sqlTrans.getSqlBvList();
System.out.println("\n-- Bind Values -----");
for (Object obj : bindVals) {
    System.out.println(obj.toString());
}

// Execute the PreparedStatement with new bind values
executePs(sqlPs, bindVals);
}
finally {

```

```

        // close the SQL statement
        if (sqlPs != null) {
            sqlPs.close();
        }
        // close the statement
        if (pgqlPs != null) {
            pgqlPs.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}

/**
 * Executes a SQL PreparedStatement with the input bind values
 */
static void executePs(PreparedStatement ps, List<Object> bindVals)
throws Exception
{
    ResultSet rs = null;
    try {
        // Set bind values
        for (int idx = 0; idx < bindVals.size(); idx++) {
            Object o = bindVals.get(idx);
            // String
            if (o instanceof java.lang.String) {
                ps.setNString(idx + 1, (String)o);
            }
            // Int
            else if (o instanceof java.lang.Integer) {
                ps.setInt(idx + 1, ((Integer)o).intValue());
            }
            // Long
            else if (o instanceof java.lang.Long) {
                ps.setLong(idx + 1, ((Long)o).longValue());
            }
            // Float
            else if (o instanceof java.lang.Float) {
                ps.setFloat(idx + 1, ((Float)o).floatValue());
            }
            // Double
            else if (o instanceof java.lang.Double) {
                ps.setDouble(idx + 1, ((Double)o).doubleValue());
            }
            // Timestamp
            else if (o instanceof java.sql.Timestamp) {
                ps.setTimestamp(idx + 1, (Timestamp)o);
            }
            else {
                ps.setString(idx + 1, bindVals.get(idx).toString());
            }
        }
    }
}

```



```

        // Execute query
        rs = ps.executeQuery();
        if (rs.next()) {
            System.out.println("\n-- Execute Query: Result has "+rs.getInt(1)+"
rows --");
        }
    }
    finally {
        // close the SQL ResultSet
        if (rs != null) {
            rs.close();
        }
    }
}

/**
 * Prints return columns for a SQL translation
 */
static void printReturnCols(PgqlColumnDescriptor[] cols) throws Exception
{
    StringBuffer buff = new StringBuffer("");

    for (int i = 0; i < cols.length; i++) {

        String colName = cols[i].getColName();
        PgqlColumnDescriptor.Type colType = cols[i].getColType();
        int offset = cols[i].getSqlOffset();

        String readableType = "";
        switch(colType) {
            case VERTEX:
                readableType = "VERTEX";
                break;
            case EDGE:
                readableType = "EDGE";
                break;
            case VALUE:
                readableType = "VALUE";
                break;
        }

        buff.append("colName=["+colName+"] colType=["+readableType+"]
offset=["+offset+"]\n");
    }
    System.out.println(buff.toString());
}
}
}

```

PgqlExample9.java has the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```

--- Return Columns -----
colName=[v1] colType=[VERTEX] offset=[1]
colName=[fname1] colType=[VALUE] offset=[3]
colName=[age] colType=[VALUE] offset=[7]

```

```

colName=[constVal] colType=[VALUE] offset=[11]
-- SQL Translation -----
SELECT n'V' AS "V1$IT",
T0$0.VID AS "V1$ID",
T0$0.T AS "FNAME1$T",
T0$0.V AS "FNAME1$V",
T0$0.VN AS "FNAME1$VN",
T0$0.VT AS "FNAME1$VT",
T0$1.T AS "AGE$T",
T0$1.V AS "AGE$V",
T0$1.VN AS "AGE$VN",
T0$1.VT AS "AGE$VT",
4 AS "CONSTVAL$T",
to_nchar(?, 'TM9', 'NLS_Numeric_Characters='.', ''') AS "CONSTVAL$V",
? AS "CONSTVAL$VN",
to_timestamp_tz(null) AS "CONSTVAL$VT"
FROM ( SELECT L.VID, L.VL, R.K, R.T, R.V, R.VN, R.VT
      FROM "SCOTT".TEST_GRAPHVD$ L,
           (SELECT * FROM "SCOTT".TEST_GRAPHVT$ WHERE K=n'fname' ) R
      WHERE L.VID = R.VID(+)
    ) T0$0,
( SELECT L.VID, L.VL, R.K, R.T, R.V, R.VN, R.VT
  FROM "SCOTT".TEST_GRAPHVD$ L,
       (SELECT * FROM "SCOTT".TEST_GRAPHVT$ WHERE K=n'age' ) R
  WHERE L.VID = R.VID(+)
) T0$1
WHERE T0$0.VID=T0$1.VID AND
((T0$0.T = 1 AND T0$0.V = ?) OR T0$1.VN < ?)

-- Bind Values -----
2.05
2.05
Bill
35
-- Execute Query: Result has 2 rows --

-- Bind Values -----
3.02
3.02
Ray
30
-- Execute Query: Result has 1 rows --

```

6.8.1.4.6 Additional Options for PGQL Translation and Execution

Several options are available to influence PGQL query translation and execution. The following are the main ways to set query options:

- Through explicit arguments to `executeQuery` and `translateQuery`
- Through flags in the `options` string argument of `executeQuery` and `translateQuery`
- Through Java JVM arguments.

The following table summarizes the available query arguments for PGQL translation and execution.

Table 6-3 PGQL Translation and Execution Options

Option	Default	Explicit Argument	Options Flag	JVM Argument
Degree of parallelism	0	parallel	none	none
Timeout	unlimited	timeout	none	none
Dynamic sampling	2	dynamicSampling	none	none
Maximum number of results	unlimited	maxResults	none	none
GT\$ table usage	on	none	USE_GT_TAB=F	- Doracle.pg.rdbms.pgql.useGtTab=false
CONNECT BY usage	off	none	USE_RW=F	-Doracle.pg.rdbms.pgql.useRW=false
Distinct recursive WITH usage	off	none	USE_DIST_RW=T	- Doracle.pg.rdbms.pgql.useDistRW=true
Maximum path length	unlimited	none	MAX_PATH_LEN=n	-Doracle.pg.rdbms.pgql.maxPathLen=n
Set partial	false	none	EDGE_SET_PARTIAL=T	- Doracle.pg.rdbms.pgql.edgeSetPartial=true
Project null properties	true	none	PROJ_NULL_PROPS=F	- Doracle.pg.rdbms.pgql.projNullProps=false
VT\$ VL column usage	on	none	USE_VL_COL=F	- Doracle.pg.rdbms.pgql.useVLCol=false

- [Query Options Controlled by Explicit Arguments](#)
- [Using the GT\\$ Skeleton Table](#)
- [Path Query Options](#)
- [Options for Partial Object Construction](#)

6.8.1.4.6.1 Query Options Controlled by Explicit Arguments

Some query options are controlled by explicit arguments to methods in the Java API.

- The `executeQuery` method of `PgqlStatement` has explicit arguments for timeout in seconds, degree of parallelism, optimizer dynamic sampling, and maximum number of results.
- The `translateQuery` method has explicit arguments for degree of parallelism, optimizer dynamic sampling, and maximum number of results. `PgqlPreparedStatement` also provides those same additional arguments for `executeQuery` and `translateQuery`.

Example 6-17 PgqlExample10.java

PgqlExample10.java shows PGQL query execution with additional options controlled by explicit arguments.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL query with various options.
 */
public class PgqlExample10
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];
        String user           = args[idx++];
        String password       = args[idx++];
        String graph          = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@"+host+": "+port+"@"+sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute query to get a ResultSet object
            String pgql =
                "SELECT v1.\"fname\" AS fname1, v2.\"fname\" AS fname2 "+
```


Example 6-18 PgqlExample11.java

PgqlExample11.java shows a query that uses the GT\$ skeleton table.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlSqlQueryTrans;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to avoid using the GT$ skeleton table for
 * PGQL query execution.
 */
public class PgqlExample11
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute query to get a ResultSet object
            String pgql =
                "SELECT id(v1), id(v2) "+
                "FROM MATCH (v1)-[knows]->(v2)";

```


6.8.1.4.6.3 Path Query Options

A few options are available for executing path queries in PGQL. There are two basic evaluation methods available in Oracle SQL: CONNECT BY or recursive WITH clauses. Recursive WITH is the default evaluation method. In addition, you can further modify the recursive WITH evaluation method to include a DISTINCT modifier during the recursive step of query evaluation. Computing distinct vertices at each step helps prevent a combinatorial explosion in highly connected graphs. The DISTINCT modifier is not added by default because it requires a specific parameter setting in the database ("`_recursive_with_control`"=8).

You can also control the maximum length of paths searched. Path length in this case is defined as the number of repetitions allowed when evaluating the * and + operators. The default maximum length is unlimited.

Path evaluation options are summarized as follows.

- **CONNECT BY:** To use CONNECT BY, specify 'USE_RW=F' in the options argument or specify `-Doracle.pg.rdbms.pgql.useRW=false` in the Java command line.
- **Distinct Modifier in Recursive WITH:** To use the DISTINCT modifier in the recursive step, first set "`_recursive_with_control`"=8 in your database session, then specify 'USE_DIST_RW=T' in the options argument or specify `-Doracle.pg.rdbms.pgql.useDistRW=true` in the Java command line. You will encounter ORA-32486: unsupported operation in recursive branch of recursive WITH clause if "`_recursive_with_control`" has not been set to 8 in your session.
- **Path Length Restriction:** To limit maximum number of repetitions when evaluating * and + to n, specify 'MAX_PATH_LEN=n' in the query options argument or specify `-Doracle.pg.rdbms.pgql.maxPathLen=n` in the Java command line.

Example 6-19 PgqlExample12.java

PgqlExample12.java shows path query translations under various options.

```
import java.sql.Connection;
import java.sql.Statement;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlSqlQueryTrans;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use various options with PGQL path queries.
 */
public class PgqlExample12
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
```



```

String sid          = args[idx++];
String user        = args[idx++];
String password    = args[idx++];
String graph       = args[idx++];

Connection conn = null;
PgqlStatement ps = null;

try {

    //Get a jdbc connection
    PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
    pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
    pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
    pds.setUser(user);
    pds.setPassword(password);
    conn = pds.getConnection();

    // Get a PGQL connection
    PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
    pgqlConn.setGraph(graph);

    // Create a PgqlStatement
    ps = pgqlConn.createStatement();

    // Set "_recursive_with_control"=8 to enable distinct optimization
    // optimization for recursive with
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("alter session set \"_recursive_with_control\"=8");
    stmt.close();

    // Path Query to illustrate options
    String pgql =
        "PATH fof AS ()-[:\"friendOf\"]->() "+
        "SELECT id(v1), id(v2) "+
        "FROM MATCH (v1)-/:fof*/->(v2) "+
        "WHERE id(v1) = 2";

    // get SQL translation with defaults - Non-distinct Recursive WITH
    PgqlSqlQueryTrans sqlTrans =
        ps.translateQuery(pgql /* query string */,
            2 /* parallel: default is 1 */,
            2 /* dynamic sampling: default is 2 */,
            -1 /* max results: -1 implies no limit */,
            "" /* options */);
    System.out.println("-- Default Path Translation -----");
    System.out.println(sqlTrans.getSqlTranslation()+"\n");

    // get SQL translation with DISTINCT reachability optimization
    sqlTrans =
        ps.translateQuery(pgql /* query string */,
            2 /* parallel: default is 1 */,
            2 /* dynamic sampling: default is 2 */,
            -1 /* max results: -1 implies no limit */,

```



```

WHERE T0$0.SVID = 2 AND
(T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL))
) UNION ALL
SELECT RW.ROOT, R.DVID
FROM (SELECT T0$0.SVID AS SVID,
T0$0.DVID AS DVID
FROM "SCOTT".TEST_GRAPHGT$ T0$0
WHERE (T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL)) R, RW
WHERE RW.DVID = R.SVID )
CYCLE DVID SET cycle_col TO 1 DEFAULT 0
SELECT ROOT SVID, DVID FROM RW))/*]Path*/) T0$0
WHERE T0$0.SVID = 2)

-- DISTINCT RW Path Translation -----
SELECT /*+ parallel(2) */ * FROM(SELECT 7 AS "id(v1)$T",
to_nchar(T0$0.SVID,'TM9','NLS_Numeric_Characters='.',') AS "id(v1)$V",
T0$0.SVID AS "id(v1)$VN",
to_timestamp_tz(null) AS "id(v1)$VT",
7 AS "id(v2)$T",
to_nchar(T0$0.DVID,'TM9','NLS_Numeric_Characters='.',') AS "id(v2)$V",
T0$0.DVID AS "id(v2)$VN",
to_timestamp_tz(null) AS "id(v2)$VT"
FROM (/*Path[/SELECT DISTINCT SVID, DVID
FROM (
SELECT 2 AS SVID, 2 AS DVID
FROM SYS.DUAL
WHERE EXISTS(
SELECT 1
FROM "SCOTT".TEST_GRAPHVT$
WHERE VID = 2)
UNION ALL
SELECT SVID,DVID FROM
(WITH RW (ROOT, DVID) AS
( SELECT ROOT, DVID FROM
(SELECT SVID ROOT, DVID
FROM (SELECT T0$0.SVID AS SVID,
T0$0.DVID AS DVID
FROM "SCOTT".TEST_GRAPHGT$ T0$0
WHERE T0$0.SVID = 2 AND
(T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL))
) UNION ALL
SELECT DISTINCT RW.ROOT, R.DVID
FROM (SELECT T0$0.SVID AS SVID,
T0$0.DVID AS DVID
FROM "SCOTT".TEST_GRAPHGT$ T0$0
WHERE (T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL)) R, RW
WHERE RW.DVID = R.SVID )
CYCLE DVID SET cycle_col TO 1 DEFAULT 0
SELECT ROOT SVID, DVID FROM RW))/*]Path*/) T0$0
WHERE T0$0.SVID = 2)

-- CONNECT BY Path Translation -----
SELECT /*+ parallel(2) */ * FROM(SELECT 7 AS "id(v1)$T",
to_nchar(T0$0.SVID,'TM9','NLS_Numeric_Characters='.',') AS "id(v1)$V",
T0$0.SVID AS "id(v1)$VN",
to_timestamp_tz(null) AS "id(v1)$VT",
7 AS "id(v2)$T",
to_nchar(T0$0.DVID,'TM9','NLS_Numeric_Characters='.',') AS "id(v2)$V",
T0$0.DVID AS "id(v2)$VN",
to_timestamp_tz(null) AS "id(v2)$VT"
FROM (/*Path[/SELECT DISTINCT SVID, DVID

```

```

FROM (
SELECT 2 AS SVID, 2 AS DVID
FROM SYS.DUAL
WHERE EXISTS(
SELECT 1
FROM "SCOTT".TEST_GRAPHVT$
WHERE VID = 2)
UNION ALL
SELECT SVID, DVID
FROM
(SELECT CONNECT_BY_ROOT T0$0.SVID AS SVID, T0$0.DVID AS DVID
FROM(
SELECT T0$0.SVID AS SVID,
T0$0.DVID AS DVID
FROM "SCOTT".TEST_GRAPHGT$ T0$0
WHERE (T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL)) T0$0
START WITH T0$0.SVID = 2
CONNECT BY NOCYCLE PRIOR DVID = SVID))/*]Path*/) T0$0
WHERE T0$0.SVID = 2)

```

The query plan for the first query with the default recursive WITH strategy should look similar to the following.

```

-- default RW
-----
-----
| Id | Operation |
Name |
-----
-----
| 0 | SELECT STATEMENT |
| 1 | TEMP TABLE TRANSFORMATION |
| 2 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6662_37AA44 |
| 3 | UNION ALL (RECURSIVE WITH) BREADTH FIRST |
| 4 | PX COORDINATOR |
| 5 | PX SEND QC (RANDOM) |
:TQ20000 |
| 6 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6662_37AA44 |
| 7 | PX PARTITION HASH ALL |
|* 8 | TABLE ACCESS BY LOCAL INDEX ROWID BATCHED |
TEST_GRAPHGT$ |
|* 9 | INDEX RANGE SCAN |
TEST_GRAPHXSG$ |
| 10 | PX COORDINATOR |
| 11 | PX SEND QC (RANDOM) |
:TQ10000 |
| 12 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6662_37AA44 |
| 13 | NESTED LOOPS |
| 14 | PX BLOCK ITERATOR |
|* 15 | TABLE ACCESS FULL |

```

```

SYS_TEMP_0FD9D6662_37AA44 |
| 16 | PARTITION HASH ALL | |
| * 17 | TABLE ACCESS BY LOCAL INDEX ROWID BATCHED | TEST_GRAPHGT$ |
| * 18 | INDEX RANGE SCAN | TEST_GRAPHXSG$ |
| 19 | PX COORDINATOR | |
| 20 | PX SEND QC (RANDOM) | :TQ30001 |
| 21 | VIEW | |
| 22 | HASH UNIQUE | |
| 23 | PX RECEIVE | |
| 24 | PX SEND HASH | :TQ30000 |
| 25 | HASH UNIQUE | |
| 26 | VIEW | |
| 27 | UNION-ALL | |
| 28 | PX SELECTOR | |
| * 29 | FILTER | |
| 30 | FAST DUAL | |
| 31 | PARTITION HASH SINGLE | |
| * 32 | INDEX SKIP SCAN | TEST_GRAPHXQV$ |
| 33 | VIEW | |
| * 34 | VIEW | |
| 35 | PX BLOCK ITERATOR | |
| 36 | TABLE ACCESS FULL | SYS_TEMP_0FD9D6662_37AA44 |
-----

```

The query plan for the second query that adds a DISTINCT modifier in the recursive step should look similar to the following.

```

-----
| Id | Operation |
Name |
-----
| 0 | SELECT STATEMENT |
| 1 | TEMP TABLE TRANSFORMATION |
| 2 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6669_37AA44 |
| 3 | UNION ALL (RECURSIVE WITH) BREADTH FIRST |
| 4 | PX COORDINATOR |
| 5 | PX SEND QC (RANDOM) |
| :TQ20000 |
| 6 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6669_37AA44 |
| 7 | PX PARTITION HASH ALL |
| * 8 | TABLE ACCESS BY LOCAL INDEX ROWID BATCHED |
TEST_GRAPHGT$ |
| * 9 | INDEX RANGE SCAN |
TEST_GRAPHXSG$ |
| 10 | PX COORDINATOR |
| 11 | PX SEND QC (RANDOM) |
| :TQ10001 |
| 12 | LOAD AS SELECT (CURSOR DURATION MEMORY) |
SYS_TEMP_0FD9D6669_37AA44 |
| 13 | SORT GROUP BY |

```

```

| 14 |      PX RECEIVE
|
| 15 |      PX SEND HASH
| :TQ10000
| 16 |      SORT GROUP BY
|
| 17 |      NESTED LOOPS
|
| 18 |      PX BLOCK ITERATOR
|
| * 19 |      TABLE ACCESS FULL
SYS_TEMP_0FD9D6669_37AA44 |
| 20 |      PARTITION HASH ALL
|
| * 21 |      TABLE ACCESS BY LOCAL INDEX ROWID BATCHED
TEST_GRAPHGT$ |
| * 22 |      INDEX RANGE SCAN
TEST_GRAPHXSG$ |
| 23 |      PX COORDINATOR
|
| 24 |      PX SEND QC (RANDOM)
| :TQ30001
| 25 |      VIEW
|
| 26 |      HASH UNIQUE
|
| 27 |      PX RECEIVE
|
| 28 |      PX SEND HASH
| :TQ30000
| 29 |      HASH UNIQUE
|
| 30 |      VIEW
|
| 31 |      UNION-ALL
|
| 32 |      PX SELECTOR
|
| * 33 |      FILTER
|
| 34 |      FAST DUAL
|
| 35 |      PARTITION HASH SINGLE
|
| * 36 |      INDEX SKIP SCAN
TEST_GRAPHXQV$ |
| 37 |      VIEW
|
| * 38 |      VIEW
|
| 39 |      PX BLOCK ITERATOR
|
| 40 |      TABLE ACCESS FULL
SYS_TEMP_0FD9D6669_37AA44 |
-----
-----
    
```

The query plan for the third query that uses CONNECTY BY should look similar to the following.

Id	Operation	Name
0	SELECT STATEMENT	
1	VIEW	
2	HASH UNIQUE	
3	VIEW	
4	UNION-ALL	
* 5	FILTER	
6	FAST DUAL	
7	PARTITION HASH SINGLE	
* 8	INDEX SKIP SCAN	TEST_GRAPHXQV\$
* 9	VIEW	
* 10	CONNECT BY WITH FILTERING	
11	PX COORDINATOR	
12	PX SEND QC (RANDOM)	:TQ10000
13	PX PARTITION HASH ALL	
* 14	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	TEST_GRAPHGT\$
* 15	INDEX RANGE SCAN	TEST_GRAPHXSG\$
16	NESTED LOOPS	
17	CONNECT BY PUMP	
18	PARTITION HASH ALL	
* 19	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	TEST_GRAPHGT\$
* 20	INDEX RANGE SCAN	TEST_GRAPHXSG\$

Example 6-20 PgqlExample13.java

PgqlExample13.java shows how to set length restrictions during path query evaluation.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use the maximum path length option for
 * PGQL path queries.
 */
public class PgqlExample13
{
    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
```

```

PgqlResultSet rs = null;

try {

    //Get a jdbc connection
    PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@"+host+": "+port +": "+sid);
pds.setUser(user);
pds.setPassword(password);
conn = pds.getConnection();

    // Get a PGQL connection
    PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
    pgqlConn.setGraph(graph);

    // Create a PgqlStatement
    ps = pgqlConn.createStatement();

    // Path Query to illustrate options
    String pgql =
        "PATH fof AS ()-[:\"friendOf\"]->() "+
        "SELECT v1.\"fname\" AS fname1, v2.\"fname\" AS fname2 "+
        "FROM MATCH (v1)-/:fof*/->(v2) "+
        "WHERE v1.\"fname\" = 'Ray'";

    // execute query for 1-hop
    rs = ps.executeQuery(pgql, " MAX_PATH_LEN=1 ");

    // print results
    System.out.println("-- Results for 1-hop -----");
    rs.print();

    // close result set
    rs.close();

    // execute query for 2-hop
    rs = ps.executeQuery(pgql, " MAX_PATH_LEN=2 ");

    // print results
    System.out.println("-- Results for 2-hop -----");
    rs.print();

    // close result set
    rs.close();

    // execute query for 3-hop
    rs = ps.executeQuery(pgql, " MAX_PATH_LEN=3 ");

    // print results
    System.out.println("-- Results for 3-hop -----");
    rs.print();
}

```



```

    // close result set
    rs.close();

  }
  finally {
    // close the result set
    if (rs != null) {
      rs.close();
    }
    // close the statement
    if (ps != null) {
      ps.close();
    }
    // close the connection
    if (conn != null) {
      conn.close();
    }
  }
}
}
}

```

PgqlExample13.java has the following output for test_graph (which can be loaded using GraphLoaderExample.java code).

```

-- Results for 1-hop -----
+-----+
| FNAME1 | FNAME2 |
+-----+
| Ray    | Ray    |
| Ray    | Susan  |
+-----+
-- Results for 2-hop -----
+-----+
| FNAME1 | FNAME2 |
+-----+
| Ray    | Susan  |
| Ray    | Ray    |
| Ray    | John   |
+-----+
-- Results for 3-hop -----
+-----+
| FNAME1 | FNAME2 |
+-----+
| Ray    | Susan  |
| Ray    | Bill   |
| Ray    | Ray    |
| Ray    | John   |
+-----+

```

6.8.1.4.6.4 Options for Partial Object Construction

When reading edges from a query result, there are two possible behaviors when adding the start and end vertex to any local caches:

- Add only the vertex ID, which is available from the edge itself. This option is the default, for efficiency.

- Add the vertex ID, and retrieve all properties for the start and end vertex. For this behavior, you can call `setPartial(true)` on each `OracleVertex` object constructed from your PGQL query result set.

Example 6-21 `PgqlExample14.java`

`PgqlExample14.java` illustrates this difference in behavior. This program first executes a query to retrieve all edges, which causes the incident vertices to be added to a local cache. The second query retrieves all vertices. The program then prints each `OracleVertex` object to show which properties have been loaded.

```
import java.sql.Connection;

import oracle.pg.rdbms.Oracle;
import oracle.pg.rdbms.OraclePropertyGraph;
import oracle.pg.rdbms.OracleVertex;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows the behavior of setPartial(true) for OracleVertex
 * objects
 * created from PGQL query results.
 */
public class PgqlExample14
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        Oracle oracle = null;
        OraclePropertyGraph opg = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
```

```

pds.setPassword(password);
conn = pds.getConnection();

// Get a PGQL connection
PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
pgqlConn.setGraph(graph);

// Create a PgqlStatement
ps = pgqlConn.createStatement();

// Query to illustrate set partial
String pgql =
    "SELECT id(e), label(e) "+
    "FROM MATCH (v1)-[e:\\"knows\\"]->(v2)";

// execute query
rs = ps.executeQuery(pgql, " ");

// print results
System.out.println("-- Results for edge query -----");
rs.print();

// close result set
rs.close();

// Create an Oracle Property Graph instance
oracle = new Oracle(conn);
opg = OraclePropertyGraph.getInstance(oracle, graph);

// Query to retrieve vertices
pgql =
    "SELECT id(v) "+
    "FROM MATCH (v)";

// Get each vertex object in result and print with toString()
rs = ps.executeQuery(pgql, " ");

// iterate through result
System.out.println("-- Vertex objects retrieved from vertex query --");
while (rs.next()) {
    Long vid = rs.getLong(1);
    OracleVertex v = OracleVertex.getInstance(opg, vid);
    System.out.println(v.toString());
}
// close result set
rs.close();

// Execute the same query but call setPartial(true) for each vertex
rs = ps.executeQuery(pgql, " ");
System.out.println("-- Vertex objects retrieved from vertex query with
setPartial(true) --");
while (rs.next()) {
    Long vid = rs.getLong(1);
    OracleVertex v = OracleVertex.getInstance(opg, vid);
    v.setPartial(true);
}

```



```

mval:bol:false, age:int:35}
Vertex ID 0 [NULL] {bval:bol:true, fname:str:Bill, lname:str:Brown, mval:str:y,
age:int:40}
Vertex ID 2 [NULL] {fname:str:Ray, lname:str:Green, mval:dat:1985-01-01 04:00:00.0,
age:int:41}
Vertex ID 1 [NULL] {bval:bol:true, fname:str:John, lname:str:Black, mval:int:27,
age:int:30}

```

6.8.1.4.7 Querying Another User's Property Graph

You can query another user's property graph data if you have been granted the appropriate privileges in the database. For example, to query GRAPH1 in SCOTT's schema, you must have READ privilege on SCOTT.GRAPH1GE\$, SCOTT.GRAPH1VT\$, SCOTT.GRAPH1GT\$, and SCOTT.GRAPH1VD\$.

Example 6-22 PgqlExample15.java

PgqlExample15.java shows how another user can query a graph in SCOTT's schema.

```

import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to query a property graph located in another user's
 * schema. READ privilege on GE$, VT$, GT$ and VD$ tables for the other
 * user's
 * property graph are required to avoid ORA-00942: table or view does not
 * exist.
 */
public class PgqlExample15
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

```

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@"+host+": "+port +":"+sid);
pds.setUser(user);
pds.setPassword(password);
conn = pds.getConnection();

// Get a PGQL connection
PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
pgqlConn.setGraph(graph);

// Set schema so that we can query Scott's graph
pgqlConn.setSchema("SCOTT");

// Create a PgqlStatement
ps = pgqlConn.createStatement();

// Execute query to get a ResultSet object
String pgql =
    "SELECT v.\"fname\" AS fname, v.\"lname\" AS lname "+
    "FROM MATCH (v)";
rs = ps.executeQuery(pgql, "");

// Print query results
rs.print();
}
finally {
    // close the result set
    if (rs != null) {
        rs.close();
    }
    // close the statement
    if (ps != null) {
        ps.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}
```

The following SQL statements create database user USER2 and grant the necessary privileges. You can also use the `OraclePropertyGraph.grantAccess` Java API to achieve the same effect.

```
SQL> grant connect, resource, unlimited tablespace to user2 identified by user2;
```

```
Grant succeeded.
```

```
SQL> grant read on scott.test_graphvt$ to user2;
```

```
Grant succeeded.
```

```
SQL> grant read on scott.test_graphge$ to user2;

Grant succeeded.

SQL> grant read on scott.test_graphgt$ to user2;

Grant succeeded.

SQL> grant read on scott.test_graphvd$ to user2;

Grant succeeded.
```

The output for `PgqlExample15.java` for the `test_graph` data set when connected to the database as `USER2` is as follows. Note that `test_graph` should have already been loaded (using `GraphLoaderExample.java` code) as `GRAPH1` by user `SCOTT` before running `PgqlExample15`.

```
+-----+
| FNAME | LNAME |
+-----+
| Susan | Blue  |
| Bill  | Brown |
| Ray   | Green |
| John  | Black |
+-----+
```

6.8.1.4.8 Using Query Optimizer Hints with PGQL

The Java API allows query optimizer hints that influence the join type when executing PGQL queries. The `executeQuery` and `translateQuery` methods in `PgqlStatement` and `PgqlPreparedStatement` accept the following strings in the options argument to influence the query plan for the corresponding SQL query.

- `ALL_EDGE_NL` – Use Nested Loop join for all joins that involve the `$GE` and `$GT` tables.
- `ALL_EDGE_HASH` – Use HASH join for all joins that involve the `$GE` and `$GT` tables.
- `ALL_VERTEX_NL` – Use Nested Loop join for all joins that involve the `$VT` table.
- `ALL_VERTEX_HASH` – Use HASH join for all joins that involve the `$VT` table.

Example 6-23 `PgqlExample16.java`

`PgqlExample16.java` shows how to use optimizer hints to influence the joins used for a graph traversal.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlSqlQueryTrans;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to use query optimizer hints with PGQL queries.
 */
public class PgqlExample16
{
```

```

public static void main(String[] args) throws Exception
{
    int idx=0;
    String host          = args[idx++];
    String port          = args[idx++];
    String sid           = args[idx++];
    String user          = args[idx++];
    String password      = args[idx++];
    String graph         = args[idx++];

    Connection conn = null;
    PgqlStatement ps = null;

    try {

        //Get a jdbc connection
        PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
        pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
        pds.setUser(user);
        pds.setPassword(password);
        conn = pds.getConnection();

        // Get a PGQL connection
        PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
        pgqlConn.setGraph(graph);

        // Create a PgqlStatement
        ps = pgqlConn.createStatement();
        // Query to illustrate join hints
        String pgql =
            "SELECT id(v1), id(v4) "+
            "FROM MATCH (v1)-[:\"friendOf\"]->(v2)-[:\"friendOf\"]-
            >(v3)-[:\"friendOf\"]->(v4)";

        // get SQL translation with hash join hint
        PgqlSqlQueryTrans sqlTrans =
            ps.translateQuery(pgql /* query string */,
                " ALL_EDGE_HASH " /* options */);
        // print SQL translation
        System.out.println("-- Query with ALL_EDGE_HASH
        -----");
        System.out.println(sqlTrans.getSqlTranslation()+"\n");

        // get SQL translation with nested loop join hint
        sqlTrans =
            ps.translateQuery(pgql /* query string */,
                " ALL_EDGE_NL " /* options */);
        // print SQL translation
        System.out.println("-- Query with ALL_EDGE_NL
        -----");
        System.out.println(sqlTrans.getSqlTranslation()+"\n");
    }
}

```



```

    finally {
        // close the statement
        if (ps != null) {
            ps.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}
}
}

```

The output for PgqlExample16.java for test_graph (which can be loaded using GraphLoaderExample.java code) is:

```

-- Query with ALL_EDGE_HASH -----
SELECT /*+ USE_HASH(T0$0 T0$1 T0$2) */ 7 AS "id(v1)$T",
to_nchar(T0$0.SVID, 'TM9', 'NLS_Numeric_Characters=','.', ''') AS "id(v1)$V",
T0$0.SVID AS "id(v1)$VN",
to_timestamp_tz(null) AS "id(v1)$VT",
7 AS "id(v4)$T",
to_nchar(T0$2.DVID, 'TM9', 'NLS_Numeric_Characters=','.', ''') AS "id(v4)$V",
T0$2.DVID AS "id(v4)$VN",
to_timestamp_tz(null) AS "id(v4)$VT"
FROM "SCOTT".TEST_GRAPHGT$ T0$0,
"SCOTT".TEST_GRAPHGT$ T0$1,
"SCOTT".TEST_GRAPHGT$ T0$2
WHERE T0$0.DVID=T0$1.SVID AND
T0$1.DVID=T0$2.SVID AND
(T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL) AND
(T0$1.EL = n'friendOf' AND T0$1.EL IS NOT NULL) AND
(T0$2.EL = n'friendOf' AND T0$2.EL IS NOT NULL)

-- Query with ALL_EDGE_NL -----
SELECT /*+ USE_NL(T0$0 T0$1 T0$2) */ 7 AS "id(v1)$T",
to_nchar(T0$0.SVID, 'TM9', 'NLS_Numeric_Characters=','.', ''') AS "id(v1)$V",
T0$0.SVID AS "id(v1)$VN",
to_timestamp_tz(null) AS "id(v1)$VT",
7 AS "id(v4)$T",
to_nchar(T0$2.DVID, 'TM9', 'NLS_Numeric_Characters=','.', ''') AS "id(v4)$V",
T0$2.DVID AS "id(v4)$VN",
to_timestamp_tz(null) AS "id(v4)$VT"
FROM "SCOTT".TEST_GRAPHGT$ T0$0,
"SCOTT".TEST_GRAPHGT$ T0$1,
"SCOTT".TEST_GRAPHGT$ T0$2
WHERE T0$0.DVID=T0$1.SVID AND
T0$1.DVID=T0$2.SVID AND
(T0$0.EL = n'friendOf' AND T0$0.EL IS NOT NULL) AND
(T0$1.EL = n'friendOf' AND T0$1.EL IS NOT NULL) AND
(T0$2.EL = n'friendOf' AND T0$2.EL IS NOT NULL)

```

The query plan for the first query that uses ALL_EDGE_HASH should look similar to the following.

```

-----
| Id | Operation | Name |
-----

```

0	SELECT STATEMENT	
* 1	HASH JOIN	
* 2	HASH JOIN	
3	PARTITION HASH ALL	
* 4	TABLE ACCESS FULL	TEST_GRAPHGT\$
5	PARTITION HASH ALL	
* 6	TABLE ACCESS FULL	TEST_GRAPHGT\$
7	PARTITION HASH ALL	
* 8	TABLE ACCESS FULL	TEST_GRAPHGT\$

The query plan for the second query that uses ALL_EDGE_NL should look similar to the following.

Id	Operation	Name
0	SELECT STATEMENT	
1	NESTED LOOPS	
2	NESTED LOOPS	
3	PARTITION HASH ALL	
* 4	TABLE ACCESS FULL	TEST_GRAPHGT\$
5	PARTITION HASH ALL	
* 6	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	TEST_GRAPHGT\$
* 7	INDEX RANGE SCAN	TEST_GRAPHXSG\$
8	PARTITION HASH ALL	
* 9	TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	TEST_GRAPHGT\$
* 10	INDEX RANGE SCAN	TEST_GRAPHXSG\$

6.8.1.4.9 Modifying Property Graphs through INSERT, UPDATE, and DELETE Statements

PGQL supports INSERT, UPDATE, and DELETE operations on Property Graphs. The method `execute` in `PgqlStatement` lets you execute such DML operations. This topic provides several examples of such operations.



Note:

JDBC connection autocommit must be off in order to be able to execute INSERT, UPDATE, and DELETE statements.

Example 6-24 PgqlExample17.java (Insert)

`PgqlExample17.java` inserts several vertices and edges into a graph. Notice that the special property `_ora_id` is used to define ID values of vertices and edges. If the property `_ora_id` is omitted, a unique ID is generated for each new vertex or edge that is inserted into the graph.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
```

```

import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL INSERT operation.
 */
public class PgqlExample17
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute insert statement
            String pgql =
                "INSERT VERTEX p1 LABELS (person) PROPERTIES (p1.\"_ora_id\" = 1,
p1.fname = 'Jake') "+
                "          , VERTEX p2 LABELS (person) PROPERTIES (p2.\"_ora_id\" = 2,
p2.fname = 'Amy') "+
                "          , VERTEX p3 LABELS (person) PROPERTIES (p3.\"_ora_id\" = 3,
p3.fname = 'Erik') "+
                "          , VERTEX p4 LABELS (person) PROPERTIES (p4.\"_ora_id\" = 4,
p4.fname = 'Jane') "+
                "          , EDGE e1 BETWEEN p1 AND p2 LABELS (knows) PROPERTIES
(e1.\"_ora_id\" = 1, e1.since = DATE '2003-04-21') "+
                "          , EDGE e2 BETWEEN p1 AND p3 LABELS (knows) PROPERTIES
(e2.\"_ora_id\" = 2, e2.since = DATE '2010-02-10') "+
                "          , EDGE e3 BETWEEN p3 AND p4 LABELS (knows) PROPERTIES

```



```

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL UPDATE operation.
 */
public class PgqlExample18
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute update statement
            String pgql =
                "UPDATE p1 SET (p1.age = 47, p1.lname = 'Red'), "+
                "          p2 SET (p2.age = 29, p2.lname = 'White'), "+
                "          e SET (e.strength = 100) "+
                "FROM MATCH (p1) -[e:knows]-> (p2) "+
                "WHERE p1.fname = 'Jake' AND p2.fname = 'Amy'";
            ps.execute(pgql, /* query string */
                    "", /* query options */
                    "" /* modify options */);

            // Execute a query to verify update
            pgql =

```



```

* This example shows how to execute a PGQL DELETE operation.
*/
public class PgqlExample19
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();

            // Execute delete statement
            String pgql =
                "DELETE e "+
                " FROM MATCH (p1) -[e:knows]-> (p2) "+
                " WHERE p1.fname = 'Jake'";
            ps.execute(pgql, /* query string */
                    "", /* query options */
                    "" /* modify options */);

            // Execute a query to verify delete
            pgql =
                "SELECT p1.fname AS fname1, p2.fname AS fname2 "+
                " FROM MATCH (p1) -[e:knows]-> (p2)";
            rs = ps.executeQuery(pgql, "");

            // Print the results
            rs.print();
        }
    }
}

```

```

    finally {
        // close the result set
        if (rs != null) {
            rs.close();
        }
        // close the statement
        if (ps != null) {
            ps.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}
}
}

```

The output for `PgqlExample19.java` applied on a graph where `PgqlExample18.java` has been previously executed is:

```

+-----+
| FNAME1 | FNAME2 |
+-----+
| Erik   | Jane   |
+-----+

```

For more examples of DELETE statement, see the relevant section of the PGQL specification [here](#).

Example 6-27 PgqlExample20.java (Multiple Modifications)

`PgqlExample20.java` executes multiple modifications in the same statement: an edge is inserted, vertex properties are updated, and another edge is deleted.

```

import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to execute a PGQL
 * INSERT/UPDATE/DELETE operation.
 */
public class PgqlExample20
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid            = args[idx++];

```



```

String user          = args[idx++];
String password     = args[idx++];
String graph        = args[idx++];

Connection conn = null;
PgqlStatement ps = null;
PgqlResultSet rs = null;

try {

    //Get a jdbc connection
    PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
    pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
    pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
    pds.setUser(user);
    pds.setPassword(password);
    conn = pds.getConnection();
    conn.setAutoCommit(false);

    // Get a PGQL connection
    PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
    pgqlConn.setGraph(graph);

    // Create a PgqlStatement
    ps = pgqlConn.createStatement();

    // Execute INSERT/UPDATE/DELETE statement
    String pgql =
        "INSERT EDGE f BETWEEN p2 AND p1 LABELS (knows) PROPERTIES (f.since
= e.since) "+
        "UPDATE p1 SET (p1.age = 30) "+
        "      , p2 SET (p2.age = 25) "+
        "DELETE e "+
        " FROM MATCH (p1) -[e:knows]-> (p2) "+
        " WHERE p1.fname = 'Erik'";
    ps.execute(pgql, /* query string */
              "", /* query options */
              "" /* modify options */);

    // Execute a query to verify INSERT/UPDATE/DELETE
    pgql =
        "SELECT p1.fname AS fname1, p1.age AS age1, "+
        "       p2.fname AS fname2, p2.age AS age2, e.since "+
        " FROM MATCH (p1) -[e:knows]-> (p2)";
    rs = ps.executeQuery(pgql, "");

    // Print the results
    rs.print();
}
finally {
    // close the result set
    if (rs != null) {
        rs.close();
    }
    // close the statement
}

```

```

        if (ps != null) {
            ps.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}

```

The output for `PgqlExample20.java` applied on a graph where `PgqlExample19.java` has been previously executed is:

```

+-----+
| FNAME1 | AGE1 | FNAME2 | AGE2 | SINCE |
+-----+
| Jane   | 25   | Erik   | 30   | 1999-01-02 16:00:00.0 |
+-----+

```

For more examples of INSERT/UPDATE/DELETE statements, see the relevant section of the PGQL specification [here](#).

- [Additional Options for PGQL Statement Execution](#)

6.8.1.4.9.1 Additional Options for PGQL Statement Execution

Several options are available to influence PGQL statement execution. The following are the main ways to set query options:

- Through flags in the `modify options` string argument of `execute`
- Through Java JVM arguments.

The following table summarizes the main options for modifying PGQL statement execution.

Table 6-4 PGQL Statement Modification Options

Option	Default	Options Flag	JVM Argument
Auto commit	true if JDBC auto commit is off, false if JDBC auto commit is on	AUTO_COMMIT=F	- Doracle.pg.rdbms.pgql.autoCommit=false
Delete cascade	true	DELETE_CASCADE=F	- Doracle.pg.rdbms.pgql.deleteCascade=false

- [Turning Off PGQL Auto Commit](#)
- [Turning Off Cascading Deletion](#)

6.8.1.4.9.1.1 Turning Off PGQL Auto Commit

When an INSERT, UPDATE, or DELETE operation is executed, a commit is performed automatically at the end of the PGQL execution so that changes are persisted on the RDBMS side.

The flag `AUTO_COMMIT=F` can be added to the `options` argument of `execute` or the flag `Doracle.pg.rdbms.pgql.autoCommit=false` can be set in the Java command line to turn off auto commit. Notice that when auto commit is off, you must perform any

necessary commits or rollbacks on the JDBC connection in order to persist or cancel graph modifications.

Example 6-28 Turn Off Auto Commit and Roll Back Changes

PgqlExample21.java turns off auto commit and performs a rollback of the changes.

```
import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to modify a PGQL graph
 * with auto commit off.
 */
public class PgqlExample21
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;
        PgqlResultSet rs = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
            pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
            pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + ":" + sid);
            pds.setUser(user);
            pds.setPassword(password);
            conn = pds.getConnection();
            conn.setAutoCommit(false);

            // Get a PGQL connection
            PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);
            pgqlConn.setGraph(graph);

            // Create a PgqlStatement
            ps = pgqlConn.createStatement();
```



```

+-----+
| 0      |
+-----+
Number of edges after rollback:
+-----+
| COUNT(e) |
+-----+
| 1        |
+-----+

```

6.8.1.4.9.1.2 Turning Off Cascading Deletion

When a vertex is deleted from a graph, all its input and output edges are also deleted automatically.

Using the flag `DELETE_CASCADE=F` in the `options` argument of `execute` or setting the flag or setting the flag `Doracle.pg.rdbms.pgql.autoCommit=false` in the Java command line lets you turn off cascading deletion. When a vertex with input or output edges is deleted and cascading deletion is off, an error is thrown to warn about the unsafe operation that you are trying to perform.

Example 6-29 Turn Off Cascading Deletion

`PgqlExample22.java` attempts to delete a vertex with an output edge when cascading deletion is off.

```

import java.sql.Connection;

import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.pg.rdbms.pgql.PgqlToSQLException;

import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows the use of DELETE_CASCADE flag.
 */
public class PgqlExample22
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String graph         = args[idx++];

        Connection conn = null;
        PgqlStatement ps = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

```


- [Basic Query Execution](#)
- [Iterating a Query Result Set](#)

6.8.1.5.1 Creating a Property Graph Using the Python Client

You can create a property graph using the `CREATE PROPERTY GRAPH` statement in Python.

Creating a Property Graph Using the Python Client

- Launch the Python client as shown:

```
./bin/opg4py --no_connect
```

- Create a PGQL connection to connect to the database as shown:

```
>>> pgql_conn = opg4py.pgql.get_connection(<user>, <password>, <jdbc_url>)
PgqlConnection(schema: GRAPHUSER, graph: None)
```

- Create a PGQL statement as shown:

```
>>> pgql_statement = pgql_conn.create_statement()
PgqlStatement(java_pgql_statement: oracle.pg.rdbms.pgql.PgqlStatement)
```

- Define and execute the `CREATE PROPERTY GRAPH` statement as shown:

```
pgql = """
CREATE PROPERTY GRAPH <graph_name>
  VERTEX TABLES (
    bank_accounts
      LABEL accounts
      PROPERTIES ALL COLUMNS
  )
  EDGE TABLES (
    bank_txns
      SOURCE KEY (from_acct_id) REFERENCES bank_accounts
      DESTINATION KEY (to_acct_id) REFERENCES bank_accounts
      LABEL transfers PROPERTIES ALL COLUMNS
  )
"""
```

where `<graph_name>` is the name of the graph.

```
pgql_statement.execute(pgql)
```

The graph gets created.

6.8.1.5.2 Dropping a Property Graph Using the Python Client

You can drop a property graph using the `DROP PROPERTY GRAPH` statement in Python.

Dropping a Property Graph Using the Python Client

- Define and execute the `DROP PROPERTY GRAPH` statement as shown:

```
>>> pgql = "DROP PROPERTY GRAPH <graph_name>"
```

where `<graph_name>` is the name of the graph.

```
>>> pgql_statement.execute(pgql)
```

The graph gets dropped.

6.8.1.5.3 Basic Query Execution

You can execute PGQL queries using the `opg4py.pgql` Python wrapper.

Executing PGQL Queries Using the Python Client

- Set the graph for querying as shown:

```
>>> pgql_conn.set_graph("<graph_name>")
```

where `<graph_name>` is the name of the graph.

- Define and execute the PGQL `SELECT` query. For example,

```
>>> pgql = "SELECT e.from_acct_id, e.to_acct_id, e.amount FROM
MATCH (n:accounts) -[e:transfers]-> (m:accounts) on bank_graph
limit 10"
```

- Execute and print the result set as shown:

```
>>> pgql_result_set = pgql_statement.execute_query(pgql)
>>> pgql_result_set.print()
```

```
+-----+
| FROM_ACCT_ID | TO_ACCT_ID | AMOUNT |
+-----+
| 781.0         | 712.0       | 1000.0 |
| 190.0         | 555.0       | 1000.0 |
| 191.0         | 329.0       | 1000.0 |
| 198.0         | 57.0        | 1000.0 |
| 220.0         | 441.0       | 1000.0 |
| 251.0         | 387.0       | 1000.0 |
| 254.0         | 188.0       | 1000.0 |
| 259.0         | 305.0       | 1000.0 |
| 261.0         | 145.0       | 1000.0 |
| 263.0         | 40.0        | 1000.0 |
+-----+
```

```
PgqlResultSet(java_pgql_result_set:
oracle.pg.rdbms.pgql.PgqlResultSet, # of results: 0)
```


6.8.1.5.4 Iterating a Query Result Set

You can iterate your query result set using the methods in `PgqlResultSet`.

You can position the cursor for iterating your query result set using the following methods:

- `first()` : boolean
- `next()` : boolean
- `previous()` : boolean
- `last()` : boolean
- `before_first()`
- `after_last()`
- `absolute(target_row_value)` : boolean
- `relative(offset_value)` : boolean

Once the cursor is positioned at the desired row, you can use the following getters to obtain values:

- `get(column_idx)` : Object
- `get(column_name)` : Object
- `get_boolean(column_idx)` : boolean
- `get_boolean(column_name)` : boolean
- `get_date(column_idx)` : `datetime.date`
- `get_date(column_name)` : `datetime.date`
- `get_float(column_idx)` : `Float`
- `get_float(column_name)` : `Float`
- `get_integer(column_idx)` : `Integer`
- `get_integer(column_name)` : `Integer`
- `get_list(column_idx)` : `List`
- `get_list(column_name)` : `List`
- `get_string(column_idx)` : `String`
- `get_string(column_name)` : `String`
- `get_time(column_idx)` : `datetime.time`
- `get_time(column_name)` : `datetime.time`
- `get_time_with_timezone(column_idx)` : `datetime.time`
- `get_time_with_timezone(column_name)` : `datetime.time`
- `get_timestamp(column_idx)` : `datetime.datetime`
- `get_timestamp(column_name)` : `datetime.datetime`
- `get_timestamp_with_timezone(column_idx)` : `datetime.datetime`
- `get_timestamp_with_timezone(column_name)` : `datetime.datetime`

- `get_value_type(column_idx)` : Integer
- `get_value_type(column_name)` : Integer
- `get_vertex_labels(column_idx)` : List
- `get_vertex_labels(column_name)` : List

See [Retrieving PGQL-on-RDBMS results](#) documentation for more information.

The following code samples illustrate cursor operations for iterating a result set using few of the cursor position and getter methods. These examples reference the query result set obtained in the [example](#) in the previous section.

```
# Call first() and retrieve value for "FROM_ACCT_ID"
>>> pgql_result_set.first()
True
>>> pgql_result_set.get("FROM_ACCT_ID")
781.0
```

```
# Call next() and retrieve value for "FROM_ACCT_ID"
>>> pgql_result_set.next()
True
>>> pgql_result_set.get("FROM_ACCT_ID")
978.0
```

```
# Call last() and retrieve value for "FROM_ACCT_ID"
>>> pgql_result_set.last()
True
>>> pgql_result_set.get("FROM_ACCT_ID")
842.0
```

```
# Call previous() and retrieve value for "FROM_ACCT_ID"
>>> pgql_result_set.previous()
True
>>> pgql_result_set.get("FROM_ACCT_ID")
838.0
```

```
# Reset the result set and offset by 6. Then retrieve value for
"FROM_ACCT_ID"
>>> pgql_result_set.before_first()
>>> pgql_result_set.relative(6)
True
>>> pgql_result_set.get("FROM_ACCT_ID")
925.0
```

```
# Reach the end of the result set and offset by -2. Then retrieve value
for "FROM_ACCT_ID"
>>> pgql_result_set.after_last()
>>> pgql_result_set.relative(-2)
True
>>> pgql_result_set.get("FROM_ACCT_ID")
838.0
```

```
# Call absolute() and provide an absolute row value. Then retrieve
value for "FROM_ACCT_ID"
>>> pgql_result_set.absolute(3)
```

```
True
>>> pgql_result_set.get_float("FROM_ACCT_ID")
900.0
```

Alternatively, you can also iterate through the query result set using the Python index operator as shown:

```
# Retrieving a value from a tuple
>>> pgql_result_set[4, "double", "FROM_ACCT_ID"]
907.0

# Retrieving a value using index value
>>> pgql_result_set[4].get("FROM_ACCT_ID")
907.0
```

6.8.1.6 Performance Considerations for PGQL Queries

Many factors affect the performance of PGQL queries in Oracle Database. The following are some recommended practices for query performance.

- [Query Optimizer Statistics](#)
- [Parallel Query Execution](#)
- [Optimizer Dynamic Sampling](#)
- [Bind Variables](#)
- [Path Queries](#)

Query Optimizer Statistics

Good, up-to-date query optimizer statistics are critical for query performance. Ensure that you run [OPG_APIS.ANALYZE_PG](#) after any significant updates to your property graph data.

Parallel Query Execution

Use parallel query execution to take advantage of Oracle's parallel SQL engine. Parallel execution often gives a significant speedup versus serial execution. Parallel execution is especially critical for path queries evaluated using the recursive WITH strategy.

See also the *Oracle Database VLDB and Partitioning Guide* for more information about parallel query execution.

Optimizer Dynamic Sampling

Due to the inherent flexibility of the graph data model, static information may not always produce optimal query plans. In such cases, dynamic sampling can be used by the query optimizer to sample data at run time for better query plans. The amount of data sampled is controlled by the dynamic sampling level used. Dynamic sampling levels range from 0 to 11. The best level to use depends on a particular dataset and workload, but levels of 2 (default), 6, or 11 often give good results.

See also Supplemental Dynamic Statistics in the *Oracle Database SQL Tuning Guide*.

Bind Variables

Use bind variables for constants whenever possible. The use of bind variables gives a very large reduction in query compilation time, which dramatically increases throughput for query workloads with queries that differ only in the constant values used. In addition, queries with bind variables are less vulnerable to injection attacks.

Path Queries

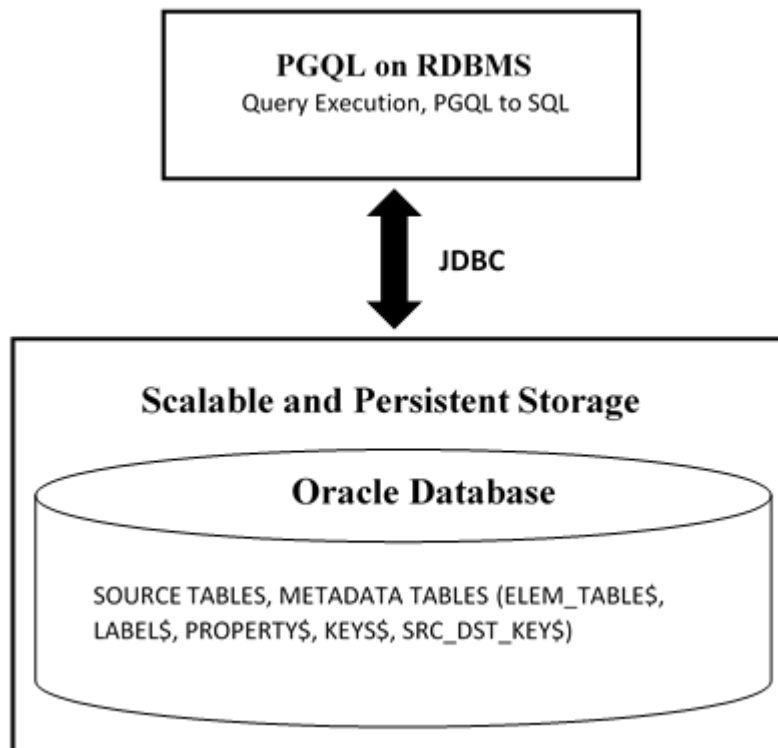
Path queries in PGQL that use the + (plus sign) or * (asterisk) operator to search for arbitrary length paths require special consideration because of their high computational complexity. You should use parallel execution and use the DISTINCT option for Recursive WITH (USE_DIST_RW=T) for the best performance. Also, for large, highly connected graphs, it is a good idea to use MAX_PATH_LEN=*n* to limit the number of repetitions of the recursive step to a reasonable number. A good strategy can be to start with a small repetition limit, and iteratively increase the limit to find more and more results.

6.8.2 Executing PGQL Queries Against Property Graph Views

This topic explains how you can execute PGQL queries directly against the property graph views on Oracle Database tables.

The PGQL query execution flow is shown in the following figure.

Figure 6-2 PGQL on Property Graph Views in Oracle Database



The basic execution flow is the same as in [PGQL on property graph schema tables](#). The only exception is that the PGQL query is translated into SQL statements using the internal metadata tables for property graph views.

- [PGQL Features Supported in Property Graph Views](#)
- [PGQL Limitations in Property Graph Views](#)
- [Performance Considerations for PGQL Queries](#)
- [Creating a Property Graph View](#)
- [Executing PGQL SELECT Queries](#)
- [Dropping A Property Graph View](#)

6.8.2.1 PGQL Features Supported in Property Graph Views

The Java API in `oracle.pg.rdbms.pgql` package provides support for executing PGQL SELECT queries with a few exceptions. See [PGQL Limitations in Property Graph Views](#) for more information.

The execution of recursive queries against property graph views are supported for the following features:

- Recursive queries are supported for the following variable-length path finding goals:
 - Reachability
 - ANY
 - ANY SHORTEST
 - TOP k SHORTEST
- Recursive queries are supported for the following horizontal aggregations:
 - LISTAGG

```
SELECT LISTAGG(src.first_name || ' ' || src.last_name, ',')
FROM MATCH TOP 2 SHORTEST ( (n:Person) ((src)-[e:knows]->)*
(m:Person) )
WHERE n.id = 1234
```

- SUM

```
SELECT SUM(e.weight + 3)
FROM MATCH TOP 2 SHORTEST ( (n:Person) -[e:knows]->* (m:Person) )
WHERE n.id = 1234
```

- COUNT

```
SELECT COUNT(e)
FROM MATCH TOP 2 SHORTEST ( (n:Person) -[e:knows]->* (m:Person) )
WHERE n.id = 1234
```

- AVG

```
SELECT AVG(dst.age)
FROM MATCH TOP 2 SHORTEST ( (n:Person) (-[e:knows]->(dst))*
```

```
(m:Person) )
WHERE n.id = 1234
```

- MIN (Only for property value or CAST expressions)

```
SELECT MIN(CAST(dst.age + 5 AS INTEGER))
FROM MATCH TOP 2 SHORTEST ( (n:Person) (-[e:knows]->(dst))*
(m:Person) )
WHERE n.id = 1234
```

- MAX (Only for property value or CAST expressions)

```
SELECT MAX(dst.birthday)
FROM MATCH TOP 2 SHORTEST ( (n:Person) (-[e:knows]->(dst))*
(m:Person) )
WHERE n.id = 1234
```

See [Performance Considerations for PGQL Queries](#) for details on recommended practices to enhance query performance for recursive queries.

6.8.2.2 PGQL Limitations in Property Graph Views

PGQL UPDATE queries are not supported for property graph views.

Also, the following PGQL SELECT features are not supported:

- The only quantifier supported for recursive queries is *.
If you attempt to use a different quantifier, it will result in an error as shown:

```
jshell> String s = "SELECT id(a) FROM MATCH ANY SHORTEST ((a) -[e]->+ (b))";
s ==> "SELECT id(a) FROM MATCH ANY SHORTEST ((a) -[e]->+ (b))"

jshell> PgqlStatement stmt = pgqlConn.createStatement();
stmt ==> oracle.pg.rdbms.pgql.PgqlExecution@27b9d5b7

jshell> stmt.execute(s);
| Exception java.lang.UnsupportedOperationException: Only zero (0)
or more path quantifier is supported
```

- Use of bind variables in path expressions.
If you attempt to use a bind variable, it will result in an error as shown:

```
jshell> String s = "SELECT id(a) FROM MATCH ANY SHORTEST (a) -[e]->* (b) WHERE id(a) = ?";
s ==> "SELECT id(a) FROM MATCH ANY SHORTEST (a) -[e]->* (b) WHERE
id(a) = ?"

jshell> PgqlPreparedStatement ps = pgqlConn.prepareStatement(s);
ps ==> oracle.pg.rdbms.pgql.PgqlExecution@7806db3f

jshell> ps.setString(1, "PERSON(3)");

jshell> ps.executeQuery();
```

```
| Exception java.lang.UnsupportedOperationException: Use of bind
variables for path queries is not supported
```

- Using subqueries.
- `in_degree` and `out_degree` functions.
- Any-directed edge patterns (for example, `-[e]-`).

6.8.2.3 Performance Considerations for PGQL Queries

The following are some recommended practices for query performance.

Recursive Queries

The following indexes are recommended in order to speed up execution of recursive queries:

- For underlying VERTEX tables of the recursive pattern, an index on the key column
- For underlying EDGE tables of the recursive pattern, an index on the source key column

Note:

You can also create index on (source key, destination key).

For example, consider the following CREATE PROPERTY GRAPH statement:

```
CREATE PROPERTY GRAPH people
  VERTEX TABLES(
    person
    KEY ( id )
    LABEL person
    PROPERTIES( name, age )
  )
  EDGE TABLES(
    knows
    key (person1, person2)
    SOURCE KEY ( person1 ) REFERENCES person
    DESTINATION KEY ( person2 ) REFERENCES person
    NO PROPERTIES
  )
  OPTIONS ( PG_VIEW )
```

And also consider the following query:

```
SELECT COUNT(*)
FROM MATCH ANY SHORTEST ( (n:Person) -[e:knows]->* (m:Person) )
WHERE n.id = 1234
```

In order to improve performance of the recursive part of the preceding query, the following indexes must exist:

- `CREATE INDEX <INDEX_NAME> ON PERSON(ID)`
- `CREATE INDEX <INDEX_NAME> ON KNOWS(PERSON1) OR`

```
CREATE INDEX <INDEX_NAME> ON KNOWS(PERSON1, PERSON2)
```

Composite Vertex Keys

For composite vertex keys, query execution can be optimized with the creation of function-based indexes on the key columns:

- For underlying VERTEX tables of the recursive pattern, a function-based index on the comma-separated concatenation of key columns
- For underlying EDGE tables of the recursive pattern, a function-based index on the comma-separated concatenation of source key columns

Note:

You can also create index on (source key columns, destination key columns).

For example, consider the following CREATE PROPERTY GRAPH statement:

```
CREATE PROPERTY GRAPH people
  VERTEX TABLES(
    person
    KEY ( id1, id2 )
    LABEL person
    PROPERTIES( name, age )
  )
  EDGE TABLES(
    knows
    key (id)
    SOURCE KEY ( id1person1, id2person1 ) REFERENCES person
    DESTINATION KEY ( id1person2, id2person2 ) REFERENCES person
    NO PROPERTIES
  )
  OPTIONS ( PG_VIEW )
```

And also consider the following query:

```
SELECT COUNT(*)
FROM MATCH ANY SHORTEST ( (n:Person) -[e:knows]->* (m:Person) )
WHERE n.id = 1234
```

In order to improve performance of the recursive part of the preceding query, the following indexes must exist:

- CREATE INDEX <INDEX_NAME> ON PERSON (ID1 || ',' || ID2)
- CREATE INDEX <INDEX_NAME> ON KNOWS (ID1PERSON1 || ',' || ID2PERSON1) or
CREATE INDEX <INDEX_NAME> ON KNOWS (ID1PERSON1 || ',' || ID2PERSON1,
ID1PERSON2 || ',' || ID2PERSON2)

If some of the columns in a composite vertex key is a string column, the column needs to be comma-escaped in the function-based index creation.

For example, if column `ID1` in table `PERSON` of the preceding example is of type `VARCHAR2(10)`, you need to escape the comma for the column as follows:

```
replace(ID1, ',', '\,')
```

So, the indexes to improve performance will result as shown:

- `CREATE INDEX <INDEX_NAME> ON PERSON (replace(ID1, ',', '\,') || ',' || ID2)`
- `CREATE INDEX <INDEX_NAME> ON KNOWS (replace(ID1PERSON1, ',', '\,') || ',' || ID2PERSON1)`

6.8.2.4 Creating a Property Graph View

You can create a property graph view using the `CREATE PROPERTY GRAPH` statement.

Creating a Property Graph View Using JShell

To create a property graph view using JShell, execute the following steps as shown:

1. Launch the JShell to work with the database as shown:

```
./bin/opg4j --no_connect
```

2. Connect to the Oracle Database as shown:

```
opg4j> var jdbcUrl="jdbc:oracle:thin:@<host_name>:<port>/<service>"
jdbcUrl ==> "jdbc:oracle:thin:@<host_name>:<port>/<db_service>"
opg4j> var conn =
DriverManager.getConnection(jdbcUrl,"<username>","<password>");
conn ==> oracle.jdbc.driver.T4CConnection@3e9a20e
```

In the preceding code:

- `<host_name>`: database host
 - `<port>`: database port
 - `<service>`: database SID
 - `<username>`: user name
 - `<password>`: database password
3. Create a PGQL Connection as shown:

```
opg4j> var pgqlConn = PgqlConnection.getConnection(conn)
pgqlConn ==> oracle.pg.rdbms.pgql.PgqlConnection@4301fa39
```

4. Execute the following commands as shown to create a property graph view:

```
opg4j> var pgqlStmt = pgqlConn.createStatement(); //create a PGQL
Statement
pgqlStmt ==> oracle.jdbc.driver.OracleStatementWrapper@6f976c
opg4j> String pgql =
...> "CREATE PROPERTY GRAPH <pgview> "
...> + "VERTEX TABLES ( bank_nodes AS Accounts "
```

```

...> + "KEY (id) "
...> + "LABEL Accounts "
...> + "PROPERTIES (id, label) "
...> + " "
...> + "EDGE TABLES ( bank_edges_amt AS Transfers "
...> + "KEY (src_id, dest_id, amount) "
...> + "SOURCE KEY (src_id) REFERENCES Accounts "
...> + "DESTINATION KEY (dest_id) REFERENCES Accounts "
...> + "LABEL Transfers "
...> + "PROPERTIES (src_id, dest_id, amount, label) "
...> + " " OPTIONS (PG_VIEW) ";
opg4j> pgqlStmt.execute(pgql);
$8 ==> false

```

The property graph view is created.

Creating a Property Graph View Using Java

The following example shows how to create a property graph view from the relational database tables using Java. The examples assumes the tables `bank_nodes` and `bank_edges_amt` already exist in the database.

```

import java.sql.Connection;
import java.sql.Statement;
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

/**
 * This example shows how to create a Property Graph View from
relational
 * data stored in Oracle Database by executing a PGQL create statement.
 */
public class CreatePgView
{

    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host          = args[idx++];
        String port          = args[idx++];
        String sid           = args[idx++];
        String user          = args[idx++];
        String password      = args[idx++];
        String pgview       = args[idx++];

        Connection conn = null;
        PgqlStatement pgqlStmt = null;

        try {

            //Get a jdbc connection
            PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

```


SELECT Query Execution on a Property Graph View Using JShell

1. Launch the JShell to work with the database as shown:

```
./bin/opg4j --no_connect
```

2. Execute the following commands to connect to the database and to run any SELECT query on a property graph view:

```
opg4j> var jdbcUrl="jdbc:oracle:thin:@<host_name>:<port>/
<db_service>"
opg4j> var conn =
DriverManager.getConnection(jdbcUrl,"<username>","<password>");
opg4j> var pgqlConn = PgqlConnection.getConnection(conn)
opg4j> var pgqlStmt = pgqlConn.createStatement() //create a PGQL
Statement
opg4j> String s = "SELECT n.id FROM MATCH (n:Accounts) ON <pgview>
LIMIT 3"
opg4j> var rs = pgqlStmt.executeQuery(s)
opg4j> rs.print() //Prints the query result set
+-----+
| ID   |
+-----+
| 434  |
| 435  |
| 436  |
+-----+
```

SELECT Query Execution on a Property Graph View Using Java

The following example shows how to execute a SELECT query on a property graph view.

```
import java.sql.Connection;
import java.sql.Statement;
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlResultSet;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;
/**
 * This example shows how to execute a SELECT query on a property graph
 * view.
 */
public class ExecuteQueryOnPgView
{
    public static void main(String[] args) throws Exception
    {
        int idx=0;
        String host           = args[idx++];
        String port           = args[idx++];
        String sid             = args[idx++];
        String user           = args[idx++];
```

```

String password          = args[idx++];
String pgview           = args[idx++];

Connection conn = null;
PgqlStatement pgqlStmt = null;
PgqlResultSet rs = null;

try {
    //Get a jdbc connection
    PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
    pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
    pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + "/" + sid);
    pds.setUser(user);
    pds.setPassword(password);
    conn = pds.getConnection();
    conn.setAutoCommit(false);

    // Get a PGQL connection
    PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);

    // Create a PGQL Statement
    pgqlStmt = pgqlConn.createStatement();

    // Execute PGQL Query
    String query = "SELECT a,b,c,e1,e2,e3 FROM MATCH (a)-[e1]->(b)-[e2]->(c)-[e3]->(a) ON " + pgview + " LIMIT 3";
    rs = pgqlStmt.executeQuery(query);

    // Print the results
    rs.print();
}
finally {
    // close the result set
    if (rs != null) {
        rs.close();
    }
    // close the statement
    if (pgqlStmt != null) {
        pgqlStmt.close();
    }
    // close the connection
    if (conn != null) {
        conn.close();
    }
}
}
}

```

The resulting output is as shown:

```

+-----+
-----+

```

```

| A          | B          | C          |
E1          |           |           |
E3          |           |           |
+-----+
| ACCOUNTS(998) | ACCOUNTS(781) | ACCOUNTS(71) |
TRANSFERS(998,781,1000) | TRANSFERS(781,71,1000) |
TRANSFERS(71,998,1000) |
| ACCOUNTS(359) | ACCOUNTS(579) | ACCOUNTS(76) |
TRANSFERS(359,579,1000) | TRANSFERS(579,76,1000) |
TRANSFERS(76,359,1000) |
| ACCOUNTS(6) | ACCOUNTS(580) | ACCOUNTS(82) |
TRANSFERS(6,580,1000) | TRANSFERS(580,82,1000) |
TRANSFERS(82,6,1000) |
+-----+

```

6.8.2.6 Dropping A Property Graph View

You can use PGQL to drop property graph views. When a `DROP PROPERTY GRAPH` statement is called, all the metadata tables for the property graph view are dropped.

Drop a Property Graph View Using JShell

1. Launch the JShell to work with the database as shown:

```
./bin/opg4j --no_connect
```

2. Execute the following commands to drop a property graph view:

```

opg4j> var jdbcUrl="jdbc:oracle:thin:@<host_name>:<port>/
<db_service>"
opg4j> var conn =
DriverManager.getConnection(jdbcUrl,"<username>","<password>")
opg4j> var pgqlConn = PgqlConnection.getConnection(conn)
opg4j> var pgqlStmt = pgqlConn.createStatement() //create a PGQL
Statement
opg4j> pgqlStmt.execute("DROP PROPERTY GRAPH <pgview>")
$9 ==> false

```

Drop a Property Graph View Using Java

The following example shows how to drop a property graph view.

```

import java.sql.Connection;
import java.sql.Statement;
import oracle.pg.rdbms.pgql.PgqlConnection;
import oracle.pg.rdbms.pgql.PgqlStatement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;
/**
 * This example shows how to drop a property graph view.
 */
public class DropPgView

```

```

{
public static void main(String[] args) throws Exception
{
    int idx=0;
    String host           = args[idx++];
    String port           = args[idx++];
    String sid            = args[idx++];
    String user           = args[idx++];
    String password       = args[idx++];
    String pgview         = args[idx++];

    Connection conn = null;
    PgqlStatement pgqlStmt = null;

    try {
        //Get a jdbc connection
        PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
        pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
        pds.setURL("jdbc:oracle:thin:@" + host + ":" + port + "/" + sid);
        pds.setUser(user);
        pds.setPassword(password);
        conn = pds.getConnection();
        conn.setAutoCommit(false);

        // Get a PGQL connection
        PgqlConnection pgqlConn = PgqlConnection.getConnection(conn);

        // Create PGQL Statement
        pgqlStmt = pgqlConn.createStatement();

        String query = "DROP PROPERTY GRAPH " + pgview;
        pgqlStmt.execute(query);
    }
    finally {
        // close the statement
        if (pgqlStmt != null) {
            pgqlStmt.close();
        }
        // close the connection
        if (conn != null) {
            conn.close();
        }
    }
}
}

```

7

Graph Visualization Application

The Graph Visualization application enables interactive exploration and visualization of property graphs. It can also visualize graphs stored in the database.

- [About the Graph Visualization Application](#)
The Graph Visualization application is a single-page web application that works with the in-memory graph server (PGX).
- [How does the Graph Visualization Application Work](#)
The Graph Visualization application exposes its own web interface and REST endpoint and can execute PGQL queries against the in-memory graph server (PGX) or the Oracle Database (PGQL on RDBMS).
- [Using the Graph Visualization Application](#)
The principal points of entry for the Graph Visualization application are the query editor and the graph lists.
- [REST Endpoints for the Graph Visualization Application](#)
- [Kerberos Enabled Authentication for the Graph Visualization Application](#)
The Graph Visualization application can authenticate users with Kerberos authentication enabled.

7.1 About the Graph Visualization Application

The Graph Visualization application is a single-page web application that works with the in-memory graph server (PGX).

The in-memory graph analytics server can be deployed in embedded mode or in Apache Tomcat or Oracle Weblogic Server. Graph Visualization application takes PGQL queries as an input and renders the result visually. A rich set of client-side exploration and visualization features can reveal new insights into your graph data.

Graph Visualization application works with the in-memory analytics server. It can visualize graphs that have been loaded into the in-memory analytics server, either preloaded when the in-memory analytics server is started, or loaded at run-time by a client application and made available through the `graph.publish()` API.

7.2 How does the Graph Visualization Application Work

The Graph Visualization application exposes its own web interface and REST endpoint and can execute PGQL queries against the in-memory graph server (PGX) or the Oracle Database (PGQL on RDBMS).

By default, it uses PGX and therefore requires a running PGX server to function. Alternatively, you can configure Graph Visualization application to directly talk to the database via PGQL on RDBMS. Graph Visualization application does not have any UI to create graphs, it can only visualize graphs which are already loaded into PGX or Oracle Database. See [REST Endpoints for the Graph Visualization Application](#) for more information on the graph visualization REST endpoints.

See [Enabling the Graph Visualization Application](#) for more information on starting the Graph Visualization application.

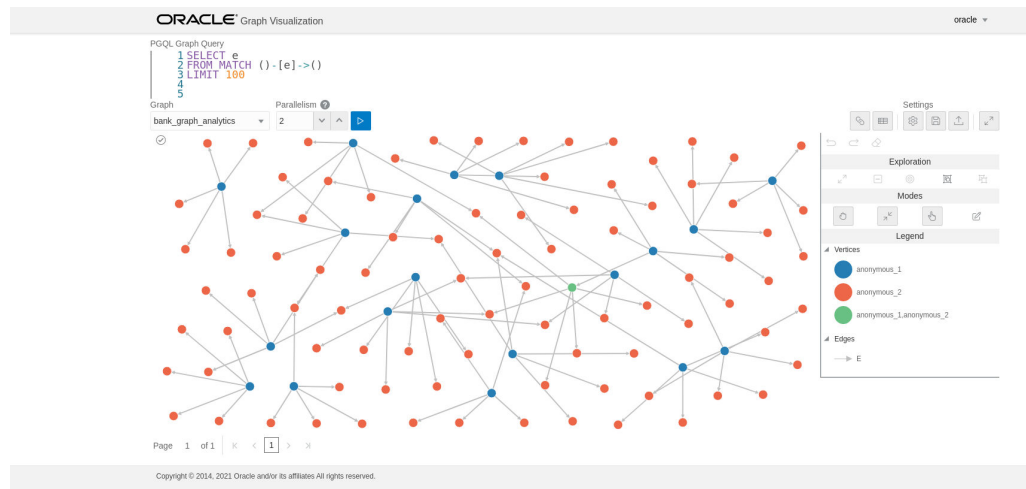
7.3 Using the Graph Visualization Application

The principal points of entry for the Graph Visualization application are the query editor and the graph lists.

When you start the graph visualization application, the graph list will be populated with the graphs loaded in PGX. To run queries against a graph, select that graph. The query lets you write PGQL queries that can be visualized. (PGQL is the SQL-like query language supported by the Graph Visualization application.)

Once the query is ready and the desired graph is selected, click the **Run** icon to execute the query. The following figure shows a query visualization identifying all edges that are directed edges from any vertex in the graph to any other vertex.

Figure 7-1 Query Visualization



When a query is successful, the graph visualization is displayed, including nodes and their connections. You can right-click a node or connection to display tooltip information, and you can drag the nodes around.

- [Graph Visualization Modes](#)
The buttons on the right let you switch between two modes: Graph Manipulation and Zoom/Move.
- [Graph Visualization Settings](#)
You can click the **Settings** gear icon to display the Graph Visualization settings window.
- [Using the Geographical Layout](#)
The Graph Visualization application offers a choice of layouts for rendering graphs. One of them is the Geographical layout that will show the graph (vertices and edges) on a global map.

- [Using Live Search](#)
Live Search lets you to search the displayed graph and add live fuzzy search score to each item, so you can create a Highlight which visually shows the results of the search in the graph immediately.
- [Using URL Parameters to Control the Graph Visualization Application](#)
You can provide the Graph Visualization application input data through URL parameters instead of using the form fields of the user interface.

7.3.1 Graph Visualization Modes

The buttons on the right let you switch between two modes: Graph Manipulation and Zoom/Move.

- **Graph Manipulation** mode lets you execute actions that modify the visualization. These actions include:
 - **Drop** removes selected vertices from visualization. Can also be executed from the tooltip.
 - **Group** selects multiple vertices and collapses them into a single one.
 - **Ungroup** selects a group of collapsed vertices and ungroups them.
 - **Expand** retrieves a configurable number of neighbors (hops) of selected vertices. Can also be executed from the tooltip.
 - **Focus**, like Expand, retrieves a configurable number of neighbors, but also drops all other vertices. Can also be executed from the tooltip.
 - **Undo** undoes the last action.
 - **Redo** redoes the last action.
 - **Reset** resets the visualization to the original state after the query.
- **Zoom/Move** mode lets you zoom in and out, as well as to move to another part of the visualization. The **Pan to Center** button resets the zoom and returns the view to the original one.

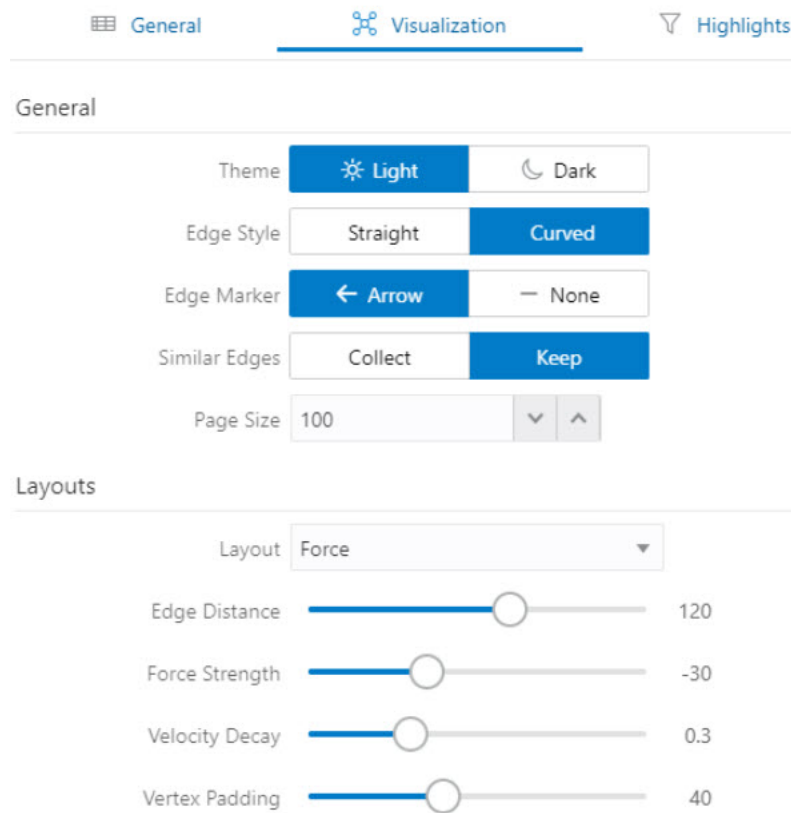
An additional mode, called **Sticky** mode, lets you cancel the action of dragging the nodes around.

7.3.2 Graph Visualization Settings

You can click the **Settings** gear icon to display the Graph Visualization settings window.

The settings window lets you modify some parameters for the visualization, and it has tabs for General, Visualization, and Highlights. The following figure shows this window, with the Visualization tab selected.

Figure 7-2 Graph Visualization Settings Window



The **General tab** includes the following:

- **Number of hops:** The configurable number of hops for the expand and focus actions.
- **Truncate label:** Truncates the label if it exceeds the maximum length.
- **Max. visible label length:** Maximum length before truncating.
- **Show Label On Hover:** Controls whether the label is shown on hover.
- **Display the graph legend:** Controls whether the legend is displayed.

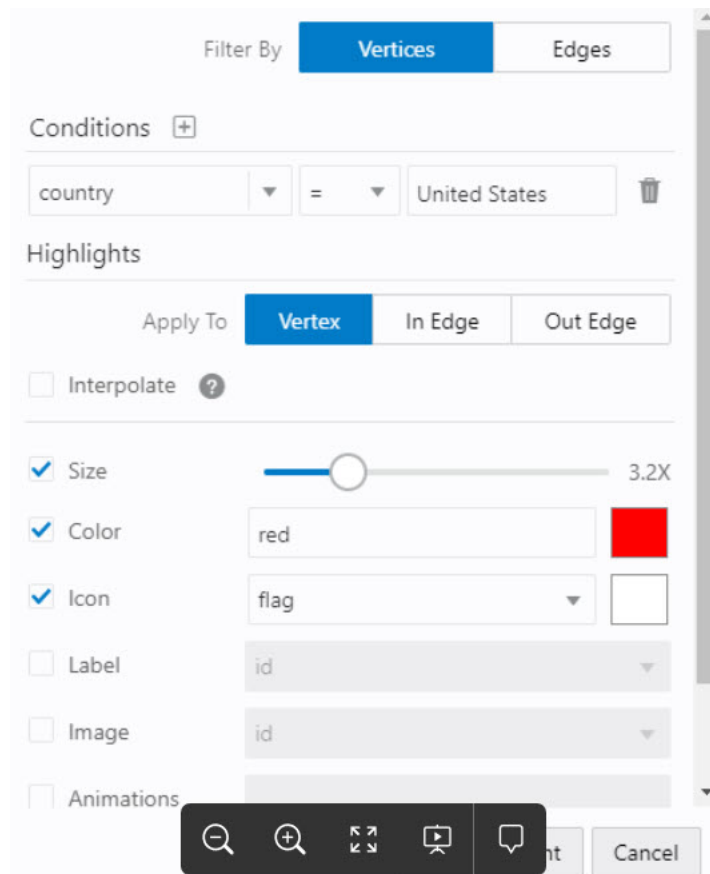
The **Visualization tab** includes the following:

- **Theme:** Select a light or dark mode.
- **Edge Style:** Select straight or curved edges.
- **Edge Marker:** Select arrows or no edge marker. This only applies to directed edges.
- **Similar Edges:** Select keep or collect.
- **Page Size:** Specify how many vertices and edges are displayed per page.
- **Layouts:** Select between different layouts (random, grid, circle, concentric, ...).
- **Vertex Label:** Select which property to use as the vertex label.
- **Vertex Label Orientation:** Select the relative position of the vertex label.

- **Edge Label:** Select which property to use as the edge label.

The **Highlights tab** includes customization options that let you modify the appearance of edges and vertices. Highlighting can be applied based on conditions (filters) on single or multiple elements. The following figure shows a condition (`country = United States`) and visual highlight options for vertices.

Figure 7-3 Highlights Options for Vertices



A filter for highlights can contain multiple conditions on any property of the element. The following conditions are supported.

- = (equal to)
- < (less than)
- <= (less than or equal to)
- > (greater than)
- >= (greater than or equal to)
- != (not equal to)
- ~ (filter is a regular expression)
- * (any: like a wildcard, can match to anything)

The visual highlight customization options include:

- Edges:
 - Width
 - Color
 - Label
 - Style
 - Animations
- Vertices:
 - Size
 - Color
 - Icon
 - Label
 - Image
 - Animations

You can export and import highlight options by clicking the Save and Import buttons in the main window. **Save** lets you persist the highlight options, and **Load** lets you apply previously saved highlight options.

When you click **Save**, a file is saved containing a JSON object with the highlights configuration. Later, you can load that file to restore the highlights of the saved session.

7.3.3 Using the Geographical Layout

The Graph Visualization application offers a choice of layouts for rendering graphs. One of them is the Geographical layout that will show the graph (vertices and edges) on a global map.

The following figure shows a graph rendered on a geographical layout in the Graph Visualization application:

Figure 7-4 Geographical Layout



In order to view your vertices on a map, they must include a geographical location, in the form of a pair of properties that contain the longitude and latitude coordinates for that vertex. For example:

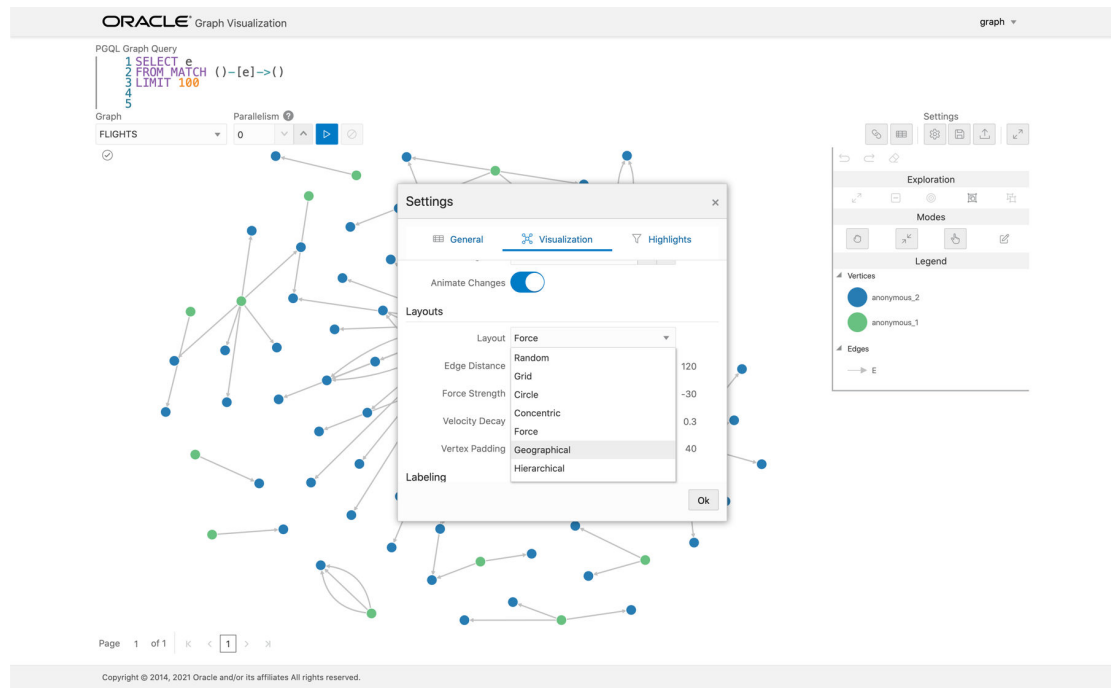
iata	city	longitude	latitude
SIN	Singapore	103.994003	1.35019
LAX	Los Angeles	-118.4079971	33.94250107
MUC	Munich	11.7861	48.353802
CDG	Paris	2.55	49.012798
LHR	London	-0.461941	51.4706

 **Note:**

You can use any name for the longitude and latitude properties (such as X and Y, or long and lat). But, you must ensure that the longitude/latitude pair are in the WGS84 system (GPS coordinates), and the coordinates are expressed in decimal degrees.

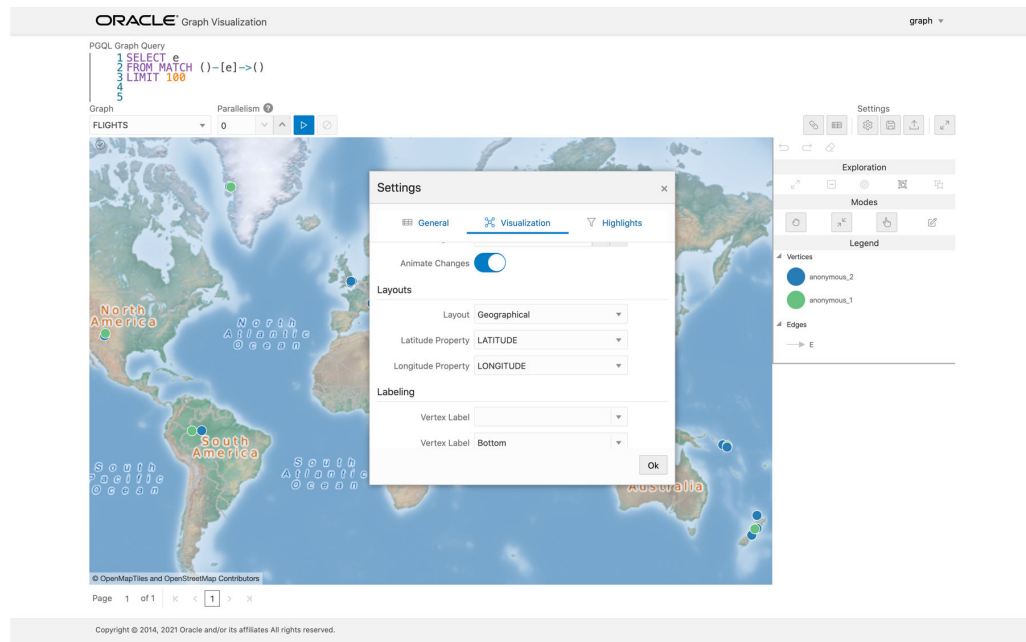
You can select the geographical layout in the Graph Visualization settings window as shown:

Figure 7-5 Setting Geographical Layout



Then, select the properties in your vertices that contain the geographical coordinates as shown:

Figure 7-6 Selecting the Coordinates for the Geographical layout



You can now move around the map and zoom in/out using your mouse or trackpad. From now on, whenever you enter a new PGQL query, the map will automatically center and zoom the vertices returned by the query.

7.3.4 Using Live Search

Live Search lets you to search the displayed graph and add live fuzzy search score to each item, so you can create a Highlight which visually shows the results of the search in the graph immediately.

If you run a query, and a graph is displayed, you can add the live search, which is on the settings dialog. On the bottom of the General tab, you will see these options.

- **Enable Live Search:** Enables the Live Search feature, adds the search input to the visualization, and lets you further customize the search.
- **Enable Search In:** You can select whether you want to search the properties of Vertices, Edges, or both.
- **Properties To Search:** Based on what you selected for Enable Search In, you can set one or more properties to search in. For example, if you disable the search for edges but you had a property from edges selected, it will be stored and added back when you enable search for the edges again. (This also works for vertices.)
- **Advanced Settings:** You can fine-tune the search even more. Each of the advanced options is documented with context help, visible upon enabling.
 - **Location:** Determines approximately where in the text the pattern is expected to be found.
 - **Distance:** Determines how close the match must be to the fuzzy location (specified by location). An exact letter match which is distance characters away from the fuzzy location would score as a complete mismatch. A distance

of 0 requires the match be at the exact location specified, a distance of 1000 would require a perfect match to be within 800 characters of the location to be found using a threshold of 0.8.

- **Maximum Pattern Length:** The maximum length of the pattern. The longer the pattern (that is, the search query), the more intensive the search operation will be. Whenever the pattern exceeds this value, an error will be thrown.
- **Min Char Match:** The minimum length of the pattern. Whenever the pattern length is below this value, an error will be thrown.

When the search is enabled, the input will be displayed in the top left part of the Graph Visualization component. If you start typing, the search will add a score to every vertex or edge, based on the settings and the search match.

To be able to see the results visually, you have to add a **Highlight** with interpolation set to a **Live Search** score and other settings based on the desired visual change.

7.3.5 Using URL Parameters to Control the Graph Visualization Application

You can provide the Graph Visualization application input data through URL parameters instead of using the form fields of the user interface.

If you supply the parameters in the URL, the Graph Visualization application automatically executes the specified query and hides the input form fields from the screen, so only the resulting visualization output is visible. This feature is useful if you want to embed the resulting graph visualization into an existing application, such as through an iframe.

The following table specifies the available URL parameters:

Table 7-1 Available URL Parameters

Parameter Name	Value (must be URL encoded)	Type	Optional?
graph	Graph name	string	No
parallelism	Degree of parallelism desired	number	Yes (defaults to server-side default parallelism)
query	PQL query	string	No

The following URL shows an example of visualizing the PGQL query `SELECT v, e MATCH (v) -[e]-> () LIMIT 10` on graph `myGraph` with parallelism 4:

```
https://myhost:7007/ui/?query=SELECT%20v%2C%20e%20MATCH%20%28v%29%20-%5Be%5D-%3E%20%28%29%20LIMIT%2010&graph=myGraph&parallelism=4
```

7.4 REST Endpoints for the Graph Visualization Application

This section explains all the REST endpoints through which you can perform various operations using the Graph Visualization Application.

The following are the available REST endpoints:

 **Note:**

The examples shown in the REST endpoints assume that:

- The PGX server is up and running on `https://localhost:7007`.
- Linux with `cURL` is installed. `cURL` is used to demonstrate how to access the `graph.publish` API using the CA certificate for verifying the graph server.

- [Login](#)
- [List Graphs](#)
- [Run a PGQL Query](#)
- [Get User](#)
- [Asynchronous REST Endpoints](#)

7.4.1 Login

HTTP Request: POST `https://localhost:7007/ui/v1/login/`

Authentication: Uses cookie-based authentication.

Table 7-2 Parameters

Parameter	Parameter Type	Value
Content-type	Header	application/x-www-form-urlencoded
username	Body	<username>
password	Body	<password>

Request

The following `curl` command signs the user in to the Graph Visualization application:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -X POST -H "Content-Type: application/x-www-form-urlencoded" -d 'username=oracle&password=<user password>' -c cookie.txt https://localhost:7007/ui/v1/login/
```

Response: None

On successful login, the server session cookie is stored in a cookie file, `cookie.txt`. Use this cookie file, in the subsequent calls to the API.

7.4.2 List Graphs

HTTP Request: GET `https://localhost:7007/ui/v1/graphs`
Request

The following `curl` command lists all the graphs that belong to the user:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt 'https://localhost:7007/ui/v1/graphs'
```

Response: The list of the available graphs for the current user. For example:

```
["hr", "bank_graph_analytics"]
```

7.4.3 Run a PGQL Query

HTTP Request: GET `https://localhost:7007/ui/v1/query?pgql=<PGQL query>&graph=<graph>¶llelism=<value>&size=<size value>`

Table 7-3 Query Parameters

Parameter	Description	Values
pgql	PGQL query string	<PGQL query>
graph	Name of the graph	<graph_name>
parallelism	Degree of Parallelism	<parallelism_value>
size	Fetch size (= the number of rows) of the query result	<size_value>

Request

The following `curl` command executes PGQL Query on a property graph:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt 'https://localhost:7007/ui/v1/query?pgql=SELECT%20e%0AMATCH%20()-%5Be%5D-%3E()%0ALIMIT%205&graph=hr&parallelism=&size=100'
```

Response: The PGQL query result in JSON format.

```
{
  "name": "bank_graph_analytics_2",
  "resultSetId": "pgql_14",
  "graph": {
    "idType": "number",
    "vertices": [
      {
        "_id": "1",
        "p": [],
        "l": [
          "Accounts"
        ],
        "g": [
          "anonymous_1"
        ]
      }
    ],
  },
}
```

```
{
  "_id": "418",
  "p": [],
  "l": [
    "Accounts"
  ],
  "g": [
    "anonymous_2"
  ]
},
{
  "_id": "259",
  "p": [],
  "l": [
    "Accounts"
  ],
  "g": [
    "anonymous_2"
  ]
}
],
"edges": [
  {
    "_id": "0",
    "p": [
      {
        "n": "AMOUNT",
        "v": "1000.0",
        "s": false
      }
    ],
    "l": [
      "Transfers"
    ],
    "g": [
      "e"
    ],
    "s": "1",
    "d": "259",
    "u": false
  },
  {
    "_id": "1",
    "p": [
      {
        "n": "AMOUNT",
        "v": "1000.0",
        "s": false
      }
    ],
    "l": [
      "Transfers"
    ],
    "g": [
      "e"
    ]
  }
]
```

```

    ],
    "s": "1",
    "d": "418",
    "u": false
  }
],
"paths": [],
"totalNumResults": 2
},
"table":
"e\nPgxEde[provider=Transfers, ID=0]\nPgxEde[provider=Transfers, ID=1]"
}

```

7.4.4 Get User

HTTP Request: GET <https://localhost:7007/ui/v1/user>

Request

The following `curl` command gets the name of the current user:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt 'https://localhost:7007/ui/v1/user'
```

Response: The name of the current user. For example:

```
"oracle"
```

7.4.5 Asynchronous REST Endpoints

Graph Visualization REST endpoints support cancellation of queries.

In order to be able to cancel queries, you need to send the query using the following asynchronous REST endpoints:

- [Run a PGQL Query Asynchronously](#)
- [Check a Query Completion](#)
- [Cancel a Query Execution](#)
- [Retrieve a Query Result](#)

7.4.5.1 Run a PGQL Query Asynchronously

HTTP Request: GET <https://localhost:7007/ui/v1/async-query?pgql=<PGQL query>&graph=<graph>¶llelism=<value>&size=<size value>>

See [Table 7-3](#) for more information on query parameters.

Request

The following `curl` command executes a PGQL query asynchronously on a property graph:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt  
'https://localhost:7007/ui/v1/async-query?pgql=SELECT%20e%0AMATCH%20()-  
%5Be%5D-%3E()%0ALIMIT%205&graph=hr&parallelism=&size=100'
```

Response: None.



Note:

An error message will be returned in case the query is malformed or if the graph does not exist.

7.4.5.2 Check a Query Completion

HTTP Request: GET `https://localhost:7007/ui/v1/async-query-complete`

Request

The following `curl` command checks if the PGQL query execution is completed:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt  
'https://localhost:7007/ui/v1/async-query-complete'
```

Response: A boolean that indicates if the query execution is completed. For example,

`true`



Note:

You do not have to specify any request ID, as the currently executing query is attached to your HTTP session. You can only have one query executing per session. For concurrent query execution, create multiple HTTP sessions by logging in multiple times.

7.4.5.3 Cancel a Query Execution

HTTP Request: DELETE `https://localhost:7007/ui/v1/async-query`

Request

The following `curl` command cancels a currently executing PGQL Query on a property graph:

```
curl -X DELETE --cacert /etc/oracle/graph/ca_certificate.pem -b  
cookie.txt 'https://localhost:7007/ui/v1/async-query'
```

Response: Confirmation of the cancellation or an error message if the query has already completed execution.

7.4.5.4 Retrieve a Query Result

HTTP Request: GET `https://localhost:7007/ui/v1/async-result?pgql=<PGQL query>&graph=<graph>¶llelism=<value>&size=<size value>`

See [Table 7-3](#) for more information on query parameters.

Request

The following `curl` command retrieves the result of a successfully completed query:

```
curl --cacert /etc/oracle/graph/ca_certificate.pem -b cookie.txt 'https://localhost:7007/ui/v1/async-result?pgql=SELECT%20e%0AMATCH%20()-%5Be%5D-%3E()%0ALIMIT%205&graph=hr&parallelism=&size=100'
```

Response: The PGQL query result in JSON format.

```
{
  "name": "bank_graph_analytics_2",
  "resultSetId": "pgql_14",
  "graph": {
    "idType": "number",
    "vertices": [
      {
        "_id": "1",
        "p": [],
        "l": [
          "Accounts"
        ],
        "g": [
          "anonymous_1"
        ]
      },
      {
        "_id": "418",
        "p": [],
        "l": [
          "Accounts"
        ],
        "g": [
          "anonymous_2"
        ]
      },
      {
        "_id": "259",
        "p": [],
        "l": [
          "Accounts"
        ],
        "g": [
          "anonymous_2"
        ]
      }
    ]
  }
}
```

```
    }
  ],
  "edges": [
    {
      "_id": "0",
      "p": [
        {
          "n": "AMOUNT",
          "v": "1000.0",
          "s": false
        }
      ],
      "l": [
        "Transfers"
      ],
      "g": [
        "e"
      ],
      "s": "1",
      "d": "259",
      "u": false
    },
    {
      "_id": "1",
      "p": [
        {
          "n": "AMOUNT",
          "v": "1000.0",
          "s": false
        }
      ],
      "l": [
        "Transfers"
      ],
      "g": [
        "e"
      ],
      "s": "1",
      "d": "418",
      "u": false
    }
  ],
  "paths": [],
  "totalNumResults": 2
},
"table":
"e\nPgxE[provider=Transfers, ID=0]\nPgxE[provider=Transfers, ID=1]"
}
```

7.5 Kerberos Enabled Authentication for the Graph Visualization Application

The Graph Visualization application can authenticate users with Kerberos authentication enabled.

Graph Visualization provides two different drivers to log in:

- **Graph Server (PGX) Driver:** To send your credentials (Kerberos ticket) to Graph Server.
- **Database Driver:** To send your credentials (Kerberos ticket) directly to the database.
- [Prerequisite Requirements for Kerberos Authentication](#)
- [Preparing the Graph Visualization Application for Kerberos Authentication](#)

7.5.1 Prerequisite Requirements for Kerberos Authentication

The system requirements for the respective PGQL drivers are as follows:

- **Graph Server (PGX) Driver:** See [Prerequisite Requirements](#) for enabling Kerberos authentication on the graph server (PGX).
- **Database Driver:**
 - The database must have Kerberos authentication enabled. See [Configuring Kerberos Authentication](#) for more information.
 - Both the database and the Kerberos Authentication Server need to be reachable from the host where the Graph Visualization application is running.
 - The database must be prepared for graph server authentication. That is, relevant graph roles have been granted to users who will log into the Graph Visualization application.

7.5.2 Preparing the Graph Visualization Application for Kerberos Authentication

In order to use Kerberos authentication, you must enter your Active Directory credentials in the Graph Visualization application login page.

To enable Kerberos authentication for the Graph Visualization application, follow the steps shown:

1. Locate the `web.xml` file for your installation.

You can locate the `WEB-INF/web.xml` inside the Graph Visualization WAR file for your installation as shown in the following table:

Table 7-4 Location of WEB-INF/web.xml file

Type of Installation	WAR file	Location
Standalone installation (RPM)	graphviz-<version>-pgviz<graphviz-version>.war	/opt/oracle/graph/graphviz
Apache Tomcat Deployment:	graphviz-<version>-pgviz<graphviz-version>-tomcat.war <version> denotes the downloaded Oracle Graph Server and Client version.	<p>a. Download oracle-graph-webapps-<version>.zip from Oracle Software Delivery Cloud</p> <p>b. Unzip the file into a directory of your choice.</p> <p>c. Locate the .war file for deploying the Graph Visualization application to Tomcat. It follows the naming pattern: graphviz-<version>-pgviz<graphviz-version>-tomcat.war</p>
Oracle WebLogic Server Deployment	graphviz-<version>-pgviz<graphviz-version>-wls.war <version> denotes the downloaded Oracle Graph Server and Client version.	<p>a. Download oracle-graph-webapps-<version>.zip from Oracle Software Delivery Cloud</p> <p>b. Unzip the file into a directory of your choice.</p> <p>c. Locate the .war file for deploying the Graph Visualization application to Oracle WebLogic Server. It follows the naming pattern: graphviz-<version>-pgviz<graphviz-version>-wls.war</p>

2. Extract the appropriate WAR file to a directory of your choice by executing the following command:

```
unzip graphviz-*.war -d <war-file-extraction-path>
```

3. Locate and open the WEB-INF/web.xml file for update using any file editor of your choice. For example:

```
cd <war-file-extraction-path>
vi WEB-INF/web.xml
```

4. Enable the graphviz.driver.auth.kerberos parameter as shown:

```
<context-param>
  <param-name>graphviz.driver.auth.kerberos</param-name>
  <param-value>>true</param-value>
</context-param>
```

Setting this flag **true** initiates the Graph Visualization application to install its own `okinit` package.

5. Optionally, set the cache directory that will be used by the Graph Visualization application to temporarily store Kerberos tickets given by clients as shown

```
<context-param>
  <param-name>graphviz.driver.auth.kerberos.cache_dir</param-name>
  <param-value>/dev/shm/graph_cache</param-value>
</context-param>
```

The default value is `/dev/shm/graph_cache`. If the directory does not exist, it will be automatically created upon server startup.

6. Optionally, set the maximum amount of concurrent Kerberos active sessions in the Graph Visualization application.

```
<context-param>
  <param-name>graphviz.driver.auth.kerberos.max_cache_size</param-
name>
  <param-value>64</param-value>
</context-param>
```

7. Optionally, modify the directory where `okinit` package will be installed, by updating the following parameter:

```
<context-param>
  <param-name>graphviz.driver.auth.kerberos.okinit-directory</param-
name>
  <param-value>/tmp</param-value>
</context-param>
```

 **Note:**

The default value is `/tmp` and you must have executable permission for the directory.

8. Optionally, set the following parameter if there is a location for an existing `okinit` package on your machine. In this case, the GraphVisualization application will not install its own `okinit` package.

```
<context-param>
  <param-
name>graphviz.driver.auth.kerberos.graphviz.driver.auth.okinit-location</
param-name>
  <param-value></param-value>
</context-param>
```

 **Note:**

The GraphVisualization application must have executable permission for the directory location.

9. Finally, after all the preceding updates, repackage the WAR file by executing the following commands:

```
cd <war-file-extraction-path>  
jar -cvf <war-file-name> *
```

10. Redeploy the WAR file to the appropriate directory for your installation.

Kerberos authentication is enabled for the Graph Visualization Application.

8

Using the Machine Learning Library (PgXML) for Graphs

The in-memory graph server (PGX) provides a machine learning library `oracle.pgx.api.mllib`, which supports graph-empowered machine learning algorithms.

The following machine learning algorithms are currently supported:

- [Using the DeepWalk Algorithm](#)
DeepWalk is a widely employed vertex representation learning algorithm used in industry.
- [Using the Supervised GraphWise Algorithm](#)
Supervised GraphWise is an inductive vertex representation learning algorithm which is able to leverage vertex feature information. It can be applied to a wide variety of tasks, including vertex classification and link prediction.
- [Using the Unsupervised GraphWise Algorithm](#)
Unsupervised GraphWise is an unsupervised inductive vertex representation learning algorithm which is able to leverage vertex information. The learned embeddings can be used in various downstream tasks including vertex classification, vertex clustering and similar vertex search.
- [Using the Pg2vec Algorithm](#)
Pg2vec learns representations of graphlets (partitions inside a graph) by employing edges as the principal learning units and thereby packing more information in each learning unit (as compared to employing vertices as learning units) for the representation learning task.

8.1 Using the DeepWalk Algorithm

DeepWalk is a widely employed vertex representation learning algorithm used in industry.

It consists of two main steps:

1. First, the random walk generation step computes random walks for each vertex (with a pre-defined walk length and a pre-defined number of walks per vertex).
2. Second, these generated walks are fed to a **Word2vec** algorithm to generate the vector representation for each vertex (which is the word in the input provided to the Word2vec algorithm). See [KDD paper](#) for more details on DeepWalk algorithm.

DeepWalk creates vertex embeddings for a specific graph and cannot be updated to incorporate modifications on the graph. Instead, a new DeepWalk model should be trained on this modified graph. Lastly, it is important to note that the memory consumption of the DeepWalk model is $O(2n*d)$ where n is the number of vertices in the graph and d is the embedding length.

The following describes the usage of the main functionalities of DeepWalk in in-memory PGX using [DBpedia](#) graph as an example with 8,637,721 vertices and 165,049,964 edges:

- [Loading a Graph](#)

- [Building a Minimal DeepWalk Model](#)
- [Building a Customized DeepWalk Model](#)
- [Training a DeepWalk Model](#)
- [Getting the Loss Value For a DeepWalk Model](#)
- [Computing Similar Vertices for a Given Vertex](#)
- [Computing Similar Vertices for a Vertex Batch](#)
- [Storing a Trained DeepWalk Model](#)
- [Loading a Pre-Trained DeepWalk Model](#)
- [Destroying a DeepWalk Model](#)

8.1.1 Loading a Graph

The following describes the steps for loading a graph:

1. Create a **Session** and an **Analyst**.
Creating a Session and an Analyst Using JShell

```
cd /opt/oracle/graph/  
./bin/opg4j  
// starting the shell will create an implicit session and analyst
```

Creating a Session and an Analyst Using Java

```
import oracle.pgx.api.*;  
import oracle.pgx.api.mllib.DeepWalkModel;  
import oracle.pgx.api.frames.*;  
...  
PgxSession session = Pgx.createSession("my-session");  
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Python

```
session = pypgx.get_session(session_name="my-session")  
analyst = session.create_analyst()
```

2. Load the **graph**.

 **Note:**

Though the DeepWalk algorithm implementation can be applied to directed or undirected graphs, currently only undirected random walks are considered.

Loading a graph using JShell

```
opg4j> var graph = session.readGraphWithProperties("<path>/<graph.json>");
```

Loading a graph using Java

```
PgxGraph graph = session.readGraphWithProperties("<path>/<graph.json>");
```

Loading a graph using Python

```
graph = session.read_graph_with_properties("<path>/<graph.json>")
```

8.1.2 Building a Minimal DeepWalk Model

You can build a DeepWalk model using the minimal configuration and default hyper-parameters as described in the following code:

Building a Minimal DeepWalk Model Using JShell

```
opg4j> var model = analyst.deepWalkModelBuilder().  
        setWindowSize(3).  
        setWalksPerVertex(6).  
        setWalkLength(4).  
        build();
```

Building a Minimal DeepWalk Model Using Java

```
DeepWalkModel model = analyst.deepWalkModelBuilder()  
    .setWindowSize(3)  
    .setWalksPerVertex(6)  
    .setWalkLength(4)  
    .build();
```

Building a Minimal DeepWalk Model Using Python

```
model =  
analyst.deepwalk_builder(window_size=3,walks_per_vertex=6,walk_length=4)
```

8.1.3 Building a Customized DeepWalk Model

You can build a DeepWalk model using customized hyper-parameters as described in the following code:

Building a Customized DeepWalk model Using JShell

```
opg4j> var model = analyst.deepWalkModelBuilder().  
        setMinWordFrequency(1).  
        setBatchSize(512).
```

```
setNumEpochs(1).  
setLayerSize(100).  
setLearningRate(0.05).  
setMinLearningRate(0.0001).  
setWindowSize(3).  
setWalksPerVertex(6).  
setWalkLength(4).  
setSampleRate(0.00001).  
setNegativeSample(2).  
setValidationFraction(0.01).  
build();
```

Building a Customized DeepWalk model Using Java

```
DeepWalkModel model= analyst.deepWalkModelBuilder()  
    .setMinWordFrequency(1)  
    .setBatchSize(512)  
    .setNumEpochs(1)  
    .setLayerSize(100)  
    .setLearningRate(0.05)  
    .setMinLearningRate(0.0001)  
    .setWindowSize(3)  
    .setWalksPerVertex(6)  
    .setWalkLength(4)  
    .setSampleRate(0.00001)  
    .setNegativeSample(2)  
    .setValidationFraction(0.01)  
    .build();
```

Building a Customized DeepWalk model Using Python

```
model = analyst.deepwalk_builder(min_word_frequency=1,  
                                batch_size=512,num_epochs=1,  
                                layer_size=100,  
                                learning_rate=0.05,  
                                min_learning_rate=0.0001,  
                                window_size=3,  
                                walks_per_vertex=6,  
                                walk_length=4,  
                                sample_rate=0.00001,  
                                negative_sample=2,  
                                validation_fraction=0.01)
```

See [DeepWalkModelBuilder](#) in Javadoc for more explanation for each builder operation along with the default values.

8.1.4 Training a DeepWalk Model

You can train a DeepWalk model with the specified default or customized settings as described in the following code:

Training a DeepWalk model Using JShell

```
opg4j> model.fit(graph);
```

Training a DeepWalk model Using Java

```
model.fit(graph);
```

Training a DeepWalk model Using Python

```
model.fit(graph)
```

8.1.5 Getting the Loss Value For a DeepWalk Model

You can fetch the loss value on a specified fraction of training data, that is set in builder using `setValidationFraction` as described in the following code:

Getting the Loss Value Using JShell

```
opg4j> var loss = model.getLoss();
```

Getting the Loss Value Using Java

```
double loss = model.getLoss();
```

Getting the Loss Value Using Python

```
loss = model.loss
```

8.1.6 Computing Similar Vertices for a Given Vertex

You can fetch the k most similar vertices for a given vertex as described in the following code:

Computing Similar Vertices for Given Vertex Using JShell

```
opg4j> var similars = model.computeSimilars("Albert_Einstein", 10);  
opg4j> similars.print();
```

Computing Similar Vertices for Given Vertex Using Java

```
PgxFrame similars = model.computeSimilars("Albert_Einstein", 10);  
similars.print();
```


Computing Similar Vertices for Given Vertex Using Python

```
similar = model.compute_similar("Albert_Einstein",10)
similar.print()
```

Searching for similar vertices for [Albert_Einstein](#) using the trained model, will result in the following output:

```
+-----+
| dstVertex | similarity |
+-----+
| Albert_Einstein | 1.0000001192092896 |
| Physics | 0.8664291501045227 |
| Werner_Heisenberg | 0.8625140190124512 |
| Richard_Feynman | 0.8496938943862915 |
| List_of_physicists | 0.8415523767471313 |
| Physicist | 0.8384397625923157 |
| Max_Planck | 0.8370327353477478 |
| Niels_Bohr | 0.8340970873832703 |
| Quantum_mechanics | 0.8331197500228882 |
| Special_relativity | 0.8280861973762512 |
+-----+
```

8.1.7 Computing Similar Vertices for a Vertex Batch

You can fetch the *k* most similar vertices for a list of input vertices as described in the following code:

Computing Similar Vertices for a Vertex Batch Using JShell

```
opg4j> var vertices = new ArrayList();
opg4j> vertices.add("Machine_learning");
opg4j> vertices.add("Albert_Einstein");
opg4j> batchedSimilar = model.computeSimilar(vertices, 10);
opg4j> batchedSimilar.print();
```

Computing Similar Vertices for a Vertex Batch Using Java

```
List vertices = Arrays.asList("Machine_learning","Albert_Einstein");
PgxFramework batchedSimilar = model.computeSimilar(vertices,10);
batchedSimilar.print();
```

Computing Similar Vertices for a Vertex Batch Using Python

```
vertices = ["Machine_learning","Albert_Einstein"]
batched_similar = model.compute_similar(vertices,10)
batched_similar.print()
```

The following describes the output result:

srcVertex	dstVertex	similarity
Machine_learning	Machine_learning	1.0000001192092896
Machine_learning	Data_mining	0.9070799350738525
Machine_learning	Computer_science	0.8963605165481567
Machine_learning	Unsupervised_learning	0.8828719854354858
Machine_learning	R_(programming_language)	0.8821185827255249
Machine_learning	Algorithm	0.8819515705108643
Machine_learning	Artificial_neural_network	0.8773092031478882
Machine_learning	Data_analysis	0.8758628368377686
Machine_learning	List_of_algorithms	0.8737979531288147
Machine_learning	K-means_clustering	0.8715602159500122
Albert_Einstein	Albert_Einstein	1.0000001192092896
Albert_Einstein	Physics	0.8664291501045227
Albert_Einstein	Werner_Heisenberg	0.8625140190124512
Albert_Einstein	Richard_Feynman	0.8496938943862915
Albert_Einstein	List_of_physicists	0.8415523767471313
Albert_Einstein	Physicist	0.8384397625923157
Albert_Einstein	Max_Planck	0.8370327353477478
Albert_Einstein	Niels_Bohr	0.8340970873832703
Albert_Einstein	Quantum_mechanics	0.8331197500228882
Albert_Einstein	Special_relativity	0.8280861973762512

8.1.8 Storing a Trained DeepWalk Model

You can store models in database. The models get stored as a row inside a model store table.

The following code shows how to store a trained DeepWalk model in database in a specific model store table:

Storing a Trained DeepWalk Model Using JShell

```
opg4j> model.export().db().
           modelstore("modelstoretablename"). // name of the model store
table
           modelname("model").                // model name (primary key
of model store table)
           description("a model description"). // description to store
alongside the model
           store();
```

Storing a Trained DeepWalk Model Using Java

```
model.export().db()
    .modelstore("modelstoretablename") // name of the model store table
    .modelname("model")                // model name (primary key of model
store table)
    .description("a model description") // description to store alongside
the model
    .store();
```

Storing a Trained DeepWalk Model Using Python

```
model.export().db(model_store="modelstoretablename",
                  model_name="model", description="a model description")
```



Note:

All the preceding examples assume that you are storing the model in the current logged in database. If you must store the model in a different database then refer to the examples in [Storing a Trained Model in Another Database](#).

- [Storing a Trained Model in Another Database](#)

8.1.8.1 Storing a Trained Model in Another Database

You can store models in a different database other than the one used for login.

The following code shows how to store a trained model in a different database:

Storing a Trained Model Using JShell

```
opg4j> model.export().db().
           username("user").           // DB user to use for storing
the model
           password("password").       // password of the DB user
           jdbcUrl("jdbcUrl").        // jdbc url to the DB
           modelstore("modelstoretablename"). // name of the model store
table
           modelname("model").         // model name (primary key of
model store table)
           description("a model description"). // description to store
alongside the model
           store();
```

Storing a Trained Model Using Java

```
model.export().db()
    .username("user")           // DB user to use for storing the model
    .password("password")      // password of the DB user
    .jdbcUrl("jdbcUrl")       // jdbc url to the DB
    .modelstore("modelstoretablename") // name of the model store table
    .modelName("model")       // model name (primary key of model
store table)
    .description("a model description") // description to store alongside the
model
    .store();
```

Storing a Trained Model Using Python

```
model.export().db(username="user",
                  password="password",
                  model_store="modelstoretablename",
                  model_name="model",
```

```
description="a model description",  
jdbc_url="jdbc_url")
```

8.1.9 Loading a Pre-Trained DeepWalk Model

You can load models from a database.

You can load a pre-trained DeepWalk model from a model store table in database as described in the following code:

Loading a Pre-Trained DeepWalk Model Using JShell

```
opg4j> var model = analyst.loadDeepWalkModel().db().  
        modelstore("modeltablename"). // name of the model store  
table  
        modelname("model").          // model name (primary key of  
model store table)  
        load();
```

Loading a Pre-Trained DeepWalk Model Using Java

```
DeepWalkModel model = analyst.loadDeepWalkModel().db()  
    .modelstore("modeltablename") // name of the model store table  
    .modelname("model")          // model name (primary key of model store  
table)  
    .load();
```

Loading a Pre-Trained DeepWalk Model Using Python

```
analyst.get_deepwalk_model_loader().db(model_store="modelstoretablename",  
                                       model_name="model")
```



Note:

All the preceding examples assume that you are loading the model from the current logged in database. If you must load the model from a different database then refer to the examples in [Loading a Pre-Trained Model From Another Database](#).

- [Loading a Pre-Trained Model From Another Database](#)

8.1.9.1 Loading a Pre-Trained Model From Another Database

You can load models from a different database other than the one used for login.

You can load a pre-trained model from a model store table in database as described in the following code:

Loading a Pre-Trained Model Using JShell

```

opg4j> var model = analyst.<modelLoader>.db().
        username("user").           // DB user to use for storing the
model
        password("password").       // password of the DB user
        jdbcUrl("jdbcUrl").         // jdbc url to the DB
        modelstore("modeltablename"). // name of the model store table
        modelname("model").         // model name (primary key of
model store table)
        load();

```

where `<modelLoader>` applies as follows:

- `loadDeepWalkModel()`: Loads a Deepwalk model
- `loadSupervisedGraphWiseModel()`: Loads a Supervised GraphWise model
- `loadUnsupervisedGraphWiseModel()`: Loads a Unsupervised GraphWise model
- `loadPg2vecModel()`: Loads a Pg2vec model

Loading a Pre-Trained DeepWalk Model Using Java

```

<modeltype> model = analyst.<modelLoader>.db()
    .username("user")           // DB user to use for storing the model
    .password("password")      // password of the DB user
    .jdbcUrl("jdbcUrl")        // jdbc url to the DB
    .modelstore("modeltablename") // name of the model store table
    .modelname("model")        // model name (primary key of model store
table)
    .load();

```

where `<modeltype>` can have the following values based on the model to be loaded:

- `DeepWalkModel`: represents a Deepwalk model
- `SupervisedGraphWiseModel`: represents a Supervised GraphWise model
- `UnsupervisedGraphWiseModel`: represents a Unsupervised GraphWise model
- `Pg2vecModel`: represents a Pg2vec model

where `<modelLoader>` applies as follows:

- `loadDeepWalkModel()`: Loads a Deepwalk model
- `loadSupervisedGraphWiseModel()`: Loads a Supervised GraphWise model
- `loadUnsupervisedGraphWiseModel()`: Loads a Unsupervised GraphWise model
- `loadPg2vecModel()`: Loads a Pg2vec model

Loading a Pre-Trained DeepWalk Model Using Python

```

analyst.<modelLoader>.db(username="user",
                        password="password",
                        model_store="modelstoretablename",
                        model_name="model",
                        jdbc_url="jdbc_url")

```

where `<modelLoader>` applies as follows:

- `get_deepwalk_model_loader()`: Loads a Deepwalk model

- `get_supervised_graphwise_model_loader()`: Loads a Supervised GraphWise model
- `get_unsupervised_graphwise_model_loader()`: Loads a Unsupervised GraphWise model
- `get_pg2vec_model_loader()`: Loads a Pg2vec model

8.1.10 Destroying a DeepWalk Model

You can destroy a DeepWalk model as described in the following code:

Destroying a DeepWalk Model Using JShell

```
opg4j> model.destroy();
```

Destroying a DeepWalk Model Using Java

```
model.destroy();
```

Destroying a DeepWalk Model Using Python

```
model.destroy()
```

8.2 Using the Supervised GraphWise Algorithm

Supervised GraphWise is an inductive vertex representation learning algorithm which is able to leverage vertex feature information. It can be applied to a wide variety of tasks, including vertex classification and link prediction.

Supervised GraphWise is based on [GraphSAGE](#) by Hamilton et al.

Model Structure

A Supervised GraphWise model consists of two graph convolutional layers followed by several prediction layers.

The forward pass through a convolutional layer for a vertex proceeds as follows:

1. A set of neighbors of the vertex is sampled.
2. The previous layer representations of the neighbors are mean-aggregated, and the aggregated features are concatenated with the previous layer representation of the vertex.
3. This concatenated vector is multiplied with weights, and a bias vector is added.
4. The result is normalized to such that the layer output has unit norm.

The prediction layers are standard neural network layers.

The following describes the usage of the main functionalities of the implementation of **GraphSAGE** in PGX using the [Cora](#) graph as an example:

- [Loading a Graph](#)
- [Building a Minimal GraphWise Model](#)

- [Advanced Hyperparameter Customization](#)
- [Training a Supervised GraphWise Model](#)
- [Getting the Loss Value For a Supervised GraphWise Model](#)
- [Inferring the Vertex Labels for a Supervised GraphWise Model](#)
- [Evaluating the Supervised GraphWise Model Performance](#)
- [Inferring Embeddings for a Supervised GraphWise Model](#)
- [Storing a Trained Supervised GraphWise Model](#)
- [Loading a Pre-Trained Supervised GraphWise Model](#)
- [Destroying a Supervised GraphWise Model](#)
- [Explaining a Prediction of a Supervised GraphWise Model](#)

8.2.1 Loading a Graph

The following describes the steps for loading a graph:

1. Create a **Session** and an **Analyst**.
Creating a Session and an Analyst Using JShell

```
cd /opt/oracle/graph/  
./bin/opg4j  
// starting the shell will create an implicit session and analyst  
import oracle.pgx.config.mllib.ActivationFunction;  
import oracle.pgx.config.mllib.WeightInitScheme;  
PgxSession session = Pgx.createSession("my-session");  
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Java

```
import oracle.pgx.api.*;  
import oracle.pgx.api.mllib.SupervisedGraphWiseModel;  
import oracle.pgx.api.frames.*;  
import oracle.pgx.config.mllib.ActivationFunction;  
import oracle.pgx.config.mllib.GraphWiseConvLayerConfig;  
import oracle.pgx.config.mllib.GraphWisePredictionLayerConfig;  
import oracle.pgx.config.mllib.SupervisedGraphWiseModelConfig;  
import oracle.pgx.config.mllib.WeightInitScheme;  
PgxSession session = Pgx.createSession("my-session");  
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Python

```
session = pypgx.get_session(session_name="my-session")  
analyst = session.create_analyst()
```

2. Load the **graph**.

Loading a graph Using JShell

```

opg4j> var fullGraph = session.readGraphWithProperties("<path>/
<full_graph.json>")
opg4j> var trainGraph = session.readGraphWithProperties("<path>/
<train_graph.json>")
opg4j> var testVertices = fullGraph.getVertices()
                                .stream()
                                .filter(v -> !trainGraph.hasVertex(v.getId()))
                                .collect(Collectors.toList());

```

Loading a graph Using Java

```

PgxGraph fullGraph = session.readGraphWithProperties("<path>/
<full_graph.json>");
PgxGraph trainGraph = session.readGraphWithProperties("<path>/
<train_graph.json>");
List<PgxVertex> testVertices = fullGraph.getVertices()
    .stream()
    .filter(v->!trainGraph.hasVertex(v.getId()))
    .collect(Collectors.toList());

```

Loading a graph Using Python

```

full_graph = session.read_graph_with_properties("<path>/
<full_graph.json>")
train_graph = session.read_graph_with_properties("<path>/
<train_graph.json>")
test_vertices = []
train_vertices = train_graph.get_vertices()
for v in full_graph.get_vertices():
    if(not train_vertices.contains(v)):
        test_vertices.append(v)

```

8.2.2 Building a Minimal GraphWise Model

You can build a GraphWise model using the minimal configuration and default hyper-parameters as described in the following code:

**Note:**

Starting from Graph Server and Client Release 21.2, you can create a model with one of the following options:

- only vertex properties
- only edge properties
- both vertex and edge properties

Building a Minimal GraphWise Model with Vertex and Edge Properties Using JShell

```
opg4j> var model = analyst.supervisedGraphWiseModelBuilder().
    setVertexInputPropertyNames("features").
    setVertexTargetPropertyName("label").
    setEdgeInputPropertyNames("cost"). //sets the edge
properties name
    build();
```

Building a Minimal GraphWise with Vertex and Edge Properties Model Using Java

```
SupervisedGraphWiseModel model =
analyst.supervisedGraphWiseModelBuilder()
    .setVertexInputPropertyNames("features")
    .setVertexTargetPropertyName("labels")
    .setEdgeInputPropertyNames("cost") //sets the edge properties name
    .build();
```

Building a Minimal GraphWise with Vertex Properties Model Using Python

```
params = dict(vertex_target_property_name="label",
              vertex_input_property_names=["features"])

model = analyst.supervised_graphwise_builder(**params)
```



Note:

Even though only one vertex and one edge property is specified in the preceding example, you can specify a list of vertex or edge properties.

8.2.3 Advanced Hyperparameter Customization

You can build a GraphWise model using rich hyperparameter customization.

This is done through the following two sub-config classes:

1. GraphWiseConvLayerConfig
2. GraphWisePredictionLayerConfig

The following code describes the implementation of the configuration using the above classes in GraphWise model:

 **Note:**

Starting from Graph Server and Client Release 21.2, you can create a model with one of the following options:

- only vertex properties
- only edge properties
- both vertex and edge properties

Building a Customized GraphWise Model with Vertex and Edge Properties Using JShell

```
opg4j> var weightProperty = analyst.pagerank(trainGraph).getName();
opg4j> var convLayerConfig = analyst.graphWiseConvLayerConfigBuilder().
    setNumSampledNeighbors(25).
    setActivationFunction(ActivationFunction.TANH).
    setWeightInitScheme(WeightInitScheme.XAVIER).
    setWeightedAggregationProperty(weightProperty).
    build();
opg4j> var predictionLayerConfig =
analyst.graphWisePredictionLayerConfigBuilder().
    setHiddenDimension(32).
    setActivationFunction(ActivationFunction.RELU).
    setWeightInitScheme(WeightInitScheme.HE).
    build();
opg4j> var model = analyst.supervisedGraphWiseModelBuilder().
    setVertexInputPropertyNames("features").
    setVertexTargetPropertyName("labels").
    setEdgeInputPropertyNames("cost"). //sets the edge
properties name
    setConvLayerConfigs(convLayerConfig).
    setPredictionLayerConfigs(predictionLayerConfig).
    build();
```

Building a Customized GraphWise Model with Vertex and Edge Properties Using Java

```
String weightProperty = analyst.pagerank(trainGraph).getName()
GraphWiseConvLayerConfig convLayerConfig =
analyst.graphWiseConvLayerConfigBuilder()
    .setNumSampledNeighbors(25)
    .setActivationFunction(ActivationFunction.TANH)
    .setWeightInitScheme(WeightInitScheme.XAVIER)
    .setWeightedAggregationProperty(weightProperty)
    .build();

GraphWisePredictionLayerConfig predictionLayerConfig =
analyst.graphWisePredictionLayerConfigBuilder()
    .setHiddenDimension(32)
    .setActivationFunction(ActivationFunction.RELU)
    .setWeightInitScheme(WeightInitScheme.HE)
```

```
.build();

SupervisedGraphWiseModel model =
  analyst.supervisedGraphWiseModelBuilder()
    .setVertexInputPropertyNames("features")
    .setVertexTargetPropertyName("labels")
    .setEdgeInputPropertyNames("cost") //sets the edge properties name
    .setConvLayerConfigs(convLayerConfig)
    .setPredictionLayerConfigs(predictionLayerConfig)
    .build();
```

Building a Customized GraphWise Model with Vertex Properties Using Python

```
weightProperty = analyst.pagerank(train_graph).name

conv_layer_config = dict(num_sampled_neighbors=25,
                        activation_fn='TANH',
                        weight_init_scheme='XAVIER',
                        neighbor_weight_property_name=weightProperty)

conv_layer = analyst.graphwise_conv_layer_config(**conv_layer_config)

pred_layer_config = dict(hidden_dim=32,
                        activation_fn='RELU',
                        weight_init_scheme='HE')

pred_layer = analyst.graphwise_pred_layer_config(**pred_layer_config)

params = dict(vertex_target_property_name="labels",
              conv_layer_config=[conv_layer],
              pred_layer_config=[pred_layer],
              vertex_input_property_names=["features"],
              seed=17)

model = analyst.supervised_graphwise_builder(**params)
```

See [SupervisedGraphWiseModelBuilder](#), [GraphWiseConvLayerConfigBuilder](#) and [GraphWisePredictionLayerConfigBuilder](#) in Javadoc for a full description of all available hyperparameters and their default values.

8.2.4 Training a Supervised GraphWise Model

You can train a Supervised GraphWise model on a graph as described in the following code:

Training a GraphWise Model Using JShell

```
opg4j> model.fit(trainGraph);
```

Training a GraphWise Model Using Java

```
model.fit(trainGraph);
```

Training a GraphWise Model Using Python

```
model.fit(train_graph)
```

8.2.5 Getting the Loss Value For a Supervised GraphWise Model

You can fetch the training loss value as described in the following code:

Getting the Loss Value Using JShell

```
opg4j> var loss = model.getTrainingLoss();
```

Getting the Loss Value Using Java

```
double loss = model.getTrainingLoss();
```

Getting the Loss Value Using Python

```
double loss = model.get_training_loss()
```

8.2.6 Inferring the Vertex Labels for a Supervised GraphWise Model

You can infer the labels for vertices on any graph (including vertices or graphs that were not seen during training) as described in the following code:

Inferring the Vertex Labels Using JShell

```
opg4j> var labels = model.inferLabels(fullGraph, testVertices);  
opg4j> labels.head().print()
```

Inferring the Vertex Labels Using Java

```
PgxFrame labels = model.inferLabels(fullGraph, testVertices);  
labels.head().print();
```

Inferring the Vertex Labels Using Python

```
labels = model.infer_labels(full_graph, full_graph.get_vertices())  
labels.print()
```

The output will be similar to the following example output:

```
+-----+  
| vertexId | label |  
+-----+
```

2	Neural Networks
6	Theory
7	Case Based
22	Rule Learning
30	Theory
34	Neural Networks
47	Case Based
48	Probabalistic Methods
50	Theory
52	Theory

Similarly, you can also get the model confidence for each class by inferring the prediction logits as described in the following code:

Getting the Model Confidence Using JShell

```
opg4j> var logits = model.inferLogits(fullGraph, testVertices);
opg4j> logits.head().print();
```

Getting the Model Confidence Using Java

```
PgxFrame logits = model.inferLogits(fullGraph, testVertices);
logits.head().print();
```

Getting the Model Confidence Using Python

```
logits = model.infer_logits(full_graph, test_vertices)
logits.print()
```

8.2.7 Evaluating the Supervised GraphWise Model Performance

You can evaluate various classification metrics for the model using the `evaluateLabels` method as described in the following code:

Evaluating the Supervised GraphWise Model Performance Using JShell

```
opg4j> model.evaluateLabels(fullGraph, testVertices).print();
```

Evaluating the Supervised GraphWise Model Performance Using Java

```
model.evaluateLabels(fullGraph, testVertices).print();
```

Evaluating the Supervised GraphWise Model Performance Using Python

```
model.evaluate_labels(full_graph, test_vertices).print()
```

The output will be similar to the following example output:

```
+-----+
| Accuracy | Precision | Recall | F1-Score |
+-----+
| 0.8488   | 0.8523   | 0.831  | 0.8367   |
+-----+
```

8.2.8 Inferring Embeddings for a Supervised GraphWise Model

You can use a trained model to infer embeddings for unseen nodes and store in the database as described in the following code:

Inferring Embeddings Using JShell

```
opg4j> var vertexVectors = model.inferEmbeddings(fullGraph,
fullGraph.getVertices()).flattenAll();
opg4j> vertexVectors.write().
    db().
    name("vertex vectors").
    tablename("vertexVectors"). // indicate the name of the table in which
the data should be stored
    overwrite(true).           // indicate that if there is a table with
the same name, it will be overwritten (truncated)
    store();
```

Inferring Embeddings Using Java

```
PgxFrame vertexVectors =
model.inferEmbeddings(fullGraph,fullGraph.getVertices()).flattenAll();
vertexVectors.write()
    .db()
    .name("vertex vectors")
    .tablename("vertexVectors") // indicate the name of the table in which
the data should be stored
    .overwrite(true)           // indicate that if there is a table with
the same name, it will be overwritten (truncated)
    .store();
```

Inferring Embeddings Using Python

```
vertex_vectors = model.infer_embeddings(full_graph,
full_graph.get_vertices()).flatten_all()
vertexVectors.write().db(name="vertex vectors", "tablename", overwrite=True)
```

The schema for the `vertexVectors` will be as follows without flattening (`flattenAll` splits the vector column into separate double-valued columns):

```
+-----+
```

vertexId	embedding

**Note:**

All the preceding examples assume that you are inferring the embeddings for a model in the current logged in database. If you must infer embeddings for the model in a different database then refer to the examples in [Inferring Embeddings for a Model in Another Database](#).

- [Inferring Embeddings for a Model in Another Database](#)

8.2.8.1 Inferring Embeddings for a Model in Another Database

You can infer embeddings on a trained model and store in a different database other than the one used for login.

The following code shows how to infer embeddings and store in a different database:

Inferring Embeddings Using JShell

```
opg-jshell> var vertexVectors = model.inferEmbeddings(fullGraph,
fullGraph.getVertices()).flattenAll()
opg-jshell> vertexVectors.write()
    .db()
    .username("user")                // DB user to use for storing the model
    .password("password")            // password of the DB user
    .jdbcUrl("jdbcUrl")             // jdbc url to the DB
    .name("vertex vectors")
    .tablename("vertexVectors") // indicate the name of the table in which the
data should be stored
    .overwrite(true)                // indicate that if there is a table with the
same name, it will be overwritten (truncated)
    .store()
```

Inferring Embeddings Using Java

```
PgxFrame vertexVectors =
model.inferEmbeddings(fullGraph,fullGraph.getVertices()).flattenAll();
vertexVectors.write()
    .db()
    .username("user")                // DB user to use for storing the model
    .password("password")            // password of the DB user
    .jdbcUrl("jdbcUrl")             // jdbc url to the DB
    .name("vertex vectors")
    .tablename("vertexVectors") // indicate the name of the table in which the
data should be stored
    .overwrite(true)                // indicate that if there is a table with the
same name, it will be overwritten (truncated)
    .store();
```

Inferring Embeddings Using Python

```
vertexVectors =
model.infer_embeddings(fullGraph,fullGraph.getVertices()).flattenAll()
vertexVectors.write().db(username="user", password="password",
```

```
jdbc_url="jdbcUrl",
name="vertex vectors", "tablename", overwrite=True)
```

8.2.9 Storing a Trained Supervised GraphWise Model

You can store models in database. The models get stored as a row inside a model store table.

The following code shows how to store a trained Supervised GraphWise model in database in a specific model store table:

Storing a Trained Supervised GraphWise Model Using JShell

```
opg4j> model.export().db().
        modelstore("modelstoretablename"). // name of the model store
table
        modelname("model"). // model name (primary key
of model store table)
        description("a model description"). // description to store
alongside the model
        store();
```

Storing a Trained Supervised GraphWise Model Using Java

```
model.export().db()
    .modelstore("modelstoretablename") // name of the model store table
    .modelname("model") // model name (primary key of model
store table)
    .description("a model description") // description to store alongside
the model
    .store();
```

Storing a Trained Supervised GraphWise Model Using Python

```
model.export().db(model_store="modeltablename",
model_name="model", description="a model description")
```



Note:

All the preceding examples assume that you are storing the model in the current logged in database. If you must store the model in a different database then refer to the examples in [Storing a Trained Model in Another Database](#).

8.2.10 Loading a Pre-Trained Supervised GraphWise Model

You can load models from a database.

You can load a pre-trained Supervised GraphWise model from a model store table in database as described in the following code:

Loading a Pre-Trained Supervised GraphWise Model Using JShell

```
opg4j> var model = analyst.loadSupervisedGraphWiseModel().db().
        modelstore("modeltablename"). // name of the model
store table
        modelname("model").          // model name (primary
key of model store table)
        load();
```

Loading a Pre-Trained Supervised GraphWise Model Using Java

```
SupervisedGraphWiseModel model =
analyst.loadSupervisedGraphWiseModel().db()
        .modelstore("modeltablename") // name of the model store table
        .modelname("model")          // model name (primary key of model
store table)
        .load();
```

Loading a Pre-Trained Supervised GraphWise Model Using Python

```
analyst.get_supervised_graphwise_model_loader().db(model_store="modelsto
retablename",

model_name="model")
```



Note:

All the preceding examples assume that you are loading the model from the current logged in database. If you must load the model from a different database then refer to the examples in [Loading a Pre-Trained Model From Another Database](#).

8.2.11 Destroying a Supervised GraphWise Model

You can destroy a GraphWise model as described in the following code:

Destroying a GraphWise Model Using JShell

```
opg4j> model.destroy();
```

Destroying a GraphWise Model Using Java

```
model.destroy();
```

Destroying a GraphWise Model Using Python

```
model.destroy();
```

8.2.12 Explaining a Prediction of a Supervised GraphWise Model

In order to understand which features and vertices are important for a prediction of the Supervised GraphWise model, you can generate a `SupervisedGnnExplanation` using a technique similar to the [GNNEexplainer](#) by Ying et al.

The explanation holds information related to:

- **graph structure:** an importance score for each vertex
- **features:** an importance score for each graph property

Note:

The vertex being explained is always assigned importance 1. Further, the feature importances are scaled such that the most important feature has importance 1.

Additionally, a `SupervisedGnnExplanation` contains the inferred embedding, logits, and label. The `inferAndGetExplanation` method can be used on all fitted `SupervisedGraphWiseModel` models that do not rely on edge features. In order to achieve best results, the features should be centered around 0.

For example, assume a simple graph that contains a feature that correlates with the label and another feature that does not. It is therefore expected that the importance of the features to differ significantly (with the feature correlating with the label being more important), while structural importance does not play a big role. In this case, you can generate an explanation as shown:

Explaining a Prediction Using JShell

```
opg4j> var simpleGraph = session.createGraphBuilder().
    addVertex(0).setProperty("label_feature",
0.5).setProperty("const_feature", 0.5).
    setProperty("label", true).
    addVertex(1).setProperty("label_feature",
-0.5).setProperty("const_feature", 0.5).
    setProperty("label", false).
    addEdge(0, 1).build();

// build and train model (var model) as explained earlier in Building a Minimal
GraphWise Model
// and Training a Supervised GraphWise Model.

// explain prediction of vertex 0
opg4j> var explanation = model.inferAndGetExplanation(simpleGraph,
simpleGraph.getVertex(0));
opg4j> var constProperty = simpleGraph.getVertexProperty("const_feature");
opg4j> var labelProperty = simpleGraph.getVertexProperty("label_feature");

// retrieve feature importances
opg4j> var featureImportances = explanation.getVertexFeatureImportance();
opg4j> var importanceConstProp = featureImportances.get(constProperty); //
small as unimportant
```

```
opg4j> var importanceLabelProp =
featureImportances.get(labelProperty); // large (1) as important

// retrieve computation graph with importances
opg4j> var importanceGraph = explanation.getImportanceGraph();

// retrieve importance of vertices
opg4j> var importanceProperty =
explanation.getVertexImportanceProperty();
opg4j> var importanceVertex0 = importanceProperty.get(0); // has
importance 1
opg4j> var importanceVertex1 = importanceProperty.get(1); // available
if vertex 1 part of computation
```

Explaining a Prediction Using Java

```
PgxGraph simpleGraph = session.createGraphBuilder()
    .addVertex(0).setProperty("label_feature",
0.5).setProperty("const_feature", 0.5)
    .setProperty("label", true)
    .addVertex(1).setProperty("label_feature",
-0.5).setProperty("const_feature", 0.5)
    .setProperty("label", false)
    .addEdge(0, 1).build();

// build and train model (SupervisedGraphWiseModel model) as explained
earlier in Building a Minimal GraphWise Model
// and Training a Supervised GraphWise Model

// explain prediction of vertex 0
SupervisedGnnExplanation<Integer> explanation =
model.inferAndGetExplanation(simpleGraph,
    simpleGraph.getVertex(0));

VertexProperty<Integer, Float> constProperty =
simpleGraph.getVertexProperty("const_feature");
VertexProperty<Integer, Float> labelProperty =
simpleGraph.getVertexProperty("label_feature");

// retrieve feature importances
Map<VertexProperty<Integer, ?>, Float> featureImportances =
explanation.getVertexFeatureImportance();
float importanceConstProp = featureImportances.get(constProperty); //
small as unimportant
float importanceLabelProp = featureImportances.get(labelProperty); //
large (1) as important

// retrieve computation graph with importances
PgxGraph importanceGraph = explanation.getImportanceGraph();

// retrieve importance of vertices
VertexProperty<Integer, Float> importanceProperty =
explanation.getVertexImportanceProperty();
```

```
float importanceVertex0 = importanceProperty.get(0); // has importance 1
float importanceVertex1 = importanceProperty.get(1); // available if vertex
1 part of computation
```

Explaining a Prediction Using Python

```
simple_graph = session.create_graph_builder()
    .add_vertex(0).set_property("label_feature",
0.5).set_property("const_feature", 0.5)
    .set_property("label", true)
    .add_vertex(1).set_property("label_feature",
-0.5).set_property("const_feature", 0.5)
    .set_property("label", false)
    .add_edge(0, 1).build();

// build and train model (model) as explained earlier in Building a Minimal
GraphWise Model
// and Training a Supervised GraphWise Model

// explain prediction of vertex 0
explanation = model.infer_and_get_explanation(simple_graph,
simple_graph.get_vertex(0));

const_property = simple_graph.get_vertex_property("const_feature");
label_property = simple_graph.get_vertex_property("label_feature");

// retrieve feature importances
feature_importances = explanation.get_vertex_feature_importance();
importance_const_prop = feature_importances[const_property]; // small as
unimportant
importance_label_prop = feature_importances[label_property]; // large (1) as
important

// retrieve computation graph with importances
importance_graph = explanation.get_importance_graph();

// retrieve importance of vertices
importance_property = explanation.get_vertex_importance_property();
importance_vertex_0 = importance_property[0]; // has importance 1
importance_vertex_1 = importance_property[1]; // available if vertex 1 part
of computation
```

8.3 Using the Unsupervised GraphWise Algorithm

Unsupervised GraphWise is an unsupervised inductive vertex representation learning algorithm which is able to leverage vertex information. The learned embeddings can be used in various downstream tasks including vertex classification, vertex clustering and similar vertex search.

Unsupervised GraphWise is based on [Deep Graph Infomax \(DGI\)](#) by Velickovic et al.

Model Structure

A Unsupervised GraphWise model consists of two graph convolutional layers followed by a DGI Layer.

The forward pass through a convolutional layer for a vertex proceeds as follows:

1. A set of neighbors of the vertex is sampled.
2. The previous layer representations of the neighbors are mean-aggregated, and the aggregated features are concatenated with the previous layer representation of the vertex.
3. This concatenated vector is multiplied with weights, and a bias vector is added.
4. The result is normalized to such that the layer output has unit norm.

The DGI Layer consists of three parts enabling unsupervised learning using embeddings produced by the convolution layers.

1. **Corruption function:** Shuffles the node features while preserving the graph structure to produce negative embedding samples using the convolution layers.
2. **Readout function:** Sigmoid activated mean of embeddings, used as summary of a graph.
3. **Discriminator:** Measures the similarity of positive (unshuffled) embeddings with the summary as well as the similarity of negative samples with the summary from which the loss function is computed.

Since none of these contains mutable hyperparameters, the default DGI layer is always used and cannot be adjusted.

The following describes the usage of the main functionalities of the implementation of DGI in PGX using the [Cora](#) graph as an example:

- [Loading a Graph](#)
- [Building a Minimal Unsupervised GraphWise Model](#)
- [Advanced Hyperparameter Customization](#)
- [Training a Unsupervised GraphWise Model](#)
- [Getting the Loss Value for a Unsupervised GraphWise Model](#)
- [Inferring Embeddings for a Unsupervised GraphWise Model](#)
- [Storing a Unsupervised GraphWise Model](#)
- [Loading a Pre-Trained Unsupervised GraphWise Model](#)
- [Destroying a Unsupervised GraphWise Model](#)

8.3.1 Loading a Graph

The following describes the steps for loading a graph:

1. Create a **Session** and an **Analyst**.

Creating a Session and an Analyst Using JShell

```
cd /opt/oracle/graph/
```

```
./bin/opg4j
// starting the shell will create an implicit session and analyst
```

Creating a Session and an Analyst Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.mllib.UnsupervisedGraphWiseModel;
import oracle.pgx.api.frames.*;
import oracle.pgx.config.mllib.ActivationFunction;
import oracle.pgx.config.mllib.GraphWiseConvLayerConfig;
import oracle.pgx.config.mllib.UnsupervisedGraphWiseModelConfig;
import oracle.pgx.config.mllib.WeightInitScheme;

PgxSession session = Pgx.createSession("my-session");
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Python

```
session = pypgx.get_session()
analyst = session.analyst
```

2. Load the graph.

Loading a graph using JShell

```
opg4j> var graph = session.readGraphWithProperties("<path/to/
graph_config.json>");
```

Loading a graph using Java

```
PgxGraph graph = session.readGraphWithProperties("<path/to/
graph_config.json>");
```

Loading a graph using Python

```
graph = session.readGraphWithProperties("<path/to/graph_config.json>")
```

You do not need to use a test graph or test vertices, since the model is trained to be unsupervised.

8.3.2 Building a Minimal Unsupervised GraphWise Model

You can build a Unsupervised GraphWise model with only vertex properties, or only edge properties or both using the minimal configuration and default hyper-parameters.

- Create a Unsupervised GraphWise model as described in the following code:

Building a Minimal Unsupervised GraphWise Model Using JShell

```
opg4j> var model = analyst.unsupervisedGraphWiseModelBuilder().
    setVertexInputPropertyNames("features").
    build()
```

Building a Minimal Unsupervised GraphWise Model Using Java

```
UnsupervisedGraphWiseModel model =
  analyst.unsupervisedGraphWiseModelBuilder()
    .setVertexInputPropertyNames("features")
    .build();
```

Building a Minimal Unsupervised GraphWise Model Using Python

```
model =
  analyst.unsupervised_graphwise_builder(vertex_input_property_names=[
    "features"])
```

8.3.3 Advanced Hyperparameter Customization

You can build a Unsupervised GraphWise model with only vertex properties or only edge properties or both using rich hyperparameter customization. This is implemented using the sub-config class, `GraphWiseConvLayerConfig`.

The following code describes the implementation of the configuration in a Unsupervised GraphWise model:

- Build a Unsupervised GraphWise model as shown in the following code:

Building a Customized Unsupervised GraphWise Model Using JShell

```
opg4j> var weightProperty = analyst.pagerank(trainGraph).getName();
opg4j> var convLayerConfig =
  analyst.graphWiseConvLayerConfigBuilder().
    setNumSampledNeighbors(25).
    setActivationFunction(ActivationFunction.TANH).
    setWeightInitScheme(WeightInitScheme.XAVIER).
    setWeightedAggregationProperty(weightProperty).
    build();

opg4j> var model = analyst.unsupervisedGraphWiseModelBuilder().
  setVertexInputPropertyNames("features").
  setConvLayerConfigs(convLayerConfig).
  build();
```

Building a Customized Unsupervised GraphWise Model Using Java

```
String weightProperty = analyst.pagerank(trainGraph).getName();
GraphWiseConvLayerConfig convLayerConfig =
  analyst.graphWiseConvLayerConfigBuilder()
    .setNumSampledNeighbors(25)
    .setActivationFunction(ActivationFunction.TANH)
    .setWeightInitScheme(WeightInitScheme.XAVIER)
    .setWeightedAggregationProperty(weightProperty)
    .build();

UnsupervisedGraphWiseModel model =
  analyst.unsupervisedGraphWiseModelBuilder()
    .setVertexInputPropertyNames("features")
```

```
.setConvLayerConfigs(convLayerConfig)
.build();
```

Building a Customized Unsupervised GraphWise Model Using Python

```
weightProperty = analyst.pagerank(train_graph).name

conv_layer_config = dict(num_sampled_neighbors=25,
                        activation_fn='TANH',
                        weight_init_scheme='XAVIER',
                        neighbor_weight_property_name=weightProperty)
conv_layer = analyst.graphwise_conv_layer_config(**conv_layer_config)
params = dict(conv_layer_config=[conv_layer],
              vertex_input_property_names=["features"])

model = analyst.unsupervised_graphwise_builder(**params)
```

See [UnsupervisedGraphWiseModelBuilder](#) and [GraphWiseConvLayerConfigBuilder](#) in Javadoc for full description of all available hyperparameters and their default values.

8.3.4 Training a Unsupervised GraphWise Model

You can train a Unsupervised GraphWise model on a graph.

- Train a Unsupervised GraphWise model as shown in the following code:

Training a Unsupervised GraphWise Model Using JShell

```
opg4j> model.fit(trainGraph);
```

Training a Unsupervised GraphWise Model Using Java

```
model.fit(trainGraph);
```

Training a Unsupervised GraphWise Model Using Python

```
model.fit(trainGraph)
```

8.3.5 Getting the Loss Value for a Unsupervised GraphWise Model

You can fetch the training loss value for a Unsupervised GraphWise Model.

- Get the loss value for a Unsupervised GraphWise model as shown in the following code:

Getting the Loss Value Using JShell

```
opg4j> var loss = model.getTrainingLoss();
```

Getting the Loss Value Using Java

```
double loss = model.getTrainingLoss();
```


vertexId	embedding

 **Note:**

All the preceding examples assume that you are inferring the embeddings for a model in the current logged in database. If you must infer embeddings for the model in a different database then refer to the examples in [Inferring Embeddings for a Model in Another Database](#).

8.3.7 Storing a Unsupervised GraphWise Model

You can store models in database. The models get stored as a row inside a model store table.

- Store a trained Unsupervised GraphWise Model as shown in following code:

Storing a Trained Unsupervised GraphWise Model Using JShell

```
opg4j> model.export().db().
        modelstore("modelstoretablename"). // name of the model
store table
        modelname("model").                // model name (primary
key of model store table)
        description("a model description"). // description to store
alongside the model
        store();
```

Storing a Trained Unsupervised GraphWise Model Using Java

```
model.export().db()
        .modelstore("modelstoretablename") // name of the model store table
        .modelname("model")                // model name (primary key of
model store table)
        .description("a model description") // description to store alongside
the model
        .store();
```

Storing a Trained Unsupervised GraphWise Model Using Python

```
model.export().db(model_store="modelstoretablename",
        model_name="model", description="a model description")
```

 **Note:**

All the preceding examples assume that you are storing the model in the current logged in database. If you must store the model in a different database then refer to the examples in [Storing a Trained Model in Another Database](#).

8.3.8 Loading a Pre-Trained Unsupervised GraphWise Model

You can load models from a database.

- Load a pre-trained Unsupervised GraphWise Model from a model store table as shown in following code:

Loading a Pre-Trained Unsupervised GraphWise Model Using JShell

```
opg4j> var model = analyst.loadUnsupervisedGraphWiseModel().db().
        modelstore("modeltablename"). // name of the model
store table
        modelname("model").           // model name
(primary key of model store table)
        load();
```

Loading a Pre-Trained Unsupervised GraphWise Model Using Java

```
UnsupervisedGraphWiseModel model =
analyst.loadUnsupervisedGraphWiseModel().db()
        .modelstore("modeltablename") // name of the model store table
        .modelname("model")           // model name (primary key of
model store table)
        .load();
```

Loading a Pre-Trained Unsupervised GraphWise Model Using Python

```
analyst.get_unsupervised_graphwise_model_loader().db(model_store="mo
delstoretablename",
                                                    model_name="model")
```

 **Note:**

All the preceding examples assume that you are loading the model from the current logged in database. If you must load the model from a different database then refer to the examples in [Loading a Pre-Trained Model From Another Database](#).

8.3.9 Destroying a Unsupervised GraphWise Model

- Destroy a Unsupervised GraphWise model as described in the following code:

Destroy a Unsupervised GraphWise Model Using JShell

```
opg-jshell> model.destroy();
```

Destroy a Unsupervised GraphWise Model Using Java

```
model.destroy();
```

Destroy a Unsupervised GraphWise Model Using Python

```
model.destroy()
```

8.4 Using the Pg2vec Algorithm

Pg2vec learns representations of graphlets (partitions inside a graph) by employing edges as the principal learning units and thereby packing more information in each learning unit (as compared to employing vertices as learning units) for the representation learning task.

It consists of three main steps:

1. Random walks for each vertex (with pre-defined length per walk and pre-defined number of walks per vertex) are generated.
2. Each edge in this random walk is mapped as a `property.edge-word` in the created document (with the document label as the graph-id) where the `property.edge-word` is defined as the concatenation of the properties of the source and destination vertices.
3. The generated documents (with their attached document labels) are fed to a [doc2vec](#) algorithm which generates the vector representation for each document, which is a graph in this case.

Pg2vec creates graphlet embeddings for a specific set of graphlets and cannot be updated to incorporate modifications on these graphlets. Instead, a new Pg2vec model should be trained on these modified graphlets.

The following represents the memory consumption of Pg2vec model.

$$O(2(n+m)*d)$$

where:

- `n`: is the number of vertices in the graph
- `m`: is the number of graphlets in the graph
- `d`: is the embedding length

The following describes the usage of the main functionalities of the implementation of Pg2vec in PGX using [NCI109](#) dataset as an example with 4127 graphs in it:

- [Loading a Graph](#)
- [Building a Minimal Pg2vec Model](#)
- [Building a Customized Pg2vec Model](#)
- [Training a Pg2vec Model](#)
- [Getting the Loss Value For a Pg2vec Model](#)

- [Computing Similar Graphlets for a Given Graphlet](#)
- [Computing Similar Graphlets for a Graphlet Batch](#)
- [Inferring a Graphlet Vector](#)
- [Inferring Vectors for a Graphlet Batch](#)
- [Storing a Trained Pg2vec Model](#)
- [Loading a Pre-Trained Pg2vec Model](#)
- [Destroying a Pg2vec Model](#)

8.4.1 Loading a Graph

The following describes the steps for loading a graph:

1. Create a **Session** and an **Analyst**.
Creating a Session and an Analyst Using JShell

```
cd /opt/oracle/graph/  
./bin/opg4j  
// starting the shell will create an implicit session and analyst
```

Creating a Session and an Analyst Using Java

```
import oracle.pgx.api.*;  
import oracle.pgx.api.mllib.Pg2vecModel;  
import oracle.pgx.api.frames.*;  
...  
PgxSession session = Pgx.createSession("my-session");  
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Python

```
session = pypgx.get_session(session_name="my-session")  
analyst = session.create_analyst()
```

2. Load the **graph**.
Loading a graph using JShell

```
opg4j> var graph = session.readGraphWithProperties("<path>/  
<graph.json>");
```

Loading a graph using Java

```
PgxGraph graph = session.readGraphWithProperties("<path>/  
<graph.json>");
```

Loading a graph using Python

```
graph = session.read_graph_with_properties("<path>/<graph.json>")
```

8.4.2 Building a Minimal Pg2vec Model

You can build a Pg2vec model using the minimal configuration and default hyper-parameters as described in the following code:

Building a Minimal Pg2vec Model Using JShell

```
opg4j> var model = analyst.pg2vecModelBuilder().
        setGraphLetIdPropertyName("graph_id").
        setVertexPropertyNames(Arrays.asList("category")).
        setWindowSize(4).
        setWalksPerVertex(5).
        setWalkLength(8).
        build();
```

Building a Minimal Pg2vec Model Using Java

```
Pg2vecModel model = analyst.pg2vecModelBuilder()
    .setGraphLetIdPropertyName("graph_id")
    .setVertexPropertyNames(Arrays.asList("category"))
    .setWindowSize(4)
    .setWalksPerVertex(5)
    .setWalkLength(8)
    .build();
```

Building a Minimal Pg2vec Model Using Python

```
model = analyst.pg2vec_model_builder(
    graph_let_id_property_name="graph_id",
    vertex_property_names(["category"]),
    window_size=4,
    walks_per_vertex=5,
    walk_length=8)
```

You can specify the property name to determine each graphlet using the `Pg2vecModelBuilder#setGraphLetIdPropertyName` operation and also employ the vertex properties in Pg2vec which are specified using the `Pg2vecModelBuilder#setVertexPropertyNames` operation.

You can also use the weakly connected component (WCC) functionality in PGX to determine the graphlets in a given graph.

8.4.3 Building a Customized Pg2vec Model

You can build a Pg2vec model using customized hyper-parameters as described in the following code:

Building a Customized Pg2vec model Using JShell

```
opg4j> var model = analyst.pg2vecModelBuilder().
    setGraphLetIdPropertyName("graph_id").
    setVertexPropertyNames(Arrays.asList("category")).
    setMinWordFrequency(1).
    setBatchSize(128).
    setNumEpochs(5).
    setLayerSize(200).
    setLearningRate(0.04).
    setMinLearningRate(0.0001).
    setWindowSize(4).
    setWalksPerVertex(5).
    setWalkLength(8).
    setUseGraphletSize(true).
    setValidationFraction(0.05).
    setGraphletSizePropertyName("<propertyName>").
    build();
```

Building a Customized Pg2vec model Using Java

```
Pg2vecModel model= analyst.pg2vecModelBuilder()
    .setGraphLetIdPropertyName("graph_id")
    .setVertexPropertyNames(Arrays.asList("category"))
    .setMinWordFrequency(1)
    .setBatchSize(128)
    .setNumEpochs(5)
    .setLayerSize(200)
    .setLearningRate(0.04)
    .setMinLearningRate(0.0001)
    .setWindowSize(4)
    .setWalksPerVertex(5)
    .setWalkLength(8)
    .setUseGraphletSize(true)
    .setValidationFraction(0.05)
    .setGraphletSizePropertyName("<propertyName>")
    .build();
```

Building a Customized Pg2vec model Using Python

```
model = analyst.pg2vec_model_builder(
    graph_let_id_property_name = "graph_id",
    vertex_property_names = ["category"],
    min_word_frequency = 1,
```

```
batch_size = 128,  
num_epochs = 5,  
layer_size = 200,  
learning_rate = 0.04,  
min_learning_rate = 0.0001,  
window_size = 4,  
walks_per_vertex = 5,  
walk_length = 8,  
use_graphlet_size = true,  
graphlet_size_property_name = "<property_name>",  
validation_fraction = 0.05)
```

See [Pg2vecModelBuilder](#) in Javadoc for more explanation for each builder operation along with the default values.

8.4.4 Training a Pg2vec Model

You can train a Pg2vec model with the specified default or customized settings as described in the following code:

Training a Pg2vec Model Using JShell

```
opg4j> model.fit(graph);
```

Training a Pg2vec Model Using Java

```
model.fit(graph);
```

Training a Pg2vec Model Using Python

```
model.fit(graph)
```

8.4.5 Getting the Loss Value For a Pg2vec Model

You can fetch the training loss value on a specified fraction of training data (set in builder using `setValidationFraction`) as described in the following code:

Getting the Loss Value Using JShell

```
opg4j> var loss = model.getLoss();
```

Getting the Loss Value Using Java

```
double loss = model.getLoss();
```


Getting the Loss Value Using Python

```
loss = model.loss
```

8.4.6 Computing Similar Graphlets for a Given Graphlet

You can fetch the k most similar graphlets for a given graphlet as described in the following code:

Computing Similar Graphlets for Given Graphlet Using JShell

```
opg4j> var similars = model.computeSimilar(52, 10);
```

Computing Similar Graphlets for Given Graphlet Using Java

```
PgxFrame similars = model.computeSimilar(52, 10);
```

Computing Similar Graphlets for Given Graphlet Using Python

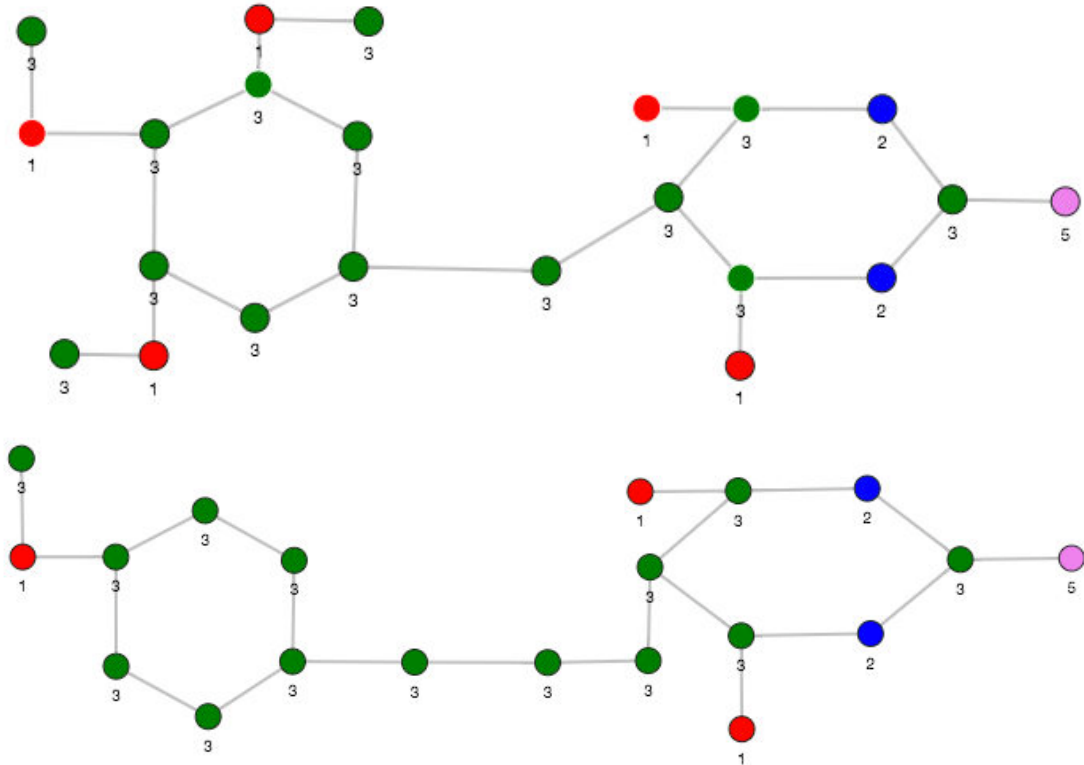
```
similars = model.compute_similar(52, 10)
```

Searching for similar vertices for graphlet with ID = 52 using the trained model and printing it with `similars.print()`, will result in the following output:

dstGraphlet	similarity
52	1.0
10	0.8748674392700195
23	0.8551455140113831
26	0.8493421673774719
47	0.8411962985992432
25	0.8281504511833191
43	0.8202780485153198
24	0.8179885745048523
8	0.796689510345459
9	0.7947834134101868

The following depicts the visualization of two similar graphlets (top: ID = 52 and bottom: ID = 10):

Figure 8-1 Pg2vec - Visualization of Two Similar Graphlets



8.4.7 Computing Similar for a Graphlet Batch

You can fetch the k most similar graphlets for a batch of input graphlets as described in the following code:

Computing Similar Graphlets for a Graphlet Batch Using JShell

```
opg4j> var graphlets = new ArrayList();
opg4j> graphlets.add(52);
opg4j> graphlets.add(41);
opg4j> var batchedSimilar = model.computeSimilar(graphlets, 10);
```

Computing Similar Graphlets for a Graphlet Batch Using Java

```
List graphlets = Arrays.asList(52,41);
PgxFramework batchedSimilar = model.computeSimilar(graphlets,10);
```

Computing Similar Graphlets for a Graphlet Batch Using Python

```
batched_similar = model.compute_similar([52,41],10)
```

Searching for similar vertices for graphlet with ID = 52 and ID = 41 using the trained model and printing it with `batched_similars.print()`, will result in the following output:

srcGraphlet	dstGraphlet	similarity
52	52	1.0
52	10	0.8748674392700195
52	23	0.8551455140113831
52	26	0.8493421673774719
52	47	0.8411962985992432
52	25	0.8281504511833191
52	43	0.8202780485153198
52	24	0.8179885745048523
52	8	0.796689510345459
52	9	0.7947834134101868
41	41	1.0
41	197	0.9653506875038147
41	84	0.9552277326583862
41	157	0.9465565085411072
41	65	0.9287481307983398
41	248	0.9177336096763611
41	315	0.9043129086494446
41	92	0.8998928070068359
41	297	0.8897411227226257
41	50	0.8810243010520935

8.4.8 Inferring a Graphlet Vector

You can infer the vector representation for a given new graphlet as described in the following code:

Inferring a Graphlet Vector Using JShell

```
opg4j> var graphlet = session.readGraphWithProperties("<path>/
<graphletConfig.json>");
opg4j> inferredVector = model.inferGraphletVector(graphlet);
opg4j> inferredVector.print();
```

Inferring a Graphlet Vector Using Java

```
PgxGraph graphlet = session.readGraphWithProperties("<path>/
<graphletConfig.json>");
PgxFrame inferredVector = model.inferGraphletVector(graphlet);
inferredVector.print();
```

Inferring a Graphlet Vector Using Python

```
PgxGraph graphlet = session.read_graph_with_properties("<path>/
<graphletConfig.json>")
inferredVector = model.infer_graphlet_vector(graphlet)
inferredVector.print()
```

The schema for the `inferredVector` will be similar to the following output:

```
+-----+
| graphlet                               | embedding           |
+-----+
```

8.4.9 Inferring Vectors for a Graphlet Batch

You can infer the vector representations for multiple graphlets (specified with different graphids in a graph) as described in the following code:

Inferring Vectors for a Graphlet Batch Using JShell

```
opg4j> var graphlet = session.readGraphWithProperties("<path>/
<graphletConfig.json>");
opg4j> inferredVectorBatched = model.inferGraphletVectorBatched(graphlets);
opg4j> inferredVectorBatched.print();
```

Inferring Vectors for a Graphlet Batch Using Java

```
PgxGraph graphlet = session.readGraphWithProperties("<path>/
<graphletConfig.json>");
PgxFrame inferredVectorBatched = model.inferGraphletVectorBatched(graphlets);
inferredVector.print();
```

Inferring Vectors for a Graphlet Batch Using Python

```
graphlets = session.read_graph_with_properties("<path>/
<graphletConfig.json>")
inferred_vector_batched = model.infer_graphlet_vector_batched(graphlets)
inferred_vector_batched.print()
```

The schema is same as for `inferGraphletVector` but with more rows corresponding to the input graphlets.

8.4.10 Storing a Trained Pg2vec Model

You can store models in database. The models get stored as a row inside a model store table.

The following code shows how to store a trained Pg2vec model in database in a specific model store table:

Storing a Trained Pg2vec Model Using JShell

```
opg4j> model.export().db().
           modelstore("modelstoretablename"). // name of the model
store table
           modelname("model").                // model name
(primary key of model store table)
           description("a model description"). // description to
store alongside the model
           store();
```

Storing a Trained Pg2vec Model Using Java

```
model.export().db()
    .modelstore("modelstoretablename") // name of the model store table
    .modelname("model")                // model name (primary key of
model store table)
    .description("a model description") // description to store
alongside the model
    .store();
```

Storing a Trained Pg2vec Model Using Python

```
model.export().db(model_store="modelstoretablename",
                  model_name="model", description="a model description")
```



Note:

All the preceding examples assume that you are storing the model in the current logged in database. If you must store the model in a different database then refer to the examples in [Storing a Trained Model in Another Database](#).

8.4.11 Loading a Pre-Trained Pg2vec Model

You can load models from a database.

You can load a pre-trained Pg2vec model from a model store table in database as described in the following:

Loading a Pre-Trained Pg2vec Model Using JShell

```
opg4j> var model = analyst.loadPg2vecModel().db().
           modelstore("modeltablename"). // name of the model
store table
```

```
        modelname("model").          // model name (primary key of
model store table)
        load();
```

Loading a Pre-Trained Pg2vec Model Using Java

```
Pg2vecModel model = analyst.loadPg2vecModel().db()
        .modelstore("modeltablename") // name of the model store table
        .modelname("model")          // model name (primary key of model store
table)
        .load();
```

Loading a Pre-Trained Pg2vec Model Using Python

```
analyst.get_pg2vec_model_loader().db(model_store="modelstoretablename",
        model_name="model")
```

Note:

All the preceding examples assume that you are loading the model from the current logged in database. If you must load the model from a different database then refer to the examples in [Loading a Pre-Trained Model From Another Database](#).

8.4.12 Destroying a Pg2vec Model

You can destroy a Pg2vec model as described in the following code:

Destroying a Pg2vec Model Using JShell

```
opg4j> model.destroy();
```

Destroying a Pg2vec Model Using Java

```
model.destroy();
```

Destroying a Pg2vec Model Using Python

```
model.destroy()
```

9

OPG_API Package Subprograms

The OPG_API package contains subprograms (functions and procedures) for working with property graphs in an Oracle database.

To use the subprograms in this chapter, you must understand the conceptual and usage information in earlier chapters of this book.

This chapter provides reference information about the subprograms, in alphabetical order.

- [OPG_API.ANALYZE_PG](#)
- [OPG_API.CF](#)
- [OPG_API.CF_CLEANUP](#)
- [OPG_API.CF_PREP](#)
- [OPG_API.CLEAR_PG](#)
- [OPG_API.CLEAR_PG_INDICES](#)
- [OPG_API.CLONE_GRAPH](#)
- [OPG_API.COUNT_TRIANGLE](#)
- [OPG_API.COUNT_TRIANGLE_CLEANUP](#)
- [OPG_API.COUNT_TRIANGLE_PREP](#)
- [OPG_API.COUNT_TRIANGLE_RENUM](#)
- [OPG_API.CREATE_EDGES_TEXT_IDX](#)
- [OPG_API.CREATE_PG](#)
- [OPG_API.CREATE_PG_SNAPSHOT_TAB](#)
- [OPG_API.CREATE_PG_TEXTIDX_TAB](#)
- [OPG_API.CREATE_STAT_TABLE](#)
- [OPG_API.CREATE_SUB_GRAPH](#)
- [OPG_API.CREATE_VERTICES_TEXT_IDX](#)
- [OPG_API.DROP_EDGES_TEXT_IDX](#)
- [OPG_API.DROP_PG](#)
- [OPG_API.DROP_PG_VIEW](#)
- [OPG_API.DROP_VERTICES_TEXT_IDX](#)
- [OPG_API.ESTIMATE_TRIANGLE_RENUM](#)
- [OPG_API.EXP_EDGE_TAB_STATS](#)
- [OPG_API.EXP_VERTEX_TAB_STATS](#)
- [OPG_API.FIND_CC_MAPPING_BASED](#)
- [OPG_API.FIND_CLUSTERS_CLEANUP](#)

- OPG_APIS.FIND_CLUSTERS_PREP
- OPG_APIS.FIND_SP
- OPG_APIS.FIND_SP_CLEANUP
- OPG_APIS.FIND_SP_PREP
- OPG_APIS.GET_BUILD_ID
- OPG_APIS.GET_GEOMETRY_FROM_V_COL
- OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS
- OPG_APIS.GET_LATLONG_FROM_V_COL
- OPG_APIS.GET_LATLONG_FROM_V_T_COLS
- OPG_APIS.GET_LONG_LAT_GEOMETRY
- OPG_APIS.GET_LATLONG_FROM_V_COL
- OPG_APIS.GET_LONGLAT_FROM_V_T_COLS
- OPG_APIS.GET_SCN
- OPG_APIS.GET_VERSION
- OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL
- OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS
- OPG_APIS.GRANT_ACCESS
- OPG_APIS.IMP_EDGE_TAB_STATS
- OPG_APIS.IMP_VERTEX_TAB_STATS
- OPG_APIS.PR
- OPG_APIS.PR_CLEANUP
- OPG_APIS.PR_PREP
- OPG_APIS.PREPARE_TEXT_INDEX
- OPG_APIS.RENAME_PG
- OPG_APIS.SPARSIFY_GRAPH
- OPG_APIS.SPARSIFY_GRAPH_CLEANUP
- OPG_APIS.SPARSIFY_GRAPH_PREP

9.1 OPG_APIS.ANALYZE_PG

Format

```
OPG_APIS.ANALYZE_PG(  
    graph_name      IN VARCHAR2,  
    estimate_percent IN NUMBER,  
    method_opt      IN VARCHAR2,  
    degree          IN NUMBER,  
    cascade         IN BOOLEAN,  
    no_invalidate   IN BOOLEAN,  
    force           IN BOOLEAN DEFAULT FALSE,  
    options         IN VARCHAR2 DEFAULT NULL);
```


Description

Hathers, for a given property graph, statistics for the VT\$, GE\$, IT\$, and GT\$ tables.

Parameters**graph_name**

Name of the property graph.

estimate_percent

Percentage of rows to estimate in the schema tables (NULL means compute). The valid range is [0.000001,100]. Use the constant `DBMS_STATS.AUTO_SAMPLE_SIZE` to have Oracle Database determine the appropriate sample size for good statistics. This is the usual default.

mrthod_opt

Accepts either of the following options, or both in combination, for the internal property graph schema tables:

- `FOR ALL [INDEXED | HIDDEN] COLUMNS [size_clause]`
- `FOR COLUMNS [size clause] column|attribute [size_clause] [,column|attribute [size_clause]...]`

`size_clause` is defined as `size_clause := SIZE {integer | REPEAT | AUTO | SKEWONLY}`

- `integer` : Number of histogram buckets. Must be in the range [1,254].
- `REPEAT` : Collects histograms only on the columns that already have histograms.
- `AUTO` : Oracle Database determines the columns to collect histograms based on data distribution and the workload of the columns.
- `SKEWONLY` : Oracle Database determines the columns to collect histograms based on the data distribution of the columns

`column` is defined as `column := column_name | (extension)`

- `column_name` : name of a column
- `extension`: Can be either a column group in the format of `(column_name, column_name [, ...])` or an expression.

The usual default is: `FOR ALL COLUMNS SIZE AUTO`

degree

Degree of parallelism for the property graph schema tables. The usual default for degree is NULL, which means use the table default value specified by the `DEGREE` clause in the `CREATE TABLE` or `ALTER TABLE` statement. Use the constant `DBMS_STATS.DEFAULT_DEGREE` to specify the default value based on the initialization parameters. The `AUTO_DEGREE` value determines the degree of parallelism automatically. This is either 1 (serial execution) or `DEFAULT_DEGREE` (the system default value based on number of CPUs and initialization parameters) according to size of the object.

cascade

Gathers statistics on the indexes for the property graph schema tables. Use the constant `DBMS_STATS.AUTO_CASCADE` to have Oracle Database determine whether index statistics are to be collected or not. This is the usual default.

no_invalidate

If `TRUE`, does not invalidate the dependent cursors. If `FALSE`, invalidates the dependent cursors immediately. If `DBMS_STATS.AUTO_INVALIDATE` (the usual default) is in effect, Oracle Database decides when to invalidate dependent cursors.

force

If `TRUE`, performs the operation even if one or more underlying tables are locked.

options

(Reserved for future use.)

Usage Notes

Only the owner of the property graph can call this procedure.

Examples

The following example gather statistics for property graph `mypg`.

```
EXECUTE OPG_APIS.ANALYZE_PG('mypg', estimate_percent=> 0.001, method_opt=>'FOR
ALL COLUMNS SIZE AUTO', degree=>4, cascade=>true, no_invalidate=>>false,
force=>true, options=>NULL);
```

9.2 OPG_APIS.CF

Format

```
OPG_APIS.CF(
    edge_tab_name    IN      VARCHAR2,
    edge_label       IN      VARCHAR2,
    rating_property  IN      VARCHAR2,
    iterations       IN      NUMBER DEFAULT 10,
    min_error        IN      NUMBER DEFAULT 0.001,
    k                IN      NUMBER DEFAULT 5,
    learning_rate    IN      NUMBER DEFAULT 0.0002,
    decrease_rate    IN      NUMBER DEFAULT 0.95,
    regularization   IN      NUMBER DEFAULT 0.02,
    dop              IN      NUMBER DEFAULT 8,
    wt_l             IN/OUT  VARCHAR2,
    wt_r             IN/OUT  VARCHAR2,
    wt_l1            IN/OUT  VARCHAR2,
    wt_r1            IN/OUT  VARCHAR2,
    wt_i             IN/OUT  VARCHAR2,
    wt_ld            IN/OUT  VARCHAR2,
    wt_rd            IN/OUT  VARCHAR2,
    tablespace       IN      VARCHAR2 DEFAULT NULL,
    options          IN      VARCHAR2 DEFAULT NULL);
```

Description

Runs collaborative filtering using matrix factorization on the given graph. The resulting factors of the matrix product will be stored on the left and right tables.

Parameters**edge_tab_name**

Name of the property graph edge table (GE\$).

edge_label

Label of the edges that hold the rating property.

rating_property

(Reserved for future use: Name of the rating property.)

iterations

Maximum number of iterations that should be performed. Default = 10.

min_error

Minimal error to reach. If at some iteration the error value is lower than this value, the procedure finishes.. Default = 0.001.

k

Number of features for the left and right side products. Default = 5.

learning_rate

Learning rate for the gradient descent. Default = 0.0002.

decrease_rate

(Reserved for future use: Decrease rate if the learning rate is too large for an effective gradient descent. Default = 0.95.)

regularization

An additional parameter to avoid overfitting. Default = 0.02

dop

Degree of parallelism. Default = 8.

wt_l

Name of the working table that holds the left side of the matrix factorization.

wt_r

Name of the working table that holds the right side of the matrix factorization.

wt_l1

Name of the working table that holds the left side intermediate step in the gradient descent.

wt_r1

Name of the working table that holds the right side intermediate step in the gradient descent.

wt_l

Name of the working table that holds intermediate matrix product.

wt_ld

Name of the working table that holds intermediate left side delta in gradient descent.

wt_rd

Name of the working table that holds intermediate right side delta in gradient descent.

tablespace

Name of the tablespace to use for storing intermediate data.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

For information about collaborative filtering with RDF data, see [SQL-Based Property Graph Analytics](#), especially [Collaborative Filtering Overview and Examples](#).

If the working tables already exist, you can specify their names for the working table-related parameters. In this case, the algorithm can continue the progress of the previous iterations without recreating the tables.

If the working tables do not exist, or if you do not want to use existing working tables, you must first call the [OPG_APIS.CF_PREP](#) procedure, which creates the necessary working tables.

The final result of the collaborative filtering algorithm are the working tables `wt_l` and `wt_r`, which are the two factors of a matrix product. These matrix factors should be used when making predictions for collaborative filtering.

If (and only if) you have no interest in keeping the output matrix factors and the current progress of the algorithm for future use, you can call the [OPG_APIS.CF_CLEANUP](#) procedure to drop all the working tables that hold intermediate tables and the output matrix factors.

Examples

The following example calls the [OPG_APIS.CF_PREP](#) procedure to create the working tables, and then the [OPG_APIS.CF](#) procedures to run collaborative filtering on the `phones` graph using the edges with the `rating` label.

```
DECLARE
  wt_l varchar2(32);
  wt_r varchar2(32);
  wt_ll varchar2(32);
  wt_rl varchar2(32);
  wt_i varchar2(32);
  wt_ld varchar2(32);
  wt_rd varchar2(32);
  edge_tab_name  varchar2(32) := 'phonesge$';
  edge_label     varchar2(32) := 'rating';
  rating_property varchar2(32) := '';
  iterations     integer      := 100;
  min_error     number       := 0.001;
  k             integer      := 5;
  learning_rate  number       := 0.001;
  decrease_rate  number       := 0.95;
  regularization number       := 0.02;
  dop           number       := 2;
  tablespace    varchar2(32) := null;
  options       varchar2(32) := null;
BEGIN
  opg_apis.cf_prep(edge_tab_name,wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd);
  opg_apis.cf(edge_tab_name,edge_label,rating_property,iterations,min_error,k,
    learning_rate,decrease_rate,regularization,dop,
    wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd,tablespace,options);
```

```
END;
/
```

The following example assumes that OPG_APIS.CF_PREP had been run previously, and it specifies the various working tables that were created during that run. In this case, the preceding example automatically assigned suffixes like '\$\$CFL57' to the names of the working tables. (The output names can be printed when they are generated or be user-defined in the call to OPG_APIS.CF_PREP.) Thus, the following example can run more iterations of the algorithm using OPG_APIS.CF without needing to call OPG_APIS.CF_PREP first, thereby continuing the progress of the previous run.

```
DECLARE
  wt_l varchar2(32) = 'phonesge$$CFL57';
  wt_r varchar2(32) = 'phonesge$$CFR57';
  wt_ll varchar2(32) = 'phonesge$$CFL157';
  wt_rl varchar2(32) = 'phonesge$$CFR157';
  wt_i varchar2(32) = 'phonesge$$CFI57';
  wt_ld varchar2(32) = 'phonesge$$CFLD57';
  wt_rd varchar2(32) = 'phonesge$$CFRD57';
  edge_tab_name varchar2(32) := 'phonesge$';
  edge_label varchar2(32) := 'rating';
  rating_property varchar2(32) := '';
  iterations integer := 100;
  min_error number := 0.001;
  k integer := 5;
  learning_rate number := 0.001;
  decrease_rate number := 0.95;
  regularization number := 0.02;
  dop number := 2;
  tablespace varchar2(32) := null;
  options varchar2(32) := null;
BEGIN
  opg_apis.cf(edge_tab_name,edge_label,rating_property,iterations,min_error,k,
             learning_rate,decrease_rate,regularization,dop,
             wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd,tablespace,options);
END;
/
```

9.3 OPG_APIS.CF_CLEANUP

Format

```
OPG_APIS.CF_CLEANUP(
  wt_l          IN/OUT  VARCHAR2,
  wt_r          IN/OUT  VARCHAR2,
  wt_ll         IN/OUT  VARCHAR2,
  wt_rl         IN/OUT  VARCHAR2,
  wt_i          IN/OUT  VARCHAR2,
  wt_ld         IN/OUT  VARCHAR2,
  wt_rd         IN/OUT  VARCHAR2,
  options       IN      VARCHAR2 DEFAULT NULL);
```

Description

Performs cleanup work after graph collaborative filtering has been done. All the working tables that hold intermediate tables and the output matrix factors are dropped.

Parameters

edge_tab_name

Name of the property graph edge table (GE\$).

wt_l

Name of the working table that holds the left side of the matrix factorization.

wt_r

Name of the working table that holds the right side of the matrix factorization.

wt_l1

Name of the working table that holds the left side intermediate step in the gradient descent.

wt_r1

Name of the working table that holds the right side intermediate step in the gradient descent.

wt_i

Name of the working table that holds intermediate matrix product.

wt_ld

Name of the working table that holds intermediate left side delta in gradient descent.

wt_rd

Name of the working table that holds intermediate right side delta in gradient descent.

options

(Reserved for future use.)

Usage Notes

Call this procedure only when you have no interest in keeping the output matrix factors and the current progress of the algorithm for future use.

Do **not** call this procedure if more predictions will be made using the resulting product factors (`wt_l` and `wt_r` tables), unless you have previously made backup copies of these two tables.

See also the information about the [OPG_APIS.CF](#) procedure.

Examples

The following example drops the working tables that were created in the example for the [OPG_APIS.CF_PREP](#) procedure.

```
DECLARE
  wt_l varchar2(32) = 'phonesge$$CFL57';
  wt_r varchar2(32) = 'phonesge$$CFR57';
  wt_l1 varchar2(32) = 'phonesge$$CFL157';
  wt_r1 varchar2(32) = 'phonesge$$CFR157';
  wt_i varchar2(32) = 'phonesge$$CFI57';
  wt_ld varchar2(32) = 'phonesge$$CFLD57';
  wt_rd varchar2(32) = 'phonesge$$CFRD57';
BEGIN
  opg_apis.cf_cleanup('phonesge$',wt_l,wt_r,wt_l1,wt_r1,wt_i,wt_ld,wt_rd);
```

```
END;  
/
```

9.4 OPG_APIS.CF_PREP

Format

```
OPG_APIS.CF_PREP(  
    wt_l          IN/OUT  VARCHAR2.  
    wt_r          IN/OUT  VARCHAR2.  
    wt_l1         IN/OUT  VARCHAR2.  
    wt_r1         IN/OUT  VARCHAR2.  
    wt_i          IN/OUT  VARCHAR2.  
    wt_ld         IN/OUT  VARCHAR2.  
    wt_rd         IN/OUT  VARCHAR2.  
    options       IN      VARCHAR2 DEFAULT NULL);
```

Description

Performs preparation work, including creating the necessary intermediate tables, for a later call to the [OPG_APIS.CF](#) procedure that will perform collaborative filtering.

Parameters

edge_tab_name

Name of the property graph edge table (GE\$).

wt_l

Name of the working table that holds the left side of the matrix factorization.

wt_r

Name of the working table that holds the right side of the matrix factorization.

wt_l1

Name of the working table that holds the left side intermediate step in the gradient descent.

wt_r1

Name of the working table that holds the right side intermediate step in the gradient descent.

wt_i

Name of the working table that holds intermediate matrix product.

wt_ld

Name of the working table that holds intermediate left side delta in gradient descent.

wt_rd

Name of the working table that holds intermediate right side delta in gradient descent.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

The names of the working tables can be specified or left as null parameters. If the name of any working table parameter is not specified, a name is automatically generated and is returned as an OUT parameter. The working table names can be used when you call the [OPG_APIS.CF](#) procedure to run the collaborative filtering algorithm.

See also the Usage Notes and Examples for [OPG_APIS.CF](#).

Examples

The following example creates the working tables for a graph named `phones`, and it prints the names that were automatically generated for the working tables.

```
DECLARE
  wt_l varchar2(32);
  wt_r varchar2(32);
  wt_ll varchar2(32);
  wt_rl varchar2(32);
  wt_i varchar2(32);
  wt_ld varchar2(32);
  wt_rd varchar2(32);
BEGIN
  opg_apis.cf_prep('phonesge$',wt_l,wt_r,wt_ll,wt_rl,wt_i,wt_ld,wt_rd);
  dbms_output.put_line(' wt_l ' || wt_l);
  dbms_output.put_line(' wt_r ' || wt_r);
  dbms_output.put_line(' wt_ll ' || wt_ll);
  dbms_output.put_line(' wt_rl ' || wt_rl);
  dbms_output.put_line(' wt_i ' || wt_i);
  dbms_output.put_line(' wt_ld ' || wt_ld);
  dbms_output.put_line(' wt_rd ' || wt_rd);
END;
/
```

9.5 OPG_APIS.CLEAR_PG

Format

```
OPG_APIS.CLEAR_PG(
  graph_name IN VARCHAR2);
```

Description

Clears all data from a property graph.

Parameters

graph_name

Name of the property graph.

Usage Notes

This procedure removes all data in the property graph by deleting data in the graph tables (VT\$, GE\$, and so on).

Examples

The following example removes all data from the property graph named `mypg`.

```
EXECUTE OPG_APIS.CLEAR_PG('mypg');
```

9.6 OPG_APIS.CLEAR_PG_INDICES

Format

```
OPG_APIS.CLEAR_PG(  
    graph_name IN VARCHAR2);
```

Description

Removes all text index metadata in the IT\$ table of the property graph.

Parameters

graph_name

Name of the property graph.

Usage Notes

This procedure does not actually remove text index data

Examples

The following example removes all index metadata of the property graph named `mypg`.

```
EXECUTE OPG_APIS.CLEAR_PG_INDICES('mypg');
```

9.7 OPG_APIS.CLONE_GRAPH

Format

```
OPG_APIS.CLONE_GRAPH(  
    orgGraph      IN VARCHAR2,  
    newGraph      IN VARCHAR2,  
    dop           IN INTEGER DEFAULT 4,  
    num_hash_ptns IN INTEGER DEFAULT 8,  
    tbs           IN VARCHAR2 DEFAULT NULL);
```

Description

Makes a clone of the original graph, giving the new graph a new name.

Parameters

orgGraph

Name of the original property graph.

newGraph

Name of the new (clone) property graph.

dop

Degree of parallelism for the operation.

num_hash_ptns

Number of hash partitions used to partition the vertices and edges tables. It is recommended to use a power of 2 (2, 4, 8, 16, and so on).

tbs

Name of the tablespace to hold all the graph data and index data.

Usage Notes

The original property graph must already exist in the database.

Examples

The following example creates a clone graph named `mypgclone` from the property graph `mypg` in the tablespace `my_ts` using a degree of parallelism of 4 and 8 partitions.

```
EXECUTE OPG_APIS.CLONE_GRAPH('mypg', 'mypgclone', 4, 8, 'my_ts');
```

9.8 OPG_APIS.COUNT_TRIANGLE

Format

```
OPG_APIS.COUNT_TRIANGLE(  
    edge_tab_name IN VARCHAR2,  
    wt_und        IN OUT VARCHAR2,  
    num_sub_ptns  IN NUMBER DEFAULT 1,  
    dop           IN INTEGER DEFAULT 1,  
    tbs           IN VARCHAR2 DEFAULT NULL,  
    options       IN VARCHAR2 DEFAULT NULL  
) RETURN NUMBER;
```

Description

Performs triangle counting in property graph.

Parameters**edge_tab_name**

Name of the property graph edge table.

wt_und

A working table holding an undirected version of the graph.

num_sub_ptns

Number of logical subpartitions used in calculating triangles. Must be a positive integer, power of 2 (1, 2, 4, 8, ...). For a graph with a relatively small maximum degree, use the value 1 (the default).

dop

Degree of parallelism for the operation. The default is 1.

tbs

Name of the tablespace to hold the data stored in working tables.

options

Additional settings for the operation:

- 'PDML=T' enables parallel DML.

Usage Notes

The property graph edge table must exist in the database, and the [OPG_APIS.COUNT_TRIANGLE_PREP](#) procedure must already have been executed.

Examples

The following example performs triangle counting in the property graph named `connections`

```
set serveroutput on
DECLARE
  wt1 varchar2(100); -- intermediate working table
  wt2 varchar2(100);
  wt3 varchar2(100);
  n number;
BEGIN
  opg_apis.count_triangle_prep('connectionsGE$', wt1, wt2, wt3);
  n := opg_apis.count_triangle(
    'connectionsGE$',
    wt1,
    num_sub_ptns=>1,
    dop=>2,
    tbs => 'MYPG_TS',
    options=>'PDML=T'
  );
  dbms_output.put_line('total number of triangles ' || n);
END;
/
```

9.9 OPG_APIS.COUNT_TRIANGLE_CLEANUP

Format

```
COUNT_TRIANGLE_CLEANUP(
  edge_tab_name IN VARCHAR2,
  wt_undBM      IN VARCHAR2,
  wt_rnmap      IN VARCHAR2,
  wt_undAM      IN VARCHAR2,
  options       IN VARCHAR2 DEFAULT NULL);
```

Description

Cleans up and drops the temporary working tables used for triangle counting.

Parameters**edge_tab_name**

Name of the property graph edge table.

wt_undBM

A working table holding an undirected version of the original graph (before renumbering optimization).

wt_rnmap

A working table that is a mapping table for renumbering optimization.

wt_undAM

A working table holding the undirected version of the graph data after applying the renumbering optimization.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- PDML=T enables parallel DML.

Usage Notes

You should use this procedure to clean up after triangle counting.

The working tables must exist in the database.

Examples

The following example performs triangle counting in the property graph named `connections`, and drops the working table after it has finished.

```
set serveroutput on

DECLARE
  wt1 varchar2(100); -- intermediate working table
  wt2 varchar2(100);
  wt3 varchar2(100);
  n number;
BEGIN
  opg_apis.count_triangle_prep('connectionsGE$', wt1, wt2, wt3);
  n := opg_apis.count_triangle_renum(
    'connectionsGE$',
    wt1,
    wt2,
    wt3,
    num_sub_ptns=>1,
    dop=>2,
    tbs => 'MYPG_TS',
    options=>'PDML=T'
  );
  dbms_output.put_line('total number of triangles ' || n);
  opg_apis.count_triangle_cleanup('connectionsGE$', wt1, wt2, wt3);
END;
/
```

9.10 OPG_APIS.COUNT_TRIANGLE_PREP

Format

```
OPG_APIS.COUNT_TRIANGLE_PREP(
  edge_tab_name IN VARCHAR2,
  wt_undBM      IN OUT VARCHAR2,
  wt_rnmap      IN OUT VARCHAR2,
  wt_undAM      IN OUT VARCHAR2,
  options       IN VARCHAR2 DEFAULT NULL);
```

Description

Prepares for running triangle counting.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_undBM

A working table holding an undirected version of the original graph (before renumbering optimization).

wt_rnmap

A working table that is a mapping table for renumbering optimization.

wt_undAM

A working table holding the undirected version of the graph data after applying the renumbering optimization.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- CREATE_UNDIRECTED=T
- REUSE_UNDIRECTED_TAB=T

Usage Notes

The property graph edge table must exist in the database.

Examples

The following example prepares for triangle counting in a property graph named connections.

```
set serveroutput on

DECLARE
  wt1 varchar2(100); -- intermediate working table
  wt2 varchar2(100);
  wt3 varchar2(100);
  n number;
BEGIN
  opg_apis.count_triangle_prep('connectionsGE$', wt1, wt2, wt3);

  n := opg_apis.count_triangle_renum(
    'connectionsGE$',
    wt1,
    wt2,
    wt3,
    num_sub_ptns=>1,
    dop=>2,
    tbs => 'MYPG_TS',
    options=>'CREATE_UNDIRECTED=T,REUSE_UNDIREC_TAB=T'
  );
  dbms_output.put_line('total number of triangles ' || n);
```

```
END;  
/
```

9.11 OPG_APIS.COUNT_TRIANGLE_RENUM

Format

```
COUNT_TRIANGLE_RENUM(  
    edge_tab_name IN VARCHAR2,  
    wt_undBM      IN VARCHAR2,  
    wt_rnmap      IN VARCHAR2,  
    wt_undAM      IN VARCHAR2,  
    num_sub_ptns  IN INTEGER DEFAULT 1,  
    dop           IN INTEGER DEFAULT 1,  
    tbs           IN VARCHAR2 DEFAULT NULL,  
    options       IN VARCHAR2 DEFAULT NULL  
) RETURN NUMBER;
```

Description

Performs triangle counting in property graph, with the optimization of renumbering the vertices of the graph by their degree.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_undBM

A working table holding an undirected version of the original graph (before renumbering optimization).

wt_rnmap

A working table that is a mapping table for renumbering optimization.

wt_undAM

A working table holding the undirected version of the graph data after applying the renumbering optimization.

num_sub_ptns

Number of logical subpartitions used in calculating triangles . Must be a positive integer, power of 2 (1, 2, 4, 8, ...). For a graph with a relatively small maximum degree, use the value 1 (the default).

dop

Degree of parallelism for the operation. The default is 1 (no parallelism).

tbs

Name of the tablespace to hold the data stored in working tables.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- PDML=T enables parallel DML.

Usage Notes

This function makes the algorithm run faster, but requires more space.

The property graph edge table must exist in the database, and the [OPG_APIS.COUNT_TRIANGLE_PREP](#) procedure must already have been executed.

Examples

The following example performs triangle counting in the property graph named `connections`. It does not perform the cleanup after it finishes, so you can count triangles again on the same graph without calling the preparation procedure.

```

set serveroutput on

DECLARE
  wt1 varchar2(100); -- intermediate working table
  wt2 varchar2(100);
  wt3 varchar2(100);
  n number;
BEGIN
  opg_apis.count_triangle_prep('connectionsGE$', wt1, wt2, wt3);
  n := opg_apis.count_triangle_renum(
    'connectionsGE$',
    wt1,
    wt2,
    wt3,
    num_sub_ptns=>1,
    dop=>2,
    tbs => 'MYPG_TS',
    options=>'PDML=T'
  );
  dbms_output.put_line('total number of triangles ' || n);
END;
/

```

9.12 OPG_APIS.CREATE_EDGES_TEXT_IDX

Format

```

OPG_APIS.CREATE_EDGES_TEXT_IDX(
  graph_owner IN VARCHAR2,
  graph_name  IN VARCHAR2,
  pref_owner  IN VARCHAR2 DEFAULT NULL,
  datastore   IN VARCHAR2 DEFAULT NULL,
  filter      IN VARCHAR2 DEFAULT NULL,
  storage     IN VARCHAR2 DEFAULT NULL,
  wordlist    IN VARCHAR2 DEFAULT NULL,
  stoplist    IN VARCHAR2 DEFAULT NULL,
  lexer       IN VARCHAR2 DEFAULT NULL,
  dop         IN INTEGER  DEFAULT NULL,
  options     IN VARCHAR2 DEFAULT NULL,);

```

Description

Creates a text index on a property graph edge table.

Parameters**graph_owner**

Owner of the property graph.

graph_name

Name of the property graph.

pref_owner

Owner of the preference.

datastore

The way that documents are stored.

filter

The way that documents can be converted to plain text.

storage

The way that the index data is stored.

wordlist

The way that stem and fuzzy queries should be expanded

stoplist

The words or themes that are not to be indexed.

lexer

The language used for indexing.

dop

The degree of parallelism used for index creation.

options

Additional settings for index creation.

Usage Notes

The property graph must exist in the database.

You must have the ALTER SESSION privilege to run this procedure.

Examples

The following example creates a text index on the edge table of property graph `mypg`, which is owned by user `SCOTT`, using the lexer `OPG_AUTO_LEXER` and a degree of parallelism of 4.

```
EXECUTE OPG_APIS.CREATE_EDGES_TEXT_IDX('SCOTT', 'mypg', 'MDSYS', null, null,  
null, null, null, 'OPG_AUTO_LEXER', 4, null);
```

9.13 OPG_APIS.CREATE_PG

Format

```
OPG_APIS.CREATE_PG(  
    graph_name    IN VARCHAR2,  
    dop           IN INTEGER DEFAULT NULL,
```



```
num_hash_ptns IN INTEGER DEFAULT 8,  
tbs           IN VARCHAR2 DEFAULT NULL,  
options      IN VARCHAR2 DEFAULT NULL);
```

Description

Creates, for a given property graph name, the necessary property graph schema tables that are necessary to store data about vertices, edges, text indexes, and snapshots.

Parameters

graph_name

Name of the property graph.

dop

Degree of parallelism for the operation.

num_hash_ptns

Number of hash partitions used to partition the vertices and edges tables. It is recommended to use a power of 2 (2, 4, 8, 16, and so on).

tbs

Name of the tablespace to hold all the graph data and index data.

options

Options that can be used to customize the creation of indexes on schema tables. (One or more, comma separated.)

- 'SKIP_INDEX=T' skips the default index creation.
- 'SKIP_ERROR=T' ignores errors encountered during table/index creation.
- 'INMEMORY=T' creates the schema tables with an INMEMORY clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY BASIC clause.

Usage Notes

You must have the CREATE TABLE and CREATE INDEX privileges to call this procedure.

By default, all the schema tables will be created with basic compression enabled.

Examples

The following example creates a property graph named `mypg` in the tablespace `my_ts` using eight partitions.

```
EXECUTE OPG_APIS.CREATE_PG('mypg', 4, 8, 'my_ts');
```

9.14 OPG_APIS.CREATE_PG_SNAPSHOT_TAB

Format

```
OPG_APIS.CREATE_PG_SNAPSHOT_TAB(  
  graph_owner IN VARCHAR2,  
  graph_name  IN VARCHAR2,  
  dop         IN INTEGER DEFAULT NULL,  
  tbs        IN VARCHAR2 DEFAULT NULL,  
  options    IN VARCHAR2 DEFAULT NULL);
```

or

```
OPG_APIS.CREATE_PG_SNAPSHOT_TAB(
  graph_name IN VARCHAR2,
  dop        IN INTEGER DEFAULT NULL,
  tbs        IN VARCHAR2 DEFAULT NULL,
  options    IN VARCHAR2 DEFAULT NULL);
```

Description

Creates, for a given property graph name, the necessary property graph schema table (<graph_name>SS\$) that stores data about snapshots for the graph.

Parameters

graph_owner

Name of the owner of the property graph.

graph_name

Name of the property graph.

dop

Degree of parallelism for the operation.

tbs

Name of the tablespace to hold all the graph snapshot data and associated index.

options

Additional settings for the operation:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

You must have the CREATE TABLE privilege to call this procedure.

The created snapshot table has the following structure, which may change between releases.

Name	Null?	Type
SSID	NOT NULL	NUMBER
CONTENTS		BLOB
SS_FILE		BINARY FILE LOB
TS		TIMESTAMP(6) WITH TIME ZONE
SS_COMMENT		VARCHAR2(512)

By default, all schema tables will be created with basic compression enabled.

Examples

The following example creates a snapshot table for property graph `mypg` in the current schema, with a degree of parallelism of 4 and using the `MY_TS` tablespace.

```
EXECUTE OPG_APIS.CREATE_PG_SNAPSHOT_TAB('mypg', 4, 'my_ts');
```

9.15 OPG_APIS.CREATE_PG_TEXTIDX_TAB

Format

```
OPG_APIS.CREATE_PG_TEXTIDX_TAB(
    graph_owner IN VARCHAR2,
    graph_name  IN VARCHAR2,
    dop         IN INTEGER DEFAULT NULL,
    tbs         IN VARCHAR2 DEFAULT NULL,
    options     IN VARCHAR2 DEFAULT NULL);
```

or

```
OPG_APIS.CREATE_PG_TEXTIDX_TAB(
    graph_name  IN VARCHAR2,
    dop         IN INTEGER DEFAULT NULL,
    tbs         IN VARCHAR2 DEFAULT NULL,
    options     IN VARCHAR2 DEFAULT NULL);
```

Description

Creates, for a given property graph name, the necessary property graph text index schema table (<graph_name>IT\$) that stores data for managing text index metadata for the graph.

Parameters

graph_owner

Name of the owner of the property graph.

graph_name

Name of the property graph.

dop

Degree of parallelism for the operation.

tbs

Name of the tablespace to hold all the graph index metadata and associated index.

options

Additional settings for the operation:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

You must have the CREATE TABLE privilege to call this procedure.

The created index metadata table has the following structure, which may change between releases.

```
(
    EIN    nvarchar2(80) not null, -- index name
    ET     number,                -- entity type 1 - vertex, 2 -edge
    IT     number,                -- index type 1 - auto   0 - manual
    SE     number,                -- search engine 1 -solr, 0 - lucene
    K      nvarchar2(3100),       -- property key use an empty space when
```

```

there is no K/V
          DT          number,          -- directory type 1 - MMAP, 2 -
FS, 3 - JDBC
          LOC          nvarchar2(3100), -- directory location (1, 2)
          NUMDIRS      number,          -- property key used to index CAN
BE NULL
          VERSION      nvarchar2(100),  -- lucene version
          USEDT         number,          -- user data type (1 or 0)
          STOREF        number,          -- store fields into lucene
          CF            nvarchar2(3100),  -- configuration name
          SS            nvarchar2(3100),  -- solr server url
          SA            nvarchar2(3100),  -- solr server admin url
          ZT            number,          -- zookeeper timeout
          SH            number,          -- number of shards
          RF            number,          -- replication factor
          MS            number,          -- maximum shards per node
          PO            nvarchar2(3100),  -- preferred owner oracle text
          DS            nvarchar2(3100),  -- datastore
          FIL           nvarchar2(3100),  -- filter
          STR           nvarchar2(3100),  -- storage
          WL            nvarchar2(3100),  -- word list
          SL            nvarchar2(3100),  -- stop list
          LXR           nvarchar2(3100),  -- lexer
          OPTS          nvarchar2(3100),  -- options
          primary key (EIN, K, ET)
        )

```

By default, all schema tables will be created with basic compression enabled.

Examples

The following example creates a property graph text index metadata table for property graph `mypg` in the current schema, with a degree of parallelism of 4 and using the `MY_TS` tablespace.

```
EXECUTE OPG_APIS.CREATE_PG_TEXTIDX_TAB('mypg', 4, 'my_ts');
```

9.16 OPG_APIS.CREATE_STAT_TABLE

Format

```
OPG_APIS.CREATE_STAT_TABLE(
    stattab IN VARCHAR2,
    tblspace IN VARCHAR2 DEFAULT NULL);
```

Description

Creates a table that can hold property graph statistics.

Parameters

stattab

Name of the table to hold statistics

tblspace

Name of the tablespace to hold the statistics table. If none is specified, then the statistics table will be created in the user's default tablespace.

Usage Notes

You must have the CREATE TABLE privilege to call this procedure.

The statistics table has the following columns. Note that the columns and their types may vary between releases.

Name	Null?	Type
STATID		VARCHAR2(128)
TYPE		CHAR(1)
VERSION		NUMBER
FLAGS		NUMBER
C1		VARCHAR2(128)
C2		VARCHAR2(128)
C3		VARCHAR2(128)
C4		VARCHAR2(128)
C5		VARCHAR2(128)
C6		VARCHAR2(128)
N1		NUMBER
N2		NUMBER
N3		NUMBER
N4		NUMBER
N5		NUMBER
N6		NUMBER
N7		NUMBER
N8		NUMBER
N9		NUMBER
N10		NUMBER
N11		NUMBER
N12		NUMBER
N13		NUMBER
D1		DATE
T1		TIMESTAMP(6) WITH TIME ZONE
R1		RAW(1000)
R2		RAW(1000)
R3		RAW(1000)
CH1		VARCHAR2(1000)
CL1		CLOB

Examples

The following example creates a statistics table named `mystat`.

```
EXECUTE OPG_APIS.CREATE_STAT_TABLE('mystat',null);
```

9.17 OPG_APIS.CREATE_SUB_GRAPH

Format

```
OPG_APIS.CREATE_SUB_GRAPH(
  graph_owner IN VARCHAR2,
  orgGraph    IN VARCHAR2,
  newGraph    IN VARCHAR2,
  nSrc        IN NUMBER,
  depth       IN NUMBER);
```

Description

Creates a subgraph, which is an expansion from a given vertex. The depth of expansion is customizable.

Parameters**graph_owner**

Owner of the property graph.

orgGraph

Name of the original property graph.

newGraph

Name of the subgraph to be created from the original graph.

nSrc

Vertex ID: the subgraph will be created by expansion from this vertex. For example, `nSrc = 1` starts the expansion from the vertex with ID 1.

depth

Depth of expansion: the expansion, following outgoing edges, will include all vertices that are within `depth` hops away from vertex `nSrc`. For example, `depth = 2` causes the to should include all vertices that are within 2 hops away from vertex `nSrc` (vertex ID 1 in the preceding example).

Usage Notes

The original property graph must exist in the database.

Examples

The following example creates a subgraph `mypgsub` from the property graph `mypg` whose owner is SCOTT. The subgraph includes vertex 1 and all vertices that are reachable from the vertex with ID 1 in 2 hops.

```
EXECUTE OPG_APIS.CREATE_SUB_GRAPH('SCOTT', 'mypg', 'mypgsub', 1, 2);
```

9.18 OPG_APIS.CREATE_VERTICES_TEXT_IDX

Format

```
OPG_APIS.CREATE_VERTICES_TEXT_IDX(
  graph_owner IN VARCHAR2,
  graph_name  IN VARCHAR2,
  pref_owner  IN VARCHAR2 DEFAULT NULL,
  datastore  IN VARCHAR2 DEFAULT NULL,
  filter      IN VARCHAR2 DEFAULT NULL,
  storage     IN VARCHAR2 DEFAULT NULL,
  wordlist   IN VARCHAR2 DEFAULT NULL,
  stoplist   IN VARCHAR2 DEFAULT NULL,
  lexer      IN VARCHAR2 DEFAULT NULL,
  dop        IN INTEGER  DEFAULT NULL,
  options    IN VARCHAR2 DEFAULT NULL,);
```

Description

Creates a text index on a property graph vertex table.

Parameters**graph_owner**

Owner of the property graph.

graph_name

Name of the property graph.

pref_owner

Owner of the preference.

datastore

The way that documents are stored.

filter

The way that documents can be converted to plain text.

storage

The way that the index data is stored.

wordlist

The way that stem and fuzzy queries should be expanded

stoplist

The words or themes that are not to be indexed.

lexer

The language used for indexing.

dop

The degree of parallelism used for index creation.

options

Additional settings for index creation.

Usage Notes

The original property graph must exist in the database.

You must have the ALTER SESSION privilege to run this procedure.

Examples

The following example creates a text index on the vertex table of property graph `mypg`, which is owned by user `SCOTT`, using the lexer `OPG_AUTO_LEXER` and a degree of parallelism of 4.

```
EXECUTE OPG_APIS.CREATE_VERTICES_TEXT_IDX('SCOTT', 'mypg', null, null, null, null,  
null, null, 'OPG_AUTO_LEXER', 4, null);
```

9.19 OPG_APIS.DROP_EDGES_TEXT_IDX

Format

```
OPG_APIS.DROP_EDGES_TEXT_IDX(  
    graph_owner IN VARCHAR2,  
    graph_name  IN VARCHAR2,  
    options     IN VARCHAR2 DEFAULT NULL);
```

Description

Drops a text index on a property graph edge table.

Parameters

graph_owner

Owner of the property graph.

graph_name

Name of the property graph.

options

Additional settings for the operation.

Usage Notes

A text index must already exist on the property graph edge table.

Examples

The following example drops the text index on the edge table of property graph `mypg` that is owned by user `SCOTT`.

```
EXECUTE OPG_APIS.DROP_EDGES_TEXT_IDX('SCOTT', 'mypg', null);
```

9.20 OPG_APIS.DROP_PG

Format

```
OPG_APIS.DROP_PG(  
    graph_name IN VARCHAR2);
```

Description

Drops (deletes) a property graph.

Parameters

graph_name

Name of the property graph.

Usage Notes

All the graph tables (VT\$, GE\$, and so on) will be dropped from the database.

Examples

The following example drops the property graph named `mypg`.

```
EXECUTE OPG_APIS.DROP_PG('mypg');
```

9.21 OPG_APIS.DROP_PG_VIEW

Format

```
OPG_APIS.DROP_PG_VIEW(  
    graph_name IN VARCHAR2);  
    options   IN VARCHAR2);
```

Description

Drops (deletes) the view definition of a property graph.

Parameters

graph_name

Name of the property graph.

options

(Reserved for future use.)

Usage Notes

Oracle supports creating physical property graphs and property graph views. For example, given an RDF model, it supports creating property graph views over the RDF model, so that you can run property graph analytics on top of the RDF graph.

This procedure cannot be undone.

Examples

The following example drops the view definition of the property graph named `mypg`.

```
EXECUTE OPG_APIS.DROP_PG_VIEW('mypg');
```

9.22 OPG_APIS.DROP_VERTICES_TEXT_IDX

Format

```
OPG_APIS.DROP_VERTICES_TEXT_IDX(  
    graph_owner IN VARCHAR2,  
    graph_name  IN VARCHAR2,  
    options     IN VARCHAR2 DEFAULT NULL);
```

Description

Drops a text index on a property graph vertex table.

Parameters

graph_owner
Owner of the property graph.

graph_name
Name of the property graph.

options
Additional settings for the operation.

Usage Notes

A text index must already exist on the property graph vertex table.

Examples

The following example drops the text index on the vertex table of property graph `mypg` that is owned by user `SCOTT`.

```
EXECUTE OPG_APIS.DROP_VERTICES_TEXT_IDX('SCOTT', 'mypg', null);
```

9.23 OPG_APIS.ESTIMATE_TRIANGLE_RENUM

Format

```
COUNT_TRIANGLE_ESTIMATE(
  edge_tab_name IN VARCHAR2,
  wt_undBM      IN VARCHAR2,
  wt_rnmap      IN VARCHAR2,
  wt_undAM      IN VARCHAR2,
  num_sub_ptns IN INTEGER DEFAULT 1,
  chunk_id      IN INTEGER DEFAULT 1,
  dop           IN INTEGER DEFAULT 1,
  tbs           IN VARCHAR2 DEFAULT NULL,
  options       IN VARCHAR2 DEFAULT NULL
) RETURN NUMBER;
```

Description

Estimates the number of triangles in a property graph.

Parameters

edge_tab_name
Name of the property graph edge table.

wt_undBM
A working table holding an undirected version of the original graph (before renumbering optimization).

wt_rnmap
A working table that is a mapping table for renumbering optimization.

wt_undAM

A working table holding the undirected version of the graph data after applying the renumbering optimization.

num_sub_ptns

Number of logical subpartitions used in calculating triangles . Must be a positive integer, power of 2 (1, 2, 4, 8, ...). For a graph with a relatively small maximum degree, use the value 1 (the default).

chunk_id

The logical subpartition to be used in triangle estimation (Only this partition will be counted). It must be an integer between 0 and $\text{num_sub_ptns} * \text{num_sub_ptns} - 1$.

dop

Degree of parallelism for the operation. The default is 1 (no parallelism).

tbs

Name of the tablespace to hold the data stored in working tables.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- PDML=T enables parallel DML.

Usage Notes

This function counts the total triangles in a portion of size $1 / (\text{num_sub_ptns} * \text{num_sub_ptns})$ of the graph; so to estimate the total number of triangles in the graph, you can multiply the result by $\text{num_sub_ptns} * \text{num_sub_ptns}$.

The property graph edge table must exist in the database, and the [OPG_APIS.COUNT_TRIANGLE_PREP](#) procedure must already have been executed.

Examples

The following example estimates the number of triangle in the property graph named `connections`. It does not perform the cleanup after it finishes, so you can count triangles again on the same graph without calling the preparation procedure.

```
set serveroutput on

DECLARE
  wt1 varchar2(100); -- intermediate working table
  wt2 varchar2(100);
  wt3 varchar2(100);
  n number;
BEGIN
  opg_apis.count_triangle_prep('connectionsGE$', wt1, wt2, wt3);
  n := opg_apis.estimate_triangle_renum(
    'connectionsGE$',
    wt1,
    wt2,
    wt3,
    num_sub_ptns=>64,
    chunk_id=>2048,
    dop=>2,
    tbs => 'MYPG_TS',
    options=>'PDML=T'
  );
```

```
    dbms_output.put_line('estimated number of triangles ' || (n * 64 * 64));  
END;  
/
```

9.24 OPG_APIS.EXP_EDGE_TAB_STATS

Format

```
OPG_APIS.EXP_EDGE_TAB_STATS(  
    graph_name    IN VARCHAR2,  
    stattab       IN VARCHAR2,  
    statid        IN VARCHAR2 DEFAULT NULL,  
    cascade       IN BOOLEAN DEFAULT TRUE,  
    statown       IN VARCHAR2 DEFAULT NULL,  
    stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Retrieves statistics for the edge table of a given property graph and stores them in the user-created statistics table.

Parameters

graph_name

Name of the property graph.

stattab

Name of the statistics table.

statid

Optional identifier to associate with these statistics within `stattab`.

cascade

If `TRUE`, column and index statistics are exported.

statown

Schema containing `stattab`.

stat_category

Specifies what statistics to export, using a comma to separate values. The supported values are `'OBJECT_STATS'` (the default: table statistics, column statistics, and index statistics) and `'SYNOPSIS'` (auxiliary statistics created when statistics are incrementally maintained).

Usage Notes

(None.)

Examples

The following example creates a statistics table, exports into this table the property graph edge table statistics, and issues a query to count the relevant rows for the newly created statistics.

```
EXECUTE OPG_APIS.CREATE_STAT_TABLE('mystat',null);  
  
EXECUTE OPG_APIS.EXP_EDGE_TAB_STATS('mypg', 'mystat', 'edge_stats_id_1', true,  
null, 'OBJECT_STATS');
```

```
SELECT count(1) FROM mystat WHERE statid='EDGE_STATS_ID_1';

153
```

9.25 OPG_APIS.EXP_VERTEX_TAB_STATS

Format

```
OPG_APIS.EXP_VERTEX_TAB_STATS(  
    graph_name    IN VARCHAR2,  
    stattab       IN VARCHAR2,  
    statid        IN VARCHAR2 DEFAULT NULL,  
    cascade       IN BOOLEAN DEFAULT TRUE,  
    statown       IN VARCHAR2 DEFAULT NULL,  
    stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Retrieves statistics for the vertex table of a given property graph and stores them in the user-created statistics table.

Parameters

graph_name

Name of the property graph.

stattab

Name of the statistics table.

statid

Optional identifier to associate with these statistics within `stattab`.

cascade

If `TRUE`, column and index statistics are exported.

statown

Schema containing `stattab`.

stat_category

Specifies what statistics to export, using a comma to separate values. The supported values are `'OBJECT_STATS'` (the default: table statistics, column statistics, and index statistics) and `'SYNOPSIS'` (auxiliary statistics created when statistics are incrementally maintained).

Usage Notes

(None.)

Examples

The following example creates a statistics table, exports into this table the property graph vertex table statistics, and issues a query to count the relevant rows for the newly created statistics.

```
EXECUTE OPG_APIS.CREATE_STAT_TABLE('mystat',null);

EXECUTE OPG_APIS.EXP_VERTEX_TAB_STATS('mypg', 'mystat', 'vertex_stats_id_1', true,  
null, 'OBJECT_STATS');
```

```
SELECT count(1) FROM mystat WHERE statid='VERTEX_STATS_ID_1';  
  
108
```

9.26 OPG_APIS.FIND_CC_MAPPING_BASED

Format

```
OPG_APIS.FIND_CC_MAPPING_BASED(  
    edge_tab_name IN VARCHAR2,  
    wt_clusters   IN OUT VARCHAR2,  
    wt_undir     IN OUT VARCHAR2,  
    wt_cluas     IN OUT VARCHAR2,  
    wt_newas     IN OUT VARCHAR2,  
    wt_delta     IN OUT VARCHAR2,  
    dop          IN INTEGER DEFAULT 4,  
    rounds       IN INTEGER DEFAULT 0,  
    tbs          IN VARCHAR2 DEFAULT NULL,  
    options      IN VARCHAR2 DEFAULT NULL);
```

Description

Finds connected components in a property graph. All connected components will be stored in the `wt_clusters` table. The original graph is treated as undirected.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_clusters

A working table holding the final vertex cluster mappings. This table has two columns (VID NUMBER, CLUSTER_ID NUMBER). Column VID stores the vertex ID values, and column CLUSTER_ID stores the corresponding cluster ID values. Cluster ID values are long integers that can have gaps between them. If an empty name is specified, a new table will be generated, and its name will be returned.

wt_undir

A working table holding an undirected version of the graph.

wt_cluas

A working table holding current cluster assignments.

wt_newas

A working table holding updated cluster assignments.

wt_delta

A working table holding changes ("delta") in cluster assignments.

dop

Degree of parallelism for the operation. The default is 4.

rounds

Maximum number of iterations to perform in searching for connected components. The default value of 0 (zero) means that computation will continue until all connected components are found.

tbs

Name of the tablespace to hold the data stored in working tables.

options

Additional settings for the operation.

- 'PDML=T' enables parallel DML.

Usage Notes

The property graph edge table must exist in the database, and the [OPG_APIS.FIND_CLUSTERS_PREP](#) procedure must already have been executed.

Examples

The following example finds the connected components in a property graph named `mypg`.

```
DECLARE
  wtClusters  varchar2(200) := 'mypg_clusters';
  wtUnDir     varchar2(200);
  wtCluas     varchar2(200);
  wtNewas     varchar2(200);
  wtDelta     varchar2(200);
BEGIN
  opg_apis.find_clusters_prep('mypgGE$', wtClusters, wtUnDir,
    wtCluas, wtNewas, wtDelta, '');
  dbms_output.put_line('working tables names ' || wtClusters || ' '
    || wtUnDir || ' ' || wtCluas || ' ' || wtNewas || ' '
    || wtDelta );

  opg_apis.find_cc_mapping_based('mypgGE$', wtClusters, wtUnDir,
    wtCluas, wtNewas, wtDelta, 8, 0, 'MYTBS', 'PDML=T');

  --
  -- logic to consume results in wtClusters
  -- e.g.:
  -- select /*+ parallel(8) */ count(distinct cluster_id)
  --   from mypg_clusters;

  -- cleanup all the working tables
  opg_apis.find_clusters_cleanup('mypgGE$', wtClusters, wtUnDir,
    wtCluas, wtNewas, wtDelta, '');

END;
/
```

9.27 OPG_APIS.FIND_CLUSTERS_CLEANUP

Format

```
OPG_APIS.FIND_CLUSTERS_CLEANUP(
  edge_tab_name  IN VARCHAR2,
  wt_clusters    IN OUT VARCHAR2,
  wt_undir       IN OUT VARCHAR2,
```

```

wt_cluas      IN OUT VARCHAR2,
wt_newas     IN OUT VARCHAR2,
wt_delta     IN OUT VARCHAR2,
options      IN VARCHAR2 DEFAULT NULL);

```

Description

Cleans up after running weakly connected components (WCC) cluster detection.

Parameters**edge_tab_name**

Name of the property graph edge table.

wt_clusters

A working table holding the final vertex cluster mappings. This table has two columns (VID NUMBER, CLUSTER_ID NUMBER). Column VID stores the vertex ID values, and column CLUSTER_ID stores the corresponding cluster ID values. Cluster ID values are long integers that can have gaps between them. If an empty name is specified, a new table will be generated, and its name will be returned.

wt_undir

A working table holding an undirected version of the graph.

wt_cluas

A working table holding current cluster assignments.

wt_newas

A working table holding updated cluster assignments.

wt_delta

A working table holding changes ("delta") in cluster assignments.

options

(Reserved for future use.)

Usage Notes

The property graph edge table must exist in the database.

Examples

The following example cleans up after performing doing cluster detection in a property graph named mypg.

```

EXECUTE OPG_APIS.FIND_CLUSTERS_CLEANUP('mypgGE$', wtClusters, wtUnDir, wtCluas,
wtNewas, wtDelta, null);

```

9.28 OPG_APIS.FIND_CLUSTERS_PREP

Format

```

OPG_APIS.FIND_CLUSTERS_PREP(
  edge_tab_name  IN VARCHAR2,
  wt_clusters    IN OUT VARCHAR2,
  wt_undir       IN OUT VARCHAR2,
  wt_cluas       IN OUT VARCHAR2,

```



```

wt_newas      IN OUT VARCHAR2,
wt_delta      IN OUT VARCHAR2,
options       IN VARCHAR2 DEFAULT NULL);

```

Description

Prepares for running weakly connected components (WCC) cluster detection.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_clusters

A working table holding the final vertex cluster mappings. This table has two columns (VID NUMBER, CLUSTER_ID NUMBER). Column VID stores the vertex ID values, and column CLUSTER_ID stores the corresponding cluster ID values. Cluster ID values are long integers that can have gaps between them.

If an empty name is specified, a new table will be generated, and its name will be returned.

wt_undir

A working table holding an undirected version of the graph.

wt_cluas

A working table holding current cluster assignments.

wt_newas

A working table holding updated cluster assignments.

wt_delta

A working table holding changes ("delta") in cluster assignments.

options

Additional settings for index creation.

Usage Notes

The property graph edge table must exist in the database.

Examples

The following example prepares for doing cluster detection in a property graph named mypg.

```

DECLARE
  wtClusters  varchar2(200);
  wtUnDir     varchar2(200);
  wtCluas     varchar2(200);
  wtNewas     varchar2(200);
  wtDelta     varchar2(200);
BEGIN
  opg_apis.find_clusters_prep('mypgGE$', wtClusters, wtUnDir,
    wtCluas, wtNewas, wtDelta, '');
  dbms_output.put_line('working tables names ' || wtClusters || ' '
    || wtUnDir || ' ' || wtCluas || ' ' || wtNewas || ' '
    || wtDelta );
END;
/

```

9.29 OPG_APIS.FIND_SP

Format

```
OPG_APIS.FIND_SP(  
    edge_tab_name  IN VARCHAR2,  
    source         IN NUMBER,  
    dest          IN NUMBER,  
    exp_tab       IN OUT VARCHAR2,  
    dop           IN INTEGER,  
    stats_freq    IN INTEGER DEFAULT 20000,  
    path_output   OUT VARCHAR2,  
    weights_output OUT VARCHAR2,  
    edge_tab_name IN VARCHAR2,  
    options       IN VARCHAR2 DEFAULT NULL,  
    scn          IN NUMBER DEFAULT NULL);
```

Description

Finds the shortest path between given source vertex and destination vertex in the property graph. It assumes each edge has a numeric weight property. (The actual edge property name is not significant.)

Parameters

edge_tab_name

Name of the property graph edge table.

source

Source (start) vertex ID.

dest

Destination (end) vertex ID.

exp_tab

Name of the expansion table to be used for shortest path calculations.

dop

Degree of parallelism for the operation.

stats_freq

Frequency for collecting statistics on the table.

path_output

The output shortest path. It consists of IDs of vertices on the shortest path, which are separated by the space character.

weights_output

The output shortest path weights. It consists of weights of edges on the shortest path, which are separated by the space character.

options

Additional settings for the operation. An optional string with one or more (comma-separated) of the following values:

- CREATE_UNDIRECTED=T
- REUSE_UNDIRECTED_TAB=T

scn

SCN for the edge table. It can be null.

Usage Notes

The property graph edge table must exist in the database, and the [OPG_APIS.FIND_SP_PREP](#) procedure must have already been called.

Examples

The following example prepares for shortest-path calculation, and then finds the shortest path from vertex 1 to vertex 35 in a property graph named `mypg`.

```
set serveroutput on
DECLARE
  w      varchar2(2000);
  wtExp  varchar2(2000);
  vPath  varchar2(2000);
BEGIN
  opg_apis.find_sp_prep('mypgGE$', wtExp, null);
  opg_apis.find_sp('mypgGE$', 1, 35, wtExp, 1, 200000, vPath, w, null, null);
  dbms_output.put_line('Shortest path ' || vPath);
  dbms_output.put_line('Path weights ' || w);
END;
/
```

The output will be similar to the following. It shows one shortest path starting from vertex 1, to vertex 2, and finally to the destination vertex (35).

```
Shortest path 1      2 35
Path weights 3 2    1 1
```

9.30 OPG_APIS.FIND_SP_CLEANUP

Format

```
OPG_APIS.FIND_SP_CLEANUP(
  edge_tab_name  IN VARCHAR2,
  exp_tab        IN OUT VARCHAR2,
  options        IN VARCHAR2 DEFAULT NULL);
```

Description

Cleans up after running one or more shortest path calculations.

Parameters**edge_tab_name**

Name of the property graph edge table.

exp_tab

Name of the expansion table used for shortest path calculations.

options

(Reserved for future use.)

Usage Notes

There is no need to call this procedure after each `OPG_APIS.FIND_SP` call. You can run multiple shortest path calculations before calling `OPG_APIS.FIND_SP_CLEANUP`.

Examples

The following example does cleanup work after doing shortest path calculations in a property graph named `mypg`.

```
EXECUTE OPG_APIS.FIND_SP_CLEANUP('mypgGE$', wtExpTab, null);
```

9.31 OPG_APIS.FIND_SP_PREP

Format

```
OPG_APIS.FIND_SP_PREP(  
    edge_tab_name  IN VARCHAR2,  
    exp_tab        IN OUT VARCHAR2,  
    options        IN VARCHAR2 DEFAULT NULL);
```

Description

Prepares for shortest path calculations.

Parameters**edge_tab_name**

Name of the property graph edge table.

exp_tab

Name of the expansion table to be used for shortest path calculations. If it is empty, an intermediate working table will be created and the table name will be returned in `exp_tab`.

options

Additional settings for the operation. An optional string with one or more (comma-separated) of the following values:

- `CREATE_UNDIRECTED=T`
- `REUSE_UNDIRECTED_TAB=T`

Usage Notes

The property graph edge table must exist in the database.

Examples

The following example does preparation work before doing shortest path calculations in a property graph named `mypg`

```
set serveroutput on  
DECLARE  
    wtExp varchar2(2000); -- name of working table for shortest path calculation  
BEGIN
```

```
    opg_apis.find_sp_prep('mypgGE$', wtExp, null);
    dbms_output.put_line('Working table name ' || wtExp);
END;
/
```

The output will be similar to the following. (Your output may be different depending on the SQL session ID.)

```
Working table name "MYPGGE$$TWFS277"
```

9.32 OPG_APIS.GET_BUILD_ID

Format

```
OPG_APIS.GET_BUILD_ID() RETURN VARCHAR2;
```

Description

Returns the current build ID of the Oracle Spatial and Graph property graph support, in YYYYMMDD format.

Parameters

(None.)

Usage Notes

(None.)

Examples

The following example returns the current build ID of the Oracle Spatial and Graph property graph support.

```
SQL> SELECT OPG_APIS.GET_BUILD_ID() FROM DUAL;
```

```
OPG_APIS.GET_BUILD_ID()
```

```
-----  
20160606
```

9.33 OPG_APIS.GET_GEOMETRY_FROM_V_COL

Format

```
OPG_APIS.GET_GEOMETRY_FROM_V_COL(  
    v          IN NVARCHAR2,  
    srid      IN NUMBER DEFAULT 8307  
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form.

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following examples show point, line, and polygon geometries.

```
SQL> select opg_apis.get_geometry_from_v_col('10.0 5.0',8307) from dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_COL('10.05.0',8307)(SDO_GTYPE, SDO_SRID,
SDO_POINT(
```

```
-----
-----
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5, NULL), NULL, NULL)
```

```
SQL> select opg_apis.get_geometry_from_v_col('LINESTRING(30 10, 10
30, 40 40)',8307) from dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_COL('LINESTRING(3010,1030,4040)',8307)
(SDO_GTYPE, S
```

```
-----
-----
SDO_GEOMETRY(2002, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
SDO_ORDINATE_ARRAY(
30, 10, 10, 30, 40, 40))
```

```
SQL> select opg_apis.get_geometry_from_v_col('POLYGON((-83.6 34.1,
-83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))', 8307) from dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_COL('POLYGON((-83.634.1,-83.634.3,-83.434.3
,-83.434
```

```
-----
-----
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARR
AY(-83.6, 34.1, -83.6, 34.3, -83.4, 34.3, -83.4, 34.1, -83.6, 34.1))
```

9.34 OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS

Format

```
OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS(  
    v      IN NVARCHAR2,  
    t      IN INTEGER,  
    srid   IN NUMBER DEFAULT 8307  
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data, a type value, and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form,

t

Value indicating the type of value represented by the v parameter. Must be 20. (A null value or any other value besides 20 returns a null SDO_GEOMETRY object.)

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following examples show point, line, and polygon geometries.

```
SQL> select opg_apis.get_geometry_from_v_t_cols('10.0 5.0', 20, 8307) from  
dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS('10.05.0',20,8307)(SDO_GTYPE, SDO_SRID,  
SDO_  
-----  
---  
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5, NULL), NULL, NULL)
```

```
SQL> select opg_apis.get_geometry_from_v_t_cols('LINESTRING(30 10, 10 30, 40  
40)', 20, 8307) from dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS('LINESTRING(3010,1030,4040)',20,8307)  
(SDO_GT  
-----  
---
```

```
SDO_GEOMETRY(2002, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
SDO_ORDINATE_ARRAY(
30, 10, 10, 30, 40, 40))
```

```
SQL> select opg_apis.get_geometry_from_v_t_cols('POLYGON((-83.6 34.1,
-83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))', 20, 8307) from dual;
```

```
OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS('POLYGON((-83.634.1,-83.634.3,-83.43
4.3,-83.
```

```
-----
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARR
AY(-83.6, 34.1, -83.6, 34.3, -83.4, 34.3, -83.4, 34.1, -83.6, 34.1))
```

9.35 OPG_APIS.GET_LATLONG_FROM_V_COL

Format

```
OPG_APIS.GET_LATLONG_FROM_V_COL(
    v          IN NVARCHAR2,
    srid      IN NUMBER DEFAULT 8307
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form.

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

This function assumes that for each vertex in the geometry in the v parameter, the **first** number is the **latitude** value and the second number is the longitude value. (This is the reverse of the order in an SDO_GEOMETRY object definition, where longitude is first and latitude is second).

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following example returns a point SDO_GEOMETRY object. Notice that the coordinate values of the input point are “swapped” in the returned SDO_GEOMETRY object.

```
SQL> select opg_apis.get_latlong_from_v_col('5.1 10.0', 8307) from dual;

OPG_APIS.GET_LATLONG_FROM_V_COL('5.110.0',8307)(SDO_GTYPE, SDO_SRID,
SDO_POINT(X
-----
---
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5.1, NULL), NULL, NULL)
```

9.36 OPG_APIS.GET_LATLONG_FROM_V_T_COLS

Format

```
OPG_APIS.GET_LATLONG_FROM_V_T_COLS(
    v      IN NVARCHAR2,
    t      IN INTEGER,
    srid   IN NUMBER DEFAULT 8307
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data, a type value, and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form.

t

Value indicating the type of value represented by the v parameter. Must be 20. (A null value or any other value besides 20 returns a null SDO_GEOMETRY object.)

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

This function assumes that for each vertex in the geometry in the v parameter, the **first** number is the **latitude** value and the second number is the longitude value. (This is the reverse of the order in an SDO_GEOMETRY object definition, where longitude is first and latitude is second).

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following example returns a point SDO_GEOMETRY object. Notice that the coordinate values of the input point are “swapped” in the returned SDO_GEOMETRY object.

```
SQL> select opg_apis.get_latlong_from_v_t_cols('5.1 10.0',20,8307) from
dual;
```

```
OPG_APIS.GET_LATLONG_FROM_V_T_COLS('5.110.0',20,8307)(SDO_GTYPE,
SDO_SRID, SDO_P
```

```
-----
-----
```

```
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5.1, NULL), NULL, NULL)
```

9.37 OPG_APIS.GET_LONG_LAT_GEOMETRY

Format

```
OPG_APIS.GET_LONG_LAT_GEOMETRY(
    x      IN NUMBER,
    y      IN NUMBER,
    srid   IN NUMBER DEFAULT 8307
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using X and Y point coordinate values, and optionally an SRID value.

Parameters

x

The X (first coordinate) value in the SDO_POINT_TYPE element of the geometry definition.

y

The Y (second coordinate) value in the SDO_POINT_TYPE element of the geometry definition.

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following example returns the geometry object for a point with X, Y coordinates 10.5, 5.0, and it uses 8307 as the SRID in the resulting geometry object.

```
SQL> select opg_apis.get_long_lat_geometry(10.0, 5.0, 8307) from dual;
```

```
OPG_APIS.GET_LONG_LAT_GEOMETRY(10.0,5.0,8307)(SDO_GTYPE, SDO_SRID,  
SDO_POINT(X,  
-----  
---  
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5, NULL), NULL, NULL)
```

9.38 OPG_APIS.GET_LATLONG_FROM_V_COL

Format

```
OPG_APIS.GET_LATLONG_FROM_V_COL(  
    v          IN NVARCHAR2,  
    srid       IN NUMBER DEFAULT 8307  
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form.

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

This function assumes that for each vertex in the geometry in the v parameter, the **first** number is the **latitude** value and the second number is the longitude value. (This is the reverse of the order in an SDO_GEOMETRY object definition, where longitude is first and latitude is second).

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following example returns a point SDO_GEOMETRY object. Notice that the coordinate values of the input point are “swapped” in the returned SDO_GEOMETRY object.

```
SQL> select opg_apis.get_latlong_from_v_col('5.1 10.0', 8307) from dual;
```

```
OPG_APIS.GET_LATLONG_FROM_V_COL('5.110.0',8307)(SDO_GTYPE, SDO_SRID,
```

```
SDO_POINT(X
-----
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5.1, NULL), NULL, NULL)
```

9.39 OPG_APIS.GET_LONGLAT_FROM_V_T_COLS

Format

```
OPG_APIS.GET_LONGLAT_FROM_V_T_COLS(
  v      IN NVARCHAR2,
  t      IN INTEGER,
  srid   IN NUMBER DEFAULT 8307
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object constructed using spatial data, a type value, and optionally an SRID value.

Parameters

v

A String containing spatial data in serialized form.

t

Value indicating the type of value represented by the v parameter. Must be 20. (A null value or any other value besides 20 returns a null SDO_GEOMETRY object.)

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

This function assumes that for each vertex in the geometry in the v parameter, the first number is the longitude value and the second number is the latitude value (which is the order in an SDO_GEOMETRY object definition).

The following example returns a point SDO_GEOMETRY object.

```
SQL> select opg_apis.get_longlat_from_v_t_cols('5.1 10.0',20,8307) from
dual;
```

```
OPG_APIS.GET_LATLONG_FROM_V_T_COLS('5.110.0',20,8307)(SDO_GTYPE,
SDO_SRID, SDO_P
```

```
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(5.1, 10, NULL), NULL, NULL)
```

9.40 OPG_APIS.GET_SCN

Format

```
OPG_APIS.GET_SCN() RETURN NUMBER;
```

Description

Returns the SCN (system change number) of the Oracle Spatial and Graph property graph support, in YYYYMMDD format.

Note:

Effective with Release 20.3, the OPG_APIS.GET_SCN function is **deprecated**. Instead, to retrieve the current SCN (system change number), use the DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER function:

```
SELECT dbms_flashback.get_system_change_number FROM DUAL;
```

Parameters

(None.)

Usage Notes

The SCN value is incremented after each commit.

Examples

The following example returns the current build ID of the Oracle Spatial and Graph property graph support.

```
SQL> SELECT OPG_APIS.GET_SCN() FROM DUAL;
```

```
OPG_APIS.GET_SCN()  
-----  
1478701
```

9.41 OPG_APIS.GET_VERSION

Format

```
OPG_APIS.GET_VERSION() RETURN VARCHAR2;
```

Description

Returns the current version of the Oracle Spatial and Graph property graph support.

Parameters

(None.)

Usage Notes

(None.)

Examples

The following example returns the current version of the Oracle Spatial and Graph property graph support.

```
SQL> SELECT OPG_APIS.GET_VERSION() FROM DUAL;
```

```
OPG_APIS.GET_VERSION()
```

```
-----  
12.2.0.1 P1
```

9.42 OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL

Format

```
OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL(  
    v      IN NVARCHAR2,  
    srid   IN NUMBER DEFAULT NULL  
) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object based on a geometry in WKT (well known text) form and optionally an SRID.

Parameters**v**

A String containing spatial data in serialized form.

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following statements return a point geometry and a line string geometry

```
SQL> select opg_apis.get_wktgeometry_from_v_col('POINT(10.0 5.1)',  
8307) from dual;
```

```
OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL('POINT(10.05.1)',8307)(SDO_GTYPE,  
SDO_SRID,
```

```

-----
---
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5.1, NULL), NULL, NULL)

SQL> select opg_apis.get_wktgeometry_from_v_col('LINESTRING(30 10, 10 30, 40
40)',8307) from dual;

OPG_APIS.GET_WKTGEOMETRY_FROM_V_COL('LINESTRING(3010,1030,4040)',8307)
(SDO_GTYPE
-----
---
SDO_GEOMETRY(2002, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
SDO_ORDINATE_ARRAY(
30, 10, 10, 30, 40, 40))

```

9.43 OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS

Format

```

OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS(
    v      IN NVARCHAR2,
    t      IN INTEGER,
    srid   IN NUMBER DEFAULT NULL
) RETURN SDO_GEOMETRY;

```

Description

Returns an SDO_GEOMETRY object based on a geometry in WKT (well known text) form, a type value, and optionally an SRID.

Parameters

v

A String containing spatial data in serialized form.

t

Value indicating the type of value represented by the v parameter. Must be 20. (A null value or any other value besides 20 returns a null SDO_GEOMETRY object.)

srid

SRID (coordinate system identifier) to be used in the resulting SDO_GEOMETRY object. The default value is 8307, the Oracle Spatial SRID for the WGS 84 longitude/latitude coordinate system.

Usage Notes

If there is incorrect syntax or a parsing error, this function returns NULL instead of generating an exception.

Examples

The following statements return a point geometry and a polygon geometry

```

SQL> select opg_apis.get_wktgeometry_from_v_t_cols('POINT(10.0
5.1)',20,8307) from dual;

```

```

OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS('POINT(10.05.1)',20,8307)
(SDO_GTYPE, SDO_
-----
-----
SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(10, 5.1, NULL), NULL, NULL)

SQL> select opg_apis.get_wktgeometry_from_v_t_cols('POLYGON((-83.6
34.1, -83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))',20,8307) from
dual;

OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS('POLYGON((-83.634.1,-83.634.3,-83
.434.3,-
-----
-----
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARR
AY(-83.6, 34.1, -83.6, 34.3, -83.4, 34.3, -83.4, 34.1, -83.6, 34.1))

```

9.44 OPG_APIS.GRANT_ACCESS

Format

```

OPG_APIS.GRANT_ACCESS(
    graph_owner IN VARCHAR2,
    graph_name  IN VARCHAR2,
    other_user  IN VARCHAR2,
    privilege   IN VARCHAR2);

```

Description

Grants access privileges on a property graph to another database user.

Parameters

graph_owner

Owner of the property graph.

graph_name

Name of the property graph.

other_user

Name of the database user to which one or more access privileges will be granted.

privilege

A string of characters indicating privileges: R for read, S for select, U for update, D for delete, I for insert, A for all. Do not use commas or any other delimiter.

If you specify A, do not specify any other values because A includes all access privileges.

Usage Notes

(None.)

Examples

The following example grants read and select (RS) privileges on the `mypg` property graph owned by database user SCOTT to database user PGUSR. It then connects as PGUSR and queries the `mypg` vertex table in the SCOTT schema.

```
CONNECT scott/<password>

EXECUTE OPG_APIS.GRANT_ACCESS('scott', 'mypg', 'pgusr', 'RS');

CONNECT pgusr/<password>

SELECT count(1) from scott.mypgVT$;
```

17

9.45 OPG_APIS.IMP_EDGE_TAB_STATS

Format

```
OPG_APIS.IMP_EDGE_TAB_STATS(
  graph_name    IN VARCHAR2,
  stattab       IN VARCHAR2,
  statid        IN VARCHAR2 DEFAULT NULL,
  cascade       IN BOOLEAN DEFAULT TRUE,
  statown       IN VARCHAR2 DEFAULT NULL,
  no_invalidate IN BOOLEAN DEFAULT FALSE,
  force         IN BOOLEAN DEFAULT FALSE,
  stat_category IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Retrieves statistics for the given property graph edge table (GE\$) from the user statistics table identified by `stattab` and stores them in the dictionary. If `cascade` is `TRUE`, all index statistics associated with the specified table are also imported.

Parameters

graph_name

Name of the property graph.

stattab

Name of the statistics table.

statid

Optional identifier to associate with these statistics within `stattab`.

cascade

If `TRUE`, column and index statistics are exported.

statown

Schema containing `stattab`.

no_invalidate

If `TRUE`, does not invalidate the dependent cursors. If `FALSE`, invalidates the dependent cursors immediately. If `DBMS_STATS.AUTO_INVALIDATE` (the usual default) is in effect, Oracle Database decides when to invalidate dependent cursors.

force

If `TRUE`, performs the operation even if the statistics are locked.

stat_category

Specifies what statistics to export, using a comma to separate values. The supported values are `'OBJECT_STATS'` (the default: table statistics, column statistics, and index statistics) and `'SYNOPSIS'` (auxiliary statistics created when statistics are incrementally maintained).

Usage Notes

(None.)

Examples

The following example creates a statistics table, exports into this table the edge table statistics, issues a query to count the relevant rows for the newly created statistics, and finally imports the statistics back.

```
EXECUTE OPG_APIS.CREATE_STAT_TABLE('mystat',null);

EXECUTE OPG_APIS.EXP_EDGE_TAB_STATS('mypg', 'mystat', 'edge_stats_id_1', true,
null, 'OBJECT_STATS');

SELECT count(1) FROM mystat WHERE statid='EDGE_STATS_ID_1';

      153

EXECUTE OPG_APIS.IMP_EDGE_TAB_STATS('mypg', 'mystat', 'edge_stats_id_1', true,
null, false, true, 'OBJECT_STATS');
```

9.46 OPG_APIS.IMP_VERTEX_TAB_STATS

Format

```
OPG_APIS.IMP_VERTEX_TAB_STATS(
    graph_name      IN VARCHAR2,
    stattab         IN VARCHAR2,
    statid          IN VARCHAR2 DEFAULT NULL,
    cascade         IN BOOLEAN DEFAULT TRUE,
    statown         IN VARCHAR2 DEFAULT NULL,
    no_invalidate   BOOLEAN DEFAULT FALSE,
    force           BOOLEAN DEFAULT FALSE,
    stat_category   IN VARCHAR2 DEFAULT 'OBJECT_STATS');
```

Description

Retrieves statistics for the given property graph vertex table (VT\$) from the user statistics table identified by `stattab` and stores them in the dictionary. If `cascade` is `TRUE`, all index statistics associated with the specified table are also imported.

Parameters

graph_name

Name of the property graph.

stattab

Name of the statistics table.

statid

Optional identifier to associate with these statistics within `stattab`.

cascade

If `TRUE`, column and index statistics are exported.

statown

Schema containing `stattab`.

no_invalidate

If `TRUE`, does not invalidate the dependent cursors. If `FALSE`, invalidates the dependent cursors immediately. If `DBMS_STATS.AUTO_INVALIDATE` (the usual default) is in effect, Oracle Database decides when to invalidate dependent cursors.

force

If `TRUE`, performs the operation even if the statistics are locked.

stat_category

Specifies what statistics to export, using a comma to separate values. The supported values are `'OBJECT_STATS'` (the default: table statistics, column statistics, and index statistics) and `'SYNOPSIS'` (auxiliary statistics created when statistics are incrementally maintained).

Usage Notes

(None.)

Examples

The following example creates a statistics table, exports into this table the vertex table statistics, issues a query to count the relevant rows for the newly created statistics, and finally imports the statistics back.

```
EXECUTE OPG_APIS.CREATE_STAT_TABLE('mystat',null);

EXECUTE OPG_APIS.EXP_VERTEX_TAB_STATS('mypg', 'mystat', 'vertex_stats_id_1', true,
null, 'OBJECT_STATS');

SELECT count(1) FROM mystat WHERE statid='VERTEX_STATS_ID_1';

108

EXECUTE OPG_APIS.IMP_VERTEX_TAB_STATS('mypg', 'mystat', 'vertex_stats_id_1', true,
null, false, true, 'OBJECT_STATS');
```

9.47 OPG_APIS.PR

Format

```
OPG_APIS.PR(  
    edge_tab_name    IN VARCHAR2,  
    d                IN NUMBER DEFAULT 0.85,  
    num_iterations   IN NUMBER DEFAULT 10,  
    convergence      IN NUMBER DEFAULT 0.1,  
    dop              IN INTEGER DEFAULT 4,  
    wt_node_pr       IN OUT VARCHAR2,  
    wt_node_nextpr   IN OUT VARCHAR2,  
    wt_edge_tab_deg  IN OUT VARCHAR2,  
    wt_delta         IN OUT VARCHAR2,  
    tablespace       IN VARCHAR2 DEFAULT NULL,  
    options          IN VARCHAR2 DEFAULT NULL,  
    num_vertices     OUT NUMBER);
```

Description

Prepares for page rank calculations.

Parameters

edge_tab_name

Name of the property graph edge table.

d

Damping factor.

num_iterations

Number of iterations for calculating the page rank values.

convergence

A threshold. If the difference between the page rank value of the current iteration and next iteration is lower than this threshold, then computation stops.

dop

Degree of parallelism for the operation.

wt_node_pr

Name of the working table to hold the page rank values of the vertices.

wt_node_pr

Name of the working table to hold the page rank values of the vertices.

wt_node_next_pr

Name of the working table to hold the page rank values of the vertices in the next iteration.

wt_edge_tab_deg

Name of the working table to hold edges and node degree information.

wt_delta

Name of the working table to hold information about some special vertices.

tablespace

Name of the tablespace to hold all the graph data and index data.

options

Additional settings for the operation. An optional string with one or more (comma-separated) of the following values:

- CREATE_UNDIRECTED=T
- REUSE_UNDIRECTED_TAB=T

num_vertices

Number of vertices processed by the page rank calculation.

Usage Notes

The property graph edge table must exist in the database, and the [OPG_APIS.PR_PREP](#) procedure must have been called.

Examples

The following example performs preparation, and then calculates the page rank value of vertices in a property graph named `mypg`.

```
set serveroutput on
DECLARE
    wt_pr  varchar2(2000); -- name of the table to hold PR value of the current
iteration
    wt_npr varchar2(2000); -- name of the table to hold PR value for the next iteration
    wt3    varchar2(2000);
    wt4    varchar2(2000);
    wt5    varchar2(2000);
    n_vertices number;
BEGIN
    wt_pr := 'mypgPR';
    opg_apis.pr_prep('mypgGE$', wt_pr, wt_npr, wt3, wt4, null);
    dbms_output.put_line('Working table names ' || wt_pr
        || ', wt_npr ' || wt_npr || ', wt3 ' || wt3 || ', wt4 ' || wt4);
    opg_apis.pr('mypgGE$', 0.85, 10, 0.01, 4, wt_pr, wt_npr, wt3, wt4, 'SYSAUX', null,
n_vertices)
;
END;
/
```

The output will be similar to the following.

```
Working table names "MYPGPR", wt_npr "MYPGGE$TWPRX277", wt3
"MYPGGE$TWPRE277", wt4 "MYPGGE$TWPRD277"
```

The calculated page rank value is stored in the `mypgpr` table which has the following definition and data.

```
SQL> desc mypgpr;
Name                               Null?    Type
-----
NODE                                NOT NULL NUMBER
PR                                    NUMBER
C                                    NUMBER

SQL> select node, pr from mypgpr;
```

NODE	PR
101	.1925
201	.2775
102	.1925
104	.74383125
105	.313625
103	.1925
100	.15
200	.15

9.48 OPG_APIS.PR_CLEANUP

Format

```
OPG_APIS.PR_CLEANUP(
    edge_tab_name  IN VARCHAR2,
    wt_node_pr     IN OUT VARCHAR2,
    wt_node_nextpr IN OUT VARCHAR2,
    wt_edge_tab_deg IN OUT VARCHAR2,
    wt_delta       IN OUT VARCHAR2,
    options        IN VARCHAR2 DEFAULT NULL);
```

Description

Performs cleanup after performing page rank calculations.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_node_pr

Name of the working table to hold the page rank values of the vertices.

wt_node_next_pr

Name of the working table to hold the page rank values of the vertices in the next iteration.

wt_edge_tab_deg

Name of the working table to hold edges and node degree information.

wt_delta

Name of the working table to hold information about some special vertices.

options

Additional settings for the operation. An optional string with one or more (comma-separated) of the following values:

- CREATE_UNDIRECTED=T
- REUSE_UNDIRECTED_TAB=T

Usage Notes

You do not need to do cleanup after each call to the [OPG_APIS.PR](#) procedure. You can run several page rank calculations before calling the [OPG_APIS.PR_CLEANUP](#) procedure.

Examples

The following example does the cleanup work after running page rank calculations in a property graph named `mypg`.

```
EXECUTE OPG_APIS.PR_CLEANUP('mypgGE$', wt_pr, wt_npr, wt3, wt4, null);
```

9.49 OPG_APIS.PR_PREP

Format

```
OPG_APIS.PR_PREP(  
    edge_tab_name    IN VARCHAR2,  
    wt_node_pr       IN OUT VARCHAR2,  
    wt_node_nextpr   IN OUT VARCHAR2,  
    wt_edge_tab_deg  IN OUT VARCHAR2,  
    wt_delta         IN OUT VARCHAR2,  
    options          IN VARCHAR2 DEFAULT NULL);
```

Description

Prepares for page rank calculations.

Parameters

edge_tab_name

Name of the property graph edge table.

wt_node_pr

Name of the working table to hold the page rank values of the vertices.

wt_node_next_pr

Name of the working table to hold the page rank values of the vertices in the next iteration.

wt_edge_tab_deg

Name of the working table to hold edges and node degree information.

wt_delta

Name of the working table to hold information about some special vertices.

options

Additional settings for the operation. An optional string with one or more (comma-separated) of the following values:

- `CREATE_UNDIRECTED=T`
- `REUSE_UNDIRECTED_TAB=T`

Usage Notes

The property graph edge table must exist in the database.

Examples

The following example does the preparation work before running page rank calculations in a property graph named `mypg`.

```

set serveroutput on
DECLARE
    wt_pr  varchar2(2000); -- name of the table to hold PR value of the current
iteration
    wt_npr varchar2(2000); -- name of the table to hold PR value for the next
iteration
    wt3    varchar2(2000);
    wt4    varchar2(2000);
    wt5    varchar2(2000);
BEGIN
    wt_pr := 'mypgPR';
    opg_apis.pr_prep('mypgGE$', wt_pr, wt_npr, wt3, wt4, null);
    dbms_output.put_line('Working table names ' || wt_pr
        || ', wt_npr ' || wt_npr || ', wt3 ' || wt3 || ', wt4 ' || wt4);
END;
/

```

The output will be similar to the following.

```

Working table names  "MYPGPR", wt_npr "MYPGGE$$TWPRX277", wt3
"MYPGGE$$TWPRE277", wt4 "MYPGGE$$TWPRD277"

```

9.50 OPG_APIS.PREPARE_TEXT_INDEX

Format

```
OPG_APIS.PREPARE_TEXT_INDEX();
```

Description

Performs preparatory work needed before a text index can be created on any NVARCHAR2 columns.

Parameters

None.

Usage Notes

You must have the ALTER SESSION to run this procedure.

Examples

The following example performs preparatory work needed before a text index can be created on any NVARCHAR2 columns.

```
EXECUTE OPG_APIS.PREPARE_TEXT_INDEX();
```

9.51 OPG_APIS.RENAME_PG

Format

```
OPG_APIS.RENAME_PG(
    graph_name      IN VARCHAR2,
    new_graph_name  IN VARCHAR2);
```


Description

Renames a property graph.

Parameters**graph_name**

Name of the property graph.

new_graph_name

New name for the property graph.

Usage Notes

The `graph_name` property graph must exist in the database.

Examples

The following example changes the name of a property graph named `mypg` to `mynewpg`.

```
EXECUTE OPG_APIS.RENAME_PG('mypg', 'mynewpg');
```

9.52 OPG_APIS.SPARSIFY_GRAPH

Format

```
OPG_APIS.SPARSIFY_GRAPH(
    edge_tab_name IN VARCHAR2,
    threshold      IN NUMBER DEFAULT 0.5,
    min_keep       IN INTEGER DEFAULT 1,
    dop            IN INTEGER DEFAULT 4,
    wt_out_tab     IN OUT VARCHAR2,
    wt_und_tab     IN OUT VARCHAR2,
    wt_hsh_tab     IN OUT VARCHAR2,
    wt_mch_tab     IN OUT VARCHAR2,
    tbs            IN VARCHAR2 DEFAULT NULL,
    options        IN VARCHAR2 DEFAULT NULL);
```

Description

Performs sparsification (edge trimming) for a property graph edge table.

Parameters**edge_tab_name**

Name of the property graph edge table (GE\$).

threshold

A numeric value controlling how much sparsification needs to be performed. The lower the value, the more edges will be removed. Some typical values are: 0.1, 0.2, ..., 0.5

min_keep

A positive integer indicating at least how many adjacent edges should be kept for each vertex. A recommended value is 1.

dop

Degree of parallelism for the operation.

wt_out_tab

A working table to hold the output, a sparsified graph.

wt_und_tab

A working table to hold the undirected version of the original graph.

wt_hsh_tab

A working table to hold the min hash values of the graph.

wt_mch_tab

A working table to hold matching count of min hash values.

tbs

A working table to hold the working table data.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

The CREATE TABLE privilege is required to call this procedure.

The sparsification algorithm used is a min hash based local sparsification. See "Local graph sparsification for scalable clustering", Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data: <https://cs.uwaterloo.ca/~tozsu/courses/CS848/W15/presentations/ElbagouryPresentation-2.pdf>

Sparsification only involves the topology of a graph. None of the properties (K/V) are relevant.

Examples

The following example does the preparation work for the edges table of `mypg`, prints out the working table names, and runs sparsification. The output, a sparsified graph, is stored in a table named `LEAN_PG`, which has two columns, `SVID` and `DVID`.

```
SQL> set serveroutput on
DECLARE
  my_lean_pg varchar2(100) := 'lean_pg'; -- output table
  wt2 varchar2(100);
  wt3 varchar2(100);
  wt4 varchar2(100);
BEGIN
  opg_apis.sparsify_graph_prep('mypgGE$', my_lean_pg, wt2, wt3, wt4, null);
  dbms_output.put_line('wt2 ' || wt2 || ', wt3 ' || wt3 || ', wt4 ' || wt4);

  opg_apis.sparsify_graph('mypgGE$', 0.5, 1, 4, my_lean_pg, wt2, wt3, wt4,
'SEMTS', null);
END;
/
```

```
wt2 "MYPGGE$$TWSPAU275", wt3 "MYPGGE$$TWSPAH275", wt4 "MYPGGE$$TWSPAM275"
```

```
SQL> describe lean_pg;
```

Name	Null?	Type
SVID		NUMBER
DVID		NUMBER

9.53 OPG_APIS.SPARSIFY_GRAPH_CLEANUP

Format

```
OPG_APIS.SPARSIFY_GRAPH_CLEANUP(
    edge_tab_name IN VARCHAR2,
    wt_out_tab    IN OUT VARCHAR2,
    wt_und_tab    IN OUT VARCHAR2,
    wt_hsh_tab    IN OUT VARCHAR2,
    wt_mch_tab    IN OUT VARCHAR2,
    options       IN VARCHAR2 DEFAULT NULL);
```

Description

Cleans up after sparsification (edge trimming) for a property graph edge table.

Parameters

edge_tab_name

Name of the property graph edge table (GE\$).

wt_out_tab

A working table to hold the output, a sparsified graph.

wt_und_tab

A working table to hold the undirected version of the original graph.

wt_hsh_tab

A working table to hold the min hash values of the graph.

wt_mch_tab

A working table to hold matching count of min hash values.

tbs

A working table to hold the working table data

options

(Reserved for future use.)

Usage Notes

The working tables will be dropped after the operation completes.

Examples

The following example does the preparation work for the edges table of `mypg`, prints out the working table names, runs sparsification, and then performs cleanup.

```
SQL> set serveroutput on
DECLARE
  my_lean_pg varchar2(100) := 'lean_pg';
  wt2 varchar2(100);
  wt3 varchar2(100);
  wt4 varchar2(100);
BEGIN
  opg_apis.sparsify_graph_prep('mypgGE$', my_lean_pg, wt2, wt3, wt4, null);
  dbms_output.put_line('wt2 ' || wt2 || ', wt3 ' || wt3 || ', wt4 ' || wt4);

  opg_apis.sparsify_graph('mypgGE$', 0.5, 1, 4, my_lean_pg, wt2, wt3, wt4,
'SEMTS', null);

  -- Add logic here to consume SVID, DVID in LEAN_PG table
  --

  -- cleanup
  opg_apis.sparsify_graph_cleanup('mypgGE$', my_lean_pg, wt2, wt3, wt4, null);
END;
/
```

9.54 OPG_APIS.SPARSIFY_GRAPH_PREP

Format

```
OPG_APIS.SPARSIFY_GRAPH_PREP(
  edge_tab_name IN VARCHAR2,
  wt_out_tab    IN OUT VARCHAR2,
  wt_und_tab    IN OUT VARCHAR2,
  wt_hsh_tab    IN OUT VARCHAR2,
  wt_mch_tab    IN OUT VARCHAR2,
  options       IN VARCHAR2 DEFAULT NULL);
```

Description

Prepares working table names that are necessary to run sparsification for a property graph edge table.

Parameters

edge_tab_name

Name of the property graph edge table (GE\$).

wt_out_tab

A working table to hold the output, a sparsified graph.

wt_und_tab

A working table to hold the undirected version of the original graph.

wt_hsh_tab

A working table to hold the min hash values of the graph.

wt_mch_tab

A working table to hold the matching count of min hash values.

options

Additional settings for operation. An optional string with one or more (comma-separated) of the following values:

- 'INMEMORY=T' is an option for creating the schema tables with an 'inmemory' clause.
- 'IMC_MC_B=T' creates the schema tables with an INMEMORY MEMCOMPRESS BASIC clause.

Usage Notes

The sparsification algorithm used is a min hash based local sparsification. See "Local graph sparsification for scalable clustering", Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data: <https://cs.uwaterloo.ca/~tozsu/courses/CS848/W15/presentations/ElbagouryPresentation-2.pdf>

Examples

The following example does the preparation work for the edges table of `mypg` and prints out the working table names.

```
set serveroutput on

DECLARE
  my_lean_pg varchar2(100) := 'lean_pg';
  wt2 varchar2(100);
  wt3 varchar2(100);
  wt4 varchar2(100);
BEGIN
  opg_apis.sparsify_graph_prep('mypgGE$', my_lean_pg, wt2, wt3, wt4, null);
  dbms_output.put_line('wt2 ' || wt2 || ', wt3 ' || wt3 || ', wt4 ' || wt4);
END;
/
```

The output may be similar to the following.

```
wt2 "MYPGGE$TWSPAU275", wt3 "MYPGGE$TWSPAH275", wt4 "MYPGGE$TWSPAM275"
```

10

OPG_GRAPHOP Package Subprograms

The OPG_GRAPHOP package contains subprograms for various operations on property graphs in an Oracle database.

To use the subprograms in this chapter, you must understand the conceptual and usage information in earlier chapters of this book.

This chapter provides reference information about the subprograms, in alphabetical order.

- [OPG_GRAPHOP.POPULATE_SKELETON_TAB](#)

10.1 OPG_GRAPHOP.POPULATE_SKELETON_TAB

Format

```
OPG_GRAPHOP.POPULATE_SKELETON_TAB(  
    graph      IN VARCHAR2,  
    dop        IN INTEGER DEFAULT 4,  
    tbs        IN VARCHAR2 DEFAULT NULL,  
    options    IN VARCHAR2 DEFAULT NULL);
```

Description

Populates the skeleton table (<graph-name>GT\$). By default, any existing content in the skeleton table is truncated (removed) before the table is populated.

Parameters

graph

Name of the property graph.

dop

Degree of parallelism for the operation.

tbs

Name of the tablespace to hold the index data for the skeleton table.

options

Options that can be used to customize the populating of the skeleton table. (One or more, comma separated.)

- 'KEEP_DATA=T' causes any existing table not to be removed before the table is populated. New rows are added after the existing ones.
- 'PDML=T' skips the default index creation.

Usage Notes

You must have the CREATE TABLE and CREATE INDEX privileges to call this procedure.

There is a unique index constraint on EID column of the skeleton table (GE\$). So if you specify the `KEEP_DATA=T` option and if the new data overlaps with existing one, then the unique key constraint will be violated, resulting in an error.

Examples

The following example populates the skeleton table of the property graph named `mypg`.

```
EXECUTE OPG_GRAPHOP.POPULATE_SKELETON_TAB('mypg',4, 'pgts', 'PDML=T');
```

Part II

In-Memory Graph Server (PGX) Advanced User Guide

Part II provides in-depth information on using the in-memory graph server (PGX) for advanced users.

Part II contains the following chapters:

- [Configuring the In-Memory Graph Server \(PGX\)](#)
This chapter explains the configuration options for the in-memory graph server (PGX) and the graph client.
- [Graphs Management](#)
You can load, publish, store and delete graphs
- [Namespaces and Sharing](#)
The in-memory graph server (PGX) supports separate namespaces that help you to organize your entities.
- [PGX Programming Guides](#)
You can avail all the PGX functionalities through asynchronous Java APIs. Each asynchronous method has a synchronous equivalent, which blocks the caller thread until the server produces a response.
- [Working with Files Using the Graph Server \(PGX\)](#)
This chapter describes in detail about working with different file formats to perform various actions like loading, storing or exporting a graph using the Graph Server (PGX).
- [Log Management in the Graph Server \(PGX\)](#)
The graph server (PGX) internally uses the SLF4J interface with Log4j as the default logger implementation.

11

Configuring the In-Memory Graph Server (PGX)

This chapter explains the configuration options for the in-memory graph server (PGX) and the graph client.

These options can be configured in `/etc/oracle/graph/pgx.conf` where the graph server is installed, or passed to the graph server programmatically.

- [Configuration Parameters for the Graph Server \(PGX\) Engine](#)
You can configure the graph server (PGX) engine and the PGX run-time library by assigning a single JSON file to the in-memory graph server (PGX) at startup.
- [Configuration Parameters for Connecting to the Graph Server \(PGX\)](#)
You can configure the graph server (PGX) to use the required options at startup.
- [Configuration Parameters for the Graph Client](#)
You can configure the PGX graph client. All the parameters are available as command-line options also.

11.1 Configuration Parameters for the Graph Server (PGX) Engine

You can configure the graph server (PGX) engine and the PGX run-time library by assigning a single JSON file to the in-memory graph server (PGX) at startup.

This file includes the parameters shown in the following table.

To specify the configuration file, see [Specifying the Configuration File to the In-Memory Graph Server \(PGX\)](#).

Note:

- Relative paths in parameter values are always resolved relative to the parent directory of the configuration file in which they are specified. For example, if the configuration file is `/pgx/conf/pgx.conf`, then the file path `graph-configs/my-graph.bin.json` inside that file would be resolved to `/pgx/conf/graph-configs/my-graph.bin.json`.
- The parameter default values are optimized to deliver the best performance across a wide set of algorithms. Depending on your workload, you may be able to improve performance further by experimenting with different strategies, sizes, and thresholds.

Table 11-1 Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
admin_request_cache_timeout	integer	After how many seconds admin request results get removed from the cache. Requests which are not done or not yet consumed are excluded from this timeout. Note: This is only relevant if PGX is deployed as a webapp.	60
allow_idle_timeout_overwrite	boolean	If true, sessions can overwrite the default idle timeout.	true
allow_override_scheduling_information	boolean	If true, allow all users to override scheduling information like task weight, task priority, and number of threads	true
allow_task_timeout_overwrite	boolean	If true, sessions can overwrite the default task timeout.	true
allow_user_auto_refresh	boolean	If true, users may enable auto refresh for graphs they load. If false, only graphs mentioned in <code>preload_graphs</code> can have auto refresh enabled.	false
allowed_remote_loading_locations	array of strings	Allow loading graphs into the PGX engine from remote locations (http, https, ftp, ftps, s3, hdfs). If empty, as by default, no remote location is allowed. If "*" is specified in the array, all remote locations are allowed. Only the value "*" is currently supported. Note that pre-loaded graphs are loaded from any location, regardless of the value of this setting.	[]
<div style="border-left: 2px solid #f0e68c; padding-left: 10px; margin-bottom: 10px;">  WARNING: This parameter reduces security and therefore use it only when needed. </div>			
basic_scheduler_config	object	Configuration parameters for the fork join pool backend.	null
bfs_iterate_que_task_size	integer	Task size for BFS iterate QUE phase.	128
bfs_threshold_parent_read_based	number	Threshold of BFS traversal level items to switch to parent-read-based visiting strategy.	0.05
bfs_threshold_read_based	integer	Threshold of BFS traversal level items to switch to read-based visiting strategy.	1024
bfs_threshold_single_threaded	integer	Until what number of BFS traversal level items vertices are visited single-threaded.	128
character_set	string	Standard character set to use throughout PGX. UTF-8 is the default. Note: Some formats may not be compatible.	utf-8

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
<code>cni_diff_factor_default</code>	integer	Default diff factor value used in the common neighbor iterator implementations.	8
<code>cni_small_default</code>	integer	Default value used in the common neighbor iterator implementations, to indicate below which threshold a subarray is considered small.	128
<code>cni_stop_recursion_default</code>	integer	Default value used in the common neighbor iterator implementations, to indicate the minimum size where the binary search approach is applied.	96
<code>dfs_threshold_large</code>	integer	Value that determines at which number of visited vertices the DFS implementation will switch to data structures that are optimized for larger numbers of vertices.	4096
<code>enable_csrf_token_checks</code>	boolean	If true, the PGX webapp will verify the Cross-Site Request Forgery (CSRF) token cookie and request parameters sent by the client exist and match. This is to prevent CSRF attacks.	true
<code>enable_gm_compiler</code>	boolean	If true, enable dynamic compilation of PGX Algorithm API (or Green-Marl code) during runtime.	true
<code>enable_shutdown_cleanup_hook</code>	boolean	If true, PGX will add a JVM shutdown hook that will automatically shutdown PGX at JVM shutdown. Notice: Having the shutdown hook deactivated and not explicitly shutting down PGX may result in pollution of your temp directory.	true
<code>enterprise_scheduler_config</code>	object	Configuration parameters for the enterprise scheduler.	null
<code>enterprise_scheduler_flags</code>	object	<i>[relevant for enterprise_scheduler]</i> Enterprise scheduler-specific settings.	null
<code>explicit_spin_locks</code>	boolean	true means spin explicitly in a loop until lock becomes available. false means using JDK locks which rely on the JVM to decide whether to context switch or spin. Setting this value to true usually results in better performance.	true
<code>file_locations</code>	array of objects	The file locations that can be used in the authorization-config.	[]
<code>graph_algorithm_language</code>	enum[GM, EGA, CY, GM, JAVA]	Front-end compiler to use.	gm

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
graph_validation_level	enum[low, high]	Level of validation performed on newly loaded or created graphs.	low
ignore_incompatible_backend_operations	boolean	If true, only log when encountering incompatible operations and configuration values in RTS or FJ pool. If false, throw exceptions.	false
in_place_update_consistency_model	enum[ALL, INCONSISTENCIES, CANCEL_TASKS]	Consistency model used when in-place updates occur. Only relevant if in-place updates are enabled. Currently updates are only applied in place if the updates are not structural (Only modifies properties). Two models are currently implemented, one only delays new tasks when an update occurs, the other also delays running tasks.	ALL
init_pgql_on_startup	boolean	If true PGQL is directly initialized on start-up of PGX. Otherwise, it is initialized during the first use of PGQL.	true
interval_to_poll_max	integer	Exponential backoff upper bound (in ms) to which -once reached, the job status polling interval is fixed	1000
java_home_dir	string	The path to Java's home directory. If set to <system-java-home-dir>, use the java.home system property.	null
large_array_threshold	integer	Threshold when the size of an array is too big to use a normal Java array. This depends on the used JVM. (Defaults to Integer.MAX_VALUE - 3)	214748364
max_active_sessions	integer	Maximum number of sessions allowed to be active at a time.	1024
max_distinct_strings_per_pool	integer	<i>[only relevant if string_pooling_strategy is indexed]</i> Number of distinct strings per property after which to stop pooling. If the limit is reached, an exception is thrown.	65536
max_http_client_request_size	long	Maximum size in bytes of any http request sent to the PGX server over the REST API. Setting it to -1 allows requests of any size.	10485760

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine


Parameter	Type	Description	Default
max_off_heap_size	integer	Maximum amount of off-heap memory (in megabytes) that PGX is allowed to allocate before an OutOfMemoryError will be thrown.	<available-physical-memory>
<div style="border: 1px solid #0070C0; padding: 10px; background-color: #E6F2FF;"> <p> Note:</p> <p>This limit is not guaranteed to never be exceeded, because of rounding and synchronization trade-offs. It only serves as threshold when PGX starts to reject new memory allocation requests.</p> </div>			
max_queue_size_per_session	integer	The maximum number of pending tasks allowed to be in the queue, per session. If a session reaches the maximum, new incoming requests of that session get rejected. A negative value means infinity or unlimited..	-1
max_snapshot_count	integer	Number of snapshots that may be loaded in the engine at the same time. New snapshots can be created via auto or forced update. If the number of snapshots of a graph reaches this threshold, no more auto-updates will be performed, and a forced update will result in an exception until one or more snapshots are removed from memory. A value of zero indicates to support an unlimited amount of snapshots.	0
memory_allocator	enum[basic_allocator, persistent_allocator]	The memory allocator to use.	basic_allocator
memory_cleanup_interval	integer	Memory cleanup interval in seconds.	600

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
min_array_compaction_threshold	number	Minimum value (<i>only relevant for graphs optimized for updates</i>) that can be used for the array_compaction_threshold value in graph configuration. If a graph configuration attempts to use a value lower than the one specified by min_array_compaction_threshold, it will use min_array_compaction_threshold instead.	0.2
min_fetch_interval_sec	integer	For delta-refresh (<i>only relevant if the graph format supports delta updates</i>), the lowest interval at which a graph source is queried for changes. You can tune this value to prevent PGX from hanging due to too frequent graph delta-refreshing.	2
min_update_interval_sec	integer	For auto-refresh, the lowest interval after which a new snapshot is created, either by reloading the entire graph or if the format supports delta-updates, out of the cached changes (<i>only relevant if the format supports delta updates</i>). You can tune this value to prevent PGX from hanging due to too frequent graph auto-refreshing.	2
ms_bfs_frontier_type_strategy	enum[auto_graph_row, show, sort, int]	The type strategy to use for MS-BFS frontiers.	auto_graph_row
num_spin_locks	integer	Number of spin locks each generated app will create at instantiation. Trade-off: a small number implies less memory consumption; a large number implies faster execution (if algorithm uses spin locks).	1024
parallelism	integer	Number of worker threads to be used in thread pool. Note: If the caller thread is part of another thread-pool, this value is ignored and the parallelism of the parent pool is used.	<number of CPUs>
pattern_matching_supernode_cache_threshold	integer	Minimum number of a node's neighbor to be a supernode. This is for the pattern matching engine.	1000
pgx_realm	object	Configuration parameters for the realm.	null
pooling_factor	number	<i>[only relevant if string_pooling_strategy is on_heap]</i> This value prevents the string pool to grow as big as the property size, which could render the pooling ineffective.	0.25
preload_graphs	array of object	List of graph configs to be registered at start-up. Each item includes path to a graph config, the name of the graph and whether it should be published.	[]

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
random_generator_strategy	enum	Method of generating random numbers in PGX.	nondeterministic
random_seed	long	<i>[relevant for deterministic random number generator only]</i> Seed for the deterministic random number generator used in pgx. The default is -24466691093057031.	-24466691093057031
release_memory_threshold	number	Threshold percentage (decimal fraction) of used memory after which the engine starts freeing unused graphs. Examples: A value of 0.0 means graphs get freed as soon as their reference count becomes zero. That is, all sessions which loaded that graph were destroyed/timed out. A value of 1.0 means graphs never get freed, and the engine will throw OutOfMemoryErrors as soon as a graph is needed which does not fit in memory anymore. A value of 0.7 means the engine keeps all graphs in memory as long as total memory consumption is below 70% of total available memory, even if there is currently no session using them. When consumption exceeds 70% and another graph needs to get loaded, unused graphs get freed until memory consumption is below 70% again.	0.85
revisit_threshold	integer	Maximum number of matched results from a node to be cached.	4096

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
scheduler	enum[basic_scheduler, enterprise_scheduler, low_latency_scheduler]	The scheduler to use. <ul style="list-style-type: none"> basic_scheduler: uses a scheduler with basic features enterprise_scheduler: uses a scheduler with advanced enterprise features for running multiple tasks concurrently and providing better performance low_latency_scheduler: uses a scheduler that privileges latency of tasks over throughput or fairness across multiple sessions. The low_latency_scheduler is only available in embedded mode. 	enterprise_scheduler
session_idle_timeout_secs	integer	Timeout of idling sessions in seconds. Zero (0) means infinity or no timeout	0
session_task_timeout_secs	integer	Timeout in seconds to interrupt long-running tasks submitted by sessions (algorithms, I/O tasks). Zero (0) means infinity or no timeout.	0
small_task_length	integer	Task length if the total amount of work is smaller than default task length (only relevant for task-stealing strategies).	128
strict_mode	boolean	If true, exceptions are thrown and logged with ERROR level whenever the engine encounters configuration problems, such as invalid keys, mismatches, and other potential errors. If false, the engine logs problems with ERROR/WARN level (depending on severity) and makes best guesses and uses sensible defaults instead of throwing exceptions.	true
string_pooling_strategy	enum[indexed, on_heap, none]	The string pooling strategy to use.	on_heap

Table 11-1 (Cont.) Configuration Parameters for the Graph Server (PGX) Engine

Parameter	Type	Description	Default
task_length	integer	Default task length (only relevant for task-stealing strategies). Should be between 100 and 10000. Trade-off: a small number implies more fine-grained tasks are generated, higher stealing throughput; a large number implies less memory consumption and GC activity.	4096
tmp_dir	string	Temporary directory to store compilation artifacts and other temporary data. If set to <system-tmp-dir>, uses the standard tmp directory of the underlying system (/tmp on Linux).	null
udf_config_directory	string	Directory path containing UDF config files.	null
use_index_for_reachability_queries	enum[auto, off]	Create index for reachability queries.	auto
use_memory_mapper_for_reading_pgb	boolean	If true, use memory mapped files for reading graphs in PGB format if possible; if false, always use a stream-based implementation.	true
use_memory_mapper_for_storing_pgb	boolean	If true, use memory mapped files for storing graphs in PGB format if possible; if false, always use a stream-based implementation.	true

Enterprise Scheduler Parameters

The following parameters are relevant only if the advanced scheduler is used. (They are ignored if the basic scheduler is used.)

- analysis_task_config**
 Configuration for analysis tasks. Type: object. Default: `prioritymediummax_threads<no-of-CPUs>weight<no-of-CPUs>`
- fast_analysis_task_config**
 Configuration for fast analysis tasks. Type: object. Default: `priorityhighmax_threads<no-of-CPUs>weight1`
- maxnum_concurrent_io_tasks**
 Maximum number of concurrent tasks. Type: integer. Default: 3
- num_io_threads_per_task**
 Configuration for fast analysis tasks. Type: object. Default: `<no-of-cpus>`

Basic Scheduler Parameters

The following parameters are relevant only if the basic scheduler is used. (They are ignored if the advanced scheduler is used.)

- num_workers_analysis**

Number of worker threads to use for analysis tasks. Type: integer. Default: <no-of-CPU>

- `num_workers_fast_track_analysis`

Number of worker threads to use for fast-track analysis tasks. Type: integer. Default: 1

- `num_workers_io`

Number of worker threads to use for I/O tasks (load/refresh/write from/to disk). This value will not affect file-based loaders, because they are always single-threaded. Database loaders will open a new connection for each I/O worker. Default: <no-of-CPU>

Example 11-1 Minimal In-Memory Graph Server (PGX) Configuration

The following example causes the in-memory graph server (PGX) to initialize its analysis thread pool with 32 workers. (Default values are used for all other parameters.)

```
{
  "enterprise_scheduler_config": {
    "analysis_task_config": {
      "max_threads": 32
    }
  }
}
```

Example 11-2 Two Pre-loaded Graphs

This example sets more fields and specifies two fixed graphs for loading into memory during the graph server (PGX) startup.

```
{
  "enterprise_scheduler_config": {
    "analysis_task_config": {
      "max_threads": 32
    },
    "fast_analysis_task_config": {
      "max_threads": 32
    }
  },
  "memory_cleanup_interval": 600,
  "max_active_sessions": 1,
  "release_memory_threshold": 0.2,
  "preload_graphs": [
    {
      "path": "graph-configs/my-graph.bin.json",
      "name": "my-graph"
    },
    {
      "path": "graph-configs/my-other-graph.adj.json",
      "name": "my-other-graph",
      "publish": false
    }
  ],
  "authorization": [{
    "pgx_role": "GRAPH_DEVELOPER",
    "pgx_permissions": [{
      "preloaded_graph": "my-graph",
      "grant": "read"
    }
  ]
}
```

```

    },
    {
      "preloaded_graph": "my-other-graph",
      "grant": "read"
    }
  ]
},
....
]
}

```

- [Configuration of the Graph Server \(PGX\) Run-Time Parameters](#)
- [Specifying the Configuration File to the In-Memory Graph Server \(PGX\)](#)
- [Memory Consumption by the Graph Server \(PGX\)](#)
The in-memory graph server (PGX) loads the graph into main memory in order to carry out analysis on the graph and its properties.

11.1.1 Configuration of the Graph Server (PGX) Run-Time Parameters

You can configure the following graph server (PGX) run-time fields.

Table 11-2 Graph Server (PGX) Run-Time Parameters

Parameter	Type	Description	Default
bfs_iterate_que_task_size	integer	Task size for BFS iterate QUE phase.	128
bfs_threshold_parent_read_base d	number	Threshold of BFS traversal level items above which to switch to parent-read-based visiting strategy.	0.05
bfs_threshold_read_based	integer	Threshold of BFS traversal level items above which to switch to read-based visiting strategy.	1024
bfs_threshold_single_threaded	integer	Number until which BFS traversal level items vertices are visited single-threaded.	128
character_set	string	Standard charset to use throughout PGX, UTF-8 will be used as default. Note: Some formats may not be compatible.	utf-8
cni_diff_factor_default	integer	Default diff factor value used in the common neighbor iterator implementations.	8
cni_small_default	integer	Default value used in the common neighbor iterator implementations, to indicate below which threshold a subarray is considered small.	128
cni_stop_recursion_default	integer	Default value used in the common neighbor iterator implementations, to indicate the minimum size where the binary search approach is applied.	96
dfs_threshold_large	integer	Value that determines at which number of visited vertices, the DFS implementation will switch to data-structures that are more optimized for larger numbers of vertices.	4096
enterprise_scheduler_flags	object	<i>[relevant for enterprise_scheduler]</i> Enterprise scheduler specific settings.	null

Table 11-2 (Cont.) Graph Server (PGX) Run-Time Parameters


Parameter	Type	Description	Default
<code>explicit_spin_locks</code>	boolean	<code>true</code> means spin explicitly in a loop until lock becomes available. <code>false</code> means using JDK locks which rely on the JVM to decide whether to context switch or spin. Our experiments showed that setting this value to <code>true</code> results in better performance.	<code>true</code>
<code>graph_validation_level</code>	enum[low, high]	Level of validation performed on newly loaded or created graphs.	low
<code>max_distinct_strings_per_pool</code>	integer	<i>[only relevant if <code>string_pooling_strategy</code> is <code>indexed</code>]</i> Amount of distinct strings per property after which to stop pooling. If the limit is reached an exception is thrown.	65536
<code>max_off_heap_size</code>	integer	Maximum amount of off-heap memory PGX is allowed to allocate in megabytes, before an <code>OutOfMemoryError</code> will be thrown.	<available-physical-memory>
<div style="border-left: 2px solid #0070C0; border-right: 2px solid #0070C0; border-bottom: 2px solid #0070C0; padding: 10px; background-color: #E6F2FF;"> <p> Note:</p> <p>This limit is not guaranteed to never be exceeded because of rounding and synchronization trade-offs. It only serves as threshold when PGX starts to reject new memory allocation requests.</p> </div>			
<code>memory_allocator</code>	enum[basic_allocator, enterprise_allocator]	Denotes which memory allocator to use.	basic_allocator
<code>ms_bfs_frontier_type_strategy</code>	enum[auto_grow, short, int]	The type strategy to use for MS-BFS frontiers.	auto_grow
<code>num_spin_locks</code>	integer	Number of spin locks each generated app will create at instantiation. Trade-off: small number implies less memory consumption. Big number implies faster execution (if algorithm uses spin locks).	1024
<code>pattern_matching_supernode_cache_threshold</code>	integer	Minimum number of a node's neighbor to be a supernode. This is for pattern matching engine.	1000

Table 11-2 (Cont.) Graph Server (PGX) Run-Time Parameters

Parameter	Type	Description	Default
pooling_factor	number	<i>[only relevant if string_pooling_strategy is on_heap]</i> This value prevents the string pool to grow as big as the property size which could render the pooling ineffective.	0.25
random_generator_strategy	enum[non_deterministic, deterministic]	Method of generating random numbers in PGX.	non_deterministic
random_seed	long	<i>[relevant for deterministic random number generator only]</i> Seed for the deterministic random number generator used in PGX. The default is -24466691093057031.	-244666 9109305 7031
revisit_threshold	integer	Maximum number of matched results from a node to be cached.	4096
scheduler	enum[basic_scheduler, enterprise_scheduler, low_latency_scheduler]	Denotes which scheduler to use. <ul style="list-style-type: none"> basic_scheduler: use scheduler with basic features. enterprise_scheduler: use scheduler with advanced, enterprise features for running multiple tasks concurrently and increased performance. low_latency_scheduler: use scheduler that privileges latency of tasks over throughput or fairness across multiple sessions. The low_latency_scheduler is only available in embedded mode 	enterprise_scheduler
small_task_length	integer	Task length, if total amount of work is small than default task length (only relevant for task-stealing strategies).	128
string_pooling_strategy	enum[indexed, on_heap, none]	Denotes which string pooling strategy to use.	on_heap
task_length	integer	Default task length (only relevant for task-stealing strategies). F/J pool documentation says this value should be between 100 and 10000. Trade-off: small number implies more fine-grained tasks are generated, higher stealing throughput. High number implies less memory consumption and GC activity.	4096
use_index_for_reachability_queries	enum[auto, off]	Create index for reachability queries.	auto
use_memory_mapper_for_reading_pgb	boolean	If true, use memory mapped files for reading graphs in PGB format if possible; false always use stream based implementation.	true
use_memory_mapper_for_storing_pgb	boolean	If true, use memory mapped files for storing in PGB format if possible; if false always use a stream based implementation.	true

11.1.2 Specifying the Configuration File to the In-Memory Graph Server (PGX)

The in-memory graph server configuration file is parsed by the in-memory graph server at startup-time whenever `ServerInstance#startEngine` (or any of its variants) is called. You can write the path to your configuration file to the in-memory graph server or specify it programmatically. This topic identifies several ways to specify the file

Programmatically

All configuration fields exist as Java enums. Example:

```
Map<PgxCfg.Field, Object> pgxCfg = new HashMap<>();
pgxCfg.put(PgxCfg.Field.MEMORY_CLEANUP_INTERVAL, 600);

ServerInstance instance = ...
instance.startEngine(pgxCfg);
```

All parameters not explicitly set will get default values.

Explicitly Using a File

Instead of a map, you can write the path to an in-memory graph server configuration JSON file. Example:

```
instance.startEngine("path/to/pgx.conf"); // file on local file system
instance.startEngine("classpath:/path/to/pgx.conf"); // file on current classpath
```

For all other protocols, you can write directly in the input stream to a JSON file.

Example:

```
InputStream is = ...
instance.startEngine(is);
```

Implicitly Using a File

If `startEngine()` is called without an argument, the in-memory graph server (PGX) looks for a configuration file at the following places, stopping when it finds the file:

- File path found in the Java system property `pgx_conf`. Example: `java -Dpgx_conf=conf/my.pgx.config.json ...`
- A file named `pgx.conf` in the root directory of the current classpath
- A file named `pgx.conf` in the root directory relative to the current `System.getProperty("user.dir")` directory

Note: Providing a configuration is optional. A default value for each field will be used if the field cannot be found in the given configuration file, or if no configuration file is provided.

Using the Shell in Embedded Mode

To change how the shell configures the embedded (local) in-memory graph server (PGX) instance, edit `$PGX_HOME/conf/pgx.conf`. Changes will be reflected the next time you invoke `$PGX_HOME/bin/pgx`.

You can also change the location of the configuration file as in the following example:

```
./bin/opg --pgx_conf path/to/my/other/pgx.conf
```

Setting System Properties

Any parameter can be set using Java system properties by writing `-Dpgx.<FIELD>=<VALUE>` arguments to the JVM that the in-memory graph server (PGX) is running on. Note that setting system properties will overwrite any other configuration. The following example sets the maximum off-heap size to 256 GB, regardless of what any other configuration says:

```
java -Dpgx.max_off_heap_size=256000 ...
```

Setting Environment Variables

Any parameter can also be set using environment variables by adding 'PGX_' to the environment variable for the JVM in which the in-memory graph server (PGX) is executed. Note that setting environment variables will overwrite any other configuration; but if a system property and an environment variable are set for the same parameter, the system property value is used. The following example sets the maximum off-heap size to 256 GB using an environment variable:

```
PGX_MAX_OFF_HEAP_SIZE=256000 java ...
```

11.1.3 Memory Consumption by the Graph Server (PGX)

The in-memory graph server (PGX) loads the graph into main memory in order to carry out analysis on the graph and its properties.

The memory consumed by the graph server for a graph is split between the memory to store the topology of the graph (the information to indicate what are the vertices and edges in the graph without their attached properties), and the memory for the properties attached to the vertices and edges. Internally, the graph server (PGX) stores the graph topology in compressed sparse row (CSR) format, a data structure which has minimal memory footprint while providing very fast read access.

- [Memory Management](#)

11.1.3.1 Memory Management

The in-memory graph server (PGX) requires both on-heap and off-heap memory to store graph data.

The allocation of memory for the graph data is as shown:

- Graph indexes and graph topology are stored off-heap.
- All primitive properties (integer, long, double, float, boolean, date, local_date, timestamp, time, point2d) are stored off-heap.
- String properties are stored on-heap.

Default Configuration of Memory Limits

You can configure both on-heap and off-heap memory limits. In case of the on-heap, if you don't explicitly set a maximum then it will default to the maximum on-heap size determined by Java Hotspot, which is based on various factors, including the total amount of physical memory available. In case of the off-heap, if you don't explicitly set a maximum then it will default to the total physical available memory on the machine.

- [Configuring On-Heap Limits](#)
- [Configuring Off-Heap Limits](#)

11.1.3.1.1 Configuring On-Heap Limits

You can configure on-heap limits using Java command-line options.

The available options are:

- `-Xmx`: to set the maximum on-heap size of the JVM.
- `-Xms`: to set the initial on-heap size of the JVM.
- `-XX:NewSize`: to set the initial size of the young generation
- `-XX:MaxNewSize`: to set the maximum size of the young generation

The following shows an example to configure the on-heap limits using `-XX:MaxNewSize` option in a Java application:

```
java -Xmx<size_mb>m -Xms<size_mb>m -XX:MaxNewSize=<size_mb>m -
XX:NewSize=<size_mb>m
```

If you are using a JShell client, then you can set the `JAVA_OPTS` environment variable before starting the shell. For example:

```
export JAVA_OPTS="-Xmx<size_gb>g -Xms<size_gb>g -
XX:MaxNewSize=<size_gb>g -XX:NewSize=<size_gb>g "
cd /opt/oracle/graph/
./bin/opg-jshell
```

11.1.3.1.2 Configuring Off-Heap Limits

You can specify the off-heap limit by setting the `max_off_heap_size` field in the graph server (PGX) configuration. See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information on the `max_off_heap_size` parameter.

WARNING:

The off-heap limit is not guaranteed to never be exceeded because of rounding and synchronization trade-offs.

The off-heap limit can be set using Java system properties using `-Dpgx.max_off_heap_size=<size_in_mb>` in the JVM argument.

You can also set the off-heap limit using an environment variable. The following example sets the maximum off-heap size to 256 GB using an environment variable:

```
PGX_MAX_OFF_HEAP_SIZE=256000 java ...
```


**Note:**

If both system property and environment variable are set for off-heap limit, then the system property value is used.

11.2 Configuration Parameters for Connecting to the Graph Server (PGX)

You can configure the graph server (PGX) to use the required options at startup.

See [Configuring the In-Memory Graph Server \(PGX\)](#)

11.3 Configuration Parameters for the Graph Client

You can configure the PGX graph client. All the parameters are available as command-line options also.

Table 11-3 Configuration Parameters for the Graph Client

Parameter	Type	Description	Default
access_token	string	The authentication token.	null
base_url	string	The base url in the format host [: port] [/path] of the PGX server REST endpoint. If the base_url is null, the default will be used which points to embedded PGX instance.	null
cctrace_out	string	<i>[relevant for enable_cctrace]</i> When cctrace is enabled, this option specifies a path to a file where cctrace should log to. If null it will use the default PGX logger on level TRACE. If it is the special value :stderr: it will log to stderr.	null
cctrace_print_stacktraces	boolean	<i>[relevant for enable_cctrace]</i> When cctrace is enabled, this flag prints the stacktrace for each request and result.	false
client_server_interaction_mode	enum[async_polling, blocking]	If async_polling the PGX client would poll the status of the future until it is completed. If blocking, the PGX client would send a request to directly get the value of the future and the server would block until the future result is ready.	async_polling
enable_cctrace	boolean	If true log every call to a Control or Core interface.	false
keystore	string	The path to the keystore to use for client connections. The keystore is used to authenticate this client at the PGX server if two-way SSL/TLS is enabled.	null
max_client_http_connections	integer	Maximum number of connections to open to the PGX server.	2
password	string	Keystore password only.	null

Table 11-3 (Cont.) Configuration Parameters for the Graph Client

Parameter	Type	Description	Default
prefetch_size	integer	Number of items to be prefetched in remote iterators.	2048
realm_client_config	object	Implementation dependent configuration options for the realm client.	null
remote_future_pending_retry_interval	integer	Number of milliseconds to wait before sending another request in case a GET request for a PgxRemoteFuture receives a 202 - Accepted response.	500
remote_future_timeout	integer	Time that a GET request for a PgxRemoteFuture will be alive, until it times out and tries again. Time in milliseconds, set it to zero for an infinite timeout. See HTTP Client SO_TIMEOUT for more details.	300000
tls_version	string	TLS version to be used by the client. For example, TLSv1.2.	tlsv1.2
truststore	string	Path to the truststore to use for client connections. The truststore is used to validate the server certificate if communicating over SSL/TLS.	null
upload_batch_size	integer	Number of items to be uploaded in a batch. This is used in Core#addAllToCollection() and Core#setProperty().	65536
username	string	Name of the user.	null

Example 11-3 Configure the Graph Client Using the Graph PGX Shell

This following is an example to configure the graph client:

```
cd /opt/oracle/graph
./bin/opg-jshell --base_url https://myhost:8080/pgx --username scott --
prefetch_size 1024 --upload_batch_size 5000 --remote_future_timeout 20000 --
pending_retry_interval 800
```

Example 11-4 Configure the Graph Client Using the Java API

The following is an example to configure the graph client programatically using the Pgx.getInstance methods:

```
public static ServerInstance getInstance(String baseUrl, String
username, String password, Integer prefetchSize,
Integer uploadBatchSize, Integer remoteFutureTimeout, Integer
remoteFuturePendingRetryInterval)
```

To specify key store and trust store for SSL connections use the standard JDK system properties:

```
System.setProperty("javax.net.ssl.trustStore", "<truststore>");  
System.setProperty("javax.net.ssl.keyStore", "<keystore>");  
System.setProperty("javax.net.ssl.keyStorePassword", "<password>");
```

12

Graphs Management

You can load, publish, store and delete graphs

- [Loading a Graph Into the Graph Server \(PGX\)](#)
- [Publishing a Graph](#)
- [Publishing a Preloaded Graph](#)
- [Deleting a Graph](#)

12.1 Loading a Graph Into the Graph Server (PGX)

Data from relational tables can be modeled as a property graph and loaded into the graph server.

The graph server (PGX) supports various data sources and data formats for loading graph data, including file system and database formats. See [Data Format Support Matrix](#) to get more information on the supported data formats.

In order to perform graph analysis with the graph server (PGX), you must first read a graph into PGX. See [Reading Graphs from Oracle Database into the Graph Server \(PGX\)](#) for details.

- [API for Loading Graphs into Memory](#)
- [Graph Configuration Options](#)
- [Preloading a Graph](#)
- [Data Loading Security Best Practices](#)
- [Data Format Support Matrix](#)
- [Immutability of Loaded Graphs](#)

12.1.1 API for Loading Graphs into Memory

The following methods in `PgxSession` can be used to load graphs into the graph server (PGX) memory:

Loading a graph using Java

```
PgxGraph readGraphWithProperties(String path)
PgxGraph readGraphWithProperties(String path, String newGraphName)
PgxGraph readGraphWithProperties(GraphConfig config)
PgxGraph readGraphWithProperties(GraphConfig config, String newGraphName)
PgxGraph readGraphWithProperties(GraphConfig config, boolean forceUpdateIfNotFresh)
PgxGraph readGraphWithProperties(GraphConfig config, boolean forceUpdateIfNotFresh,
String newGraphName)
PgxGraph readGraphWithProperties(GraphConfig config, long maxAge, TimeUnit
maxAgeTimeUnit)
PgxGraph readGraphWithProperties(GraphConfig config, long maxAge, TimeUnit
maxAgeTimeUnit, boolean blockIfFull, String newGraphName)
```

Loading a graph using Python

```
read_graph_with_properties(self, config, max_age=9223372036854775807,
max_age_time_unit='days',
                                block_if_full=False,
update_if_not_fresh=True, graph_name=None)
```

The first argument (`path` to a graph config file or a parsed `config` object) is the meta-data of the graph to be read. The meta-data includes the following information:

- Location of the graph data: file location and name, DB location and connection information and so on
- Format of the graph data: plain text formats, XML-based formats, Binary formats and so on
- Types and Names of the properties to be loaded

The `forceUpdateIfNotFresh` and `maxAge` arguments can be used to fine-control the age of the snapshot to be read. The graph server (PGX) will return an existing graph snapshot if the given graph specification was already loaded into memory by a different session. So, the `maxAge` argument becomes important if reading from a database in which the data might change frequently. If no `forceUpdateIfNotFresh` or `maxAge` is specified, PGX will favor cached data over reading new snapshots into memory.

12.1.2 Graph Configuration Options

The following table lists the JSON fields that are common to all graph configurations:

Table 12-1 Graph Config JSON Fields

Field	Type	Description	Default
<code>name</code>	<code>string</code>	Name of the graph.	Required
<code>array_compaction_threshold</code>	<code>number</code>	<i>[only relevant if the graph is optimized for updates]</i> Threshold used to determined when to compact the delta-logs into a new array. If lower than the engine <code>min_array_compaction_threshold</code> value, <code>min_array_compaction_threshold</code> will be used instead	0.2
<code>attributes</code>	<code>object</code>	Additional attributes needed to read and write the graph data.	null
<code>edge_id_strategy</code>	<code>enum[no_ids, keys_as_ids, unstable_generated_ids]</code>	Indicates what ID strategy should be used for the edges of this graph. If not specified (or set to null), the strategy will be determined during loading or using a default value.	null

Table 12-1 (Cont.) Graph Config JSON Fields

Field	Type	Description	Default
edge_id_type	enum[long]	Type of the edge ID. Setting it to long requires the IDs in the edge providers to be unique across the graphs; those IDs will be used as global IDs. Setting it to null (or omitting it) will allow repeated IDs across different edge providers and PGX will automatically generate globally-unique IDs for the edges.	null
edge_providers	array of object	List of edge providers in this graph.	[]
error_handling	object	Error handling configuration.	null
external_stores	array of object	Specification of the external stores where external string properties reside.	[]
jdbc_url	string	JDBC URL pointing to an RDBMS instance	null
keystore_alias	string	Alias to the keystore to use when connecting to database.	null
loading	object	Loading-specific configuration to use.	null
local_date_form at	array of string	array of local_date formats to use when loading and storing local_date properties. See DateTimeFormatter for more details of the format string	[]
max_prefetched_ rows	integer	Maximum number of rows prefetched during each round trip resultset-database.	10000
num_connections	integer	Number of connections to read and write data from or to the RDBMS table.	<no- of- cpus>
optimized_for	enum[read, updates]	Indicates if the graph should use data-structures optimized for read-intensive scenarios or for fast updates.	read
password	string	Password to use when connecting to database.	null
point2d	string	Longitude and latitude as floating point values separated by a space.	0.0 0.0
redaction_rules	array of object	Array of redaction rules.	[]
rules_mapping	array of object	Mapping for redaction rules to users and roles.	[]
schema	string	Schema to use when reading or writing RDBMS objects	null
time_format	array of string	The time format to use when loading and storing time properties. See DateTimeFormatter for a documentation of the format string.	[]
time_with_timestz one_format	array of string	The time with timezone format to use when loading and storing time with timezone properties. Please see DateTimeFormatter for more information of the format string.	[]

Table 12-1 (Cont.) Graph Config JSON Fields

Field	Type	Description	Default
timestamp_format	array of string	The timestamp format to use when loading and storing timestamp properties. See DateTimeFormatter for more information of the format string.	[]
timestamp_with_timezone_format	array of string	The timestamp with timezone format to use when loading and storing timestamp with timezone properties. See DateTimeFormatter for more information of the format string.	[]
username	string	Username to use when connecting to an RDBMS instance.	null
vector_component_delimiter	character	Delimiter for the different components of vector properties.	;
vertex_id_strategy	enum[no_ids, keys_as_ids, unstable_generated_ids]	Indicates what ID strategy should be used for the vertices of this graph. If not specified (or set to null), the strategy will be automatically detected.	null
vertex_id_type	enum[int, integer, long, string]	Type of the vertex ID. For homogeneous graphs, if not specified (or set to null), it will default to a specific value (depending on the origin of the data).	null
vertex_providers	array of object	List of vertex providers in this graph.	[]



Note:

Database connection fields specified in the graph configuration will be used as default in case underlying data provider configuration does not specify them.

Provider Configuration JSON file Options

You can specify the meta-information about each provider's data using provider configurations. Provider configurations include the following information about the provider data:

- Location of the data: a file, multiple files or database providers
- Information about the properties: name and type of the property

Table 12-2 Provider Configuration JSON file Options

Field	Type	Description	Default
format	enum[pgb, csv, rdbms]	Provider format.	Required

Table 12-2 (Cont.) Provider Configuration JSON file Options

Field	Type	Description	Default
name	string	Entity provider name.	Required
attributes	object	Additional attributes needed to read and write the graph data.	null
destination_vertex_provider	string	Name of the destination vertex provider to be used for this edge provider.	null
error_handling	object	Error handling configuration.	null
has_keys	boolean	Indicates if the provided entities data have keys.	true
key_type	enum[int, integer, long, string]	Type of the keys.	long
keystore_alias	string	Alias to the keystore to use when connecting to database.	null
label	string	label for the entities loaded from this provider.	null
loading	object	Loading-specific configuration.	null
local_date_formats	array of string	Array of local_date formats to use when loading and storing local_date properties. See DateTimeFormatter for a documentation of the format string.	[]
password	string	Password to use when connecting to database.	null
point2d	string	Longitude and latitude as floating point values separated by a space.	0.0 0.0
props	array of object	Specification of the properties associated with this entity provider.	[]
source_vertex_provider	string	Name of the source vertex provider to be used for this edge provider.	null
time_format	array of string	The time format to use when loading and storing time properties. See DateTimeFormatter for a documentation of the format string.	[]
time_with_timezone_format	array of string	The time with timezone format to use when loading and storing time with timezone properties. See DateTimeFormatter for a documentation of the format string.	[]
timestamp_formats	array of string	The timestamp format to use when loading and storing timestamp properties. See DateTimeFormatter for a documentation of the format string.	[]
timestamp_with_timezone_format	array of string	The timestamp with timezone format to use when loading and storing timestamp with timezone properties. See DateTimeFormatter for a documentation of the format string.	[]
vector_component_delimiter	character	Delimiter for the different components of vector properties.	;

Provider Labels

The `label` field in the provider configuration can be used to set a label for the entities loaded from the provider. If no `label` is specified, all entities from the provider are labeled with the name of the provider. It is only possible to set the same label for two different providers if they have exactly the same properties (same names and same types).

Property Configuration

The `props` entry in the `Provider` configuration is an object with the following JSON fields:

Table 12-3 Property Configuration


Field	Type	Description	Default
<code>name</code>	<code>string</code>	Name of the property.	Required
<code>type</code>	<code>enum[boolean, integer, vertex, edge, float, long, double, string, date, local_date, time, timestamp, time_with_timezone, timestamp_with_timezone, point2d]</code>	Type of the property .	Required
			<div style="border: 1px solid #0070C0; padding: 5px; background-color: #E6F2FF;"> <p> Note: <code>date</code> is deprecated, use one of <code>local_date/time/timestamp/time_with_timezone/timestamp_with_timezone</code> instead).</p> </div>
<code>aggregate</code>	<code>enum[identity, group_key, min, max, avg, sum, concat, count]</code>	<code>vertex/edge</code> are place-holders for the type specified in <code>vertex_id_type/edge_id_type</code> fields. [currently unsupported] which aggregation function to use, aggregation always happens by vertex key.	<code>null</code>
<code>column</code>	<code>value</code>	Name or index (starting from 0) of the column holding the property data. If it is not specified, the loader will try to use the property name as column name (for CSV format only).	<code>null</code>

Table 12-3 (Cont.) Property Configuration

Field	Type	Description	Default
default	value	Default value to be assigned to this property if datasource does not provide it. In case of date type: string is expected to be formatted with yyyy-MM-dd HH:mm:ss. If no default is present (null), non-existent properties will contain default Java types (primitives) or empty string (string) or 01.01.1970 00:00 (date).	null
dimension	integer	Dimension of property.	0
drop_after_loading	boolean	[currently unsupported] indicating helper properties only used for aggregation, which are dropped after loading	false
field	value	Name of the JSON field holding the property data. Nesting is denoted by dot - separation. Field names containing dots are possible, in this case the dots need to be escaped using backslashes to resolve ambiguities. Only the exactly specified object are loaded, if they are non existent, the default value is used.	null
format	array of string	Array of formats of property.	[]
group_key	string	[currently unsupported] can only be used if the property / key is part of the grouping expression.	null
max_distinct_strings_per_pool	integer	<i>[only relevant if string_pooling_strategy is indexed]</i> Amount of distinct strings per property after which to stop pooling. If the limit is reached an exception is thrown. If set to null, the default value from the global PGX configuration will be used.	null
stores	array of object	A list of storage identifiers that indicate where this property resides.	[]
string_pooling_strategy	enum[indexed, on_heap, none]	Indicates which string pooling strategy to use. If set to null, the default value from the global PGX configuration will be used.	null

Loading Configuration

The loading entry is a JSON object with the following fields:

Table 12-4 Loading Configuration

Field	Type	Description	Default
create_key_mapping	boolean	If true, a mapping between entity keys and internal IDs is prepared during loading.	true
filter	string	[currently unsupported] the filter expression	null

Table 12-4 (Cont.) Loading Configuration

Field	Type	Description	Default
grouping_by	array of string	[currently unsupported] array of edge properties used for aggregator. For Vertices, only the ID can be used (default)	[]
load_labels	boolean	Whether or not to load the entity label if it is available.	false
strict_mode	boolean	If true, exceptions are thrown and logged with ERROR level whenever loader encounters problems with input file, such as invalid format, repeated keys, missing fields, mismatches and other potential errors. If false, loader may use less memory during loading phase, but behave unexpectedly with erratic input files.	true

Error Handling Configuration

The error_handling entry is a JSON object with the following fields:

Table 12-5 Error Handling Configuration

Field	Type	Description	Default
on_missed_property	enum[silent, log_warn, log_warn_once, error]	Error handling for a missing property key.	log_warn_once
on_missing_vertex	enum[ignore_edge, ignore_edge_log, ignore_edge_log_once, create_vertex, create_vertex_log, create_vertex_log_once, error]	Error handling for a missing source or destination vertex of an edge in a vertex data source.	error
on_parsing_issue	enum[silent, log_warn, log_warn_once, error]	Error handling for incorrect data parsing. If set to silent, will attempt to continue loading. Some parsing issues may not be recoverable and provoke the end of loading.	error
on_property_conversion	enum[silent, log_warn, log_warn_once, error]	Error handling when encountering a different property type other than the one specified, but coercion is possible.	log_warn_once

Table 12-5 (Cont.) Error Handling Configuration

Field	Type	Description	Default
on_type_mismatch	enum[silent, log_warn, log_warn_once, error]	Error handling when encountering a different property type other than the one specified, but coercion is <i>not</i> possible.	error
on_vector_length_mismatch	enum[silent, log_warn, log_warn_once, error]	Error handling for a vector property that does not have the correct dimension.	error

**Note:**

The only supported setting for the `on_missing_vertex` error handling configuration is `ignore_edge`.

12.1.3 Preloading a Graph

You can configure the graph server (PGX) to preload graphs in memory at startup-time. This can be useful when you want the graph server (PGX) to startup automatically and have a graph (or multiple graphs) ready for its users. For example, deploying the graph server (PGX) on Kubernetes can be one such scenario.

The configuration for this is done through the `preload_graphs` configuration field in the graph server (PGX) configuration file.

The following is an example for preloading a graph using the graph configuration file:

```
{
  "preload_graphs": [
    {
      "path": "<path-to-graph-config>",
      "name": "my-graph"
    }
  ],
  "authorization": [{
    "pgx_role": "GRAPH_DEVELOPER",
    "pgx_permissions": [{
      "preloaded_graph": "my-graph",
      "grant": "read"
    }]
  }],
  ....
}
```

You can access a preloaded graph by its name using the `getGraph()` method of the session object.

```
PgxGraph g = session.getGraph("my-graph");
```

12.1.4 Data Loading Security Best Practices

Loading graph from the database requires authentication and it is therefore recommended to adhere to the following guidelines when configuring access to this kind of data source:

- The user or role used to access the data should be a read-only account that only has access to the required graph data.
- The graph data should be marked as read-only, for example, with non-updateable views in the case of the database.

12.1.5 Data Format Support Matrix

The following table illustrates how the different data formats differ in the way IDs, labels and vector properties are handled.



Note:

The table refers to limitations of the PGX implementation of the format and not necessarily to limitations of the format itself.

Table 12-6 Data Format Support Matrix

Format	Vertex IDs	Edge IDs	Vertex Labels	Edge Labels	Vector properties
PGB	int, long, string	long	multiple	single	supported (vectors can be of type integer, long, float or double)
CSV	int, long, string	long	multiple	single	supported (vectors can be of type integer, long, float or double)
ADJ_LIST	int, long, string	not supported	not supported	not supported	supported (vectors can be of type integer, long, float or double)
EDGE_LIST	int, long, string	not supported	multiple	single	supported (vectors can be of type integer, long, float or double)
GRAPHML	int, long, string	not supported	not supported	not supported	not supported
TWO_TABLES	int, long, string	long	multiple	single	only in text datastore (vectors can be of type integer, long, float or double)

Table 12-6 (Cont.) Data Format Support Matrix

Format	Vertex IDs	Edge IDs	Vertex Labels	Edge Labels	Vector properties
PG (FLAT_FILE)	int, long	long		single	not supported

12.1.6 Immutability of Loaded Graphs

The graph, once loaded into the graph server (PGX), the graph and its properties are automatically marked as immutable.

The immutability of loaded graphs is due to the following design choices:

- Typical graph analyses happen on a snapshot of a graph instance, and therefore they do not require mutations of the graph instance.
- Immutability allows PGX to use an internal graph representation optimized for fast analysis.
- In remote mode, the graph instance might be shared among multiple clients.

However, the graph server (PGX) also provides methods to privatize and mutate graph instances for the purpose of analysis. See [Graph Mutation and Subgraphs](#) for more information.

12.2 Publishing a Graph

Publishing a Single Graph Snapshot

The `publish()` methods in `PgxGraph` can be used to publish the current selected snapshot of the graph.



Note:

Calling `publish()` without arguments publishes the snapshot with its persistent properties but does not publish transient properties.

This operation will move the graph name from the session-private namespace to the public namespace (see [Namespaces and Sharing](#) for more information about namespaces). If a graph with the same name has been already published, the `publish()` method will fail with an exception.



Note:

Graphs published with snapshots and single published snapshots share the same namespace.

For example, see [Example 4-1](#) to publish a graph using `publish()` method.

If you want to publish specific transient properties, you must list them within the `publish()` call.

Publishing a Graph with Transient Properties Using JShell

```
opg4j> var prop1 = graph.createVertexProperty(PropertyType.INTEGER,
"prop1")
opg4j> prop.fill(0)
opg4j> var cost = graph.createEdgeProperty(PropertyType.DOUBLE, "cost")
opg4j> cost.fill(0d)
opg4j> graph.publish(List.of(prop1), List.of(cost))
```

Publishing a Graph with Transient Properties Using Java

```
VertexProperty<Integer, Integer> prop1 =
graph.createVertexProperty(PropertyType.INTEGER, "prop1");
prop.fill(0);
EdgeProperty<Double> cost =
graph.createEdgeProperty(PropertyType.DOUBLE, "cost");
cost.fill(0d);
List<VertexProperty<Integer, Integer> vertexProps = Arrays.asList(prop);
List<EdgeProperty<Double>> edgeProps = Arrays.asList(cost);
graph.publish(vertexProps, edgeProps);
```

Publishing a Graph with Transient Properties Using Python

```
prop = graph.create_vertex_property("integer", "prop1")
prop.fill(0)
cost = graph.create_edge_property("double", "cost")
cost.fill(0d)
vertex_props = [prop]
edge_props = [cost]
graph.publish(vertex_props, edge_props)
```

Publishing a Graph with Snapshots

If you want to make all snapshots of the graph visible to other sessions, use the `publishWithSnapshots()` methods instead. When a graph is published with snapshots, the `GraphMetaData` information of each snapshot is also made available to the other sessions, with the exception of the graph configuration, which is `null`.

With publishing, all persistent properties of all snapshots are also published and made visible to the other sessions, while transient properties are session-private and thus should be published explicitly. Once published, all properties become read-only. Hence, transient properties are not published when calling `publishWithSnapshots()` without arguments.

Similar to publishing a single graph snapshot, `publishWithSnapshots()` method will move the graph name from the session-private namespace to the public namespace (see [Namespaces and Sharing](#) for more information about namespaces). If a graph with the same name has been already published, the `publishWithSnapshots()` method will fail with an exception.

If you want to publish specific transient properties, you should list them within the `publishWithSnapshots()` call, as in the following example.

Publishing a Graph with Transient Properties Using JShell

```
opg4j> var prop1 = graph.createVertexProperty(PropertyType.INTEGER, "prop1")
opg4j> prop.fill(0)
opg4j> var cost = graph.createEdgeProperty(PropertyType.DOUBLE, "cost")
opg4j> cost.fill(0d)
opg4j> graph.publishWithSnapshots(List.of(prop1), List.of(cost))
```

Publishing a Graph with Transient Properties Using Java

```
VertexProperty<Integer, Integer> prop1 =
graph.createVertexProperty(PropertyType.INTEGER, "prop1");
prop.fill(0);
EdgeProperty<Double> cost = graph.createEdgeProperty(PropertyType.DOUBLE,
"cost");
cost.fill(0d);
List<VertexProperty<Integer, Integer> vertexProps = Arrays.asList(prop);
List<EdgeProperty<Double>> edgeProps = Arrays.asList(cost);
graph.publishWithSnapshots(vertexProps, edgeProps);
```

Publishing a Graph with Transient Properties Using Python

```
VertexProperty<Integer, Integer> prop1 =
graph.createVertexProperty(PropertyType.INTEGER, "prop1")
prop.fill(0)
EdgeProperty<Double> cost = graph.createEdgeProperty(PropertyType.DOUBLE,
"cost")
cost.fill(0d)
List<VertexProperty<Integer, Integer> vertexProps = Arrays.asList(prop)
List<EdgeProperty<Double>> edgeProps = Arrays.asList(cost)
graph.publishWithSnapshots(vertexProps, edgeProps)
```



Note:

The published properties, like the original transient properties, are associated to the specific snapshot they had been created on, so **they are not visible on other snapshots**.

Referencing a Published Graph from Another Session

Other sessions can reference a published graph by its name via the `getGraph()` method of the session object.

The following example references a published graph of `session1`, `myGraph`, in `session2`.

Referencing a Published Graph Using JShell

```
opg4j> var session2 = instance.createSession("session2")
opg4j> var graph2 = session2.getGraph(Namespace.PUBLIC, "myGraph")
```


Referencing a Published Graph Using Java

```
PgxSession session2 = instance.createSession("session2");  
PgxGraph graph2 = session2.getGraph(Namespace.PUBLIC, "myGraph");
```

Referencing a Published Graph Using Python

```
session2 = pypgx.get_session("session2");  
PgxGraph graph2 = session2.get_graph("myGraph")
```

`session2` can see only the published snapshot. If the graph has been published without snapshots, calls to the `getAvailableSnapshots()` method of `session2` return an empty queue.

Instead, if also the snapshots have been published, the call to `getGraph()` returns the most recent snapshot available. `session2` can see all the available snapshots via `getAvailableSnapshots()` and set a specific one via the `setSnapshot()` method of `PgxSession`.

Note:

You must remember to release every graph you reference, when you do not need it anymore. See [Deleting a Graph](#) for more information.

Publishing a Property

After publishing (a single snapshot or all of them), you can still publish transient properties individually:

Publishing Transient Properties Using JShell

```
opg4j> graph.getVertexProperty("prop1").publish()  
opg4j> graph.getEdgeProperty("cost").publish()
```

Publishing Transient Properties Using Java

```
graph.getVertexProperty("prop1").publish();  
graph.getEdgeProperty("cost").publish();
```

Publishing Transient Properties Using Python

```
graph.get_vertex_property("prop1").publish()  
graph.get_edge_property("cost").publish()
```

Note:

Published properties are associated to the specific snapshot they have been created on and thus visible only on that snapshot.

Getting a Published Property in Another Session

Sessions referencing a published graph (with or without snapshots) can reference a published property via the usual `getVertexProperty` and `getEdgeProperty` calls of `PgxGraph`.

Getting a Published Property Using JShell

```
opg4j> var session2 = instance.createSession("session2")
opg4j> var graph2 = session2.getGraph(Namespace.PUBLIC, "myGraph")
opg4j> var vertexProperty = graph2.getVertexProperty("prop1")
opg4j> var edgeProperty = graph2.getEdgeProperty("cost")
```

Getting a Published Property Using Java

```
PgxSession session2 = instance.createSession("session2");
PgxGraph graph2 = session2.getGraph(Namespace.PUBLIC, "myGraph");
VertexProperty<Integer, Integer> vertexProperty =
graph2.getVertexProperty("prop1");
EdgeProperty<Double> edgeProperty = graph2.getEdgeProperty("cost");
```

Getting a Published Property Using Python

```
session2 = pypgx.get_session(session_name = "session2")
graph2 = session2.get_graph("myGraph")
vertex_property = graph2.get_vertex_property("prop1")
edge_property = graph2.get_edge_property("cost")
```

`session2` now has a reference to the published graph of `session1` called `myGraph` and can reference its published properties via `myGraph` itself.

12.3 Publishing a Preloaded Graph

The publishing behavior for preloaded graphs can be controlled in the configuration. Unless a different behavior is configured, (only) the first loaded snapshot of a graph is published. Preloaded published graphs remain in memory even if they are not used by any session.

There are two options to control the publishing behavior:

- Set the optional flag `publish` to `true`, to publish only the graph but no future snapshots of the graph. This is the default behavior as the default value of this flag is `true`.
- Set the optional flag `publish_with_snapshots` to `true`, to publish the graph and all future snapshots of the graph. The default value is `false`.

Only one of these two flags can be set to `true` at a time. However, publishing the graph with snapshots does also publish the first version of the graph.

Example 12-1 Sample Configuration File for Preloading Graphs

This example `pgx.conf` specifies two graphs for loading into memory during the graph server (PGX) startup-time. `my-graph` is published with snapshots while `my-other-graph` is published without snapshots.

```
{
  "enterprise_scheduler_config": {
    "analysis_task_config": {
      "max_threads": 32
    }
  },
  "preload_graphs": [
    {
      "path": "graph-configs/my-graph.bin.json",
      "name": "my-graph",
      "publish": false,
      "publish_with_snapshots": true
    },
    {
      "path": "graph-configs/my-other-graph.adj.json",
      "name": "my-other-graph"
    }
  ],
  "authorization": [{
    "pgx_role": "GRAPH_DEVELOPER",
    "pgx_permissions": [{
      "preloaded_graph": "my-graph",
      "grant": "read"
    }],
    {
      "preloaded_graph": "my-other-graph",
      "grant": "read"
    }
  ]
},
  ....
]
```

The two preloaded graphs can be accessed as follows:

```
PgxGraph g1 = session.getGraph("my-graph"); //returns the most recent
available snapshot
PgxGraph g2 = session.getGraph("my-other-graph");
```

12.4 Deleting a Graph

In order to reduce the memory usage of the graph server (PGX), the session must drop the unused `PgxGraph` graph objects that it created via `PgxSession.getGraph()` by invoking the `destroy()` method. This step not only destroys the specified graph, but all of its associated properties, including transient properties as well. In addition, all of the collections related to the graph instance (for example, a `VertexSet`) are also destroyed automatically. If a session holds multiple `PgxGraph` objects referencing the same

graph, invoking `destroy()` on any of them will invalidate all the `PgxGraph` objects referencing that graph, making any operation on those objects fail:

Deleting a Graph Using Java

```
PgxGraph graph1 = session.getGraph("myGraphName")
// graph2 references the same graph of graph1
PgxGraph graph2 = session.getGraph("myGraphName")
// both calls throw an exception, as both references are not valid anymore
Set<VertexProperty<?, ?>> properties = graph1.getVertexProperties();
properties = graph2.getVertexProperties()
```

Deleting a Graph Using Python

```
graph1 = session.get_graph("myGraphName")

# graph2 references the same graph of graph1
graph2 = session.get_graph("myGraphName")

# both calls throw an exception, as both references are not valid anymore
properties = graph1.get_vertex_properties()
properties = graph2.get_vertex_properties()
```

The same behavior occurs when multiple `PgxGraph` objects reference the same snapshot. Since a snapshot is effectively a graph, destroying a `PgxGraph` object referencing a certain snapshot invalidates all `PgxGraph` objects referencing the same snapshot, but does not invalidate those referencing other snapshots:

```
// get a snapshot of "myGraphName"
PgxGraph graph1 = session.getGraph("myGraphName");
// graph2 and graph3 reference the same snapshot as graph1
PgxGraph graph2 = session.getGraph("myGraphName");
PgxGraph graph3 = session.getGraph("myGraphName");

// we assume another snapshot is created ...

// make graph3 references the latest snapshot available
session.setSnapshot(graph3, PgxSession.LATEST_SNAPSHOT);
graph2.destroy();
// both calls throw an exception, as both references are not valid anymore
Set<VertexProperty<?, ?>> properties = graph1.getVertexProperties();
properties = graph2.getVertexProperties();

// graph3 is still valid, so the call succeeds
properties = graph3.getVertexProperties();
```



Note:

Even if a graph is destroyed by a session, the graph data may still remain in the server memory if the graph is currently shared by other sessions. In such a case, the graph may still be visible among the available graphs via `PgxSession.getGraphs()`.

As a safe alternative to manual destruction of each graph, the PGX API supports some implicit resource management features which allow developers to safely omit the `destroy()` call. See [Resource Management Considerations](#) for more information.

13

Namespaces and Sharing

The in-memory graph server (PGX) supports separate namespaces that help you to organize your entities.

Each client session has its own session-private namespace and can choose any name without affecting other sessions. There is also a public namespace for published graphs (for example, published via the `publishWithSnapshots()` or the `publish()` methods).

Similarly, each published graph defines a public namespace for published properties as well as a private namespace per session. So different sessions can create properties with the same name on a published graph.

- [Defining Graph Names](#)
- [Retrieving Graphs by Name](#)
- [Checking Used Names](#)
- [Property Name Resolution and Graph Mutations](#)

13.1 Defining Graph Names

Graphs that are created in a session either through loading (for example, calling `readGraphWithProperties()`) or through mutations will take up a name in the session-private namespace. A graph will be placed in the public namespace only through publishing (that is, when calling the `publishWithSnapshots()` or the `publish()` methods). Publishing a graph will move its name from the session-private namespace to the public namespace.

There can only be one graph with a given name in a given namespace, but a name can be used in different namespaces to refer to different graphs. An operation that creates a new graph (for example, `readGraphWithProperties()`) will fail if the chosen name of the new graph already exists in the session-private namespace. Publishing a graph fails if there is already a graph in the public namespace with the same name.

13.2 Retrieving Graphs by Name

You can retrieve a graph by name by the following two ways:

- `getGraph(Namespace, String)`: with explicitly mentioning the namespace
- `getGraph(String)`: without explicitly mentioning the namespace

With `getGraph(Namespace, String)`, you need to provide the namespace (either session private or public). In this case, the graph will be looked up in the given namespace only.

With `getGraph(String)`, the provided name will be first looked up in the private namespace. If no graph with the given name is found there, then the graph name will be looked up in the public namespace. In other words, if a graph with the same name is defined in both the public and the private namespaces, `getGraph(String)` will return the private graph and you need to use `getGraph(Namespace, String)` to get hold of the public graph with that name.

13.3 Checking Used Names

To see the currently used names in a namespace you can use the [PgxFSession.getGraphs\(Namespace\)](#) method, which will list all the names in the given namespace. The names in the returned collection can be used in a [getGraph\(Namespace, String\)](#) call to retrieve the corresponding [PgxFGraph](#).

13.4 Property Name Resolution and Graph Mutations

Property names behave in a similar way as graph names. All property names of a non-published graph are in the session-private namespace. Once a graph is published with [PgxFGraph.publishWithSnapshots\(\)](#) or the [PgxFGraph.publish\(\)](#) methods, its properties are published as well and their names move into the public namespace.

Once a graph is published, newly created properties will still be private to the session and their names will be in the private namespace. Those properties can be published individually with the [Property.publish\(\)](#) method, as long as no other property with the same name is already published for that graph.

Additionally, new private properties can be created with the same name of an already-published properties (since the names are part of separate namespaces). To handle such situations and retrieve the correct property, the PGX API offers the [getVertexProperty\(Namespace, String\)](#) and the [getEdgeProperty\(Namespace, String\)](#) methods, which allow specifying the namespace where the property name should be looked up.

Similar to graphs, if you search a property without specifying the namespace, the private namespace is searched first and if the property is not found, the search proceeds to the public namespace. This case applies for [getVertexProperty\(String\)](#) or the [getEdgeProperty\(String\)](#) methods and for PGQL queries.

Likewise, when a mutation on a graph reads or writes a property referred to by name and two properties exist with the same name, the property in the private namespace is selected. To override the default selection, some mutation mechanisms accept a collection of specific [Property](#) objects to be copied into the mutated graph. For example, such mechanism is supported for filter expressions. See [Creating Subgraphs](#) for more details.

PGX Programming Guides

You can avail all the PGX functionalities through asynchronous Java APIs. Each asynchronous method has a synchronous equivalent, which blocks the caller thread until the server produces a response.

These APIs may perform one or any combination of:

- Complex, non-blocking Java applications on top of PGX
- Simple, sequential Java scripts executed by JShell
- ShellPerforming interactive graph analysis in the JShell

Layers of PGX API

The PGX API is composed of a few different Java interfaces. Each interface provides a distinct layer of abstraction for PGX, as shown in the following table:

Table 14-1 PGX API Interface

Interface	Description
ServerInstance	The <code>ServerInstance</code> class encapsulates access to a PGX server instance and can be used to create sessions, start and stop the PGX engine, monitor the engine status and perform other administrative tasks. If the instance points to a remote instance, access to the administrative functions requires special authorization on the HTTP level by default.
PgxSession	A <code>PgxSession</code> represents an active user currently connected to an instance. Each session gets its own workspace on the server side which can be used to read graphs, create in-memory data structures, hold analysis results and custom algorithms. The <code>PgxSession</code> class provides various methods to create new transient data (currently collections). If a session is idling for too long, the PGX engine will automatically destroy it to ensure no resources are wasted.
PgxGraph	A <code>PgxGraph</code> represents a client-side handle to the graph data managed by the PGX server. A graph may contain an arbitrary amount of properties of type <code>VertexProperty</code> and/or <code>EdgeProperty</code> .

Note:

The PGX currently only supports non-partitioned graphs, meaning every vertex/edge has the same properties with the same names and types as all the other vertices/edges.

`PgxGraph` class provides various methods to create new transient data (including maps and collections) as well as graph mutation operations, such as undirecting, sorting and filtering.

Table 14-1 (Cont.) PGX API Interface

Interface	Description
Analyst	The Analyst API contains all of the built-in algorithms PGX provides. Analyst objects keep track of all the transient data they created during algorithm invocations to hold analysis results. Once an Analyst gets destroyed, all the results it created get freed on the server-side automatically.
CompiledProgram	The CompiledProgram class (PGX Algorithm API) encapsulates runtime-compiled custom algorithms and allows invocation of those algorithms using PGX data objects, such as PgxGraph or VertexProperty, as arguments.

Please see the [oracle.pgx.api](#) package in the Javadoc for more details.

- [Design of the Graph Server \(PGX\) API](#)
This guide focuses on the design of the graph server (PGX) API.
- [Data Types and Collections in the Graph Server \(PGX\)](#)
This guide provides you the list of the supported data types and collections in the graph server (PGX).
- [Handling Asynchronous Requests in Graph Server \(PGX\)](#)
This guide explains in detail the asynchronous methods supported by the PGX API.
- [Graph Client Sessions](#)
The graph server (PGX) assumes there may be multiple concurrent clients, and each client submits request to the shared PGX server independently.
- [Graph Mutation and Subgraphs](#)
This guide discusses the several methods provided by the graph server (PGX) for mutating graph instances.
- [Managing Transient Data](#)
This guide discusses how to handle transient properties and collections.
- [Graph Versioning](#)
This guide describes the different ways to work with graph snapshots.
- [Labels and Properties](#)
You can perform various actions on the graph property and label values by executing PGQL queries.
- [Filter Expressions](#)
This guide explains the usage of filter expressions.
- [Advanced Task Scheduling Using Execution Environments](#)
This guide shows how you can use the advanced scheduling features of the enterprise scheduler.
- [Admin API](#)
This guide shows how to use the graph server (PGX) Admin API to inspect the server state including sessions, graphs, tasks, memory and thread pools.
- [PgxFrames Tabular Data-Structure](#)

14.1 Design of the Graph Server (PGX) API

This guide focuses on the design of the graph server (PGX) API.

The design of the PGX API reflects consideration of the following situations:

- Multiple clients may concurrently be accessing a single running instance of PGX, sharing its resources. Each client needs to maintain its own isolated workspace (session).
- Graph and property data can be large in size and therefore that data only resides on the server side.
- Some graph analysis may take a significant amount of time.
- Clients may not reside in the same address space (JVM) as PGX. Actually, clients may not even be Java applications.

Client Sessions

In PGX, each client maintains its own session, an isolated, private workspace. Therefore, clients first have to obtain a `PgxSession` object from a `PGX ServerInstance` before they can perform any analysis.

Asynchronous Execution

The PGX API is designed for asynchronous execution. That means that each computationally intensive method in the PGX API *immediately* returns a `PgxFuture` object without waiting for the request to finish. The `PgxFuture` class implements the `Future` interface, which can be used to retrieve the result of a computation at some point in the future.

Note:

The asynchronous execution aspect of this design facilitates multiple (remote) clients submitting requests to a single server. A request from one client may be queued up to wait until PGX resources become available. The asynchronous API allows the client (or calling thread) to work on other tasks until PGX completes the request.

No Direct References

The PGX API does not return objects with direct reference to PGX internal objects (such as the graph or its properties) to the client. This is because:

- The client might not be in the same JVM as the server
- The graph instance might be shared by multiple clients

Instead, the PGX API only returns lightweight, stateless pointer objects to those objects. These pointer objects only hold the `ID(name)` of the server-side object to which they are pointing.

Resource Management Considerations

The in-memory graph server (PGX), being an *in-memory* analytic engine, might allocate large amounts of memory to hold the graph data of clients. Therefore, it is important that client

sessions clean up their resources once they have ended. The PGX API supports several features to make this easier:

- Every object returned by the PGX API pointing to a server-side resource implements the `Destroyable` interface, which means all memory-consuming client-side objects can be destroyed the same way. For example:

```
PgxGraph myGraph = ...
myGraph.destroyAsync(); // request destruction of myGraph, don't
wait for response
try {
    myGraph.destroy(); // blocks caller thread until destruction
was done
} catch (ExecutionException e) {
    // destruction failed
}
```

- `Destroyable` extends [AutoClosable](#), so users can leverage Java's built-in resource management syntax:

```
try (PgxGraph myGraph = session.readGraphWithProperties(config)) {
    // do something with myGraph
}
// myGraph is destroyed
```

- **Session time out.** In some cases, the PGX server will remove the session and all its data automatically. This can occur when a client fails to destroy either the data or its session, or if it does not hear from the session after a configurable timeout. See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information to configure timeout parameters.

14.2 Data Types and Collections in the Graph Server (PGX)

This guide provides you the list of the supported data types and collections in the graph server (PGX).

Primitive Data Types

The following section explains the primitive data types supported by the graph server (PGX) and their limitations.

PGX supports the following primitive data types.:

- **Numeric Types:** `integer`, `long`, `float`, and `double`. These types have the same size, range and precision of the corresponding Java primitive data type.
- **Boolean Type:** The `boolean` data type has only two possible values, `true` and `false`. As with Java and C++, its size is not precisely defined.
- **String:** `String` is a primitive data type in PGX. PGX follows the Java conventions for `String` representation.
- **Datetime Types:** `date`, `time`, `timestamp`, `time with time zone`, and `timestamp with time zone`. These types correspond to the Java types shown in [Table 14-2](#) from the standard library package `java.util.time`.

- **Vertex and Edge:** The type `vertex` or `edge` of the graph itself is a proper type in PGX.



Note:

- `vertex` and `edge` is itself a valid primitive data type. For instance, in a path-finding algorithm, each `vertex` can have a temporary property `predecessor` that stores which incoming neighbor is the predecessor vertex in the path. Such a property would have the type `vertex`.
- `local_date` must be used instead of `date` in the graph configuration file. See [Using Datetime Data Types](#) for more examples on usage of datetime data types.

All properties and scalar variables must be one of the above preceding data types. See [Managing Transient Data](#) for more information on handling transient properties and scalar variables.

The following table presents the overview of the supported data types, their integration in different languages and APIs and their minimum and maximum value limitations.



Note:

- For float and double types, the smallest absolute value is included in the table, the minimum value is the negative of maximum value for these types.
- For string values, PGX supports arbitrary long strings.

Table 14-2 Overview of Data types

Data Type	Loading & Storing	PGX Java API	PGQL and Filter Expression	Minimum Value Limitation	Maximum Value Limitation
string	string	String	STRING	-	-
int/integer	int/integer	int	INT/INTEGER	-2147483648	2147483647
long	long	long	LONG	-9223372036854775808	9223372036854775807
float	float	float	FLOAT	1.4E-45	3.4028235e+38
double	double	double	DOUBLE	4.9E-324	1.7976931348623157E308
boolean	boolean	boolean	BOOLEAN	-	-
date	local_date	LocalDate	DATE	-5877641-06-23	5881580-07-11
time	time	LocalTime	TIME	00:00:00.000	23:59:59.999
timestamp	timestamp	LocalDateTime	TIMESTAMP	-292275055-05-17 00:00:00.000	292278994-08-17 07:12:55.807
time with time zone	time_with_timezone	OffsetTime	TIME WITH TIME ZONE	00:00:00.000+18:00	23:59:59.999-18:00

Table 14-2 (Cont.) Overview of Data types

Data Type	Loading & Storing	PGX Java API	PGQL and Filter Expression	Minimum Value Limitation	Maximum Value Limitation
timestamp with time zone	timestamp_with_timezone	OffsetDateTime	TIMESTAMP WITH TIME ZONE	-292275055-05-17 00:00:00.000+18:00	292278994-08-17 07:12:55.807-18:00
vertex	-	PgxVertex	-	-	-
edge	-	PgxEdge	-	-	-

Collections

The in-memory graph server (PGX) supports three different collection types: `sequence`, `set` and `order`. All of these collections can contain values of the `vertex` type, but each has different semantics regarding uniqueness and preserving the order of its elements:

- **Sequence:** a `sequence` works basically like a list. It preserves the order of the elements added to it, and the same element can appear multiple times.
- **Set:** a `set` can contain the same value once at the most. Adding a value that is already in the `set` will have no effect. `set` does not preserve the order of the elements it contains.
- **Order:** just like the `set`, the `order` collection will contain each element once at the most. But the `order` preserves the order of the elements inserted into it (that is, it is a FIFO data structure).

See [Collection Data Types](#) for examples on creation and usage of the different collections.

Immutable Collections

Some operations, like `PgxGraph.getVertices()` and `PgxGraph.getEdges()` return immutable collections. These collections behave like normal collections, but cannot be modified by operations like `addAll` or `removeAll` and `clear`.

An immutable collection can be transformed into a mutable collection by using the `toMutable` method, which returns a mutable copy of the collection. If `toMutable` is called on a collection that is already mutable, the method has the same result as the method `clone`.

To check if a collection is mutable, use the `isMutable` method.

Maps

PGX provides the following two kinds of maps:

- Graph-bound maps can hold mappings between types in `PropertyType`. This is the kind of maps to use if the key or value types are graph-related like `VERTEX` and `EDGE` otherwise using session-bound maps is recommended.
- Session-bound maps can map between non graph-related types and are directly bound to the session.

See [Map Data Types](#) for examples on creation and usage of maps.

- [Using Collections and Maps](#)
- [Using Datetime Data Types](#)

14.2.1 Using Collections and Maps

This section explains with examples, the creation and usages of collections and maps.

You must first create a session before getting started with the collection and map data types.

Example 14-1 Creating a session using JShell

```
cd /opt/oracle/graph/  
./bin/opg-jshell // starting the shell will create an implicit session
```

Example 14-2 Creating a session using Java

```
import oracle.pgx.api.*;  
...  
PgxSession session=Pgx.createSession("<session_name>");
```

Example 14-3 Creating a session using Python

```
from pypgx import get_session  
session = get_session(session_name="<session_name>")
```

- [Collection Data Types](#)
- [Map Data Types](#)

14.2.1.1 Collection Data Types

The in-memory graph server (PGX) defines two types of collections:

- **Graph-bound collections:** such as vertex and edge collections. These collections belong to the graph.
- **Session-bound collections:** belong to the session.
- [Graph-Bound Collections](#)
- [Session-Bound Collections](#)

14.2.1.1.1 Graph-Bound Collections

The following describes the usage of graph-bound collections.

You must first load the graph to work with vertex and edge collections as shown in [Loading a Graph Into the Graph Server \(PGX\)](#).

Vertex Collections

You can create a vertex collection as shown in the following code:

Creating a Vertex Collection Using JShell

```
v0 = graph.getVertex(100) // 'graph' is the loaded graph object. '100' -  
> '103' are vertex ids that supposedly  
v1 = graph.getVertex(101) // exist in the graph  
v2 = graph.getVertex(102)  
v3 = graph.getVertex(103)  
  
myVertexSet = graph.createVertexSet("myVertexSet") // A name is  
automatically generated if none given  
myVertexSet.add(v0) // Adds vertex 'v0'  
to the set  
myVertexSet.addAll([v1, v2, v3]) // Supports  
variadic parameter as well: myVertexSet.addAll(v1, v2, v3)
```

Creating a Vertex Collection Using Java

```
import java.util.Arrays;  
import oracle.pgx.api.*;  
...  
PgxVertex v0 = graph.getVertex(100);  
PgxVertex v1 = graph.getVertex(101);  
PgxVertex v2 = graph.getVertex(102);  
PgxVertex v3 = graph.getVertex(103);  
  
VertexSet myVertexSet = graph.createVertexSet("myVertexSet"); // A  
name is automatically generated if none given  
myVertexSet.add(v0);  
myVertexSet.addAll(Arrays.asList(v1, v2, v3));
```

Creating a Vertex Collection Using Python

```
...  
v0 = graph.get_vertex(100)  
v1 = graph.get_vertex(101)  
v2 = graph.get_vertex(102)  
v3 = graph.get_vertex(103)  
  
my_vertex_set = graph.create_vertex_set("myVertexSet")  
my_vertex_set.add(v0)  
my_vertex_set.add_all([v1,v2,v3])
```

Edge Collections

You can create an edge collection as shown in the following code:

Creating an Edge Collection Using JShell

```
e0 = graph.getEdge(100) // 'graph' is the loaded graph object. '100' ->  
'103' are edge ids that supposedly  
e1 = graph.getEdge(101) // exist in the graph.  
e2 = graph.getEdge(102)
```

```
e3 = graph.getEdge(103)

myEdgeSequence = graph.createEdgeSequence("myEdgeSequence")
myEdgeSequence.add(e0)
myEdgeSequence.addAll([e1, e2, e3])
```

Creating an Edge Collection Using Java

```
import java.util.Arrays;
import oracle.pgx.api.*;
...
PgxEde e0 = graph.getEdge(100);
PgxEde e1 = graph.getEdge(101);
PgxEde e2 = graph.getEdge(102);
PgxEde e3 = graph.getEdge(103);

EdgeSequence myEdgeSequence = graph.createEdgeSequence("myEdgeSequence");
myEdgeSequence.add(e0);
myEdgeSequence.addAll(Arrays.asList(e1, e2, e3));
```

Creating an Edge Collection Using Python

```
e0 = graph.get_edge(100)
e1 = graph.get_edge(101)
e2 = graph.get_edge(102)
e3 = graph.get_edge(103)

my_edge_sequence = graph.create_edge_sequence("my_edge_sequence")
my_edge_sequence.add(e0)
my_edge_sequence.add_all([e1, e2, e3])
```

14.2.1.1.2 Session-Bound Collections

You can create and manipulate collections directly in the session without the need for a graph. Session-bound collections can be further passed as parameters to graph algorithms or used like any other collection object. The following sub-sections describe the currently supported types for these collections.

Scalar Collections

Scalar collections contain simple data types like Integer, Long, Float, Double and Boolean. They can be managed by the `PgxSession` APIs:

Creation of a Scalar Collection

You can use `createSet()` and `createSequence()` methods to create a scalar collection as shown in the following code:

Creating a Scalar Collection Using JShell

```
myIntSet = session.createSet(PropertyType.INTEGER, "myIntSet")
myDoubleSequence = session.createSequence(PropertyType.DOUBLE) // A name will be
automatically generated if none is provided.
```



```
println myDoubleSequence.getName() // Display the
generated name.
```

Creating a Scalar Collection Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.common.types.*;
...
ScalarSet myIntSet = session.createSet(PropertyType.INTEGER, "myIntSet");
ScalarSequence myDoubleSequence = session.createSequence(PropertyType.DOUBLE);
System.out.println(myDoubleSequence.getName());
```

Run Operations on a Scalar Collection

You can run several operations on a scalar collection as shown in the following code:

Running Operations on a Scalar Collection Using JShell

```
myIntSet.add(10)
myIntSet.addAll([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
myIntSet.addAll([0,1,2]) // Element uniqueness. This operation
has no effect on the set.
println myIntSet

myIntSet.contains(1) // Checks the presence of an element.
This code returns `true`.
myIntSet.remove(10)
myIntSet.removeAll([4, 5, 6, 7, 8, 9]) // Leaves only elements `0, 1, 2, 3`.
println myIntSet
```

Running Operations on a Scalar Collection Using Java

```
import java.util.Arrays;
import oracle.pgx.api.*;
...
myIntSet.add(10);
myIntSet.addAll(Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));
myIntSet.addAll(Arrays.asList(0, 1, 2))

myIntSet.contains(1); // Returns `true`.
myIntSet.remove(10);
myIntSet.removeAll(Arrays.asList(4, 5, 6, 7, 8, 9));
```

Traversal of a Scalar Collection

You can traverse a scalar collection either using an iterator or using the new [Stream](#) API. You can add elements of a sequence to a set, traverse a sequence and filter out elements not required, and then add the rest to another scalar collection.

Traversing a Scalar Collection Using JShell

```
myIntSet.forEach({x -> print x + "\n"})
myIntSet.stream().filter({x -> x % 2 == 0}).forEach({x ->
myDoubleSequence.add(x)})
println myDoubleSequence
```

Traversing a Scalar Collection Using Java

```
import java.util.Iterator;
import java.util.stream.Stream;
import oracle.pgx.api.*;
...
```

```
myIntSet.forEach(x -> System.out.println(x))
myIntSet.stream().filter(x -> x % 2 == 0).forEach(myDoubleSequence::add)
```

14.2.1.2 Map Data Types

The in-memory graph server (PGX) defines two types of maps:

- **Graph-bound maps:** These maps support any key or value type and are created using a graph object.
- **Session-bound maps:** Keys or values in these maps are of any type except from graph-related types (that is, vertices or edges). These maps belong to the session.
- [Graph-Bound Maps](#)
- [Session-Bound Maps](#)

14.2.1.2.1 Graph-Bound Maps

Some data types like `VERTEX` or `EDGE` depend on the graph. Consequently, mappings involving these data types also depend on the graph. PGX provides `PgxGraph` and `PgxMap` APIs to manage such maps.

The following describes the usage of graph-bound maps.

You must first load the graph to work with vertex and edge maps.

You can create a graph-bound map using vertices as keys as shown in the following code:

Creating a Graph-bound Map with Vertices as Keys Using JShell

```
v0 = graph.getVertex(100)
v1 = graph.getVertex(101)
v2 = graph.getVertex(102)
v3 = graph.getVertex(103)

vertexToLongMap = graph.createMap(PropertyType.VERTEX, PropertyType.LONG,
"vertexToLongMap")
vertexToLongMap.put(v0, v0.getDegreeAsync().get())
vertexToLongMap.put(v1, v1.getDegreeAsync().get())
vertexToLongMap.put(v2, v2.getDegreeAsync().get())
vertexToLongMap.put(v3, v3.getDegreeAsync().get())
```

Creating a Graph-bound Map with Vertices as Keys Using Java

```
import java.util.Arrays;
import oracle.pgx.api.*;
...
PgxVertex v0 = graph.getVertex(100);
PgxVertex v1 = graph.getVertex(101);
PgxVertex v2 = graph.getVertex(102);
PgxVertex v3 = graph.getVertex(103);

PgxMap<PgxVertex, Long> vertexToLongMap =
graph.createMap(PropertyType.VERTEX, PropertyType.LONG, "vertexToLongMap");
vertexToLongMap.put(v0, v0.getDegree());
```

```
vertexToLongMap.put(v1, v1.getDegree());  
vertexToLongMap.put(v2, v2.getDegree());  
vertexToLongMap.put(v3, v3.getDegree());
```

Creating a Graph-bound Map with Vertices as Keys Using Python

```
v0 = graph.get_vertex(100)  
v1 = graph.get_vertex(101)  
v2 = graph.get_vertex(102)  
v3 = graph.get_vertex(103)  
  
vertex_to_long_map = graph.create_map("vertex", "long",  
"vertex_to_long_map")  
vertex_to_long_map.put(v0, v0.degree)  
vertex_to_long_map.put(v1, v1.degree)  
vertex_to_long_map.put(v2, v2.degree)  
vertex_to_long_map.put(v3, v3.degree)
```

You can create graph-bound maps using edges as keys as shown in the following code:

Creating a Graph-bound Map with Edges as Keys Using JShell

```
e0 = graph.getEdge(100)  
e1 = graph.getEdge(101)  
e2 = graph.getEdge(102)  
e3 = graph.getEdge(103)  
  
edgeToVertexMap = graph.createMap(PropertyType.EDGE,  
PropertyType.VERTEX, "edgeToVertexMap")  
edgeToVertexMap.put(e0, e0.getSource())  
edgeToVertexMap.put(e1, e1.getSource())  
edgeToVertexMap.put(e2, e2.getSource())  
edgeToVertexMap.put(e3, e3.getSource())
```

Creating a Graph-bound Map with Edges as Keys Using Java

```
import java.util.Arrays;  
import oracle.pgx.api.*;  
...  
PgxEdge e0 = graph.getEdge(100);  
PgxEdge e1 = graph.getEdge(101);  
PgxEdge e2 = graph.getEdge(102);  
PgxEdge e3 = graph.getEdge(103);  
  
PgxMap<PgxEdge, PgxVertex> edgeToVertexMap =  
graph.createMap(PropertyType.EDGE, PropertyType.VERTEX,  
"edgeToVertexMap");  
edgeToVertexMap.put(e0, e0.getSource());  
edgeToVertexMap.put(e1, e1.getSource());  
edgeToVertexMap.put(e2, e2.getSource());  
edgeToVertexMap.put(e3, e3.getSource());
```

Creating a Graph-bound Map with Edges as Keys Using Python

```
e0 = graph.get_edge(100)
e1 = graph.get_edge(101)
e2 = graph.get_edge(102)
e3 = graph.get_edge(103)

edge_to_long_map = graph.create_map("edge", "long", "edge_to_long_map")
edge_to_long_map.put(e0, e0.source)
edge_to_long_map.put(e1, e1.source)
edge_to_long_map.put(e2, e2.source)
edge_to_long_map.put(e3, e3.source)
```



Note:

If you destroy the graph you will lose the map. Consider using a session-bound maps instead if your map does not involve any graph-related key or value type.

14.2.1.2.2 Session-Bound Maps

You can directly create maps in the session. But, you cannot use any graph-related data type as the map key or value type. Session-bound maps can be further passed as parameters to graph algorithms or used like any other map object. They are managed by `PgxSession` and `PgxMaps` APIs.

Scalar collections contain simple data types like `Integer`, `Long`, `Float`, `Double` and `Boolean`. They can be managed by the `PgxSession` APIs.

Creation of a Session-bound Map

You can use `createMap()` method and its overloads to create a session-bound map.

Creating a Session-bound Map Using JShell

```
intToDouble = session.createMap(PropertyType.INTEGER, PropertyType.DOUBLE,
"intToDouble")
intToTime = session.createMap(PropertyType.INTEGER, PropertyType.TIME) // A name will
be automatically generated.
println intToTime.getName()
println intToTime.getSessionId()
println intToTime.getGraph() // `null`: Not
bound to a graph.
println intToTime.getKeyType()
println intToTime.getValueType()
```

Creating a Session-bound Map Using Java

```
import java.time.LocalDateTime;
import oracle.pgx.api.*;
import oracle.pgx.common.types.*;
...
PgxMap<Integer, Double> intToDouble = session.createMap(PropertyType.INTEGER,
PropertyType.DOUBLE, "intToDouble");
PgxMap<Integer, LocalDateTime> intToTime = session.createSequence(PropertyType.INTEGER,
PropertyType.TIME);
System.out.println(intToTime.getName());
```

```
System.out.println(intToTime.getSessionId());
System.out.println(intToTime.getGraph()); // `null`: Not bound to a graph.
System.out.println(intToTime.getKeyType());
System.out.println(intToTime.getValueType());
```

Run Operations on a Session-bound Map

You can run important operations such as setting, removing and checking existence of entries on a session-bound map as shown in the following code:

Running Operations on a Session-bound Map Using JShell

```
intToDouble.put(0, 0.314)
intToDouble.put(1, 3.14)
intToDouble.put(2, 31.4)
intToDouble.put(3, 314)

println intToDouble.size()           // 4
println intToDouble.get(1)
println intToDouble.get(3)
println intToDouble.get(10)         // null

println intToDouble.containsKey(0)  // `true`
intToDouble.remove(0)
println intToDouble.containsKey(0)  // `false`
println intToDouble.containsKey(10) // `false`
intToDouble.remove(10)
println intToDouble.containsKey(10) // `false`

println intToDouble.put(1, 999)     // previous mapped value (`3.14`) is
replaced by `999`
intToDouble.destroy()
```

Running Operations on a Session-bound Map Using Java

```
import java.util.Arrays;
import oracle.pgx.api.*;

...

intToDouble.put(0, 0.314);
intToDouble.put(1, 3.14);
intToDouble.put(2, 31.4);
intToDouble.put(3, 314);

System.out.println(intToDouble.size());           // 4
System.out.println(intToDouble.get(1));
System.out.println(intToDouble.get(3));
System.out.println(intToDouble.get(10));         // null

System.out.println(intToDouble.containsKey(0));  // `true`
intToDouble.remove(0);
System.out.println(intToDouble.containsKey(0));  // `false`
System.out.println(intToDouble.containsKey(10)); // `false`
intToDouble.remove(10);
System.out.println(intToDouble.containsKey(10)); // `false`

System.out.println(intToDouble.put(1, 999));     // previous mapped value
(`3.14`) is replaced by `999`
intToDouble.destroy();
```

Traversal of a Session-bound Map

You can traverse a session-bound map, using `entries()` method to get an iterable of map entries and `keys()` method to get an iterable of map keys.

Traversing a Session-bound Map Using JShell

```
intToDouble.entries().forEach {it -> println (it)}
intToDouble.keys().forEach {it -> println (it)}
```

Traversing a Session-bound Map Using Java

```
import java.util.Iterable;
import java.util.stream.Stream;
import oracle.pgx.api.*;
...
Iterable<Map.Entry> entries = intToDouble.entries();
entries.forEach(System.out::println);
Iterable<Map.Entry> keys = intToDouble.keys();
keys.forEach(System.out::println);
```

14.2.2 Using Datetime Data Types

This section explains in detail working of datetime data types such as date, time and timestamp.

Overview of Datetime Data Types in In-Memory Graph Server (PGX)

Table 14-3 presents the overview of the five `datetime` data types supported by PGX along with example values.



Note:

PGX also supports custom format specification when loading data into PGX.

Table 14-3 Overview of Datetime Data Types in PGX

Data Type	Loading and Storing	PGX Java API	PGQL and Filter Expression	Example Value-1	Example Value-1
date	local_date	LocalDate	DATE	2001-01-29	2018-10-08
time	time	LocalTime	TIME	10:15	10:30:01.000
timestamp	timestamp	LocalDateTime	TIMESTAMP	2001-01-29 10:15	2018-10-08 10:30:01.000
time with time zone	time_with_timezone	OffsetTime	TIME WITH TIME ZONE	10:15+01:00	10:30:01.000-08:00
timestamp with time zone	timestamp_with_timezone	OffsetDateTime	TIMESTAMP WITH TIME ZONE	2001-01-29 10:15+01:00	2018-10-08 10:30:01.000-08:00

- [Loading Datetime Data](#)
- [Specifying Custom Datetime Formats](#)
- [APIs for Accessing Datetime Data](#)
- [Querying Datetime Data Using PGQL](#)

- [Accessing Datetimes from PGQL Result Sets](#)

14.2.2.1 Loading Datetime Data

You must first load a graph to work with datetime data. See [Loading a Graph Into the Graph Server \(PGX\)](#) for more information on graph loading.

The following example shows how to load a graph that has three vertices representing persons and zero edges.

Example 14-4 Loading Datetime Data

1. Create an EDGE_LIST file `persons.edge_list` as shown:

```
1*Judy,1989-01-15,1989-01-15 10:15-08:00
2*Klara,2001-01-29,2001-01-29 21:30-08:00
3*Pete,1995-08-01,1995-08-01 03:00-08:00
```

2. Create a corresponding graph configuration file `persons.edge_list.json` as shown:

```
{
  "format": "edge_list",
  "uri": "persons.edge_list",
  "vertex_id_type": "long",
  "vertex_props": [
    {
      "name": "name",
      "type": "string"
    },
    {
      "name": "date_of_birth",
      "type": "local_date"
    },
    {
      "name": "timestamp_of_birth",
      "type": "timestamp_with_timezone",
      "format": ["yyyy-MM-dd H[H]:m[m][:s[s]][XXX]" ]
    }
  ],
  "edge_props": [
  ],
  "separator": ", "
}
```

3. You can now load the data as shown in the following code:

Loading the graph data Using JShell

```
opg4j> var graph =
session.readGraphWithProperties("persons.edge_list.json",
"people_graph")
```

Loading the graph data Using Java

```
import oracle.pgx.api.*;
...
PgxGraph graph =
session.readGraphWithProperties("persons.edge_list.json", "people_graph");
```

Loading the graph data Using Python

```
graph =
session.read_graph_with_properties("persons.edge_list.json", graph_name="pe
ople_graph")
```

14.2.2.2 Specifying Custom Datetime Formats

You can also manually specify the datetime format(s) of your data.

By default, PGX tries to parse datetime values using a set of predefined formats. If this fails, an exception like the following is thrown:

```
property timestamp_of_birth: could not parse value at line 1 for property of
temporal type OffsetDateTime using any of the given formats
```

In such a case, you can custom format the datetime data.

There are two ways of specifying datetime formats:

- on a *per-property* basis
- on a *per-type* basis

Property-Specific Datetime format:

You can custom format the property `timestamp_of_birth` used in [Example 14-4](#) to the format `yyyy-MM-dd H[H]:m[m][:s[s]][XXX]` as shown:

Example 14-5 Specifying Property-Specific Datetime format:

```
{
  "name": "timestamp_of_birth",
  "type": "timestamp_with_timezone",
  "format": ["yyyy-MM-dd H[H]:m[m][:s[s]][XXX]"]
}
```

where `yyyy-MM-dd H[H]:m[m][:s[s]][XXX]` specifies that the timestamp values consist of:

- a four-digit year
- a hyphen followed by a two-digit month
- a hyphen followed by a two-digit day
- a space
- an hour, specified as either one or two digits
- a colon followed by a minute, specified as either one or two digits

- an *optional* part that consists of a colon followed by a second that is specified as either one or two digits
- an *optional* timezone

 **Note:**

- `H[H]:m[m]` allows the value `01:15` as well as the value `1:15`.
- `yyyy-MM-dd` allows the value `1989-01-15` but not the value `1989-1-15`. However, if two-digit months and days are needed, a format like `yyyy-M[M]-d[d]` can be used.

Also the format specification takes a *list* of formats. In the preceding example, the list contains only a single format, but you may specify any number of formats. If more than one format is specified, then when parsing the datetime data, the formats are tried from left to right until parsing succeeds. In this way, you can even load data that contains a mixture of values in different formats.

Type-Specific Datetime format:

You can also specify datetime formats on a *per-type* basis. This is useful in cases when there are multiple properties that have the same type as well as the same format because you will only need to specify the datetime format only once.

In case of the per-type specification, the format is used for each vertex or edge property that has the particular type.

The following example shows two type-specific formats (`local_date_format` and `timestamp_with_timezone_format`):

Example 14-6 Specifying Type-Specific Datetime format:

```
...
  "edge_props": [
    ],
    "separator": ",",
    "local_date_format": ["yyyy-MM-dd"],
    "timestamp_with_timezone_format": ["yyyy-MM-dd H[H]:m[m][:s[s]]
[XXX]" ]
  }
```

In the example, properties of type date (`local_date`) have the format `yyyy-MM-dd` while properties of type timestamp with time zone (`timestamp_with_timezone`) have the format `yyyy-MM-dd H[H]:m[m][:s[s]][XXX]`.

 **Note:**

Property-specific formats always overrides type-specific formats. If you specify a type-specific format, and the property of the particular type also has a property-specific format, then only the property-specific format is used to parse the datetime data.

14.2.2.3 APIs for Accessing Datetime Data

The in-memory graph server (PGX) uses the new [Java 8 temporal data types](#) for accessing datetime data through the Java API:

- `date` in PGX maps to `LocalDate` in Java
- `time` in PGX maps to `LocalTime` in Java
- `timestamp` in PGX maps to `LocalDateTime` in Java
- `time with time zone` in PGX maps to `OffsetTime` in Java
- `timestamp with time zone` in PGX maps to `OffsetDateTime` in Java

You can retrieve a date as shown in the following code:

Retrieve a Date Using JShell

```
opg4j> var dateOfBirthProperty = graph.getVertexProperty("date_of_birth")
opg4j> var birthdayOfJudy = dateOfBirthProperty.get(1)
```

Retrieve a Date Using Java

```
import java.time.LocalDate;
import oracle.pgx.api.*;
...
VertexProperty<LocalDate> dateOfBirthProperty =
graph.getVertexProperty("date_of_birth")
LocalDate birthdayOfJudy = dateOfBirthProperty.get(1);
```

Retrieve a Date Using Python

```
date_of_birth_property = graph.get_vertex_property("date_of_birth")
birthday_of_judy = date_of_birth_property.get(1)
```

14.2.2.4 Querying Datetime Data Using PGQL

You can perform various operations such as *extracting* values from datetimes, *comparing* datetime values, and, *converting* between different datetime types. on datetime data using PGQL.

The following are example PGQL queries that show different operations that involve datetime data:

Retrieving Datetime Properties

The following query retrieves the `date_of_birth` and `timestamp_of_birth` properties from the all the persons in the graph.

```
SELECT n.name AS name, n.date_of_birth AS birthday, n.timestamp_of_birth
AS timestamp
```

```
FROM MATCH (n) ON people_graph
ORDER BY birthday
```

The result of the query is as follows:

name	birthday	timestamp
Judy	1989-01-15	1989-01-15T10:15-08:00
Pete	1995-08-01	1995-08-01T03:00-08:00
Klara	2001-01-29	2001-01-29T21:30-08:00

Comparing Datetime Values

The following query provides an overview of persons who are older than other persons in the graph:

```
SELECT n.name AS person1, 'is older than' AS relation, m.name AS person2
FROM MATCH (n) ON people_graph, (m) ON people_graph
WHERE n.date_of_birth > m.date_of_birth
ORDER BY person1, person2
```

The result of the query is as follows:

person1	relation	person2
Klara	is older than	Judy
Klara	is older than	Pete
Pete	is older than	Judy

Extracting Values from Datetimes

The following query extracts the year, month, and day from the date_of_birth values:

```
SELECT n.name AS name
, n.date_of_birth AS dob
, EXTRACT(YEAR FROM n.date_of_birth) AS year
, EXTRACT(MONTH FROM n.date_of_birth) AS month
, EXTRACT(DAY FROM n.date_of_birth) AS day
FROM MATCH (n) ON people_graph
ORDER BY name
```

The result of the query is as follows:

name	dob	year	month	day
Judy	1989-01-15	1989	1	15
Klara	2001-01-29	2001	1	29

```
| Pete | 1995-08-01 | 1995 | 8 | 1 |
+-----+
```

Converting Between Different Types of Datetime Values

The following query converts the `timestamp_of_birth` property into values of the following three datetime types:

- a timestamp (without time zone)
- a time with time zone
- a time (without time zone)

```
SELECT n.name AS name
       , n.timestamp_of_birth AS original_timestamp
       , CAST(n.timestamp_of_birth AS TIMESTAMP) AS utc_timestamp
       , CAST(n.timestamp_of_birth AS TIME WITH TIME ZONE) AS timezoned_time
       , CAST(n.timestamp_of_birth AS TIME) AS utc_time
FROM MATCH (n) ON people_graph
ORDER BY original_timestamp
```

The result of the query is as follows:

```
+-----+
---+
| name | original_timestamp | utc_timestamp | timezoned_time |
| utc_time |
+-----+
---+
| Judy | 1989-01-15T10:15-08:00 | 1989-01-15T18:15 | 10:15-08:00 |
| 18:15 |
| Pete | 1995-08-01T03:00-08:00 | 1995-08-01T11:00 | 03:00-08:00 |
| 11:00 |
| Klara | 2001-01-29T21:30-08:00 | 2001-01-30T05:30 | 21:30-08:00 |
| 05:30 |
+-----+
---+
```

14.2.2.5 Accessing Datetimes from PGQL Result Sets

You can use the following APIs for retrieving datetime values from PGQL result sets.

```
LocalDate getDate(int elementIdx)
LocalDate getDate(String variableName)
LocalTime getTime(int elementIdx)
LocalTime getTime(String variableName)
LocalDateTime getTimestamp(int elementIdx)
LocalDateTime getTimestamp(String variableName)
OffsetTime getTimeWithTimezone(int elementIdx)
OffsetTime getTimeWithTimezone(String variableName)
OffsetDateTime getTimestampWithTimezone(int elementIdx)
OffsetDateTime getTimestampWithTimezone(String variableName)
```

The following example prints the birthdays of all the persons in the graph is as follows:

Retrieving Datetime Values Using JShell

```
opg4j> var resultSet = session.queryPgql("""
    SELECT n.name, n.date_of_birth
    FROM MATCH (n) ON people_graph
ORDER BY n.name
""")
opg4j> while (resultSet.next()) {
...> System.out.println(resultSet.getString(1) + " has birthday " +
resultSet.getDate(2));
...> }
opg4j> resultSet.close()
```

Retrieving Datetime Values Using Java

```
import java.time.LocalDate;
import oracle.pgx.api.*;
...
PgqlResultSet resultSet = session.queryPgql(
    " SELECT n.name, n.date_of_birth\n" +
    " FROM MATCH (n) ON people_graph\n" +
    "ORDER BY n.name");

while (resultSet.next()) {
    System.out.println(resultSet.getString(1) + " has birthday " +
resultSet.getDate(2));
}

resultSet.close();
```

The result of the query is as follows:

```
Judy has birthday 1989-01-15
Klara has birthday 2001-01-29
Pete has birthday 1995-08-01
```

In addition to the Java types from the new `java.time` package, the legacy `java.util.Date` is also supported through the following APIs:

```
Date getLegacyDate(int elementIdx)
Date getLegacyDate(String variableName)
```

Note:

The legacy `java.util.Date` can store dates, times, as well as timestamps, so these two APIs can be used for accessing values of any of the five datetime types.

14.3 Handling Asynchronous Requests in Graph Server (PGX)

This guide explains in detail the asynchronous methods supported by the PGX API.

The PGX API is designed to be asynchronous. This means that all of its core methods ending with *Async* **do not** block the caller thread until the request is completed. Instead, a `PgxFuture` object is instantly returned.

You can perform the following three actions on the returned `PgxFuture` object:

- Block
- Chain
- Cancel
- [Blocking Operation](#)
- [Chaining Operation](#)
- [Cancelling Operation](#)
- [Handling Concurrent Asynchronous Operations](#)

14.3.1 Blocking Operation

You can easily get the result by calling the `get()` method on the `PgxFuture`. The `get()` blocks the caller thread until the result is available:

```
PgxFuture<PgxSession> sessionPromise = instance.createSessionAsync("my-session");
try {
    // block caller thread
    PgxSession session = sessionPromise.get();
    // do something with session
    ...
} catch (InterruptedException e) {
    // caller thread was interrupted while waiting for result
} catch (ExecutionException e) {
    // an exception was thrown during asynchronous computation
    Throwable cause = e.getCause(); // the actual exception is nested
}
```

PGX provides blocking convenience methods for every *Async* method, which calls the `get()` method. Typically, those methods have the same name as the asynchronous method they wrap, but without the *Async* suffix. For example, the preceding code snippet is equal to:

```
try {
    // block caller thread
    PgxSession session = instance.createSession("my-session");
    // do something with session
    ...
} catch (InterruptedException e) {
    // caller thread was interrupted while waiting for result
} catch (ExecutionException e) {
    // an exception was thrown during asynchronous computation
}
```

```

        Throwable cause = e.getCause(); // the actual exception is nested
    }

```

14.3.2 Chaining Operation

The in-memory graph server (PGX) ships a version of Java 8's `CompletableFuture` named `PgxFuture`, a monadic enhancement of the `Future` interface.

The `CompletableFuture` allows chaining of asynchronous computations without polling or the need of deeply nested callbacks (also known as callback hell). All `PgxFuture` instances returned by PGX APIs are instances of `CompletableFuture` and can be chained without the need of Java 8.

```

import java.util.concurrent.CompletableFuture

...

final GraphConfig graphConfig = ...
instance.createSessionAsync("my-session")
    .thenCompose(new Fun<PgxSession, CompletableFuture<PgxGraph>>() {
        @Override
        public CompletableFuture<PgxGraph> apply(PgxSession session) {
            return session.readGraphWithPropertiesAsync(graphConfig);
        }
    }).thenAccept(new Action<PgxGraph>() {
        @Override
        public void accept(PgxGraph graph) {
            // do something with loaded graph
        }
    });

```

The asynchronous chaining in the preceding example is explained as follows:

- The first line in the code makes an asynchronous call to `createSessionAsync()` to create a session. Once the promise is resolved, it returns a `PgxFuture` object, which is the newly created `PgxSession`.
- The code then calls the `.thenCompose()` handler by passing a function which takes the `PgxSession` object as an argument. Inside the function, there is another asynchronous `readGraphWithPropertiesAsync()` request which return another `PgxFuture` object. The outer `PgxFuture` object returned by `.thenCompose()` gets resolved when the `readGraphWithPropertiesAsync()` request completes.
- This is followed by the `.thenAccept()` handler. The function that is passed to `.thenAccept()` does not return anything. Therefore, the future return type of `.thenAccept()` is `PgxFuture<Void>`.

Blocking Versus Chaining

For most use cases, you can block the caller thread. However, blocking can quickly lead to poor performance or deadlocks once things get more complex. As a rule, use

blocking to quickly analyze selected graphs in a sequential manner, for example, in shell scripts or during interactive analysis using the interactive PGX shell.

Use chaining for applications built on top of PGX.

14.3.3 Cancelling Operation

You can cancel a pending request by invoking the `cancel` method of the returned `PgxFuture` instance.

For example:

```
PgxFuture<Object> promise=...
// do something else
promise.cancel(); // will cancel computation
```

Any subsequent calls to `promise.get()` will result in a `CancellationException` being thrown.



Note:

Due to Java's cooperative threading model, it might take some time before PGX actually stops the computation.

14.3.4 Handling Concurrent Asynchronous Operations

Using the `PgxSession#runConcurrently` API provided by the in-memory graph server (PGX), you can submit a list of suppliers of asynchronous APIs to run concurrently in the PGX server.

For example:

```
import oracle.pgx.api.*;

    Supplier<PgxFuture<?>> asyncRequest1 = () ->
session.readGraphWithPropertiesAsync(...);
    Supplier<PgxFuture<?>> asyncRequest2 = () ->
session.getAvailableSnapshotsAsync(...);

    List<Supplier<PgxFuture<?>>> supplierList = Arrays.asList(asyncRequest1,
asyncRequest2);

    //executing the async requests with the enabled optimization feature
List<?> results = session.runConcurrently(supplierList);

    //the supplied requests are mapped to their results and orderly collected
PgxGraph graph = (PgxGraph) results.get(0);
Deque<GraphMetaData> metaData = (Deque<GraphMetaData>) results.get(1);
```


14.4 Graph Client Sessions

The graph server (PGX) assumes there may be multiple concurrent clients, and each client submits request to the shared PGX server independently.

Each session has its own workspace in PGX and is isolated from other sessions.

You can share graphs or properties among sessions.

Creating Sessions

The following methods in the `ServerInstance` class are used to create sessions:

```
PgxFuture<PgxSession> createSessionAsync(String source)
PgxFuture<PgxSession> createSessionAsync(String source, long
idleTimeout, long taskTimeout, TimeUnit unit)
```

PGX offers blocking convenience wrappers around the preceding methods:

Creating a Session Using Java

```
PgxSession createSession(String source)
PgxSession createSession(String source, long idleTimeout, long
taskTimeout, TimeUnit unit)
```

The preceding methods accept the following arguments:

- `source` is any arbitrary string that describes the client. Currently, this string is only used for logging purposes.
- The user can specify the idle timeout (`idleTimeout`) and task timeout (`taskTimeout`) when creating a new session. If these values are not specified, default values are used.
See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information on graph server (PGX) configuration options.

Creating a Session Using Python

```
import pypgx
session = pypgx.get_session()
```

Destroying Sessions

To destroy a session, simply call:

```
session.destroyAsync();
```

Destroying a Session Using Java

```
session.destroy();
```

Destroying a Session Using Python

```
session.destroy()
```

Administrators can destroy sessions by ID using the following code:

```
PgxFuture<Void> promise = instance.killSessionAsync(sessionId);  
instance.killSession(sessionId); // blocking version
```



Note:

Calling administrative methods by default requires special authorization in client/server mode.

When a session is destroyed, PGX reclaims all of the resources associated with the session. Specifically, all transient data is destroyed immediately. See [Managing Transient Data](#) for more information on transient data.

However, PGX may choose to keep the loaded graph instance in memory for caching purposes, especially if a graph instance is shared by multiple clients. In summary, every graph remains in memory until no client is using it.



Note:

A session can be destroyed automatically via the session time-out mechanism. See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for more information on graph server (PGX) configuration options.

14.5 Graph Mutation and Subgraphs

This guide discusses the several methods provided by the graph server (PGX) for mutating graph instances.

You can use the mutation and subgraph methods that are defined in the `PgxGraph` class, to mutate a graph.



Note:

All of the mutating methods create a new graph or snapshot instance as the mutated version of the original graph, rather than mutating the original graph directly.

- [Altering Graphs](#)
- [Simplifying and Copying Graphs](#)
- [Transposing Graphs](#)

- [Undirecting Graphs](#)
- [Advanced Multi-Edge Handling](#)
- [Creating a Subgraph](#)
- [Creating a Bipartite Subgraph](#)
- [Creating a Sparsified Subgraph](#)

14.5.1 Altering Graphs

This section explains the graph alteration mutation used to add or remove vertex and edge providers of a graph.

You can add or remove vertex and edge providers in a graph that has been loaded or created previously. The mutation can either create a new independent graph, or create a new snapshot for the graph.

You must first create a graph-alteration builder to start altering an existing graph. For example, the following code shows how to start a graph alteration on a graph that is stored in a variable `graph`:

Starting a Graph Alteration Using JShell

```
opg-jshell> var alterationBuilder = graph.alterGraph();
```

Starting a Graph Alteration Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.graphalteration.GraphAlterationBuilder;

GraphAlterationBuilder alterationBuilder = graph.alterGraph();
```

The following topics explain in detail on adding and removing vertex and edge providers:

- [Loading Or Removing Additional Vertex or Edge Providers](#)

14.5.1.1 Loading Or Removing Additional Vertex or Edge Providers

You can alter your graph by adding or removing vertex or edge providers from a specific datasource.

Keys in Additionally Loaded Providers

The vertex and edge providers that are loaded must provide the respective keys in accordance with the vertex ID and edge ID strategy of the graph being altered. If the ID strategy is `KEYS_AS_IDS`, the provider must create a key mapping. But, if the ID strategy is `UNSTABLE_GENERATED_IDS`, it must not create the key mapping.

- [Loading Vertex Providers](#)
- [Loading Edge Providers](#)
- [Removing Vertex or Edge Providers](#)
- [Applying the Alteration and Building a Graph or Snapshot](#)

14.5.1.1.1 Loading Vertex Providers

You can add a vertex provider, by calling `alterationBuilder.addVertexProvider(EntityProviderConfig vertexProviderConfig)`. `vertexProviderConfig` is a vertex provider configuration and it provides configuration details such as:

- location of the datasource to load from
- the stored format
- properties of the vertex provider

Additionally, you can also add the provider by calling `alterationBuilder.addVertexProvider(String pathToVertexProviderConfig)` where `pathToVertexProviderConfig` points to a file accessible from the client that contains a JSON representation of a vertex provider configuration.

For example, the vertex provider can be added in the alteration as shown:

```
// loading by indicating the path to the JSON file
alterationBuilder.addVertexProvider("<path-to-vertex-provider-
configuration>");

// or by first loading the content of a JSON file into an
EntityProviderConfig object
EntityProviderConfig vertexProviderConfig = new
AnyFormatEntityProviderConfigFactory().fromPath("<path-to-vertex-provider-
configuration>");
alterationBuilder.addVertexProvider(vertexProviderConfig);
```

Alternatively, the vertex provider configuration can be built programmatically:

```
FileEntityProviderConfigBuilder vertexProviderConfigBuilder = new
FileEntityProviderConfigBuilder().
    setFormat().
    setName("typicalVertexProvider").
    setUris("").
    setKeyColumn(1).
    addProperty("prop1", PropertyType.STRING, null, 2).
    addProperty("prop2", PropertyType.LOCAL_DATE, null, 3);

EntityProviderConfig vertexProviderConfig =
vertexProviderConfigBuilder.build();

alterationBuilder.addVertexProvider(vertexProviderConfig);
```

14.5.1.1.2 Loading Edge Providers

You can add an edge provider, by calling `alterationBuilder.addEdgeProvider(EntityProviderConfig edgeProviderConfig)` where `edgeProviderConfig`. `edgeProviderConfig` is an edge provider configuration and it provides configuration details such as:

- location of the datasource to load from

- the stored format
- properties of the edge provider

The source and destination vertex providers to which it is linked must either be already in the base graph (and not removed in the alteration), or added with the alteration.

Additionally, you can also add the provider by calling `alterationBuilder.addVertexProvider(String pathToVertexProviderConfig)` where `pathToVertexProviderConfig` points to a file accessible from the client that contains a JSON representation of an vertex provider configuration.

For example, an edge provider can be added in the alteration as shown:

```
// loading by indicating the path to the JSON file
alterationBuilder.addEdgeProvider("<path-to-edge-provider-configuration>");

// or by first loading the content of a JSON file into an
EntityProviderConfig object
EntityProviderConfig edgeProviderConfig = new
AnyFormatEntityProviderConfigFactory().fromPath("<path-to-edge-provider-configuration>");
alterationBuilder.addEdgeProvider(edgeProviderConfig);
```

Alternatively, the edge provider configuration can be built programmatically:

```
FileEntityProviderConfigBuilder edgeProviderConfigBuilder = new
FileEntityProviderConfigBuilder().
    setFormat().
    setName("typicalEdgeProvider").
    hasHeader(true).
    setUri("").
    setSourceVertexProvider("typicalVertexProvider").
    setDestinationVertexProvider("anotherTypicalVertexProvider").
    setSourceColumn("source").
    setDestinationColumn("destination").
    setKeyColumn("EID").
    createKeyMapping(true).
    setErrorHandlingOnMissingVertex(OnMissingVertex.IGNORE_EDGE).
    addProperty("cost", PropertyType.DOUBLE);

EntityProviderConfig edgeProviderConfig =
edgeProviderConfigBuilder.build();

alterationBuilder.addEdgeProvider(edgeProviderConfig);
```

14.5.1.1.3 Removing Vertex or Edge Providers

You can remove an edge provider by calling `alterationBuilder.removeEdgeProvider(String edgeProviderName)`, where `edgeProviderName` is the name of the edge provider to be removed from the graph.

Similarly, calling `alterationBuilder.removeVertexProvider(String vertexProviderName)` will result in the graph to not contain that specific vertex

provider. If that vertex provider was the source or destination provider for some edge providers in the base graph, those edge providers should also be removed before the application of the alteration or an exception will be thrown.

It is possible to indicate that the edge providers associated to a removed vertex provider should be automatically removed by calling

```
alterationBuilder.cascadeEdgeProviderRemovals(boolean  
cascadeEdgeProviderRemovals) with cascadeEdgeProviderRemovals set to true.
```

14.5.1.1.4 Applying the Alteration and Building a Graph or Snapshot

You must call `alterationBuilder.build()`, once the different vertex and edge providers have been added or removed in the alteration to actually apply the operation. By calling `alterationBuilder.build()`, a new graph is created and that graph contains all the providers of the base graph excluding the removed providers, and the additionally loaded providers.

You can also call `alterationBuilder.buildNewSnapshot()`, in which case, a new snapshot for the base graph is created and that snapshot contains all the providers of the base graph excluding the removed providers, and the additionally loaded providers.

14.5.2 Simplifying and Copying Graphs

You can create a simplified version of the graph by calling the `simplify()` method.

Simplify a Graph Using Java

```
PgxGraph simplify(Collection<VertexProperty<?, ?>> vertexProps,  
Collection<EdgeProperty<?>> edgeProps, MultiEdges multiEdges,  
SelfEdges selfEdges, TrivialVertices trivialVertices,  
Mode mode, String newGraphName)
```

Simplify a Graph Using Python

```
simplify(self, vertex_properties=True, edge_properties=True, keep_multi_edges=False,  
keep_self_edges=False, keep_trivial_vertices=False, in_place=False,  
name=None)
```

The first two arguments (`vertexProps` and `edgeProps`) list which properties will be copied into the newly created simplified graph instance. PGX provides convenience constants `VertexProperty.ALL`, `EdgeProperty.ALL` and `VertexProperty.NONE`, `EdgeProperty.NONE` to specify all properties or none properties to be stored, respectively.

The next three arguments determine which operations will be performed to simplify the graph.

- `multiEdges`: if `MultiEdges.REMOVE_MULTI_EDGES`, eliminate multiple edges between a source vertex and a destination vertex, that is, leave at most one edge between two vertices. `MultiEdges.KEEP_MULTI_EDGES` indicates to keep them. By default, PGX picks one edge out of the multi-edges and takes its properties. See [Advanced Multi-Edge Handling](#) for more fine-grained control over the edge properties during simplification.
- `selfEdges`: if `SelfEdges.REMOVE_SELF_EDGES`, eliminate every edge whose source and destination are the same vertex. `SelfEdges.KEEP_MULTI_EDGES` indicates to keep them.
- `trivialVertices`: if `TrivialVertices.REMOVE_TRIVIAL_VERTICES`, eliminate all the vertices that have neither incoming edges nor outgoing edges. `TrivialVertices.KEEP_TRIVIAL_VERTICES` indicates to keep them.

The `mode` argument, if set to `Mode.MUTATE_IN_PLACE`, requests that the mutation occurs directly on the specified graph instance without creating a new one. If set to `Mode.CREATE_COPY`, the method will create a new graph instance with the new name in `newGraphName`. If `newGraphName` is omitted (or `null`), PGX will generate a unique graph name.

The return value of this method is the simplified `PgxGraph` instance.

The `Mode.MUTATE_IN_PLACE` option is only applicable if the graph is marked as mutable. Every graph is immutable by default when loaded into PGX. To make a `PgxGraph` mutable, the client should create a private copy of the graph first, using one of the following methods:

Copying a Graph Using Java

```
PgxGraph clone()
PgxGraph clone(String newGraphName)
PgxGraph clone(Collection<VertexProperty<?, ?>> vertexProps,
Collection<EdgeProperty<?>> edgeProps, String newGraphName)
```

Copying a Graph Using Python

```
clone(self, vertex_properties=True, edge_properties=True, name=None)
```

As with `simplify()`, the user can specify optional properties of the graph to copy with `vertexProps` and `edgeProps`. If no properties are specified, all of the original graph's properties will be copied into the new graph instance. The user can specify the name of the newly created graph instance with `newGraphName`.

14.5.3 Transposing Graphs

You can create a transposed version of the graph.

Transposing a Graph Using Java

```
PgxGraph transpose(Collection<VertexProperty<?, ?>> vertexProps,
Collection<EdgeProperty<?>> edgeProps,
Map<String, String> edgeLabelMapping,
Mode mode, String newGraphName)
```

Transposing a Graph Using Python

```
transpose(self, vertex_properties=True, edge_properties=True,
edge_label_mapping=None, in_place=False,
name=None)
```

The `edgeLabelMapping` argument can be used to rename edge labels. If any key in the given map does not exist as an edge label, it will be ignored.

`edgeLabelMapping` argument can also be an empty Map or `null`.

- `null`: if argument is `null`, edge labels from source graph will be removed on transposed graph. (default behavior when using convenience methods).
- `empty Map`: if argument is an empty Map, edge labels from source graph will be neither removed or renamed. Instead, it will be kept as it is in source graph.

See [Simplifying and Copying Graphs](#) for the meaning of the other parameters.

Additionally, the graph server (PGX) provides the following convenience methods from the `PgxGraph` class for the common operation of copying all vertex and edge properties into the transposed graph instance:

- `transpose(Mode mode, String newGraphName)`
- `transpose(String newGraphName)`
- `transpose(Mode mode)`

14.5.4 Undirecting Graphs

The following methods create the undirected version of a graph instance:

Creating Undirected Version of a Graph Using Java

```
PgxGraph undirect()
PgxGraph undirect(String newGraphName)
PgxGraph undirect(MultiEdges multiEdges, SelfEdges selfEdges, TrivialVertices
trivialVertices, Mode mode, String newGraphName)
PgxGraph undirect(Collection<VertexProperty<?, ?>> vertexProps,
Collection<EdgeProperty<?>> edgeProps, MultiEdges multiEdges, SelfEdges selfEdges,
Mode mode, String newGraphName)
```

Creating Undirected Version of a Graph Using Python

```
undirect(self, vertex_properties=True, edge_properties=True, keep_multi_edges=True,
keep_self_edges=True,
        keep_trivial_vertices=True, in_place=False, name=None)
```

The first two methods create an undirected version of the graph while copying all of the vertex properties. `newGraphName` is an optional argument to specify the name of the newly created graph instance.

In contrast, the third and fourth methods concurrently perform *undirecting* and *simplifying* of a graph. See [Simplifying and Copying Graphs](#) for the meaning of each parameter.

All methods return an object of the undirected `PgxGraph` type.

An undirected graph has some restrictions. Some algorithms are only supported on directed graphs or are not yet supported for undirected graphs. Further, PGX does not support to store undirected graphs nor reading from undirected formats. Since the edges do not have a direction anymore, the behavior of `pgxEdge.getSource()` or `pgxEdge.getDestination()` can be ambiguous. In order to provide deterministic results, PGX will always return the vertex with the smaller internal id as source and the other as destination vertex.

14.5.5 Advanced Multi-Edge Handling

Both `simplify()` and `undirect()` support the removal of multi-edges using `MultiEdges.REMOVE_MULTI_EDGES`. If this parameter is set, all multi-edges in this graph are removed, that is, collapsed. Whenever several multi-edges with edge properties are collapsed into one edge, you can choose one of the following two strategies supported by the graph server (PGX) to decide how to treat the corresponding properties:

- Picking
- Merging

If you choose picking, the graph server (PGX) picks one edge out of every set of multi-edges and copies all its properties including the edge label and key into the new graph. In the case

of merging, the graph server (PGX) creates a completely new edge out for every set of multi-edges. PGX determines the properties of these new edges by applying a `MergingFunction` on every property of the multi-edges.

If there are no multi-edges between two vertices, that is, zero or only one edge, the chosen strategy does not have an effect on the outcome. The edge is kept with all its properties as it is.

- [Picking](#)
- [Merging](#)
- [StrategyBuilder in General](#)

14.5.5.1 Picking

This strategy can be used to pick an edge out of multi-edges. The graph server (PGX) allows the user to define several picking criteria. You can pick by:

- Property
- Label
- Edge-ID

Every picking criteria has to be combined with a `PickingStrategyFunction`. PGX supports either `PickingStrategyFunction.MIN` and `PickingStrategyFunction.MAX`, which picks the edge whose property/label/id is either minimal or maximal. If one does not specify a picking criteria, PGX will non-deterministically pick an edge out of the multi-edges.

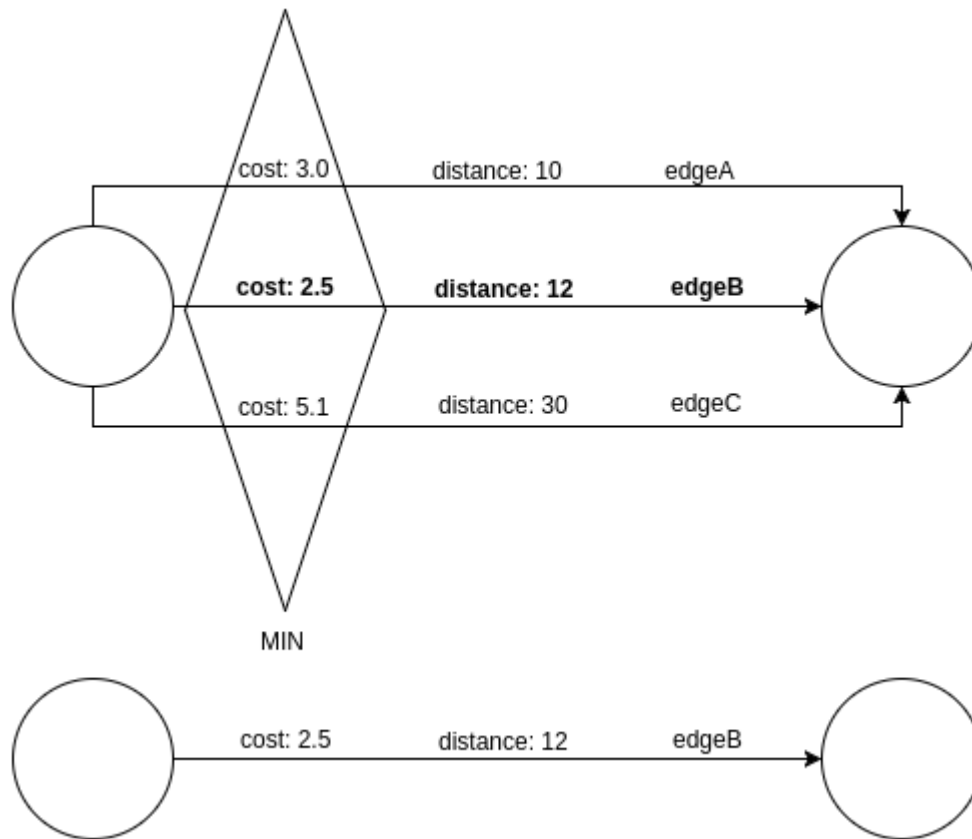
A `PickingStrategy` can be created using a `PickingStrategyBuilder`, which can be retrieved by calling `createPickingStrategyBuilder()` on the target graph.

You can call one of the following functions as per your chosen picking criteria:

```
PickingStrategyBuilder setPickByEdgeId(PickingStrategyFunction
pickingStrategyFunction)
PickingStrategyBuilder setPickByLabel(PickingStrategyFunction
pickingStrategyFunction)
PickingStrategyBuilder setPickByProperty(EdgeProperty edgeProperty,
PickingStrategyFunction pickingStrategyFunction)
PickingStrategyBuilder setPickByProperty(String propertyName,
PickingStrategyFunction pickingStrategyFunction)
```

The following figure shows how PGX picks the edge with the *minimal* cost and takes all its properties.

Figure 14-1 Picking Strategy



14.5.5.2 Merging

This strategy can be used to merge the properties of multi-edges. The graph server (PGX) allows the user to define a `MergingFunction` for every property. Currently, PGX supports the following functions:

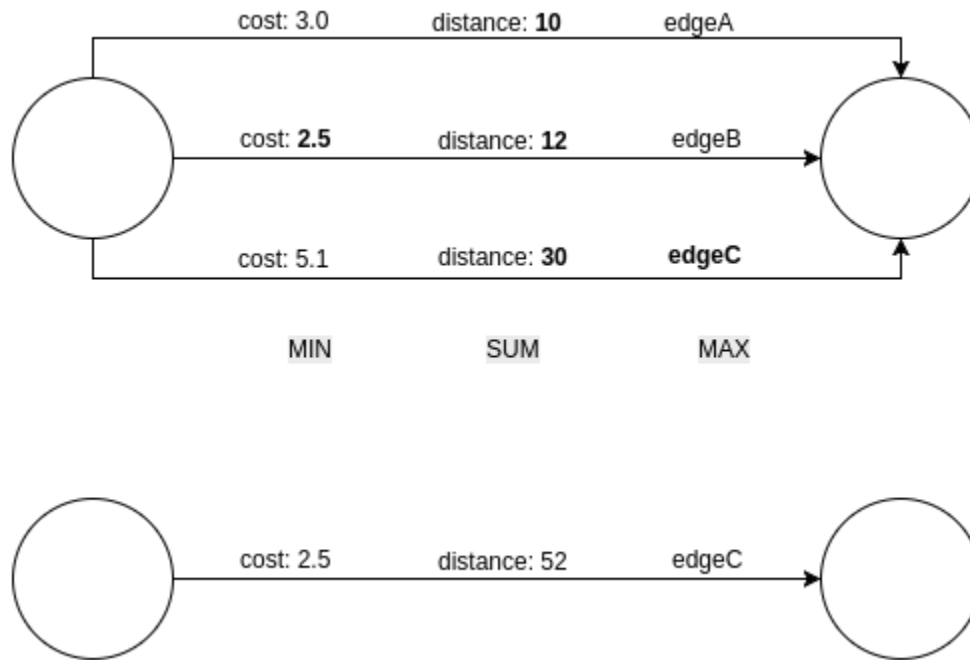
- `MergingFunction.MIN`
- `MergingFunction.MAX`
- `MergingFunction.SUM`

 **Note:**

SUM is only defined on numeric properties.

The following figure shows how the graph server (PGX) merges the different edge properties and labels. It takes the *minimal* cost, the *sum* of distances and the *maximal* edge label.

Figure 14-2 Merging Strategy



14.5.5.3 StrategyBuilder in General

By default, both the StrategyBuilders use the same values as in the convenience methods of `simplify()` and `undirect()`. This includes that all properties are kept by default. If one wants to drop specific properties, one can either use the `dropVertexProperty()` or `dropEdgeProperty()` functions.

```
MutationStrategyBuilder setNewGraphName(String newGraphName)
MutationStrategyBuilder setCopyMode(Mode mode)
MutationStrategyBuilder setTrivialVertices(TrivialVertices
trivialVertices)
MutationStrategyBuilder setSelfEdges(SelfEdges selfEdges)
MutationStrategyBuilder setMultiEdges(MultiEdges multiEdges)
MutationStrategyBuilder
dropVertexProperties(Collection<VertexProperty<?, ?>> vertexProperty)
MutationStrategyBuilder dropEdgeProperties(Collection<EdgeProperty<?>>
edgeProperty)
MutationStrategyBuilder dropVertexProperty(VertexProperty<?, ?>
vertexProperty)
MutationStrategyBuilder dropEdgeProperty(EdgeProperty<?> edgeProperty)
MutationStrategy build()
```

`Simplify()` and `undirect()` can be called using a `MutationStrategy` as follows:

```
MutationStrategy strategy = strategyBuilder.build()
PgxGraph simplifiedGraph graph.simplify(strategy)
//OR
PgxGraph undirectedGraph graph.undirect(strategy)
```

14.5.6 Creating a Subgraph

PGX provides the following methods for creating subgraphs via a filter (see [Filter Expressions](#) for more information) expression:

Creating a Subgraph Using Java

```
PgxGraph filter(GraphFilter graphFilter)
PgxGraph filter(GraphFilter graphFilter, String newGraphName)
PgxGraph filter(Collection<VertexProperty<?, ?>> vertexProps, Collection<EdgeProperty<?
>> edgeProps, GraphFilter graphFilter, String newGraphName)
```

Creating a Subgraph Using Python

```
filter(self, graph_filter, vertex_properties=True, edge_properties=True, name=None)
```

As in the other graph mutating methods, the user has the option to specify the name of the subgraph with the `newGraphName` parameter and of choosing the vertex and edge properties to be copied into the subgraph (`vertexProps` and `edgeProps`). All of the preceding methods return a `PgxGraph` object which represents the created subgraph.

All filter methods require a `GraphFilter` argument containing a filter expression. Fundamentally, the filter expression is a Boolean expression that is evaluated for every vertex and edge in the original graph (in parallel). If the expression is evaluated as `true` for the vertex or edge, then that vertex or edge is included in the subgraph.

See [Creating Subgraphs](#) for more information on how to create subgraphs from graphs loaded into memory.

14.5.7 Creating a Bipartite Subgraph

The graph server (PGX) enables the client to create a bipartite subgraph. The following methods return the created `BipartiteGraph` instance:

Creating a Bipartite Subgraph using Java

```
BipartiteGraph bipartiteSubGraphFromLeftSet(VertexSet<?> vertexSet)
BipartiteGraph bipartiteSubGraphFromLeftSet(VertexSet<?> vertexSet, String
newGraphName)
BipartiteGraph bipartiteSubGraphFromLeftSet(Collection<VertexProperty<?, ?>>
vertexProps, Collection<EdgeProperty<?>> edgeProps, VertexSet<?> vertexSet, String
newGraphName)
BipartiteGraph bipartiteSubGraphFromLeftSet(Collection<VertexProperty<?, ?>>
vertexProps, Collection<EdgeProperty<?>> edgeProps, VertexSet<?> vertexSet, String
newGraphName, String isLeftPropName)
```

Creating a Bipartite Subgraph using Python

```
bipartite_sub_graph_from_left_set(self, vset, vertex_properties=True,
edge_properties=True, name=None, is_left_name=None)
```

These methods require an additional argument `vertexSet`, which points to a set of vertices (see [Using Collections and Maps](#) for more information) whose elements (vertices) would contain the left vertices (that is, vertices on the left side of the bipartite graph that have only edges to vertices on the right side) in the resulting bipartite graph.

When creating the bipartite subgraph, PGX automatically inserts an additional boolean vertex property `isLeft`. The value of this property is set `true` for the left vertices and `false` for the

right vertices in the bipartite subgraph. The name of the `isLeft` vertex property can be obtained with `getIsLeftPropertyAsync()` on the returned `BipartiteGraph` object.

The user has the option to specify a name for the newly created graph (`newGraphName`) as well as a custom name for the Boolean left-vertex indicating property (`isLeftPropName`). The user can also specify the vertex and edge properties to be copied into the newly created graph instance (`vertexProps` and `edgeProps`).

14.5.8 Creating a Sparsified Subgraph

The graph server (PGX) supports creating a sparsified subgraph of a graph:

Creating a Sparsified Subgraph Using Java

```
PgxGraph sparsify(double e)
PgxGraph sparsify(double e, String newGraphName)
PgxGraph sparsify(Collection<VertexProperty<?, ?>> vertexProps,
Collection<EdgeProperty<?>> edgeProps, double e, String newGraphName)
```

Creating a Sparsified Subgraph Using Python

```
sparsify(self, sparsification, vertex_properties=True, edge_properties=True,
name=None)
```

The `double` argument `e` is the sparsification coefficient with a value between 0.0 and 1.0.

The user again has the option to specify the name for the newly created graph (`newGraphName`) as well as the vertex and edge properties to be copied into the newly created graph instance (`vertexProps` and `edgeProps`).

The returned `PgxGraph` object represents a sparsified subgraph which has fewer edges than the original graph.

14.6 Managing Transient Data

This guide discusses how to handle transient properties and collections.

The graph server (PGX) allows each client to maintain its own isolated workspace, called session. Clients may create additional data objects in their own session, which they can then use for analysis.

- [Managing Transient Properties](#)
- [Managing Collections and Scalars](#)

14.6.1 Managing Transient Properties

The graph server (PGX) adopts the Property Graph data model. Once a graph is loaded into PGX, the graph instance itself and its original properties are set as immutable. However, the client can create and attach additional properties to the graph dynamically. These extra properties are referred to as *transient* properties and are mutable by the client

The methods for creating transient properties are available in `PgxGraph`:

```
PgxFuture<VertexProperty<ID, V>> createVertexPropertyAsync(PropertyType type)
PgxFuture<VertexProperty<ID, V>> createVertexPropertyAsync(PropertyType
type, String name)
PgxFuture<EdgeProperty<V>> createEdgePropertyAsync(PropertyType type)
PgxFuture<EdgeProperty<V>> createEdgePropertyAsync(PropertyType type, String
name)
```

In the preceding code:

- `PropertyType`: is an enum for the data type of the property, which must be one of the primitive types supported by PGX.
- `name`: is an optional argument to assign a unique name to the newly created property. If no name is specified, PGX will assign one to the client.

Note:

Names must be unique. There cannot be two different vertex or edge properties for the same graph and with the same name.

All methods return a `Property` object, which represent the newly created transient property. Both of the underlying classes, `VertexProperty<ID, V>` and `EdgeProperty<V>`, are parametrized with the value type `V` the property holds. `V` matches the given `PropertyType`. `VertexProperty<ID, V>` is additionally parametrized with the vertex ID type. This is due to PGX support of several types of vertex identifiers. See our graph configuration chapter on how to specify the vertex ID type of a graph. `EdgeProperty<V>` is not parameterized with the edge ID type, because PGX only supports edge identifiers of type `long`.

Creating Transient Properties Using Java

```
GraphConfig config = GraphConfigBuilder.forFileFormats(...)
...
.setVertexIdType(IdType.LONG)
...
.build();

PgxGraph G = session.readGraphWithProperties(config);
VertexProperty<Long, String> p1 =
G.createVertexProperty(PropertyType.STRING);
EdgeProperty<Double> p2 = G.createEdgeProperty(PropertyType.DOUBLE);
```

Creating Transient Properties Using Python

```
G = session.read_graph_with_properties(config)
p1 = G.create_vertex_property("string")
p2 = G.create_edge_property("double")
```

To delete a transient property from the session, call `destroyAsync()` (or `destroy()`) on the property object.

14.6.2 Managing Collections and Scalars

The client can create graph-bound vertex and edge collections to use during the analysis with the following methods in `PgxGraph`:

```
PgxFuture<VertexSequence<E>> createVertexSequenceAsync()
PgxFuture<VertexSequence<E>> createVertexSequenceAsync(String name)
PgxFuture<VertexSet<E>> createVertexSetAsync()
PgxFuture<VertexSet<E>> createVertexSetAsync(String name)
PgxFuture<EdgeSequence> createEdgeSequenceAsync()
PgxFuture<EdgeSequence> createEdgeSequenceAsync(String name)
PgxFuture<EdgeSet> createEdgeSetAsync()
PgxFuture<EdgeSet> createEdgeSetAsync(String name)
```

PGX also supports scalar collections such as `set` and `sequence`. Each of these collections can hold elements of various primitive data types like `INTEGER`, `LONG`, `FLOAT`, `DOUBLE` or `BOOLEAN`. Scalar collections are session-bound and can be created with the following methods in `PgxSession`:

```
PgxFuture<ScalarSet<T>> createSetAsync(PropertyType contentType, String
name)
PgxFuture<ScalarSequence<T>> createSequenceAsync(PropertyType
contentType, String name)
PgxFuture<ScalarSet<T>> createSetAsync(PropertyType contentType)
PgxFuture<ScalarSequence<T>> createSequenceAsync(PropertyType
contentType)
```

In the preceding code, the optional argument (`name`) specifies the name of the newly created collection. If omitted, PGX chooses a name for the client. As with properties, the collections holding vertices are parametrized with the `ID` type of the vertices. Refer to graph configuration chapter to learn how to specify the vertex ID type of a graph.

The return value is the collection object which points to the newly created empty collection.

To drop a collection from the session, call `destroyAsync()` (or `destroy()`) on the collection object.

To check which collections are currently allocated for a graph you can use the following method:

```
PgxFuture<Map<String, PgxCollection<? extends PgxEntity<?>, ?>>>
getCollectionsAsync()
```

Checking Collections for a Graph Using Java

```
Map<String, PgxCollection<? extends PgxEntity<?>, ?>> getCollections()
```

Checking Collections for a Graph Using Python

```
get_collections(self)
```

The returned map contains the names of the collections as keys and the collections as values. The collections can be casted to the matching collection subclass.

PGX supports special `Map` collection types and allows users to map between different data types (`oracle.pgx.common.types.PropertyType`). Maps can be created using `PgxGraph` or `PgxSession` APIs, the difference is that the latter supports only non graph-related types, and that the created maps directly depend on the session:

```
PgxFuture<PgxMap<K, V>> createMapAsync(PropertyType keyType, PropertyType valType)
PgxFuture<PgxMap<K, V>> createMapAsync(PropertyType keyType, PropertyType valType, String mapName)
```

Creating Map Collections Using Java

```
PgxMap<K, V> createMap(PropertyType keyType, PropertyType valType)
PgxMap<K, V> createMap(PropertyType keyType, PropertyType valType, String mapName)
```

Similarly, scalar variables can be created in the client session using the following methods:

```
PgxFuture<Scalar<T>> createScalarAsync(PropertyType type, String newScalarName)
PgxFuture<Scalar<T>> createScalarAsync(PropertyType type)
```

Creating Scalar Variables Using Java

```
Scalar<T> createScalar(PropertyType type, String newScalarName)
Scalar<T> createScalar(PropertyType type)
```

Creating Scalar Variables Using Python

```
create_scalar(self, data_type, name=None)
```

These collections and scalar variables can then be passed as arguments to graph algorithms. See [Using Custom PGX Graph Algorithms](#) for more information.

14.7 Graph Versioning

This guide describes the different ways to work with graph snapshots.

A graph can have multiple snapshots associated with it, reflecting different versions of the graph. All snapshots of a graph have the same graph configuration associated.

The following topics explain the various operations you can perform on graph snapshots:

- [Configuring the Snapshots Source](#)
- [Creating a Snapshot via Refreshing](#)
- [Creating a Snapshot via ChangeSet](#)
- [Checking Out the Latest Snapshots of a Graph](#)

- [Checking Out Different Snapshots of a Graph](#)
- [Directly Loading a Specific Snapshot of a Graph](#)

14.7.1 Configuring the Snapshots Source

Snapshots can be created from two sources: **Refreshing** and **ChangeSet**.

Refreshing is available for graphs that are read from a persistent data source, that is, a file. When the data source has changed with respect to the version stored in the graph server (PGX), it can be read again manually by calling the `PgxSession.readGraphWithProperties()` method. Similarly, if auto-refresh is set for the graph, the graph server (PGX) automatically reads the data source and creates new snapshots when the data source has changed.

Instead, a `ChangeSet` is a set of changes to a graph that the user creates and populates via the PGX `ChangeSet` API. Once a `ChangeSet` is created and populated with the desired changes, the user can simply call `GraphChangeSet.buildNewSnapshot()` to create a new snapshot for the graph. In this way, you are empowered to integrate changes coming from any source into the graph and build snapshots out of them.

Only one source of snapshots is allowed for a single graph and is chosen during graph configuration via the `snapshots_source` option, which can be set to either `REFRESH` or `CHANGE_SET`. In case the `snapshots_source` option is not explicitly set by the user, the following default settings apply:

- If the graph is from a persistent data source, the default value is `REFRESH`, so that snapshots can be created only by calling `PgxSession.readGraphWithProperties()` (or via auto-refresh, if configured).
- If the graph is transient, that is, built from a graph builder, the default value is `CHANGE_SET`, since the graph is not backed by a persistent data source from which changes can be read. It is for this reason, `CHANGE_SET` is the only admissible value for transient graphs.

Additionally, the following restrictions apply:

- If auto-refresh is enabled, then snapshots come from reading the backing data source and hence only `REFRESH` is admissible for the `snapshots_source` option.
- If the user attempts to create snapshots in a way that is different from the configuration (for example, by calling `GraphChangeSet.buildNewSnapshot()` when the graph's `snapshots_source` is `REFRESH`), the operation is invalid and an exception is thrown.

14.7.2 Creating a Snapshot via Refreshing

You can create a snapshot via refreshing by performing the following steps:

1. Create a session and load the graph into memory.
2. Check the available snapshots of the graph with `PgxSession.getAvailableSnapshots()` method.

Get the Available Snapshots Using JShell

```
opg4j> session.getAvailableSnapshots(G)
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669
```

```
(2016-01-20 10:38:42.669), creationTimestamp=1453315122685 (2016-01-20
10:38:42.685), vertexIdType=integer, edgeIdType=long]
```

Get the Available Snapshots Using Java

```
Deque<GraphMetaData> snapshots = session.getAvailableSnapshots(G);
for( GraphMetaData metaData : snapshots ) {
    System.out.println( metaData );
}
```

Get the Available Snapshots Using Python

```
snapshots = session.get_available_snapshots(G)
for metadata in snapshots:
    print(metadata)
```

3. Edit the source file to contain an additional vertex and an additional edge or insert two rows in the database.
4. Reload the updated graph within the same session as you loaded the original graph. A new snapshot is created.

Load an Updated Graph Using JShell

```
opg4j> var G = session.readGraphWithProperties( G.getConfig(), true )
==> PGX Graph named 'sample_2' bound to PGX session 'a1744e86-65fb-4bd1-
b2dc-5458b20954a9' registered at PGX Server Instance running in embedded
mode
```

```
pgx> session.getAvailableSnapshots(G)
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669
(2016-01-20 10:38:42.669), creationTimestamp=1453315122685 (2016-01-20
10:38:42.685), vertexIdType=integer, edgeIdType=long]
==> GraphMetaData [getNumVertices()=5, getNumEdges()=5, memoryMb=3,
dataSourceVersion=1452083654000, creationRequestTimestamp=1453314938744
(2016-01-20 10:35:38.744), creationTimestamp=1453314938833 (2016-01-20
10:35:38.833), vertexIdType=integer, edgeIdType=long]
```

Load an Updated Graph Using Java

```
G = session.readGraphWithProperties( G.getConfig(), true );

Deque<GraphMetaData> snapshots = session.getAvailableSnapshots( G );
```

Load an Updated Graph Using Python

```
G = session.read_graph_with_properties(G.config,update_if_not_fresh=True)
```

Note that there are two `GraphMetaData` objects in the call for available snapshots, one with 4 vertices and 4 edges and one with 5 vertices and 5 edges.

5. Verify that the graph variable points to the newly loaded graph using `getNumVertices()` and `getNumEdges()` methods.

Get the Number of Vertices and Edges in a Graph Using JShell

```
opg4j> G.getNumVertices()
==> 5
```

```
opg4j> G.getNumEdges()
==> 5
```

Get the Number of Vertices and Edges in a Graph Using Java

```
int vertices = G.getNumVertices();
long edges = G.getNumEdges();
```

Get the Number of Vertices and Edges in a Graph Using Python

```
vertices = G.num_vertices
edges = G.num_edges
```

14.7.3 Creating a Snapshot via ChangeSet

You can create a graph snapshot with ChangeSet via the PGX Java API. When you want to create the graph from a persistent data source, you can use `PgxSession.readGraphWithProperties()` with the `snapshots_source` configuration option set to `CHANGE_SET`.

You can create a snapshot via ChangeSet by performing the following steps:

1. Create a snapshot of a transient graph from database:
Creating a Graph Snapshot Using JShell

```
opg4j> String statement =
...> "CREATE PROPERTY GRAPH bank_graph "
...> + "VERTEX TABLES ( BANK_NODES as ACCOUNTS "
...> + "KEY (ID) "
...> + "LABEL ACCOUNTS "
...> + "PROPERTIES (ID, LABEL) "
...> + ") "
...> + "EDGE TABLES ( BANK_EDGES_AMT "
...> + "KEY (SRC_ID, DEST_ID, AMOUNT) "
...> + "SOURCE KEY (SRC_ID) REFERENCES ACCOUNTS "
...> + "DESTINATION KEY (DEST_ID) REFERENCES ACCOUNTS "
...> + "LABEL TRANSFERS "
...> + "PROPERTIES (SRC_ID, DEST_ID, AMOUNT, LABEL) "
...> + ") ";
statement ==> "CREATE PROPERTY GRAPH bank_graph VERTEX TABLES
( BANK_NODES as ACCOUNTS KEY (ID) LABEL ACCOUNTS PROPERTIES (ID,
LABEL) ) EDGE TABLES ( BANK_EDGES_AMT KEY (SRC_ID, DEST_ID, AMOUNT)
SOURCE KEY (SRC_ID) REFERENCES ACCOUNTS DESTINATION KEY (DEST_ID)
REFERENCES ACCOUNTS LABEL TRANSFERS PROPERTIES (SRC_ID, DEST_ID,
AMOUNT, LABEL) ) "
opg4j> session.executePgql(statement);
opg4j> session.getGraph("bank_graph");
```

Creating a Graph Snapshot Using Java

```
import oracle.pgx.api.*;
pgqlStmt = pgqlConn.createStatement();
String pgql = "CREATE PROPERTY GRAPH " + bank_graph + " "+
"VERTEX TABLES ( BANK_NODES as ACCOUNTS " +
"KEY (ID) " +
"LABEL ACCOUNTS " +
```

```
"PROPERTIES (ID, LABEL) " +
") " +
"EDGE TABLES ( BANK_EDGES_AMT " +
"KEY (SRC_ID, DEST_ID, AMOUNT) " +
"SOURCE KEY (SRC_ID) REFERENCES ACCOUNTS " +
"DESTINATION KEY (DEST_ID) REFERENCES ACCOUNTS " +
"LABEL TRANSFERS " +
"PROPERTIES (SRC_ID, DEST_ID, AMOUNT, LABEL) " +
)";
pgqlStmt.execute(pgql);
```

2. Create a ChangeSet from `graph` and populate it. The following example shows adding a new edge between vertices 1 and 4:

Creating a ChangeSet Using JShell

```
opg4j> var changeSet = graph.<Integer>createChangeSet()
opg4j> changeSet.addEdge(6, 1, 4)
```

Creating a ChangeSet Using Java

```
import oracle.pgx.api.*;
GraphChangeSet<Integer> changeSet = graph.createChangeSet();
changeSet.addEdge(6, 1, 4);
```

Creating a ChangeSet Using Python

```
changeSet = graph.create_change_set()changeSet.add_edge(1,4,6)
```

3. Create a second snapshot using `GraphChangeSet.buildNewSnapshot()` as shown in the following code:

Creating a ChangeSet with GraphChangeSet API Using JShell

```
opg4j> var secondSnapshot = changeSet.buildNewSnapshot()
opg4j> session.getAvailableSnapshots(secondSnapshot).size()
==> 2
```

Creating a ChangeSet with GraphChangeSet API Using Java

```
PgxGraph secondSnapshot = changeSet.buildNewSnapshot();
System.out.println( session.getAvailableSnapshots(secondSnapshot).size() )
;
```

Creating a ChangeSet with GraphChangeSet API Using Python

```
second_snapshot = change_set.build_new_snapshot()
print(len(session.get_available_snapshots()))
```

Thus two snapshots, referenced via the variables `graph` and `secondSnapshot` are created.

14.7.4 Checking Out the Latest Snapshots of a Graph

With multiple snapshots of a graph being available and regardless of their source, you can check out a specific snapshot using the `PgxSession.setSnapshot()` method. You can use the `LATEST_SNAPSHOT` constant of `PgxSession` to easily check out the latest available snapshot, as shown in the following example:

Get the Latest Snapshot Using JShell

```
opg4j> session.setSnapshot( G, PgxSession.LATEST_SNAPSHOT )
==> null
opg4j> session.getCreationTimestamp()
==> 1453315122685
```

Get the Latest Snapshot Using Java

```
session.setSnapshot( G, PgxSession.LATEST_SNAPSHOT );
System.out.println( session.getCreationTimestamp() )
```

See the printed timestamp to verify the most recent snapshot.

14.7.5 Checking Out Different Snapshots of a Graph

You can also check out a specific snapshot, again using the `PgxSession.setSnapshot()`.

For example, consider the following two snapshots of a graph:

```
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669
(2016-01-20 10:38:42.669), creationTimestamp=1453315122685 (2016-01-20
10:38:42.685), vertexIdType=integer, edgeIdType=long]
==> GraphMetaData [getNumVertices()=5, getNumEdges()=5, memoryMb=3,
dataSourceVersion=1452083654000, creationRequestTimestamp=1453314938744
(2016-01-20 10:35:38.744), creationTimestamp=1453314938833 (2016-01-20
10:35:38.833), vertexIdType=integer, edgeIdType=long]
```

To check out a specific snapshot of the graph, you must pass the `creationTimestamp` of the snapshot you want to load to `setSnapshot()`.

For example, if `G` is pointing to the newest graph with 5 vertices and 5 edges, but you want to analyze the older graph, you need to set the snapshot to 1453315122685.

Get a Specific Snapshot Using JShell

```
opg4j> G.getNumVertices()
==> 5
opg4j> G.getNumEdges()
==> 5
opg4j> session.setSnapshot( G, 1453315122685 )
==> null
opg4j> G.getNumVertices()
```

```
==> 4
opg4j> G.getNumEdges()
==> 4
```

Get a Specific Snapshot Using Java

```
session.setSnapshot(G,1453315122685);
```

Get a Specific Snapshot Using Python

```
session.set_snapshot(G,1453315122685)
```

Note that setting the snapshot, changes the number of vertices and edges from 5 to 4.

Alternatively, you can also retrieve the creation timestamp of each snapshot from its associated `GraphMetaData` object via the `GraphMetaData.getCreationTimestamp()` method. The easiest way to get the `GraphMetaData` information of all the snapshots is to use the the `PgxSession.getAvailableSnapshots()` method, which returns a collection of `GraphMetaData` information of each snapshot ordered by creation timestamp from the most recent to the oldest.

14.7.6 Directly Loading a Specific Snapshot of a Graph

You can also load a specific snapshot of a graph directly using the `PgxSession.readGraphAsOf()` method. This is a shortcut for loading a graph with `readGraphWithProperties()` followed by a `setSnapshot()`.

Consider two snapshots of a graph that are already loaded into the PGX session. The following example shows how to get a reference to a specific snapshot:

1. Get a graph configuration for the graph
Get the Graph Configuration Using JShell

```
opg4j> var config =
GraphConfigFactory.forAnyFormat().fromPath("<path_to_json>")
==> { "format": "adj_list", ... }
```

Get the Graph Configuration Using Java

```
GraphConfig config =
GraphConfigFactory.forAnyFormat().fromPath("<path_to_json>");
```

Get the Graph Configuration Using Python

```
config = GraphConfigFactory.for_any_format().from_path("<path_to_json>")
```

2. Check the loaded snapshots for this graph config using `getAvailableSnapshots()`:
Get the Available Snapshots Using JShell

```
opg4j> session.getAvailableSnapshots(G)
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669
```

```
(2016-01-20 10:38:42.669), creationTimestamp=1453315122685  
(2016-01-20 10:38:42.685), vertexIdType=integer, edgeIdType=long]  
==> GraphMetaData [getNumVertices()=5, getNumEdges()=5, memoryMb=3,  
dataSourceVersion=1452083654000,  
creationRequestTimestamp=1453314938744 (2016-01-20 10:35:38.744),  
creationTimestamp=1453314938833 (2016-01-20 10:35:38.833),  
vertexIdType=integer, edgeIdType=long]
```

Get the Available Snapshots Using Java

```
Deque<GraphMetaData> snapshots = session.getAvailableSnapshots(G);
```

Get the Available Snapshots Using Python

```
session.get_available_snapshots(G)
```

3. Check out the snapshot of the graph which has 4 vertices and 4 edges and having the timestamp 1453315122685:

Load a Specific Snapshot Using JShell

```
opg4j> var G = session.readGraphAsOf( config, 1453315122685 )  
==> PGX Graph named 'sample' bound to PGX session  
'a1744e86-65fb-4bd1-b2dc-5458b20954a9' registered at PGX Server  
Instance running in embedded mode  
opg4j> G.getNumVertices()  
==> 4  
opg4j> G.getNumEdges()  
==> 4
```

Load a Specific Snapshot Using Java

```
PgxGraph G = session.readGraphAsOf( config, 1453315122685 )a
```

Load a Specific Snapshot Using Python

```
G = read_graph_as_of(config, creation_timestamp=1453315122685)
```

14.8 Labels and Properties

You can perform various actions on the graph property and label values by executing PGQL queries.

- [Setting and Getting Property Values](#)
- [Getting Label Values](#)

14.8.1 Setting and Getting Property Values

Getting Property Values

You can obtain the vertex or edge property values by executing a `SELECT PGQL` query on the graph.

For example:

Getting a Property Value Using JShell

```
opg4j> session.queryPgql("SELECT e.src_id, e.dest_id, e.amount FROM MATCH
(n:Account) -[e:Transfers]-> (m:Account) on bank_graph").print();
```

Getting a Property Value Using Java

```
...
...
PgxGraph g = session.getGraph("bank_graph");
String query =
    "SELECT e.src_id, e.dest_id, e.amount FROM MATCH (n:Account) -
[e:Transfers]-> (m:Account)";
g.queryPgql(query).print();
```

The resulting property values may appear as:

```
+-----+
| src_id | dest_id | amount |
+-----+
| 1      | 259    | 1000   |
| 1      | 418    | 1000   |
| 1      | 584    | 1000   |
| 1      | 644    | 1000   |
| 1      | 672    | 1000   |
| 2      | 493    | 1000   |
| 2      | 546    | 1000   |
| 2      | 693    | 1000   |
| 2      | 833    | 1000   |
| 2      | 840    | 1000   |
+-----+
```

Setting Property Values

You can set the vertex or edge property values by executing insert or update `PGQL` queries on the graph.

For example, to set a new vertex account ID on a graph using `INSERT` query:

Setting a Property Value Using JShell

```
opg4j> PgxGraph g = session.getGraph("bank_graph_analytics");
g ==> PgxGraph[name=bank_graph_analytics,N=1000,E=5001,created=1616312153556]
opg4j> PgxGraph g_mutable = g.clone("bank_graph_analytics_copy");
```



```
g_mutable ==>
PgxGraph[name=bank_graph_analytics_copy,N=1000,E=5001,created=1616312413
799]
opg4j> g_mutable.executePgql("INSERT VERTEX v LABELS (Accounts)
PROPERTIES ( v.id = 1001)");
```

Setting a Property Value Using Java

```
...
...
PgxGraph g1 =
session.readGraphWithProperties("bank_graph_analytics.json");
PgxGraph g2 = g1.clone("bank_graph_analytics_copy");
g2.executePgql("INSERT VERTEX v " +
"          LABELS ( Accounts ) " +
"          PROPERTIES ( v.id = 1001 )");
```

14.8.2 Getting Label Values

You can retrieve the vertex or edge label values of a graph as shown:

```
PgxGraph g = session.getGraph("bank_graph_analytics");
String query =
    "SELECT LABEL(v), COUNT(*) "
    + "FROM MATCH (v) "
    + "GROUP BY LABEL(v) "
    + "ORDER BY COUNT(v) DESC";
PgqlResultSet resultSet = g.queryPgql(query);
resultSet.print();
```

The result may appear as shown:

```
+-----+
| LABEL(n) | COUNT(*) |
+-----+
| ACCOUNT  | 1000     |
+-----+
```

14.9 Filter Expressions

This guide explains the usage of filter expressions.

Filter expressions are applied in the following scenarios:

- **Path-Finding:** Include only specific vertices and edges in a path
- **Sub-Graphs:** Include only specific vertices and edges in a subgraph
- **Set creation:** Create a vertex or edge set and include only specific vertices or edges

There are two types of filter expressions:

- **Vertex filters::** Evaluated on each vertex

- **Edge filters:** Evaluated on each edge, including the two vertices it connects.

These filter expressions will evaluate to `true` if the current edge or vertex matches the expression or to `false` if it does not. Filter expressions are stateless and side-effect free.

The following short example below will evaluate to `true` for all edges where the source vertex's string property name is "PGX".

```
src.name="PGX"
```

- [Syntax](#)
- [Type System](#)
- [Path Finding Filters](#)
- [Subgraph Filters](#)
- [Operations on Filter Expressions](#)

14.9.1 Syntax

Trivial Expressions

Always evaluates to `true`:

```
true
```

Always evaluates to `false`:

```
false
```

Constants

Legal constants are integer, long and floating point numbers of single and double precision as well as strings literals and `true` and `false`. Long constants need to be suffixed with `L` or `l`. Floating point numbers are treated as double precision numbers by default. To force a certain precision you can use `f` or `F` for single precision and `d` or `D` for double precision floating point numbers. String literals are UTF-8 character sequences, surrounded by single or double quotation marks.

```
25
4294967296L
0.62f
0.33d
"Double quoted string"
'Single quoted string'
```

Vertex and Edge Identifiers

Depending on the filter type, different identifiers are valid.

Vertex Filter

Vertex filter expressions have only one keyword that addresses the vertex in the current context.

`vertex` denotes the vertex that is currently being evaluated by the filter expression.

`vertex`

Edge Filter

Edge filter expressions have several keywords that addresses the edge or its vertices in the current context.

`edge` denotes the edge that is currently being evaluated by the filter expression.

`edge`

`dst` denotes the destination vertex of the current edge. `dst` is only valid in the subgraph context.

`dst`

`src` denotes the source vertex of the current edge. `src` is only valid in the subgraph context.

`src`

Properties

Filter expressions can access the values of vertex and edge properties.

`<id>.<property>`

where:

- `<id>`: is any vertex or edge identifier (that is, `src`, `dst`, `vertex`, `edge`).
- `<property>`: is the name of a vertex or edge property.

Note:

This has to be the name of an edge property if the identifier is `edge`. Otherwise it has to be a vertex property.

If the property name is a reserved name in the filter expression syntax or contains spaces, it must be quoted in single or double quotes.

The following code accesses the 'cost' property of the source vertex.

`src.cost`

Temporal properties support values comparison (constants and property values) using special constructors. The default temporal formats are shown in the following table:

Table 14-4 Default Temporal Formats

Property Type	Constructor
DATE	<code>date ('yyyy-MM-dd HH:mm:ss')</code>
LOCAL_DATE	<code>date 'yyyy-MM-dd'</code>
TIME	<code>time 'HH:mm:ss'</code>
TIME_WITH_TIMEZONE	<code>time 'HH:mm:ss+/-XXX'</code>

Table 14-4 (Cont.) Default Temporal Formats

Property Type	Constructor
TIMESTAMP	timestamp 'yyyy-MM-dd HH:mm:ss'
TIMESTAMP_WITH_TIMEZONE	timestamp 'yyyy-MM-dd HH:mm:ss+/-XXX'

The following expression accesses the property 'timestamp_withTZ' of an edge and checks if it is equal to 3/27/2007 06:00+01:00.

```
edge.timestamp_withTZ = timestamp'2007-03-2706:00:00+01:00'
```

Note:

Properties of type date can only be checked for equality. date type usage is deprecated since version 2.5, instead use local date or timestamp types that support all operations.

Methods

Filter expressions support the following functions:

Degree Functions

1. outDegree() returns the number of outgoing edges of the vertex identifier. degree() is a synonym for outDegree.

```
int <id>.degree()
int <id>.outDegree()
```

The following example determines whether the out-degree of the source vertex is greater than three:

```
src.degree() > 3
```

2. inDegree() returns the number of incoming edges of the vertex identifier.

```
int <id>.inDegree()
```

Label Functions

1. hasLabel() checks if a vertex has a label.

```
boolean <id>.hasLabel('<label>')
```

The following example determines whether a vertex has the label "city":

```
vertex.hasLabel('city')
```

2. label() returns the label of an edge.

```
string <id>.label()
```

The following expression checks whether the label of an edge is "clicked_by":

```
edge.label() = 'clicked_by'
```

Relational Expressions

To compare values (e.g., property values or constants), filter expressions provide the comparison operators listed below. Note: Both `==` and `=` are synonyms.

```
==
=
!=
<
<=
>
>=
```

The following example checks whether the "cost" property of the source vertex is lower than or equals to 1.23.

```
src.cost <= 1.23
```

Vertex ID Comparison

It is also possible to filter for vertices with a specific vertex ID.

```
<id> = <vertex_id>
```

The following example determines whether the source vertex of an edge has the vertex ID "San Francisco"

```
src = "San Francisco"
```

Regular Expressions

Strings can be matched using regular expressions.

```
<string expression> =~ '<regexexpression>'
```

The following example checks if the edge label starts with a lowercase letter and ends with a number:

```
edge.label() =~ '^[a-z].[0-9]$'
```



Note:

The syntax followed for the pattern on the right-hand side, is [Java REGEX](#).

Type Conversions

The following syntax allows converting the type of `<expression>` to `<type>`.

```
(<type>) <expression>
```

The following example converts the value of the 'cost' property of the source vertex to an integer value:

```
(int) src.cost
```

Boolean Expressions

Filter expressions can be composed to form other filter expressions. This can be done using the Boolean operators `&&` (and), `||` (or) and `!` (not).

**Note:**

Only boolean operands can be composed.

```
(! true) || false
edge.cost < INF && dst.visited = false
src.degree() < 10 || !(dst.visited)
```

Arithmetic Expressions

Any numeric expression can be combined using arithmetic expressions. The available arithmetic operators are: +, -, *, /, %.

**Note:**

These operators only work on numeric operands.

```
1 + 5
-vertex.degree()
edge.cost * 2 > 5
src.value * 2.5 = (dst.inDegree() + 5) / dst.outDegree()
```

Operator Precedence

Operator precedences are shown in the following list, from highest precedence to the lowest. An operator on a higher level is evaluated before an operator on a lower level.

1. + (unary plus), - (unary minus)
2. *, /, %
3. +, -
4. =, !=, <, >, <=, >=, =~
5. NOT
6. AND
7. OR

Syntactic Sugar

`both` and `any` denote the source and destination vertex of the current edge. They can be used to express a condition that should be `true` for both or at least either one of the two vertices. These keywords are only valid in an edge filter expression. To use them in a vertex filter results in a runtime type-checking exception.

```
both
any
```

The filter expressions inside the following examples are equivalent:

```
both.property = 1
src.property = 1 && dst.property = 1

any.degree() > 1
src.degree() > 1 || dst.degree() > 1
```

14.9.2 Type System

Filter expressions are a very simple type system. There are only the following 13 types:

1. `integer` (can be abbreviated in expressions with `int`)
2. `long`
3. `float`
4. `double`
5. `boolean`
6. `string`
7. `date`
8. `time`
9. `time with timezone`
10. `timestamp`
11. `timestamp with timezone`
12. `vertex`
13. `edge`

Conversions are only allowed from one numeric type to another numeric type (i.e. `integer`, `float`, `double`, `long`).

Comparisons require both sides to be of the same (or convertible) type.

14.9.3 Path Finding Filters

Filters can be used to limit the analyzed edges when searching for a shortest path between a source and destination vertex in a graph.

An edge filter expression is evaluated against each edge that is visited during the traversal of the graph. If the filter evaluates to `false` on an edge, this edge will be ignored and will not appear in the resulting shortest path.

It is also possible to use a vertex filter for path finding.

A vertex filter expression is evaluated against each vertex that is visited during the traversal of the graph, except for the source and destination vertex.

If the filter evaluates to `false` on a vertex, the edge to this vertex and all outgoing edges of the vertex will be ignored. The vertex will not appear in the resulting shortest path.

The source and destination vertex can be any vertex in the graph and the filter is not evaluated for them.

14.9.4 Subgraph Filters

Edge Filters

An edge filter expression is evaluated for each edge in the graph. The edge filter has access to the source and destination vertex of each edge and all of its properties.

If the filter expression evaluates to true, the edge and both the source and destination vertex will appear in the subgraph.

Vertex Filters

A vertex filter expression is evaluated for every vertex in the graph.

Every vertex for which the filter expression evaluates to true will appear in the subgraph.

Every edge connecting two vertices for which the expression evaluates to true will also appear in the subgraph.

Result Set Filters

Result set edge and vertex filters allow the creation of edge and vertex sets out of a given PGQL result set.

Vertex and Edge Collection Filters

Vertex and edge collection filters allow the creation of edge and vertex filters out of a given vertex and edge collection.

14.9.5 Operations on Filter Expressions

This section explains the various operations that you can perform on filter expressions.

- [Defining Filter Expressions](#)
- [Defining Result Set Filters](#)
- [Creating a Subgraph from PGQL Result Set](#)
- [Defining Collection Filters](#)
- [Creating a Subgraph from Collection Filters](#)
- [Combining Filter Expressions](#)

14.9.5.1 Defining Filter Expressions

You can define a new vertex filter, as shown in the following code:

Defining a Vertex Filter Using JShell

```
opg4j> var vertexFilter = VertexFilter.fromExpression("vertex.name = 'PGX'")
```

Defining a Vertex Filter Using Java

```
VertexFilter vertexFilter = VertexFilter.fromExpression("vertex.name = 'PGX'");
```

Defining a Vertex Filter Using Python

```
vertex_filter = VertexFilter("vertex.name = 'PGX'")
```

You can define a new edge filter, as shown in the following code:

Defining a Edge Filter Using JShell

```
opg4j> var edgeFilter = EdgeFilter.fromExpression("edge.cost > 5")
```


Defining a Edge Filter Using Java

```
EdgeFilter edgeFilter = EdgeFilter.fromExpression("edge.cost > 5");
```

Defining a Edge Filter Using Python

```
vertex_filter = EdgeFilter("edge.cost > 5")
```

14.9.5.2 Defining Result Set Filters

You can define a result set vertex filter, as shown in the following code:

Defining a Result Set Vertex Filter Using JShell

```
// Evaluates query on graph g to obtain a result set
opg4j> var resultSet = g.queryPgql("SELECT x FROM MATCH (x) WHERE x.age
> 24")
// Define a filter on the result set for the column "x"
opg4j> var vertexFilter = VertexFilter.fromPgqlResultSet(resultSet, "x")
// Obtain a vertex set
opg4j> var vertexSet = g.getVertices(vertexFilter)
```

Defining a Result Set Vertex Filter Using Java

```
// Evaluates query on graph g to obtain result set
PgqlResultSet resultSet = g.queryPgql("SELECT x FROM MATCH (x) WHERE
x.age > 24");
// Define a filter on the result set for the column "x"
VertexFilter vertexFilter = VertexFilter.fromPgqlResultSet(resultSet,
"x");
// Obtain a vertex set
VertexSet vertexSet = g.getVertices(vertexFilter);
```

You can define a result set edge filter, as shown in the following code:

Defining a Result Set Edge Filter Using JShell

```
// Evaluates query on graph g to obtain result set
opg4j> var resultSet = g.queryPgql("SELECT e FROM MATCH ()-[e]->()
WHERE e.weight >= 8")
// Define a filter on the result set for the column "e"
opg4j> var edgeFilter = EdgeFilter.fromPgqlResultSet(resultSet, "e")
// Obtain an edge set
opg4j> var edgeSet = g.getEdges(edgeFilter)
```

Defining a Result Set Edge Filter Using Java

```
// Evaluates query on graph g to obtain result set
PgqlResultSet resultSet = g.queryPgql("SELECT e FROM MATCH ()-[e]->()
WHERE e.weight >= 8");
// Define a filter on the result set for the column "e"
EdgeFilter edgeFilter = EdgeFilter.fromPgqlResultSet(resultSet, "e");
// Obtain an edge set
EdgeSet edgeSet = g.getEdges(edgeFilter);
```

14.9.5.3 Creating a Subgraph from PGQL Result Set

A subgraph can be obtained from a PGQL result set using result set filters.

You can create a subgraph from a result set vertex filter, as shown in the following code:

Creating a Subgraph from PGQL Result Set Vertex Filter Using JShell

```
// Evaluates query on graph g to obtain result set
opg4j> var resultSet = g.queryPgql("SELECT x FROM MATCH (x) WHERE x.age >
24")
// Define a filter on the result set for the column "x"
opg4j> var resultSetVertexFilter = VertexFilter.fromPgqlResultSet(resultSet,
"x")
// Create a subgraph of g containing the matched vertices in the resultSet
and the edges that connect them if any.
opg4j> var newGraph = g.filter(resultSetVertexFilter)
```

Creating a Subgraph from PGQL Result Set Vertex Filter Using Java

```
// Evaluates query on graph g to obtain result set
PgqlResultSet resultSet = g.queryPgql("SELECT x MATCH (x) WHERE x.age > 24")
// Define a filter on the result set for the column "x"
VertexFilter resultSetVertexFilter =
VertexFilter.fromPgqlResultSet(resultSet, "x")
// Create a subgraph of g containing the matched vertices in the resultSet
and the edges that connect them if any.
PgxGraph newGraph = g.filter(resultSetVertexFilter)
```

You can create a subgraph from a result set edge filter, as shown in the following code:

Creating a Subgraph from PGQL Result Set Edge Filter Using JShell

```
// Evaluates query on graph g to obtain result set
opg4j> var resultSet = g.queryPgql("SELECT e FROM MATCH ()-[e]->() WHERE
e.cost < 100")
// Define a filter on the result set for the column "e"
opg4j> var resultSetEdgeFilter = EdgeFilter.fromPgqlResultSet(resultSet, "e")
// Create a subgraph of g containing the matched edges in the resultSet and
their corresponding source and destination vertices.
opg4j> var newGraph = g.filter(resultSetEdgeFilter)
```

Creating a Subgraph from PGQL Result Set Edge Filter Using Java

```
// Evaluates query on graph g to obtain result set
PgqlResultSet resultSet = g.queryPgql("SELECT e FROM MATCH ()-[e]->() WHERE
e.cost < 100")
// Define a filter on the result set for the column "e"
EdgeFilter resultSetEdgeFilter = EdgeFilter.fromPgqlResultSet(resultSet, "e")
// Create a subgraph of g containing the matched edges in the resultSet and
their corresponding source and destination vertices.
PgxGraph newGraph = g.filter(resultSetEdgeFilter)
```

14.9.5.4 Defining Collection Filters

You can define a vertex collection filter, as shown in the following code:

Defining a Vertex Collection Filter Using JShell

```
// Obtain a vertex collection from an algorithm, query execution or any
other way
opg4j> VertexCollection<?> vertexCollection = ...
// Define a filter from the collection
opg4j> var vertexFilter = VertexFilter.fromCollection(vertexCollection)
```

Defining a Vertex Collection Filter Using Java

```
// Obtain a vertex collection from an algorithm, query execution or any
other way
VertexCollection<?> vertexCollection = ...
// Define a filter from the collection
VertexFilter vertexFilter =
VertexFilter.fromCollection(vertexCollection);
```

You can define an edge collection filter, as shown in the following code:

Defining an Edge Collection Filter Using JShell

```
// Obtain an edge collection from an algorithm, query execution or any
other way
opg4j> EdgeCollection edgeCollection = ...
// Define a filter from the collection
opg4j> var edgeFilter = EdgeFilter.fromCollection(edgeCollection)
```

Defining an Edge Collection Filter Using Java

```
// Obtain an edge collection from an algorithm, query execution or any
other way
EdgeCollection edgeCollection = ...
// Define a filter from the collection
EdgeFilter edgeFilter = EdgeFilter.fromCollection(edgeCollection);
```

14.9.5.5 Creating a Subgraph from Collection Filters

A subgraph can be obtained by using vertex or edge collection filters.

You can create a subgraph from a vertex collection filter, as shown in the following code:

Creating a Subgraph from a Vertex Collection Filter Using JShell

```
// Obtain a vertex collection from an algorithm, query execution or any
other way
opg4j> VertexCollection<?> vertexCollection = ...
// Define a filter from the collection
opg4j> var vertexFilter = VertexFilter.fromCollection(vertexCollection)
```

```
// Create a subgraph of g containing the matched vertices in the vertex
collection and the edges that connect them if any.
opg4j> var newGraph = g.filter(vertexFilter)
```

Creating a Subgraph from Vertex Collection Filter Using Java

```
// Obtain a vertex collection from an algorithm, query execution or any
other way
VertexCollection<?> vertexCollection = ...
// Define a filter from the collection
VertexFilter vertexFilter = VertexFilter.fromCollection(vertexCollection);
// Create a subgraph of g containing the matched vertices in the vertex
collection and the edges that connect them if any.
PgxGraph newGraph = g.filter(vertexFilter);
```

You can create a subgraph from edge collection filter, as shown in the following code:

Creating a Subgraph from Edge Collection Filter Using JShell

```
// Obtain an edge collection from an algorithm, query execution or any other
way
opg4j> EdgeCollection edgeCollection = ...
// Define a filter from the collection
opg4j> var edgeFilter = EdgeFilter.fromCollection(edgeCollection)
// Create a subgraph of g containing the matched edges in the collection and
their corresponding source and destination vertices.
opg4j> var newGraph = g.filter(edgeFilter)
```

Creating a Subgraph from Edge Collection Filter Using Java

```
// Obtain an edge collection from an algorithm, query execution or any other
way
EdgeCollection edgeCollection = ...
// Define a filter from the collection
EdgeFilter edgeFilter = EdgeFilter.fromCollection(edgeCollection);
// Create a subgraph of g containing the matched edges in the collection and
their corresponding source and destination vertices.
PgxGraph newGraph = g.filter(edgeFilter);
```

14.9.5.6 Combining Filter Expressions

Any filter expression used for subgraph filtering, can be combined with any other filter expression to form a new filter expression.

Filters can be combined using the following operations:

- intersection
- union

The intersection of two filters will only keep a vertex or edge, if both filters would accept it.

**Note:**

The intersection of two filters will not behave as an AND in the filter expression.

The union of two filters will keep a vertex or edge, if one of the filters would accept it.

**Note:**

The union of filters will not behave as an OR in the filter expression.

In the following example an edge filter is intersected with a vertex filter. The resulting subgraph will only include vertices that have the name 'PGX' and will only include edges that have a cost greater than 5.

Intersecting an Edge Filter with a Vertex Filter Using JShell

```
opg-jshell> var edgeFilter = EdgeFilter.fromExpression("edge.cost > 5")
opg-jshell> var vertexFilter = VertexFilter.fromExpression("vertex.name = 'PGX'")
opg-jshell> var combinedFilter = edgeFilter.intersect(vertexFilter)
```

Intersecting an Edge Filter with a Vertex Filter Using Java

```
EdgeFilter edgeFilter = EdgeFilter.fromExpression("edge.cost > 5");
VertexFilter vertexFilter = VertexFilter.fromExpression("vertex.name = 'PGX'");
GraphFilter combinedFilter = edgeFilter.intersect(vertexFilter);
```

Intersecting an Edge Filter with a Vertex Filter Using Python

```
edge_filter = EdgeFilter("edge.cost > 5")
vertex_filter = VertexFilter("vertex.name = 'PGX'")
combined_filter = edge_filter.intersect(vertex_filter)
```

In contrast, the subgraph created by the union of those filters will include vertices that either have the name 'PGX' or that has an incoming or outgoing edge with a cost greater than 5. It will also include edges with a cost greater than 5, as well as edges for which the source and destination vertex have the name 'PGX'.

14.10 Advanced Task Scheduling Using Execution Environments

This guide shows how you can use the advanced scheduling features of the enterprise scheduler.

The enterprise scheduler features of the graph server (PGX) are currently only available for Linux (x86_64), macOS (x86_64) and Solaris (x86_64, sparc).

The following topics provide more detailed information on enabling and scheduling tasks using the execution environment:

- [Enterprise Scheduler Configuration Guide](#)
- [Enabling Enterprise Scheduler Features](#)

- [Retrieving and Inspecting the Execution Environment](#)
- [Modifying and Submitting Tasks Under an Updated Environment](#)
- [Using Lambda Syntax](#)

14.10.1 Enterprise Scheduler Configuration Guide

This chapter describes the extra configuration options for the enterprise scheduler.



Note:

These configuration options are only available if the `scheduler` configuration variable is set to `enterprise_scheduler` in [Configuration Parameters for the Graph Server \(PGX\) Engine](#).

The configuration is divided into the following two parts:

1. `enterprise_scheduler_config`: for setting details about how tasks should be scheduled
2. `enterprise_scheduler_flags`: where you can configure the enterprise scheduler in more detail

Enterprise Scheduler Fields

Field	Type	Description	Default
<code>analysis_tasks_config</code>	object	Configuration for analysis tasks.	weight <no-of-CPUs> priority medium max_threads <no-of-CPUs>
<code>fast_analysis_task_config</code>	object	Configuration for fast analysis tasks.	weight 1 priority high max_threads <no-of-CPUs>
<code>max_num_concurrent_io_tasks</code>	integer	Maximum number of concurrent io tasks at a time.	3
<code>num_io_threads_per_task</code>	integer	Number of io threads to use per task.	<no-of-cpus>

Analysis Task Config Fields

Field	Type	Description	Default
max_threads	integer	A hard limit on the number of threads to use for a task.	<i>required</i>
priority	enum[high, medium, low]	The priority of the task. Threads are given to the task with the highest priority at the moment of execution. If there are more threads that have the highest priority, threads are given to the tasks according to their weight	<i>required</i>
weight	integer	The weight of the task. Threads are given to tasks proportionally to their weight. Tasks with higher weight will get more threads than tasks with lower weight. Tasks with the same weight will get the same amount of threads.	<i>required</i>

Enterprise Scheduler Flags

Field	Type	Description	Default
show_allocations	boolean	If true show memory allocation information.	false
show_environment	boolean	If true show version numbers and main environment settings at startup.	false
show_logging	boolean	If true enable summary logging. This is available even in non-debug builds and includes information such as the machine hardware information obtained at start-up, and per-job / per-loop information about the workload.	false
show_profiling	boolean	If true show profiling information.	false
show_scheduler_state	boolean	If true dump scheduler state on each update.	false
show_warnings	boolean	If true enable warnings. These are non-fatal errors. For example, if a NUMA-aware allocation cannot be placed on the intended socket.	true

Example 14-7 Custom Enterprise Scheduler Configuration

This configuration sets the number of io threads per task to 16, increases the maximum number of concurrent io tasks to 5. It also sets the configuration for fast analysis tasks to have a weight of 1, priority of "high" and sets a limit to the maximum number of threads used to 1.

```
{
  "enterprise_scheduler_config": {
    "num_io_threads_per_task": 16,
    "max_num_concurrent_io_tasks": 5,
    "fast_analysis_task_config": {
      "weight": 1,
      "priority": "high",
      "max_threads": 1
    }
  }
}
```

```

    }
  }
}

```

Example 14-8 Using the Enterprise Scheduler Flags

This configuration enables extra logging output from the enterprise scheduler.

```

{
  "enterprise_scheduler_flags": {
    "show_logging": true
  }
}

```

14.10.2 Enabling Enterprise Scheduler Features

You can enable the enterprise scheduler features, by setting the flag `allow_override_scheduling_information` of the the graph server (PGX) configuration file to `true`:

```

{"allow_override_scheduling_information":true}

```

See [Configuration Parameters for the Graph Server \(PGX\) Engine](#) for all configuration options of the graph server (PGX).

14.10.3 Retrieving and Inspecting the Execution Environment

Execution environments are bound to a session. You can retrieve the execution environment for a session by calling `getExecutionEnvironment()` on a `PgxSession`:

Retrieving the Execution Environment Using JShell

```

opg4j> execEnv.getValues()
==> [analysis-pool.max_num_threads=4, analysis-pool.weight=4, analysis-
pool.priority=MEDIUM, io-pool.num_threads_per_task=4, fast-track-analysis-
pool.max_num_threads=4, fast-track-analysis-pool.weight=1, fast-track-
analysis-pool.priority=HIGH]

```

Retrieving the Execution Environment Using Java

```

import oracle.pgx.api.*;
import java.util.List;
import java.util.Map.Entry;

List<Entry<String, Object>> currentValues = execEnv.getValues();
for (Entry<String, Object> value : currentValues) {
    System.out.println(value.getKey() + " = " + value.getValue());
}

```

See [Enterprise Scheduler Configuration Guide](#) for the values of an unmodified execution environment.

To retrieve the sub-environments use the `getIoEnvironment()`, `getAnalysisEnvironment()` and `getFastAnalysisEnvironment()` methods. Each sub-environment has their own `getValues()` method for retrieving the configuration of the sub-environment.

Retrieving the Sub-Execution Environment Using JShell

```
opg4j> var ioEnv = execEnv.getIoEnvironment()
ioEnv ==> IoEnvironment[pool=io-pool]
opg4j> ioEnv.getValues()
$5 ==> {num_threads_per_task=4}

opg4j> var analysisEnv = execEnv.getAnalysisEnvironment()
analysisEnv ==> CpuEnvironment[pool=analysis-pool]
opg4j> analysisEnv.getValues()
$7 ==> {max_num_threads=4, weight=4, priority=MEDIUM}

opg4j> var fastAnalysisEnv = execEnv.getFastAnalysisEnvironment()
fastAnalysisEnv ==> CpuEnvironment[pool=fast-track-analysis-pool]
opg4j> fastAnalysisEnv.getValues()
$9 ==> {max_num_threads=4, weight=1, priority=HIGH}
```

Retrieving the Sub-Execution Environment Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.executionenvironment.*;
import java.util.Map;

IoEnvironment ioEnv = execEnv.getIoEnvironment();
CpuEnvironment analysisEnv = execEnv.getAnalysisEnvironment();
CpuEnvironment fastAnalysisEnv = execEnv.getFastAnalysisEnvironment();

for (Entry<String, Object> value : ioEnv.getValues().getEntrySet()) {
    System.out.println(value.getKey() + " = " + value.getValue());
}

for (Entry<String, Object> value :
analysisEnv.getValues().getEntrySet()) {
    System.out.println(value.getKey() + " = " + value.getValue());
}

for (Entry<String, Object> value :
fastAnalysisEnv.getValues().getEntrySet()) {
    System.out.println(value.getKey() + " = " + value.getValue());
}
```

14.10.4 Modifying and Submitting Tasks Under an Updated Environment

You can modify an Input/Output (IO) environment in the number of threads by using the `setNumThreadsPerTask()` method of the `IoEnvironment`. The value is updated immediately and all tasks that are submitted after updating it are executed with the updated value.

Modifying the Execution Environment Using JShell

```
opg4j> ioEnv.setNumThreadsPerTask(8)
opg4j> var g = session.readGraphWithProperties(...)
==> PgxGraph[name=graph,N=3,E=6,created=0]
```

Modifying the Execution Environment Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.executionenvironment.*;

ioEnv.setNumThreadsPerTask(8);
PgxGraph g = session.readGraphWithProperties(...)
```

You can reset an environment to their initial values by calling the `ioEnv.reset()` method. Additionally, you can reset all environments at once by calling `execEnv.reset()` on the `ExecutionEnvironment` class.

You can modify CPU environments in their weight, priority and maximum number of threads using the `setWeight()`, `setPriority()` and `setMaxThreads()` methods:

Modifying the CPU Environment Using JShell

```
opg4j> analysisEnv.setWeight(50)
opg4j> fastAnalysisEnv.setMaxNumThreads(1)
opg4j> var rank = analyst.pagerank(g)
rank ==> VertexProperty[name=pagerank,type=double,graph=my-graph]
```

Modifying the CPU Environment Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.executionenvironment.*;

analysisEnv.setWeight(50);
fastAnalysisEnv.setMaxThreads(1);
Analyst analyst = session.createAnalyst();
VertexProperty rank = analyst.pagerank(g);
```

14.10.5 Using Lambda Syntax

Generally you can perform the following actions in the environment:

1. Set up the execution environment
2. Execute task
3. Reset execution environment

All these actions can be combined and performed in a single step using the `set` method. For each `set` method there is a method using the `with` prefix which takes the updated value and a lambda which should be executed using the updated value.

For example, use `withNumThreadsPerTask()` instead of `setNumThreadsPerTask()` as shown:

Using Lambda in the Execution Environment Using JShell

```
opg-jshell> var g = ioEnv.withNumThreadsPerTask(8, () ->
session.readGraphWithPropertiesAsync(...))
==> PgxGraph[name=graph,N=3,E=6,created=0]
```

Using Lambda in the Execution Environment Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.executionenvironment.*;

PgxGraph g = ioEnv.withNumThreadsPerTask(8, () ->
session.readGraphWithPropertiesAsync(...));
```

The preceding code execution is equivalent to the following sequence of actions:

```
var oldValue = ioEnv.getNumThreadsPerTask()
ioEnv.setNumThreadsPerTask(currentValue)
var g = session.readGraphWithProperties(...)
ioEnv.setNumThreadsPerTask(oldValue)
```

14.11 Admin API

This guide shows how to use the graph server (PGX) Admin API to inspect the server state including sessions, graphs, tasks, memory and thread pools.

- [Get a Server Instance](#)
- [Get Inspection Data](#)
- [Get Active Sessions](#)
- [Get Cached Graphs](#)
- [Get Published Graphs](#)
- [Get Currently Loading Graphs](#)
- [Get Tasks](#)
- [Get Available Memories](#)

14.11.1 Get a Server Instance

You can get a PGX Instance as shown in the following code:

Get a PGX Instance Using Java

```
import oracle.pgx.api.*;
ServerInstance instance = Pgx.getInstance(Pgx.EMBEDDED_URL);
```

Get a PGX Instance Using Python

```
instance = pypgx.get_session(base_url = "url")
```

14.11.2 Get Inspection Data

Inspection data is information about the server state.

You can get the inspection data using the following code:

Get the Inspection Data Using Java

```
JsonNode serverState = instance.getServerState();
```

Get the Inspection Data Using Python

```
server_state = instance.get_server_state()
```

This returns a `JsonNode` which contains all administration information, such as, number of graphs loaded, number of sessions, memory usage for graphs and properties and so on.

```
{
  "cached_graphs": [],
  "published_graphs": [],
  "graphs_currently_loading": [],
  "sessions": [],
  "tasks": [],
  "pools": [],
  "memory": {}
}
```

14.11.3 Get Active Sessions

`serverState.get("sessions")` returns an array of current active sessions. Each entry contains information about a session.

```
{
  "session_id": "530b5f9a-75c4-4838-9cc3-44df44b035c5",
  "source": "testServerState",
  "task_timeout_ms": 0,
  "idle_timeout_ms": 0,
  "alive_ms": 237,
  "total_analysis_time_ms": 115,
  "state": "RELEASED",
  "private_graphs": [
    {
      "name": "anonymous_graph_1",
      "creation_timestamp": 1589317879755,
      "is_transient": true,
      "memory": {
        "topology_bytes": 46,
        "key_mapping_bytes": 30,
        "persistent_property_mem_bytes": 0,
        "transient_property_mem_bytes": 0
      },
      "vertices_num": 1,
      "edges_num": 0,
      "persistent_vertex_properties": [
      ],
      "persistent_edge_properties": [
      ],
      "transient_vertex_properties": [
      ],
    ]
  ],
}
```

```

        "transient_edge_properties":[
            ]
        }
    ],
    "published_graphs":[
        {
            "name":"multigraph",
            "creation_timestamp":1589317879593,
            "is_transient":false,
            "memory":{
                "topology_bytes":110,
                "key_mapping_bytes":56,
                "persistent_property_mem_bytes":64,
                "transient_property_mem_bytes":0
            },
            "vertices_num":2,
            "edges_num":6,
            "persistent_vertex_properties":[
                {
                    "loaded":true,
                    "mem_size_bytes":16,
                    "name":"tProp",
                    "type":"string"
                }
            ],
            "persistent_edge_properties":[
                {
                    "loaded":true,
                    "mem_size_bytes":48,
                    "name":"cost",
                    "type":"double"
                }
            ],
            "transient_vertex_properties":[
            ],
            "transient_edge_properties":[
            ]
        }
    ]
}

```

The following table explains session information fields:

Table 14-5 Session Information Options

Field	Description
sessionID	Session' ID generated by PGX server.
source	Descriptive string identifying the client session.

Table 14-5 (Cont.) Session Information Options

Field	Description
task_timeout_ms	Timeout to interrupt long-running tasks submitted by sessions (algorithms, I/O tasks) in milliseconds. Set to zero for infinity/no timeout.
idle_timeout_ms	Timeout of idling sessions in milliseconds. Set to zero for infinity/no timeout.
alive_ms	Session's age in milliseconds.
total_analysis_time_ms	Total session's executing time in milliseconds.
state	Current session of the session can be Idle, Submitted, released or terminating.
private_graphs	Session bounded graphs.
published_graphs	Published graphs pointed to from the session.

Note:

The `is_transient` field indicates if the graph is transient. A graph is transient if it is not loaded from an external source.

14.11.4 Get Cached Graphs

The server state contains also cached graph information

`serverState.get("cached_graphs")` which returns a collection of graphs cached in memory. Each entry contains information about a graph as shown:

```
{
  "name": "sf-1589317879394",
  "creation_timestamp": 1589317879394,
  "vertex_properties": [
    {
      "loaded": true,
      "mem_size_bytes": 478504,
      "name": "prop1",
      "type": "double"
    }
  ],
  "edge_properties": [
    {
      "loaded": true,
      "mem_size_bytes": 1197720,
      "name": "cost",
      "type": "double"
    },
    {
      "loaded": true,
      "mem_size_bytes": 598860,
      "name": "0",
      "type": "integer"
    }
  ]
}
```

```

    }
  ],
  "memory": {
    "topology_bytes": 3921814,
    "key_mapping_bytes": 1407466,
    "property_mem_bytes": 2275084
  },
  "vertices_num": 59813,
  "edges_num": 149715
}

```

The following table explains graph information fields:

Table 14-6 Graph Information

Field	Description
name	Name of the graph.
creation_timestamp	Creation timestamp of the graph.
vertex_properties	List of vertex properties, each entry contains the name, type, memory size used by the property, and a boolean flag to indicate if the property is loaded into memory.
edge_properties	List of edges properties, similar to vertex properties.
memory	Memory size used by the whole graph (topology, key mappings and properties).
vertices_num	Number of vertices.
edges_num	Number of edges.

14.11.5 Get Published Graphs

`serverState.get("published_graphs")` returns a list of published graphs.

Each graph entry contains information about the published graph, similar to `cached_graphs`.

14.11.6 Get Currently Loading Graphs

`serverState.get("graphs_currently_loading")` returns progress information about graphs which are currently loading.

Each entry, corresponding to one graph, is shown as follows:

```

{
  "name": "anonymous_graph_1",
  "session_id": "530b5f9a-75c4-4838-9cc3-44df44b035c5",
  "start_loading_timestamp": 1605468453030,
  "elapsed_loading_time_ms": 281742,
  "num_vertices_read": 10000000,
  "num_edges_read": 196500000,
  "num_edge_providers_loaded": 1,
  "num_edge_providers_remaining": 9,
  "num_vertex_providers_loaded": 1,
  "num_vertex_providers_remaining": 0,
}

```

```

    "loading_phase": "reading edges",
    "loading_phase_start_timestamp": 1605468453085,
    "loading_phase_elapsed_time_ms": 281687,
    "loading_phase_state": "current vertex provider index: 1, number of
vertices read for prorvider: 0, current edge provider index: 1, number of
edges read for prorvider: 76,500,000"
}

```

The `name` field contains a temporary name of the graph. It may not be equal to the name that is assigned to graph after loading.

Fields indicating the number of read vertices and edges are updated in regular intervals of 10,000 entities.

The field `loading_phase` indicates the current phase during graph loading. Valid values are "reading edges" or "building graph indices". For some loading phases, the field `loading_phase_state` contains a string with additional information on the phase. However, not all loading phases provide this additional information.



Note:

`graphs_currently_loading` is supported for data formats CSV, ADJ_LIST, EDGE_LIST, TWO_TABLES and PG (FLAT_FILE) for homogeneous graphs and for formats CSV and RDBMS for partitioned graphs.

14.11.7 Get Tasks

`serverState.get("tasks")` returns the last 100 queued tasks.

Each task has a `type`, the pool to be executed on (the task might be already executed) and other status fields (`{Queued|Started|Done}` time), and a `sessionId` if the task belongs to a session.

14.11.8 Get Available Memories

This section contains a map of available memories, the key is the hostname and the value is a list of current available memories (managed and unmanaged). Each entry contains how much memory is free, used and the maximum available memory.

14.12 PgxFrames Tabular Data-Structure

`PgxFrame` is a data-structure to load, store and manipulate tabular data. It contains rows and columns. A `PgxFrame` can contain multiple columns where each column consist of elements of the same data type, and has a name. The list of the columns with their names and data types defines the schema of the frame. (The number of rows in the `PgxFrame` is not part of the schema of the frame.)

`PgxFrame` provides some operations that also output `PgxFrames` (described later in the tutorial). Those operations can be performed in-place (meaning that the frame is mutated during the operation) in order to save memory. In place operations should be used whenever possible. However, we provide out-place variants, i.e., a new frame is created during the operation.

The following table lists all the in-place operations along with the respective out-place operations:

Table 14-7 Mapping between In-Place and Out-Place Operations

In-place operations	Out-place operations
headInPlace	head
tailInPlace	tail
flattenAllInPlace	flattenAll
renameColumnInPlace	renameColumn
renameColumnsInPlace	renameColumns
selectInPlace	select

- [Loading a PgxFFrame from a Database](#)
- [Loading a PgxFFrame from Client-Side Data](#)
- [Printing the Content of a PgxFFrame](#)
- [Destroying a PgxFFrame](#)
- [Storing a PgxFFrame to a Database](#)
- [Loading and Storing Vector Properties](#)
- [Flattening Vector Properties](#)
- [Union of PGX Frames](#)
- [Joining PGX Frames](#)
- [PgxFFrame Helpers](#)
- [PgxFFrame-PgqlResultSet Conversions](#)
- [Creating a Graph from Multiple PgxFFrame Objects](#)

14.12.1 Loading a PgxFFrame from a Database

`PgxFFrame(S)` can also be loaded from relational tables in an Oracle database. Each column of the relational table will correspond to a column in the loaded frame. When loading `PgxFFrames` from the database, the default behavior is to detect what columns the table has, and to load them all. If not specified explicitly, the connection details of the current user and session are used and the columns are detected automatically.

The following describes the steps to load a `PgxFFrame` from a database table:

1. Create a **Session** and an **Analyst**.
Creating a Session and an Analyst Using JShell

```
cd /opt/oracle/graph/
./bin/opg4j
// starting the shell will create an implicit session and analyst
opg4j> import static
oracle.pgx.api.frames.functions.ColumnRenaming.renaming
opg4j> import static
oracle.pgx.api.frames.schema.ColumnDescriptor.columnDescriptor
```

```
opg4j> import oracle.pgx.api.frames.schema.*
opg4j> import oracle.pgx.api.frames.schema.datatypes.*
```

Creating a Session and an Analyst Using Java

```
import oracle.pgx.api.*;
import oracle.pgx.api.frames.*;
import oracle.pgx.api.frames.functions.*;
import oracle.pgx.api.frames.schema.*;
import oracle.pgx.api.frames.schema.datatypes.*;
import static oracle.pgx.api.frames.functions.ColumnRenaming.renaming;
import static
oracle.pgx.api.frames.schema.ColumnDescriptor.columnDescriptor;

PgxSession session = Pgx.createSession("my-session");
Analyst analyst = session.createAnalyst();
```

Creating a Session and an Analyst Using Python

```
session = pypgx.get_session(session_name="my-session")
analyst = session.create_analyst()
```

2. Load a PgxFrame.

Loading a PgxFrame Using JShell

```
opg4j> var exampleFrame = session.readFrame().
    db().
    name("framename"). // name of the frame
    tablename("tablename"). // name of the table from where the data must
be loaded
    connections(16). // indicates that 16 connections can be used
to load in parallel
    load();
```

Loading a PgxFrame Using Java

```
PgxFrame exampleFrame = session.readFrame()
    .db()
    .name("framename") // name of the frame
    .tablename("tablename") // name of the table from where the data must
be loaded
    .connections(16) // indicates that 16 connections can be used
to load in parallel
    .load();
```

3. If only a subset of the columns must be loaded, then you must specify the columns with `FrameReader.columns()`.

Loading a PgxFrame for a Subset of Columns Using JShell

```
// You must specify jdbc connection, keystore and the columns to load
opg4j> session.registerKeystore("keystore", pathToKeystore,
keystorePassword)
opg4j> var exampleFrame = session.readFrame().
```

```

    db().
    name("framename").
    tablename("tablename").      // name of the table from where
the data must be loaded
    jdbcUrl("jdbcUrl").
    username("user").
    keystoreAlias("keytore").
    owner("owner").             // necessary if the table is owned
by another user
    connections(16).           // indicates that 16 connections
can be used to load in parallel
    columns(exampleFrameSchema). // columns to load
    load();

```

Loading a PgxFrames for a Subset of Columns Using Java

```

import oracle.pgx.api.frames.schema.datatypes.DataTypes;
import oracle.pgx.api.frames.schema.ColumnDescriptor;
// You must specify jdbc connection, keystore and the columns to
load
session.registerKeystore("keystore", pathToKeystore,
keystorePassword)
PgxFrames exampleFrame = session.readFrame()
    .db()
    .name("framename")
    .tablename("tablename")      // name of the table from where
the data must be loaded
    .jdbcUrl("jdbcUrl")
    .username("user")
    .keystoreAlias("keytore")
    .owner("owner")             // necessary if the table is owned
by another user
    .connections(16)           // indicates that 16 connections
can be used to load in parallel
    .columns(exampleFrameSchema) // columns to load
    .load();

```

You can also create a graph from the `PgxFrames(s)`. See [Creating a Graph from Multiple PgxFrames Objects](#) for more information.

14.12.2 Loading a PgxFrames from Client-Side Data

You can also load `PgxFrames(s)` directly from client-side data.

The following describes the steps to load a `PgxFrames` from client-side data:

1. Create a **Session** and an **Analyst**.
See step-1 in [Loading a PgxFrames from a Database](#) for the code examples.
2. Define a frame schema to load a `PgxFrames` from client side data. For example, the following shows a frame schema defined with various data types:

Defining a Frame Schema Using JShell

```

opg4j> var exampleFrameSchema = List.of(
    columnDescriptor("name", DataTypes.STRING_TYPE),

```

```

        columnDescriptor("age", DataTypes.INTEGER_TYPE),
        columnDescriptor("salary", DataTypes.DOUBLE_TYPE),
        columnDescriptor("married", DataTypes.BOOLEAN_TYPE),
        columnDescriptor("tax_rate", DataTypes.FLOAT_TYPE),
        columnDescriptor("random", DataTypes.LONG_TYPE),
        columnDescriptor("date_of_birth", DataTypes.LOCAL_DATE_TYPE)
    )

```

Defining a Frame Schema Using Java

```

List<ColumnDescriptor> exampleFrameSchema = Arrays.asList(
    columnDescriptor("name", DataTypes.STRING_TYPE),
    columnDescriptor("age", DataTypes.INTEGER_TYPE),
    columnDescriptor("salary", DataTypes.DOUBLE_TYPE),
    columnDescriptor("married", DataTypes.BOOLEAN_TYPE),
    columnDescriptor("tax_rate", DataTypes.FLOAT_TYPE),
    columnDescriptor("random", DataTypes.LONG_TYPE),
    columnDescriptor("date_of_birth", DataTypes.LOCAL_DATE_TYPE)
);

```

Defining a Frame Schema Using Python

```

example_frame_schema = [
    ("name", "STRING_TYPE"),
    ("age", "INTEGER_TYPE"),
    ("salary", "DOUBLE_TYPE"),
    ("married", "BOOLEAN_TYPE"),
    ("tax_rate", "FLOAT_TYPE"),
    ("random", "LONG_TYPE"),
    ("date_of_birth", "LOCAL_DATE_TYPE")
]

```

3. Define data as per the schema.

Defining Data for the Schema Using JShell

```

opg4j> Map<String, Iterable<?>> exampleFrameData = Map.of(
    "name", Arrays.asList("Alice", "Bob", "Charlie"),
    "age", Arrays.asList(25, 27, 29),
    "salary", Arrays.asList(10000.0, 15000.0, 20000.0),
    "married", Arrays.asList(false, false, true),
    "tax_rate", Arrays.asList(0.21, 0.26, 0.32),
    "random", Arrays.asList(2394293898324L, 45640604960495L,
12312323409087654L),
    "date_of_birth", Arrays.asList(
        LocalDate.of(1990, 9, 15),
        LocalDate.of(1991, 11, 4),
        LocalDate.of(1993, 10, 4)
    )
);

```

Defining Data for the Schema Using Java

```

Map<String, Iterable<?>> exampleFrameData = new HashMap<>();
exampleFrameData.put("name", Arrays.asList("Alice", "Bob",
"Charlie"));
exampleFrameData.put("age", Arrays.asList(25, 27, 29));
exampleFrameData.put("salary", Arrays.asList(10000.0, 15000.0,
20000.0));
exampleFrameData.put("married", Arrays.asList(false, false, true));
exampleFrameData.put("tax_rate", Arrays.asList(0.21, 0.26, 0.32));
exampleFrameData.put("random", Arrays.asList(2394293898324L,
45640604960495L, 12312323409087654L));
exampleFrameData.put("date_of_birth",
    Arrays.asList(LocalDate.of(1990, 9, 15),
        LocalDate.of(1991, 11, 4),
        LocalDate.of(1993, 10, 4)
    )
);

```

Defining Data for the Schema Using Python

```

from datetime import date

example_frame_data = {
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 27, 29],
    "salary": [10000.0, 15000.0, 20000.0],
    "married": [False, False, True],
    "tax_rate": [0.21, 0.26, 0.32],
    "random": [2394293898324, 45640604960495, 12312323409087654],
    "date_of_birth": [date(1990, 9, 15),
        date(1991, 11, 4),
        date(1993, 10, 4)]
}

```

4. Load the frame as shown:

Loading the Frame Using JShell

```

opg4j> var exampleFrame = session.createFrame(exampleFrameSchema,
exampleFrameData, "example frame");

```

Loading the Frame Using Java

```

PgxFrame exampleFrame = session.createFrame(exampleFrameSchema,
exampleFrameData, "example frame");

```

Loading the Frame Using Python

```

example_frame=session.create_frame(example_frame_schema,example_fram
e_data,'example frame')

```

5. You can also load the frame incrementally as you receive more data:

Incrementally Loading the Frame Using JShell

```

opg4j> var exampleFrameBuilder =
session.createFrameBuilder(exampleFrameSchema);
opg4j> exampleFrameBuilder.addRow(exampleFrameData)
opg4j> Map<String, Iterable<?>> exampleFrameDataPart2 = Map.of(
    "name", Arrays.asList("Dave"),
    "age", Arrays.asList(26),
    "salary", Arrays.asList(18000.0),
    "married", Arrays.asList(true),
    "tax_rate", Arrays.asList(0.30),
    "random", Arrays.asList(456783423423L),
    "date_of_birth", Arrays.asList(LocalDate.of(1989, 9, 15))
);
opg4j> exampleFrameBuilder.addRow(exampleFrameDataPart2);
opg4j> var exampleFrame = exampleFrameBuilder.build("example frame");

```

Incrementally Loading the Frame Using Java

```

PgxFramerBuilder exampleFrameBuilder =
session.createFrameBuilder(exampleFrameSchema);
exampleFrameBuilder.addRow(exampleFrameData);
Map<String, Iterable<?>> exampleFrameDataPart2 = new HashMap<>();
exampleFrameDataPart2.put("name", Arrays.asList("Dave"));
exampleFrameDataPart2.put("age", Arrays.asList(26));
exampleFrameDataPart2.put("salary", Arrays.asList(18000.0));
exampleFrameDataPart2.put("married", Arrays.asList(true));
exampleFrameDataPart2.put("tax_rate", Arrays.asList(0.30));
exampleFrameDataPart2.put("random", Arrays.asList(456783423423L));
exampleFrameDataPart2.put("date_of_birth",
    Arrays.asList(LocalDate.of(1989, 9, 15))
);
exampleFrameBuilder.addRow(exampleFrameDataPart2);
PgxFramer exampleFrame = exampleFrameBuilder.build("example frame");

```

Incrementally Loading the Frame Using Python

```

example_frame_builder = session.create_frame_builder(example_frame_schema)
example_frame_builder.add_rows(example_frame_data)
example_frame_data_part_2 = {
    "name": ["Dave"],
    "age": [26],
    "salary": [18000.0],
    "married": [True],
    "tax_rate": [0.30],
    "random": [456783423423],
    "date_of_birth": [date(1989, 9, 15)]
}
example_frame_builder.add_rows(example_frame_data_part_2)
example_frame = example_frame_builder.build("example frame")

```

6. Finally, you can also load a frame from a Pandas dataframe in Python as shown:

Loading the Frame from Pandas dataframe

```
import pandas as pd
example_pandas_dataframe = pd.DataFrame(data=example_frame_data)
example_frame =
session.pandas_to_pgx_frame(example_pandas_dataframe, "example
frame")
```

You can also create a graph from the `PgxFrames`(s) . See [Creating a Graph from Multiple PgxFrames Objects](#) for more information.

14.12.3 Printing the Content of a PgxFrames

You can observe the contents of a frame using the `print` functionality as shown:

Printing a PgxFrames Using JShell

```
opg4j> exampleFrame.print();
```

Printing a PgxFrames Using Java

```
exampleFrame.print();
```

Printing a PgxFrames Using Python

```
example_frame.print()
```

The output appears as follows:

```
+-----+
+-----+
| name      | age | salary      | married | tax_rate | random      |
| date_of_birth |    |            |         |         |            |
+-----+
+-----+
| John      | 27  | 4133300.0   | true    | 11.0    | 123456782 |
| 1985-10-18 |    |            |         |         |            |
| Albert    | 23  | 5813000.5   | false   | 12.0    | 124343142 |
| 2000-01-14 |    |            |         |         |            |
| Heather   | 28  | 1.0130302E7 | true    | 10.5    | 827520917 |
| 1985-10-18 |    |            |         |         |            |
| Emily     | 24  | 9380080.5   | false   | 13.0    | 128973221 |
| 1910-07-30 |    |            |         |         |            |
| "D'Juan"  | 27  | 1582093.0   | true    | 11.0    | 92384      |
| 1955-12-01 |    |            |         |         |            |
+-----+
+-----+
```

14.12.4 Destroying a PgxFFrame

`PgxFrames` consumes a lot of memory on the graph server (PGX) if they have a lot of rows or columns. Hence it is necessary to close them with the `close()` operation. After this operation, the content of the `PgxFFrame` is not available anymore.

You can close a frame as shown:

Destroying a PgxFFrame Using JShell

```
opg4j> exampleFrame.close();
```

Destroying a PgxFFrame Using Java

```
exampleFrame.close();
```

Destroying a PgxFFrame Using Python

```
exampleFrame.close()
```

14.12.5 Storing a PgxFFrame to a Database

When storing a `PgxFFrame` into the database, the frame is stored as a table, where the columns correspond to the columns of the `PgxFFrame` and the rows correspond to the rows of the `PgxFFrame`.



Note:

Column order preservation may or may not happen when storing a `PgxFFrame` in the database.

Overwrite Mode

When storing as a table in the database, you can overwrite an already existing table (with correct structure).

In overwrite mode, the previous table is truncated (emptied), and the data is then inserted. By default, it is set to `false` so that if a table already exists, it will throw an error to the user unless `overwrite` is set to `true`.

Storing a PgxFFrame by Overwriting a table Using JShell

```
// store as table in the database using jdbc + username + password
opg4j> exampleFrame.write().
    db().                // select the "format" to be relational db
    name("framename").  // name of the frame
    tablename("tablename"). // name of the table in which the data must be
stored
    overwrite(true).    // indicates that if there is a table with the
```



```

same name, it will be overwritten (truncated)
    connections(16).           // indicates that 16 connections can be
used to store in parallel
    jdbcUrl("jdbcUrl").
    username("user").
    password("password").
    store();

```

Storing a PgxFrame by Overwriting a table Using Java

```

exampleFrame.write()
    .db()                       // select the "format" to be relational db
    .name("framename")         // name of the frame
    .tablename("tablename")    // name of the table in which the data
must be stored
    .overwrite(true)          // indicates that if there is a table
with the same name, it will be overwritten (truncated)
    .connections(16)          // indicates that 16 connections can be
used to store in parallel
    .jdbcUrl("jdbcUrl")
    .username("user")
    .password("password")
    .store();

```

14.12.6 Loading and Storing Vector Properties

You can load or store vector properties which are fundamental for PgxML functionality in the graph server (PGX) using PgxFrames.

In order to load a PgxFrame with vector properties, follow the steps as shown:

1. Create the PgxFrame schema, defining the columns as shown:

Creating PgxFrame Schema Using JShell

```

opg-jshell> var vecFrameSchema = List.of(
    columnDescriptor("intProp", DataTypes.INTEGER_TYPE),
    columnDescriptor("intProp2", DataTypes.INTEGER_TYPE),
    columnDescriptor("vectProp",
DataTypes.vector(DataTypes.FLOAT_TYPE, 3)),
    columnDescriptor("stringProp", DataTypes.STRING_TYPE),
    columnDescriptor("vectProp2",
DataTypes.vector(DataTypes.FLOAT_TYPE, 2))
).toArray(new ColumnDescriptor[0])

```

Creating PgxFrame Schema Using Java

```

ColumnDescriptor[] vecFrameSchema = {
    columnDescriptor("intProp", DataTypes.INTEGER_TYPE),
    columnDescriptor("intProp2", DataTypes.INTEGER_TYPE),
    columnDescriptor("vectProp",
DataTypes.vector(DataTypes.FLOAT_TYPE, 3)),
    columnDescriptor("stringProp", DataTypes.STRING_TYPE),
    columnDescriptor("vectProp2",

```

```
DataTypes.vector(DataTypes.FLOAT_TYPE, 2))
};
```

2. Load the PgxFrames with the given schema from the specified path:
Loading the PgxFrames With the Schema Using JShell

```
opg4j> var vecFrame = session.readFrame().
    db().
    name("vector PgxFrames").
    tablename("tablename").      // name of the table from where the data
must be loaded
    jdbcUrl("jdbcUrl").
    username("user").
    owner("owner").             // necessary if the table is owned by
another user
    connections(16).           // indicates that 16 connections can be
used to load in parallel
    columns(vecFrameSchema).    // columns to load
    load();
```

Loading the PgxFrames With the Schema Using Java

```
PgxFrames vecFrame = session.readFrame()
    .db()
    .name("vector PgxFrames")
    .tablename("tablename")     // name of the table from where the data
must be loaded
    .jdbcUrl("jdbcUrl")
    .username("user")
    .owner("owner")            // necessary if the table is owned by
another user
    .connections(16)          // indicates that 16 connections can be
used to load in parallel
    .columns(vecFrameSchema)   // columns to load
    .load();
```

The final result in the PgxFrames may appear as follows:

intProp	intProp2	vectProp	stringProp	vectProp2
0	2	0.1;0.2;0.3	testProp0	0.1;0.2
1	1	0.1;0.2;0.3	testProp10	0.1;0.2
1	2	0.1;0.2;0.3	testProp20	0.1;0.2
2	3	0.1;0.2;0.3	testProp30	0.1;0.2
3	1	0.1;0.2;0.3	testProp40	0.1;0.2

14.12.7 Flattening Vector Properties

You can split the vector properties into multiple columns using the `flattenAll()` operation.

For example, you can flatten the vector properties for the example explained in [Loading and Storing Vector Properties](#) as shown:

Flattening Vector Properties Using JShell

```
opg4j> vecFrame.flattenAll();
```

Flattening Vector Properties Using Java

```
vecFrame.flattenAll();
```

The resulting flattened `PgxFrame` may appear as shown:

```
+-----+
+-----+
| intProp | intProp2 | vectProp_0 | vectProp_1 | vectProp_2 |
stringProp | vectProp2_0 | vectProp2_1 |
+-----+
+-----+
| 0      | 2      | 0.1      | 0.2      | 0.3      |
testProp0 | 0.1    | 0.2      |          |          |
| 1      | 1      | 0.1      | 0.2      | 0.3      |
testProp10 | 0.1    | 0.2      |          |          |
| 1      | 2      | 0.1      | 0.2      | 0.3      |
testProp20 | 0.1    | 0.2      |          |          |
| 2      | 3      | 0.1      | 0.2      | 0.3      |
testProp30 | 0.1    | 0.2      |          |          |
| 3      | 1      | 0.1      | 0.2      | 0.3      |
testProp40 | 0.1    | 0.2      |          |          |
+-----+
+-----+
```

14.12.8 Union of PGX Frames

You can join two `PgxFrames` that have compatible columns (i.e. same type and order).

Creating a Union of PgxFrames Using JShell

```
opg4j> <first-frame>.union(<secondframe>).print();
```

Creating a Union of PgxFrames Using Java

```
<first-frame>.union(<secondframe>).print();
```

The rows of the resulting `PgxFrame` are the union of the rows from the two original frames.

Note:

The union operation will not remove duplicate rows that resulted from the union operation.

14.12.9 Joining PGX Frames

You can join two frames whose rows are correlated through one of the columns using the `join` functionality. This allows us to combine frames by checking for equality between rows for a specific column.

The following example shows joining two `PgxFrames` `exampleFrame` and `moreInfoFrame` on the `name` column by calling the `join` method.

Joining PgxFrames Using JShell

```
opg4j> exampleFrame.join(moreInfoFrame, "name", "leftFrame",
"rightFrame").print();
```

Joining PgxFrames Using Java

```
exampleFrame.join(moreInfoFrame, "name", "leftFrame", "rightFrame").print();
```

The result may appear as shown:

```
+-----+
+-----+
| leftFrame_name | leftFrame_age | leftFrame_salary | leftFrame_married |
leftFrame_tax_rate | leftFrame_random | leftFrame_date_of_birth |
rightFrame_name | rightFrame_title | rightFrame_reports |
+-----+
+-----+
| John          | 27          | 4133300.0       | true              |
11.0           | 123456782   | 1985-10-18     |                   |
John           | Software Engineering Manager | 5              |
| Albert        | 23          | 5813000.5       | false             |
12.0           | 124343142   | 2000-01-14     |                   |
Albert         | Sales Manager | 10             |                   |
| Emily         | 24          | 9380080.5       | false             |
13.0           | 128973221   | 1910-07-30     |                   |
Emily          | Operations Manager | 20             |
+-----+
+-----+
+-----+
```

The joined frame contains the columns of the two frames involved in the operation for the rows with the same `name`.



Note:

The column prefixes specified in the `join()` call, `leftFrame` and `rightFrame`.

14.12.10 PgxFrames Helpers

PgxFrames supports the following operations:

- head
- tail
- select
- renameColumns

Head Operation

The head operation can be used to only keep the first rows of a PgxFrames. (The result is deterministic only for ordered PgxFrames.)

Applying Head Operation on a PgxFrames Using JShell

```
opg4j> vecFrame.head(2).print();
```

Applying Head Operation on a PgxFrames Using Java

```
vecFrame.head(2).print();
```

The output appears as follows:

```
+-----+
| intProp | intProp2 | vectProp   | stringProp | vectProp2 |
+-----+
| 0       | 2       | 0.1;0.2;0.3 | testProp0  | 0.1;0.2   |
| 1       | 1       | 0.1;0.2;0.3 | testProp10 | 0.1;0.2   |
+-----+
```

Tail Operation

The tail operation can be used to only keep the last rows of a PgxFrames. (The result is deterministic only for ordered PgxFrames.)

Applying Tail Operation on a PgxFrames Using JShell

```
opg4j> vecFrame.tail(2).print();
```

Applying Tail Operation on a PgxFrames Using Java

```
vecFrame.tail(2).print();
```

The output appears as follows:

```
+-----+
| intProp | intProp2 | vectProp   | stringProp | vectProp2 |
+-----+
| 2       | 3       | 0.1;0.2;0.3 | testProp30 | 0.1;0.2   |
+-----+
```

```
| 3          | 1          | 0.1;0.2;0.3 | testProp40 | 0.1;0.2    |
+-----+-----+
```

Select Operation

The `select` operation can be used to keep only a specified list of columns of an input `PgxFrame`.

Applying Select Operation on a `PgxFrame` Using JShell

```
opg4j> var vecFrame_selected = vecFrame.select("vectProp2", "vectProp",
"stringProp")
```

Applying Select Operation on a `PgxFrame` Using Java

```
PgxFrame vecFrame_selected =
vecFrame.select("vectProp2","vectProp","stringProp");
```

Applying Select Operation on a `PgxFrame` Using Python

```
vec_frame_selected=vec_frame.select("vectProp2","vectProp","stringProp")
```

The result may appear as follows:

```
+-----+-----+
| vectProp2 | vectProp    | stringProp |
+-----+-----+
| 0.1;0.2   | 0.1;0.2;0.3 | testProp0  |
| 0.1;0.2   | 0.1;0.2;0.3 | testProp10 |
| 0.1;0.2   | 0.1;0.2;0.3 | testProp20 |
| 0.1;0.2   | 0.1;0.2;0.3 | testProp30 |
| 0.1;0.2   | 0.1;0.2;0.3 | testProp40 |
+-----+-----+
```

Rename `PgxFrame` Columns

You can rename the columns in a `PgxFrame` to customized names as follows:

Renaming `PgxFrame` Columns Using JShell

```
opg4j> var vecFrame_renamed = vecFrame.renameColumns(
    renaming("vectProp2", "vectProp2_renamed"),
    renaming("vectProp", "vectProp_renamed"),
    renaming("stringProp", "stringProp_renamed")
)
```

Renaming `PgxFrame` Columns Using Java

```
vecFrame_renamed = vecFrame.renameColumns(renaming("vectProp2",
"vectProp2_renamed"),
                                           renaming("vectProp",
"vectProp_renamed"),
```

```
renaming("stringProp",
"stringProp_renamed"));
```

The renamed PgxFrames appears as follows:

```
+-----+
-----+
| intProp | intProp2 | vectProp_renamed | stringProp_renamed |
vectProp2_renamed |
+-----+
-----+
| 0      | 2      | 0.1;0.2;0.3      | testProp0          |
0.1;0.2  |        |                   |                    |
| 1      | 1      | 0.1;0.2;0.3      | testProp10         |
0.1;0.2  |        |                   |                    |
| 1      | 2      | 0.1;0.2;0.3      | testProp20         |
0.1;0.2  |        |                   |                    |
| 2      | 3      | 0.1;0.2;0.3      | testProp30         |
0.1;0.2  |        |                   |                    |
| 3      | 1      | 0.1;0.2;0.3      | testProp40         |
0.1;0.2  |        |                   |                    |
+-----+
-----+
```

14.12.11 PgxFrames-PgqlResultSet Conversions

You can perform conversions between PgxFrames and PgqlResultSets.

PgxFrames to PgqlResultSet

You can convert a PgxFrames to PgqlResultSet as follows:

Converting PgxFrames to PgqlResultSet Using JShell

```
opg4j> var resultSet = exampleFrame.toPgqlResultSet();
```

Converting PgxFrames to PgqlResultSet Using Java

```
PgqlResultSet resultSet = exampleFrame.toPgqlResultSet();
```

Converting PgxFrames to PgqlResultSet Using Python

```
result_set = example_frame.to_pgql_result_set()
```

You can view the content of the result set through the usual PgqlResultSet APIs. The output appears as follows:

```
+-----+
-----+
| name      | age | salary      | married | tax_rate | random      |
date_of_birth |
+-----+
```

```

-----+
| John      | 27 | 4133300.0 | true  | 11.0 | 123456782 |
1985-10-18 |   |           |       |      |           |
| Albert   | 23 | 5813000.5 | false | 12.0 | 124343142 |
2000-01-14 |   |           |       |      |           |
| Heather  | 28 | 1.0130302E7 | true  | 10.5 | 827520917 |
1985-10-18 |   |           |       |      |           |
| Emily    | 24 | 9380080.5 | false | 13.0 | 128973221 |
1910-07-30 |   |           |       |      |           |
| "D'Juan" | 27 | 1582093.0 | true  | 11.0 | 92384     |
1955-12-01 |   |           |       |      |           |
+-----+
----+

```

PgqlResultSet to PgxFrames

You can convert a `PgqlResultSet` to a `PgxFrames` as follows:

Converting a `PgqlResultSet` to a `PgxFrames` Using `JShell`

```

opg4j> var query = ...;
opg4j> var graph = ...;
opg4j> var resultSet = graph.queryPgql(query);
opg4j> resultSet.toFrame();

```

Converting a `PgqlResultSet` to a `PgxFrames` Using Java

```

String query = ...;
PgxGraph graph = ...;
PgqlResultSet resultSet = graph.queryPgql(query);
resultSet.toFrame();

```

14.12.12 Creating a Graph from Multiple `PgxFrames` Objects

You can create a `PgxGraph` with vertex `PgxFrames`(s) and edge `PgxFrames`(s).

Consider the following `PgxFrames` objects:

```

people
+-----+
| id | name  |
+-----+
| 1  | Alice |
| 2  | Bob   |
| 3  | Charlie |
+-----+

houses
+-----+
| identification | location |
+-----+
| 1              | Road 1  |
| 2              | Street 5 |
+-----+

```



```
| 3 | Avenue 4 |
+-----+
```

knows

```
+-----+
| src | dst |
+-----+
| 1   | 1   |
| 2   | 3   |
| 3   | 2   |
+-----+
```

lives

```
+-----+
| source | destination |
+-----+
| 1      | 2            |
| 2      | 1            |
| 3      | 3            |
+-----+
```

You can now create a `PgxGraph` as shown in the following examples:

Creating a Graph from Multiple `PgxFrame` Objects Using JShell

```
opg4j> var graphFromFramesCreator =
session.createGraphFromFrames("example graph")
opg4j> graphFromFramesCreator.vertexProvider("people", people)
opg4j> graphFromFramesCreator.vertexProvider("houses",
houses).vertexKeyColumn("identification")
opg4j> graphFromFramesCreator.edgeProvider("knows", "people", "people",
knows)
opg4j> var edge_provider = graphFromFramesCreator.edgeProvider("lives",
"people", "houses", lives)
opg4j> edge_provider.sourceVertexKeyColumn("source")
opg4j> edge_provider.destinationVertexKeyColumn("destination")
opg4j> graphFromFramesCreator.partitioned(true)
opg4j> var graph = graphFromFramesCreator.create()
```

Creating a Graph from Multiple `PgxFrame` Objects Using Java

```
PgxGraphFromFramesCreator graphFromFramesCreator =
session.createGraphFromFrames("example graph");
graphFromFramesCreator.vertexProvider("people", people);
graphFromFramesCreator.vertexProvider("houses",
houses).vertexKeyColumn("identification");
graphFromFramesCreator.edgeProvider("knows", "people", "people", knows);
PgxEdgeProviderFromFramesCreator edgeProvider =
graphFromFramesCreator.edgeProvider("lives", "people", "houses", lives);
edgeProvider.sourceVertexKeyColumn("source");
edgeProvider.destinationVertexKeyColumn("destination");
graphFromFramesCreator.partitioned(true);
PgxGraph graph = graphFromFramesCreator.create();
```

Creating a Graph from Multiple PgxFrames Objects Using Python

```
vertex_providers_from_frames = [  
    session.vertex_provider_from_frame("person",  
                                       people),  
    session.vertex_provider_from_frame("house",  
                                       frame = houses,  
                                       vertex_key_column = "identification")  
]  
edge_providers_from_frames = [  
    session.edge_provider_from_frame("person_knows_person",  
                                     source_provider = "person",  
                                     destination_provider = "person",  
                                     frame = knows),  
    session.edge_provider_from_frame("person_lives_at_house",  
                                     source_provider = "person",  
                                     destination_provider = "house",  
                                     frame = lives,  
                                     source_vertex_column="source",  
                                     destination_vertex_column="destination")  
]  
graph = session.graph_from_frames("example graph",  
vertex_providers_from_frames, edge_providers_from_frames, partitioned=True)
```

15

Working with Files Using the Graph Server (PGX)

This chapter describes in detail about working with different file formats to perform various actions like loading, storing or exporting a graph using the Graph Server (PGX).

- [Loading Graph Data from Files](#)
- [Loading Graph Data in Parallel from Multiple Files](#)
- [Exporting Graphs Into a File](#)
- [Exporting a Graph into Multiple Files](#)

15.1 Loading Graph Data from Files

You can load graph data from files by either of the two ways:

- using the header format specified in the files
- by directly calling the graph builder API

Creating a graph using file header format

The graph server (PGX) uses the header of the files to determine the name and types of the properties to load. It also infers the column to be used as vertex ID, the columns that indicate the source and destination vertex ID for edges, and the column to be loaded as vertex or edge label.

Creating a graph using graph builder API

You can also use [PgxSession.readGraphFiles\(\)](#) to load the graph. This method takes the following three arguments:

- path to the vertex file
- path to the edge file
- name of the graph to be created

Loading the Graph Data from a File Using JShell

```
opg4j> var loadedGraph = session.readGraphFiles("<path/vertices.csv>", "<path/edges.csv>", "<graph_name>")
```

Loading the Graph Data from a File Using Java

```
import oracle.pgx.api.PgxSession;  
import oracle.pgx.api.PgxGraph;  
  
PgxSession session = Pgx.createSession("NewSession");
```

```
PgxGraph loadedGraph = session.readGraphFiles("<path/vertices.csv\"",
"<path/edges.csv\"", "<graph_name\"")
```

Loading the Graph Data from a File Using Python

```
session = pypgx.get_session(session_name="<session_name\"")
loaded_graph = session.read_graph_files("<path/vertices.csv\"", "<path/
edges.csv\"", "<graph_name\"")
```

The graph server (PGX) supports loading graph data from files for the following data formats:

- Plain Text Formats
- XML File Formats
- Binary File Formats
- [Graph Configuration for Loading from File](#)
- [Specifying the File Path](#)
- [Supported File Access Protocols](#)
- [Plain Text Formats](#)
- [XML File Formats](#)
- [Binary File Formats](#)

15.1.1 Graph Configuration for Loading from File

The following table presents the graph configuration options to load graph data from all supported file formats to the graph server (PGX).

Table 15-1 Loading from File - Graph Configuration Options

Field	Type	Description	Default
array_compaction_threshold	number	<i>[only relevant if the graph is optimized for updates]</i> Threshold used to determine when to compact the delta-logs into a new array. If lower than the engine <code>min_array_compaction_threshold</code> value, <code>min_array_compaction_threshold</code> will be used instead.	0.2
attributes	object	Additional attributes needed to read and write the graph data.	null
detect_gzip	boolean	Enable or disable automatic gzip compression detection when loading graphs.	true

Table 15-1 (Cont.) Loading from File - Graph Configuration Options

Field	Type	Description	Default
edge_id_strategy	enum[no_ids, keys_as_ids, unstable_generated_ids]	Indicates what ID strategy should be used for the edges of this graph. If not specified (or set to null), the strategy will be determined during loading or using a default value.	null
edge_id_type	enum[long]	Type of the edge ID. For homogeneous graphs, if not specified (or set to null), it will default to long.	null
edge_props	array of object	Specification of edge properties associated with graph.	[]
edge_uris	array of string	List of unified resource identifiers.	[]
error_handling	object	Error handling configuration.	null
external_stores	array of object	Specification of the external stores where external string properties reside.	[]
format	enum[pgb, edge_list, adj_list, graphml, pg, rdf, two_tables]	Graph format to be used.	null
header	boolean	First line of file is meant for headers. For example, 'EdgeId, SourceId, DestId, EdgeProp1, EdgeProp2'	false
keystore_alias	string	Alias to the keystore to use when connecting to the database.	null
loading	object	Loading-specific configuration.	null
local_date_formats	array of string	Array of local_date formats to use when loading and storing local_date properties. See DateTimeFormatter for documentation of the format string.	[]
optimized_for	enum[read, updates]	Indicates if the graph must use data-structures optimized for read-intensive scenarios or for fast updates.	read
partition_while_loading	enum[by_label, no]	Indicates if the graph must be partitioned while loading.	null
password	string	Password to use when connecting to database.	null
point2d	string	Longitude and latitude as floating point values separated by a space.	0.0 0.0
separator	string	A series of single-character separators for tokenizing. The characters ", {, } and \n cannot be used as separators. Default value is ", " for CSV files, and "\t " for other formats. The first character will be used as a separator when storing.	null
storing	object	Storing-specific configuration.	null

Table 15-1 (Cont.) Loading from File - Graph Configuration Options

Field	Type	Description	Default
time_format	array of string	The time format to use when loading and storing time properties. See DateTimeFormatter for documentation of the format string.	[]
time_with_timezone_format	array of string	The time with timezone format to use when loading and storing time with timezone properties. See DateTimeFormatter for documentation of the format string.	[]
timestamp_format	array of string	The timestamp format to use when loading and storing timestamp properties. See DateTimeFormatter for documentation of the format string.	[]
timestamp_with_timezone_format	array of string	The timestamp with timezone format to use when loading and storing timestamp with timezone properties. See DateTimeFormatter for documentation of the format string.	[]
vector_component_delimiter	character	Delimiter for the different components of vector properties.	;
vertex_id_strategy	enum[no_ids, keys_as_ids, unstable_generated_ids]	Indicates what ID strategy should be used for the vertices of this graph. If not specified (or set to null), the strategy will be automatically detected.	null
vertex_id_type	enum[int, integer, long, string]	Type of the vertex ID. For homogeneous graphs, if not specified (or set to null), it will default to a specific value (depending on the origin of the data).	null
vertex_props	array of object	Specification of vertex properties associated with graph.	[]
vertex_uris	array of string	List of unified resource identifiers.	[]

In the CSV format, the columns used to specify the vertex ID column, vertex labels column, edge ID column, edge source ID column, edge destination ID column and the edge label column can be configured with the CSV specific fields as shown in the following table:

Table 15-2 CSV Specific Options

Field	Type	Description	Default
array_compaction_threshold	number	<i>[only relevant if the graph is optimized for updates]</i> Threshold used to determined when to compact the delta-logs into a new array. If lower than the engine <code>min_array_compaction_threshold</code> value, <code>min_array_compaction_threshold</code> will be used instead.	0.2
attributes	object	Additional attributes needed to read and write the graph data.	null
detect_gzip	boolean	Enable or disable automatic gzip compression detection when loading graphs.	true
edge_destination_column	value	Name or index (starting from 1) of column corresponding to edge destination (for CSV format only).	null
edge_id_column	value	Name or index (starting from 1) of column corresponding to edge id (for CSV format only).	null
edge_id_strategy	enum[no_ids, keys_as_ids, unstable_generated_ids]	Indicates what ID strategy should be used for the edges of this graph. If not specified (or set to null), the strategy will be determined during loading or using a default value.	null
edge_id_type	enum[long]	Type of the edge ID. For homogeneous graphs, if not specified (or set to null), it will default to long.	null
edge_label_column	value	Name or index (starting from 1) of column corresponding to edge label (for CSV format only).	null
edge_props	array of object	Specification of edge properties associated with graph.	[]
edge_source_column	value	Name or index (starting from 1) of column corresponding to edge source (for CSV format only).	null
error_handling	object	Error handling configuration.	null
external_stores	array of object	Specification of the external stores where external string properties reside.	[]
format	enum[pgb, edge_list, adj_list, graphml, pg, rdf, two_tables]	Graph format to be used.	null
header	boolean	First line of file is meant for headers. For example, 'EdgeId, SourceId, DestId, EdgeProp1, EdgeProp2'.	false
keystore_alias	string	Alias to the keystore to use when connecting to database.	null
loading	object	Loading-specific configuration.	null

Table 15-2 (Cont.) CSV Specific Options

Field	Type	Description	Default
local_date_format	array of string	array of local_date formats to use when loading and storing local_date properties. See DateTimeFormatter for documentation of the format string	[]
optimized_for	enum[read, updates]	Indicates if the graph should use data-structures optimized for read-intensive scenarios or for fast updates.	read
partition_while_loading	enum[by_label, no]	Indicates if the graph should be partitioned while loading.	null
password	string	Password to use when connecting to database.	null
point2d	string	Longitude and latitude as floating point values separated by a space.	0.0 0.0
separator	string	a series of single-character separators for tokenizing. The characters ", {, } and \n cannot be used as separators. Default value is ", " for CSV files, and "\t " for other formats. The first character will be used as a separator when storing.	null
storing	object	Storing-specific configuration.	null
time_format	array of string	The time format to use when loading and storing time properties. See DateTimeFormatter for documentation of the format string	[]
time_with_timezone_format	array of string	The time with timezone format to use when loading and storing time with timezone properties. See DateTimeFormatter for documentation of the format string.	[]
timestamp_format	array of string	The timestamp format to use when loading and storing timestamp properties. See DateTimeFormatter for documentation of the format string.	[]
timestamp_with_timezone_format	array of string	The timestamp with timezone format to use when loading and storing timestamp with timezone properties. See DateTimeFormatter for documentation of the format string.	[]
vector_component_delimiter	character	Delimiter for the different components of vector properties.	;
vertex_id_column	value	Name or index (starting from 1) of column corresponding to vertex id (for CSV format only).	null
vertex_id_strategy	enum[no_ids, keys_as_ids, unstable_generated_ids]	Indicates what ID strategy should be used for the vertices of this graph. If not specified (or set to null), the strategy will be automatically detected.	null
vertex_id_type	enum[int, integer, long, string]	Type of the vertex ID. For homogeneous graphs, if not specified (or set to null), it will default to a specific value (depending on the origin of the data).	null

Table 15-2 (Cont.) CSV Specific Options

Field	Type	Description	Default
vertex_labels_column	value	Name or index (starting from 1) of column corresponding to vertex labels (for CSV format only).	null
vertex_props	array of object	Specification of vertex properties associated with graph.	[]

15.1.2 Specifying the File Path

The following examples show how to specify the file path for various file formats.

For formats that contain vertices and edges specified in one file (for example, EdgeList), use `uris` as shown in the following code:

```
{"uris":["path/to/file.format"]}
```

For formats that require separate files for edges and vertices (for example, FlatFile), use `vertex_uris` and `edge_uris` as shown in the following code:

```
{"vertex_uris":["vertices1.format","vertices2.format"],"edge_uris":["edges1.format","edges2.format"]}
```

PGX will parse graphs in most of the plain text formats in parallel if the graph data is split into multiple files, as shown in the following code:

```
{"uris":["file1.format","file2.format",...,"fileN.format"]}
```

15.1.3 Supported File Access Protocols

The graph server (PGX) supports loading from graph configuration files and graph data files over various protocols and virtual file systems. The type of file system or protocol is determined by the scheme of the uniform resource identifier (URI):

- local file system (`file:`) - this is also the default if the given URI does not contain any scheme
- classpath (`classpath:` or `res:`)
- HDFS (`hdfs:`)
- HTTPS (`https:`)
- FTPS (`ftps:`)
- various archive formats (`zip:`, `jar:`, `tar:`, `tgz:`, `tbz2:`, `gz:` and `bz2:`). The URI format is `scheme://arch-file-uri[!absolute-path]` (if you would like to use the `!` as a literal file-name character it must be escaped using `%21`).
For example, `jar:../lib/classes.jar!/META-INF/graph.json`.

Paths may be nested as in `tar:gz:https://anyhost/dir/mytar.tar.gz!/mytar.tar!/path/in/tar/graph.data`.

**Note:**

Relative paths are always resolved relative to the parent directory of the configuration file.

15.1.4 Plain Text Formats

The graph server (PGX) supports the following plain-text formats:

- Comma-Separated Values (CSV)
- Adjacency List (ADJ_LIST)
- Edge List (EDGE_LIST)
- Two Tables (TWO_TABLES)
- Flat File (FLAT_FILE)

Parsing of Vertices

PGX supports three types of vertex identifies (`id`): `integer`, `long` and `string`. The type defaults to `integer`, but can be configured through the `vertex_id_type` option in the graph configuration.

Parsing of Edges

Of the various formats and protocols supported by graph server (PGX), only CSV and flat file parsing support edge identifiers. For all other data sources, the `id` of an edge is PGX's internal `id`, which is an integer from zero to `num_edges - 1`.

Parsing of Properties

`string` properties, spatial properties (currently only `point2d`) and temporal properties (`date`, `local_date`, `time`, `timestamp`, `time_with_timezone` and `timestamp_with_timezone`) must be quoted ("`<string>`") only if they contain a separator character (usually `,` for CSV and `'` for Edge List and Adjacency List) or if they contain `"` or `\n`.

`date` properties are parsed using Java's `SimpleDateFormat` utility, instantiated with the format string `yyyy-MM-dd HH:mm:ss` unless specified otherwise in the graph configuration. All other types of temporal properties are parsed using Java's `DateTimeFormatter` utility.

`point2d` can be specified by its longitude followed by its latitude, separated by a space. Both longitude and latitude are doubles. For example, `"-74.0445 40.6892"` is the representation of a `point2d` instance representing the location of the Statue of Liberty.

Boolean values are interpreted as `true` if the value is `true` (ignoring case), `Y` (ignoring case) or `1`, `false` otherwise. The suggested notation for `false` is `false` (ignoring case), `N` (ignoring case) or `0`. All other types are parsed using the `parseXXX()` functions of its corresponding Java type, for example, `Integer.parseInt(...)` for integer types.

Vector properties are supported in the Adjacency List (ADJ_LIST), Comma-Separated Values (CSV), Edge List (EDGE_LIST), and Two Tables text (TWO_TABLES) formats. Vector properties with vector components of type `integer`, `long`, `float` and `double`

can be loaded from these formats. In order to specify that a vertex or edge property is a vector property, the `dimension` field of the graph property configuration must be set to the dimension of the vector and be a strictly positive integer value. A vector value is represented in the supported text formats by the list of the vector components values separated by the vector component delimiter. By default the vector component delimiter is `;`, but this delimiter can be changed by changing the `vector_component_delimiter` graph configuration entry. Therefore a 3-dimensional vector of doubles could for example look like `0.1;0.0004;3.14` in the text file if the vector component delimiter is `;`.

Separators

When using single file formats, IDs and properties are separated with `tab` or one single space (`"\t "`) by default, for multiple file formats comma (`" , "`) is used instead. However, PGX allows to configure the separator string.

Parallel Loading

The following formats support parallel loading from multiple files:

- CSV (specify multiple files in `vertex_uris` and/or `edge_uris`)
- Adjacency List (specify multiple files in `uris`)
- Edge List (specify multiple files in `uris`)
- Two Tables (specify multiple files in `vertex_uris` and/or `edge_uris`)
- Flat File (specify multiple files in `vertex_uris` and/or `edge_uris`)

Legend

The following abbreviations are used to specify text formats:

- V = Vertex Key
- VG = Neighbor Vertex
- VL = Vertex Labels
- VP = Vertex Property
- VPK = Vertex Property Key
- VPT = Vertex Property Type
- EL = Edge Label
- EP = Edge Property
- EPK = Edge Property Key
- EPT = Edge Property Type

For example `<V-2, VG-4>` or `<V-2, VG-4>` denotes the 4th neighbor of the 2nd vertex.

- [Comma-Separated Values \(CSV\)](#)
- [Adjacency List \(ADJ_LIST\)](#)
- [Edge List \(EDGE_LIST\)](#)
- [Two Tables \(TWO_TABLES\)](#)
- [Flat File \(FLAT_FILE\)](#)

15.1.4.1 Comma-Separated Values (CSV)

The CSV format is a text file format with vertices and edges stored in different files. Each line of the files represents a vertex or an edge. The vertex key and labels, the edge key, source, destination and label, and the attached properties are stored in the order specified by the file header (first line) and the configuration.

A graph with V vertices, having N vertex properties and K neighbors each, and E edges, having M edge properties, would be represented in CSV as shown:

vertices.csv

```
<V-1>,<VL-1>,<V-1, NP-1>,...,<V-1, NP-N>
<V-2>,<VL-2>,<V-2, NP-1>,...,<V-2, NP-N>
...
<V-V>,<VL-N>,<V-V, NP-1>,...,<V-V, NP-N>
```

edges.csv

```
<E-1>,<V-1>,<V-1, VG-1>,<EL-1>,<E-1, EP-1>,...,<E-1, EP-M>
...
<E-K>,<V-1>,<V-1, VG-K>,<EL-N>,<E-K, EP-1>,...,<E-K, EP-M>
<E-K+1>,<V-2>,<V-2, VG-1>,<EL-N+1>,<E-K+1, EP-1>,...,<E-K+1, EP-M>
...
<E-V*K>,<V-V>,<V-V, VG-K>,<EL-V*K>,<E-V*K, EP-1>,...,<E-V*K, EP-M>
```

Example 15-1 Loading graph from a CSV file with header details

The following examples shows a graph configuration file for loading a graph with two vertices and two edges:

vertices.csv

```
key,integer_prop,string_prop
1,33,"Alice"
2,42,"Bob"
```

edges.csv

```
source,dest,integer_prop,string_prop
1,2,0,"baz"
2,2,-12,"bat"
```

The corresponding graph configuration file is as shown:

```
{
  "format": "csv",
  "header": true,
  "vertex_id_column": "key",
  "edge_source_column": "source",
  "edge_destination_column": "dest",
  "vertex_uris": ["vertices.csv"],
  "edge_uris": ["edges.csv"],
  "vertex_props": [
    {
```

```

        "name": "integer_prop",
        "type": "integer"
    },
    {
        "name": "string_prop",
        "type": "string"
    }
],
"edge_props": [
    {
        "name": "integer_prop",
        "type": "integer"
    },
    {
        "name": "string_prop",
        "type": "string"
    }
]
}

```

Example 15-2 Loading graph from a CSV file without header details

The following examples shows a graph configuration file for loading a graph with two vertices and two edges:

vertices.csv

```

1,33,"Alice"
2,42,"Bob"

```

edges.csv

```

1,2,0,"baz"
2,2,-12,"bat"

```

The corresponding graph configuration file is as shown:



Note:

The column indices are given in place of the column names.

```

{
    "format": "csv",
    "header": false,
    "vertex_id_column": 1,
    "edge_source_column": 1,
    "edge_destination_column": 2,
    "vertex_uris": ["vertices.csv"],
    "edge_uris": ["edges.csv"],
    "vertex_props": [
        {

```

```

        "name": "integer_prop",
        "type": "integer",
        "column": 2
    },
    {
        "name": "string_prop",
        "type": "string",
        "column": 3
    }
],
"edge_props": [
    {
        "name": "integer_prop",
        "type": "integer",
        "column": 3
    },
    {
        "name": "string_prop",
        "type": "string",
        "column": 4
    }
]
}

```

If no column indices are set in the configuration file, the columns are assumed to be in the following order:

- For vertex files: - Vertex ID - Vertex labels (if present) - Vertex properties in the order they are declared in the configuration
- For edge files: - Edge ID (if present) - Edge source - Edge destination - Edge label (if present) - Edge properties in the order they are declared in the configuration

Therefore the earlier configuration is equivalent to:

```

{
  "format": "csv",
  "header": false,
  "vertex_uris": ["vertices.csv"],
  "edge_uris": ["edges.csv"],
  "vertex_props": [
    {
      "name": "integer_prop",
      "type": "integer"
    },
    {
      "name": "string_prop",
      "type": "string"
    }
  ],
  "edge_props": [
    {
      "name": "integer_prop",
      "type": "integer"
    },
    {

```

```

        "name": "string_prop",
        "type": "string"
    }
}

```

15.1.4.2 Adjacency List (ADJ_LIST)

The Adjacency List format is a text file format containing a list of neighbors from a vertex, per line. The format is extended to encode properties. The following shows a graph with V vertices, having N vertex properties and M edge properties:

```

<V-1> <V-1, VP-1> ... <V-1, VP-N> <V-1, VG-1> <EP-1> ... <EP-M> <V-1, VG-2> <EP-1> ...
<EP-M>
<V-2> <V-2, VP-1> ... <V-2, VP-N> <V-2, VG-1> <EP-1> ... <EP-M> <V-2, VG-2> <EP-1> ...
<EP-M>
...
<V-V> <V-V, VP-1> ... <V-V, VP-N> <V-V, VG-1> <EP-1> ... <EP-M> <V-V, VG-2> <EP-1> ...
<EP-M>

```

Note:

Trailing separators will be considered as errors. For example, if whitespace is used to separate the properties, any trailing whitespace will cause an exception to be raised.

Example 15-3 Graph in Adjacency List Format

This example shows a graph with 4 vertices (1, 2, 3 and 4), each having a double and a string property, and 3 edges, each having a boolean and a date property, encoded in Adjacency List format:

```

1 8.0 "foo"
2 4.3 "bar" 1 false "1985-10-18 10:00:00"
3 6.1 "bax" 2 true "1961-12-30 14:45:14" 4 false "2001-01-15 07:00:43"
4 17.78 "f00"

```

Note:

ADJ_LIST is more space efficient than EDGE_LIST. This is because vertices are first defined and then the edges are being created, indicating that we are repeating each vertex at least once.

15.1.4.3 Edge List (EDGE_LIST)

The Edge List format is a text file format starting with a section with one vertex per line, followed by a section with one edge per line. If a vertex does not have any labels or properties, it is possible to omit the vertex in the first section, but still specify edges for the vertex in the second section.

```

EdgeList      := {Vertex '\n'}* '\n' {Edge '\n'}*

Vertex       := VertexId '*' VertexLabels? PropertyValue*
VertexId     := Integer | Long | String
VertexLabels := {' String* '}

Edge         := SrcVertex DstVertex EdgeLabel? PropertyValue*
SrcVertex    := VertexId
DstVertex    := VertexId
EdgeLabel    := String

PropertyValue := Integer | Long | Double | Float | Boolean | String | Date

```

The vertices start with an identifier (`VertexId`), followed by a `*`, an optional set of vertex labels (`VertexLabels?`) and the vertex properties (`PropertyValue*`). A vertex identifier is either an `Integer`, a `Long`, or a `String`. Furthermore, vertex labels are zero or more `Strings` between curly braces (`'{ String* }'`).

The edges start with source and destination vertex identifiers (`SrcVertex DstVertex`), followed by optional edge label (`EdgeLabel?`) and the edge properties (`PropertyValue*`). The edge label is a `String`.

Example 15-4 Graph in Edge List format

This example shows a graph with two vertices and two edges, with labels and properties:

```

1 * { "Person" "Male" } "Mario" 15
2 * { "Person" "Male" } "Luigi" 14
1 2 "likes" 3.5
2 1 "likes" 2.1

```

The two vertices (lines 1-2) have identifiers 1 and 2 and both have the labels "Person" and "Male", a string property ("Mario" and "Luigi") and an integer property (15 and 14). There is an edge from vertex 1 to vertex 2 (line 3) with label "likes" and a double property with value 3.5, and another edge from vertex 2 to vertex 1 with label "likes" and a double property with value 2.1.

The following shows the corresponding graph configuration:

```

{
  "format": "edge_list",
  "uri": "example.edgelist",
  "vertex_id_type": "long",
  "vertex_labels": true,
  "edge_label": true,
  "vertex_props": [
    {
      "name": "name",
      "type": "string"
    },
    {
      "name": "age",
      "type": "int"
    }
  ],
  "edge_props": [

```



```

    {
      "name": "rating",
      "type": "double"
    }
  ],
  "loading_options": {
    "load_vertex_labels": true,
    "load_edge_label": true
  },
  "separator": " "
}

```

15.1.4.4 Two Tables (TWO_TABLES)

When configured to use `file` as datastore, the Two Tables format becomes a text file format similar to the Edge List format, with the only difference that the vertices and edges are stored in two different files. The vertices file contains vertex IDs followed by vertex properties. The edges file contains the source vertices and target vertices, followed by edge properties.

A graph with V vertices, having N vertex properties and M edge properties would be represented in two files as shown in the following:

vertices.ttt:

```

<V-1> <V-1, NP-1> ... <V-1, NP-N>
<V-2> <V-2, NP-1> ... <V-2, NP-N>
...
<V-V> <V-V, NP-1> ... <V-V, NP-N>

```

edges.ttt:

```

<V-1> <V-1, VG-1> <EP-1> ... <EP-M>
<V-1> <V-1, VG-2> <EP-1> ... <EP-M>
...
<V-V> <V-V, VG-1> <EP-1> ... <EP-M>

```

Example 15-5 Graph in Two Tables Text format

The following example shows the graph of 4 vertices (1, 2, 3 and 4), each having a `double` and a `string` property, and 3 edges, each having a `boolean` and a `date` property, encoded in Two Tables Text format:

vertices.ttt:

```

1 8.0 "foo"
2 4.3 "bar"
3 6.1 "bax"
4 17.78 "f00"

```

edges.ttt:

```

2 1 false "1985-10-18 10:00:00"
3 2 true "1961-12-30 14:45:14"
3 4 false "2001-01-15 07:00:43"

```

**Note:**

If you are planning on storing big graphs you must consider Two Tables Text format in order to save disk space.

15.1.4.5 Flat File (FLAT_FILE)

The Flat File format is a text file format containing two description files, one for vertices and one for edges. Each file consists of a list of properties with the following format:

vertices.opv

vertex_ID, key_name, value_type, value, value, value

```
<V-1> <V-1, VPK-1> <V-1, VPT-1> [<V-1, VP-1> <V-1, VP-1> <V-1, VP-1>]
...
<V-1> <V-1, VPK-N> <V-1, VPT-1> [<V-1, VP-N> <V-1, VP-N> <V-1, VP-N>]
<V-2> <V-2, VPK-1> <V-2, VPT-1> [<V-2, VP-1> <V-2, VP-1> <V-2, VP-1>]
...
<V-2> <V-2, VPK-N> <V-2, VPT-N> [<V-2, VP-N> <V-2, VP-N> <V-2, VP-N>]
...
<V-V> <V-V, VPK-N> <V-V, VPT-N> [<V-V, VP-N> <V-V, VP-N> <V-V, VP-N>]
```

edges.ope

edge_ID, source_vertex_ID, destination_vertex_ID, edge_label, key_name,
value_type, value, value, value

```
<E-1> <V-1, VG-1> <E-1, EL-1> <E-1, EPK-1> <E-1, EPT-1> [<E-1, EP-1> <E-1, EP-1>
<E-1, EP-1>]
...
<E-1> <V-N, VG-N> <E-1, EL-N> <E-1, EPK-N> <E-1, EPT-N> [<E-1, EP-N> <E-1, EP-N>
<E-1, EP-N>]
<E-2> <V-1, VG-1> <E-2, EL-1> <E-2, EPK-1> <E-2, EPT-1> [<E-2, EP-1> <E-2, EP-1>
<E-2, EP-1>]
...
<E-2> <V-N, VG-N> <E-2, EL-N> <E-2, EPK-N> <E-2, EPT-N> [<E-2, EP-N> <E-2, EP-N>
<E-2, EP-N>]
...
<E-E> <V-N, VG-N> <E-E, EL-N> <E-E, EPK-N> <E-E, EPT-N> [<E-E, EP-N> <E-E, EP-N>
<E-E, EP-N>]
```

Special Considerations when Using Flat File Format

- When no properties are defined for a certain vertex or edge, %20 is used instead of the key name:

```
Vertices: 1,%20,,,
Edges: 1,2,1,"label",%20,,,
```

- Values that are not numeric nor date go in the first field; numeric values go in the second, and dates in the third.
- The following shows the mapping between PGX property type and flat file value_type:

Table 15-3 Mapping between PGX Property Type and Flat File value_type

PGX property type	Flat file value_type
STRING	1
INTEGER	2
FLOAT	3
DOUBLE	4
DATE	5
LOCAL_DATE	5
TIME	5
TIMESTAMP	5
TIME_WITH_TIMEZONE	5
TIMESTAMP_WITH_TIMEZONE	5
BOOLEAN	6
LONG	7
POINT2D	200

 **Note:**

When loading a graph in flat file format into PGX, the graph configuration is used to find the right temporal or spatial type.

- The standard for the flat file format defines comma as the only valid delimiter, therefore any delimiter set in the graph configuration is ignored and comma is used instead.
- Strings must not be quoted, however the following encoding is needed for some characters:
 - '%' -> '%25'
 - '\t' -> '%09'
 - '' -> '%20'
 - '\n' -> '%0A'
 - ';' -> '%2C'
- When storing a graph into flat file format, vertex labels will be ignored. Also, when a graph has no edge label, an empty string ("") will be stored instead.
- When loading a graph in parallel using flat file format, all information regarding a specific vertex or edge must be contained in the same partition otherwise unexpected behavior might occur.

Example 15-6 Graph in Flat File Text format

The following example shows a graph of 4 vertices (1, 2, 3 and 4), each having a double and a string property, and 3 edges, each having a boolean and a date property, encoded in Flat File Text format:

vertices.opv:

```
1,doubleProp,4,,8.0,
1,stringProp,1,foo,,
2,doubleProp,4,,4.3,
```

```

2,stringProp,1,bar,,
3,doubleProp,4,,6.1,
3,stringProp,1,bax,,
4,doubleProp,4,,17.78,
4,stringProp,1,f00,,

edges.ope:

1,2,1,label,boolProp,6,false,,
1,2,1,label,dateProp,5,,,1985-10-18%2010:00:00
2,3,2,label,boolProp,6,true,,
2,3,2,label,dateProp,5,,,1961-12-30%2014:45:14
3,3,4,label,boolProp,6,false,,
3,3,4,label,dateProp,5,,,2001-01-15%2007:00:43

```

15.1.5 XML File Formats

Graph ML

The graph server (PGX) supports loading graphs from files using the XML-based Graph ML format. Graphs already in memory may also be exported into GraphML files. See [GraphML specification](#) for a detailed description of the XML schema.

PGX GraphML Limitation

PGX does not support all features of the GraphML format. Some of the limitations are:

- If the graph is undirected (`edgedefault="undirected"`), then edge properties are not supported
- All vertices (edges) must have the same amount and type of vertex (edge) properties
- `port`, `default`, and `hyperedge` are not supported

Example 15-7

The following example graph consists of 3 vertices and 3 edges. Each vertex has an integer property named `number` and each edge has a string property named `label`. Note that the edges are directed and that the strings for the property do not have to be put in (double) quotation marks.

```

<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <key attr.name="number" attr.type="integer" for="node" id="number"/>
  <key attr.name="label" attr.type="string" for="edge" id="label"/>
  <graph edgedefault="directed">
    <node id="1">
      <data key="number">2</data>
    </node>
    <node id="2">
      <data key="number">45</data>
    </node>
    <node id="3">
      <data key="number">83</data>
    </node>
    <edge target="2" source="1">
      <data key="label">this graph</data>
    </edge>
  </graph>
</graphml>

```

```

</edge>
<edge source="3" target="2">
  <data key="label">forms a</data>
</edge>
<edge target="1" source="3">
  <data key="label">triangle</data>
</edge>
</graph>
</graphml>

```

▲ Caution:

Due to the verbose nature of XML, the GraphML format comes with a large overhead compared to other file-based graph formats. You must use a different format if you want to consider the load or store performance and file size as important factors.

15.1.6 Binary File Formats

PGX Binary Format (PGB)

PGX binary format (.pgb) is the proprietary binary format for graph server (PGX), which allows fast and efficient file processing. Fundamentally, the file is a binary dump of the graph and property data. Bytes are written in network byte order (big endian).

Type Encoding

Table 15-4 Type Encoding

Value	Type	Size in bytes
0	Boolean	1
1	Integer	4
2	Long	8
3	Float	4
4	Double	8
7	String	varies
11	Vertex labels	varies
13	Local date	4
14	Time	4
15	Timestamp	8
16	Time with time zone	8
17	Timestamp with time zone	12
18	Vector property	variable: <sizeof component-type> * <dimension>

File Layout

Table 15-5 File Layout

Size in bytes	Description	Required	Comment
4	magic word	Yes	0x99191191
4	vertex size	Yes	Allowed values are 4 and 8.
4	edge size	Yes	Allowed values are 4 and 8.
<vertex size>	number of vertices	Yes	
<edge size>	number of edges	Yes	
<edge size> * (<numVertices> + 1)	edge begin array	Yes	
<vertex size> * <numEdges>	destination vertex array	Yes	
1	component bitmap	Yes	<ul style="list-style-type: none"> • 0x0001: node keys • 0x0002: vertex labels • 0x0004: edge label • 0x0008: edge keys • other bits: reserved
4	vertexKey type	No	Only present if <i>component bitmap</i> & 0x0001 == 0x0001. See Table 15-4 for type encoding.
<vertex key layout>	vertex keys	No	Only present if <i>component bitmap</i> & 0x0001 == 0x0001.
4	edgeKey type	No	Only present if <i>component bitmap</i> & 0x0008 == 0x0008. See Table 15-4 for type encoding
<numEdges> * 8	edge keys	No	Only present if <i>component bitmap</i> & 0x0008 == 0x0008.
4	number of vertex properties	Yes	
<num vertex properties> * <property layout>	property data	Yes	See Table 15-11 .
4	number of edge properties	Yes	
<num edge properties> * <property layout>	property data	Y	See Edge Property Layout .
<vertex labels layout>	vertex labels	No	Only present if <i>component bit</i> & 0x0002 == 0x0002.
<edge labels layout>	edge label	No	Only present if <i>component bit</i> & 0x0004 == 0x0004.
4	number of shared pools	Yes	
<shared pools size>	shared pools	No	

Table 15-5 (Cont.) File Layout

Size in bytes	Description	Required	Comment
<property names size>	property names	No	Only present if <i>component bit</i> & 0x0010 == 0x0010. See Table 15-20 .

Vertex Key Layout

The layout of vertex keys depends on the vertexKey type. PGB supports integer, long and string vertex keys.

Table 15-6 Integer Vertex Keys

Size in bytes	Description	Required	Comment
<numVertices> * 4	key data	Yes	For each vertex, the corresponding integer key value.

Table 15-7 Long Vertex Keys

Size in bytes	Description	Required	Comment
<numVertices> * 8	key data	Yes	For each vertex, the corresponding long key value.

Table 15-8 String Vertex Keys

Size in bytes	Description	Required	Comment
4	compression scheme	Yes	reserved (must be 0)
8	property size	Yes	size of each element in bytes in the following data
<number of keys> * <string key element layout>	string key data	Yes	content of the vertex keys (see Table 15-6)

Table 15-9 String Key Element Layout

Size in bytes	Description	Required	Comment
4	string length	Yes	length of the string in bytes
<string length>	string key data	Yes	content of the string as bytes, No zero-character

Property Layout

The following shows the special layout for string properties, and for vector properties:

Table 15-10 Primitive Type Layout

Size in bytes	Description	Required	Comment
4	property type	Yes	See Table 15-4 for type encoding.
8	property size	Yes	Size of the property data in bytes
<property size>	property data	Yes	Stored as <numVertices/ numEdges> * <type size>

Table 15-11 Vector Property Layout

Size in bytes	Description	Comment
4	vector type mark	Always equal to 18.
8	size of vector property data and extra fields	dataSize = <sizeof component-type> * <dimension> + 8 (The 8 extra bytes are for the added following 2 extra fields in the vector property header.)
4	vector component data type	Valid types are integer, long, float, double. Encoded with the value specified in Table 15-4 .
4	vector dimension	Number of components per vector value. Must be greater than 0 to be a valid vector property.
dataSize - 8	data	Stored as array of length *` in which the value of the j-th component of the vector for the i-th entity is at position i * + j`.

Table 15-12 String Type Layout

Size in bytes	Description	Required	Comment
4	property type	Yes	Must be 7.
8	property size	Yes	Size of the following data in bytes.
1	reserved	Yes	Reserved (must be 0).
<dictionary layout>	dictionary	Yes	String dictionary used in the property
<numVertices/ numEdges> * 8	property content	Yes	Content of the string property, stored as IDs that refer to the strings in the dictionary.

Table 15-13 String Dictionary Layout

Size in bytes	Description	Required	Comment
1	reserved	Yes	Reserved (must be 0).
8	number of strings	Yes	Number of strings in the following dictionary.

Table 15-13 (Cont.) String Dictionary Layout

Size in bytes	Description	Required	Comment
<number of strings> * <dictionary element layout>	dictionary data	Yes	See Table 15-14 .

Table 15-14 String Dictionary Element Layout

Size in bytes	Description	Required	Comment
8	string id	Yes	Unique ID of the string.
4	string length	Yes	Length of the string in bytes.
<string length>	string data	Yes	Content of the string as bytes, No zero-character

Vertex Labels Layout**Table 15-15 Vertex Labels Layout**

Size in bytes	Description	Required	Comment
4	type	Yes	Must be 11.
8	size	Yes	Size of the following data in bytes.
<dictionary layout>	dictionary	Yes	String dictionary used in the vertex labels.
<numVertices + 1> * 8	string id begin array	Yes	<string ids> offset array for each vertex.
8	number of string ids	Yes	The number of string ids.
<number of string ids> * 8	string ids	Yes	Array of string ids in the string dictionary.

Edge Label Layout

The edge label layout follows the string type layout.

Shared Pools Layout**Table 15-16 Shared Pools Layout**

Size in bytes	Description	Required	Comment
1	type	Yes	1: enum, 2: prefixed

Table 15-17 Type == Enum

Size in bytes	Description	Required	Comment
8	num strings	Yes	
<number of strings> * <string table layout>	dictionary data	Yes	See Table 15-19 .

Table 15-18 Type == Prefix

Size in bytes	Description	Required	Comment
8	num prefixes	Yes	
<number of prefixes> * <string table layout>	dictionary data	Yes	See Table 15-19 .
8	num suffixes	Yes	
<number of suffixes> * <string table layout>	dictionary data	Yes	See Table 15-19 .

Table 15-19 String Table for Shared Pools

Size in bytes	Description	Required	Comment
8	string id	Yes	String can be literal (in case of enum) or prefix/suffix (in case of prefix).
4	string length	Yes	
<string length>	string data	Yes	

Property Names Layout**Table 15-20 Property Names Layout**

Size in bytes	Description	Required	Comment
8	size	Yes	String can be literal (in case of enum) or prefix/suffix (in case of prefix).
<sum of size of vertex property names>	vertex property names	No	Follows the String Key Element Layout. See Table 15-9 .

Table 15-20 (Cont.) Property Names Layout

Size in bytes	Description	Required	Comment
<sum of size of edge property names>	edge property names	No	Follows the String Key Element Layout. See Table 15-9 .

15.2 Loading Graph Data in Parallel from Multiple Files

You can load a graph in parallel using multiple files.

The following example demonstrates how to load graph data from multiple files.

For example, consider a vertex file split into four partitions as shown:

```
vertex_file1

1,Color,1,red,,
2,Color,1,yellow,,

vertex_file2

3,Color,1,blue,,
4,Color,1,green,,

vertex_file3

5,Color,1,orange,,
6,Color,1,white,,

vertex_file4

7,Color,1,black,,
```

The edge file is split into two partitions as shown:

```
edge_file1

1,1,2,edge1,Weight,4,,1.0,
2,2,3,edge2,Weight,4,,2.0,
3,3,4,edge3,Weight,4,,3.0,

edge_file2

4,4,5,edge4,Weight,4,,4.0,
5,5,6,edge5,Weight,4,,5.0,
6,6,7,edge6,Weight,4,,6.0,
```

The following graph configuration can be used to load the graph data from four vertex files and two edge files into the same graph. Note that all the `uris` are specified inside the JSON graph configuration.

```
{
  "format": "flat_file",
  "vertex_uris": ["vertex_file1", "vertex_file2", "vertex_file3",
"vertex_file4"],
  "edge_uris": ["edge_file1", "edge_file2"],
  "separator": ",",
  "edge_props": [
    {
      "name": "Weight",
      "type": "double"
    }
  ],
  "vertex_props": [
    {
      "name": "Color",
      "type": "string"
    }
  ]
}
```

You can also create a graph configuration with multiple file partitions using Java as shown:

```
FileGraphConfig config = GraphConfigBuilder
    .forFileFormat(Format.FLAT_FILE)
    .setSeparator(",")
    .addVertexUri("vertex_file1")
    .addVertexUri("vertex_file2")
    .addVertexUri("vertex_file3")
    .addVertexUri("vertex_file4")
    .addEdgeUri("edge_file1")
    .addEdgeUri("edge_file2")
    .addVertexProperty("Color", PropertyType.STRING)
    .addEdgeProperty("Weight", PropertyType.DOUBLE)
    .build();
```

 **Note:**

The graph configuration in the preceding codes include one double edge property named "Weight" and one string vertex property named "Color".

You can now load the graph data from the files as explained in [Creating a graph using graph builder API](#).

The graph server (PGX) will automatically load the graph in parallel, using one thread for each file. This means that a graph can be loaded in parallel with as many threads as files are given depending on the configured parallelism for the graph server (PGX) instance.

**Note:**

Since the graph config will be used for all of the specified files, it is crucial to use the same format for all these files, that is, using the same separator, having the same defined properties, complying with the same format specification.

15.3 Exporting Graphs Into a File

The graph server (PGX) allows the client to export a currently loaded graph into a file.

Using the `store()` method on any `PgxGraph` object, the client can specify which file format to store the graph in. The client can also dynamically select the set of properties to be stored with the graph, that is, not all the properties need to be exported. The client can specify a `CompressionScheme` to use when storing as shown:

Table 15-21 Files CompressionScheme

CompressionScheme	Supported Formats
NONE	All formats
GZIP	ADJ_LIST, EDGE_LIST, FLAT_FILE, TWO_TABLES (text)

The client can export to multiple files as well.

When PGX exports the specified graph into a file, PGX also creates a graph config which the client receives as return value. This is to help loading the created graph instance later.

When exporting graph data into multiple files a `FileGraphStoringConfig` can be used which contains the following JSON fields:

Table 15-22 Graph Configuration when Exporting Graph into Multiple Files

Field	Type	Description	Default
<code>base_path</code>	string	Base path to use for storing a graph; file paths will be constructed using the following format <code>_. _</code> , that is, <code>parent_path/my_graph_1.edges</code> .	null
<code>compression_scheme</code>	enum[none, gzip]	The scheme to use for compression, or none to disable compression.	none
<code>delimiter</code>	character	Delimiter character used as separator when storing. The characters <code>"</code> , <code>{</code> , <code>}</code> and <code>\n</code> cannot be used as delimiters.	null
<code>edge_extension</code>	string	The extension to use when creating edge file partitions.	edges

Table 15-22 (Cont.) Graph Configuration when Exporting Graph into Multiple Files

Field	Type	Description	Default
<code>initial_partition_index</code>	integer	The value used as initial partition index, that is, <code>initial_partition_index=1024 -> my_graph_1024.edges</code> , <code>my_graph_1025.edges</code> .	1
<code>num_partitions</code>	integer	The number of partitions that should be created, when exporting to multiple files.	1
<code>row_extension</code>	string	The extension to use when creating row file partitions.	rows
<code>vertex_extension</code>	string	The extension to use when creating vertex file partitions.	nodes

- [Exporting a Graph to Disk](#)

15.3.1 Exporting a Graph to Disk

You can save a graph loaded into memory to the disk in various formats. Therefore you can make sub-graphs and graph data computed at runtime through analytics persistent, for future use. The resulting file can be used later as input for the graph server (PGX).

Consider the following example where a graph is loaded into memory and PageRank analysis is executed on the graph.

Loading a Graph and Executing PageRank Analysis Using JShell

```
var g = session.readGraphWithProperties("<path_to_json>")
var rank = analyst.pagerank(g, 0.001, 0.85, 100)
```

Loading a Graph and Executing PageRank Analysis Using Java

```
PgxGraph g = session.readGraphWithProperties("<path_to_json>");
Analyst analyst = session.createAnalyst();
VertexProperty<Integer, Double> rank = analyst.pagerank(g, 0.001, 0.85, 100);
```

Loading a Graph and Executing PageRank Analysis Using Python

```
g = session.read_graph_with_properties("<path_to_json>")
analyst = session.create_analyst()
rank = analyst.pagerank(g, 0.001, 0.85, 100)
```

You can now store the graph, together with the result of the PageRank analysis and all original edge properties, as a file in edge-list format, on disk. When a graph is stored, you need to specify the graph format, a path where the file should be stored, the properties to store and a flag that specifies whether or not a file should be overwritten should a file with the same name already exist.

Storing a Graph Using JShell

```
var config = g.store(Format.EDGE_LIST, "<file-path>", List.of(rank), EdgeProperty.ALL, false)
```

Storing a Graph Using Java

```
var config = g.store(Format.EDGE_LIST, "<file-path>", List.of(rank), EdgeProperty.ALL, false);
```

Storing a Graph Using Python

```
config = g.store('edge_list', "<file-path>", vertex_properties = [rank], overwrite=False)
```

The graph data can now be found under the file path. The graph configuration returned by the `store` method can be used to load the new graph back into memory. To persist the graph configuration to disk as well, you can use the config's `toString` method to get a JSON representation:

Reloading a Graph Using JShell

```
var path = Paths.get("<file-path>")
Files.writeString(path, config.toString())
```

Reloading a Graph Using Java

```
import apache.commons.io.*; // PGX contains a version of Apache Commons IO
...
FileUtils.write(new File("<file-path>"), config.toString());
```

Reloading a Graph Using Python

```
with open("<file-path>", "w"):
    f.write(str(config))
```

15.4 Exporting a Graph into Multiple Files

You can store a graph into multiple files using the `store` method. Most parameters are the same, as if storing to a single file. However, the main difference lies in specifying how to partition the data.

You can partition the data in either of the following two ways:

- specifying a `FileGraphStoringConfig` (see [Table 15-22](#) for more information)
- specifying a base path and the number of partitions

Export into Multiple Files Using `FileGraphStoringConfig`

You can specify a more detailed way of creating the multiple partitions used to store the graph by using the `FileGraphStoringConfig`. You can create a `FileGraphStoringConfig` object using a `FileGraphStoringConfigBuilder`.

For example, the following code specifies that the storing should be done into four partitions using the specified base path and using zero as the initial index for the partitioning. It also

contains the file extension to use for vertex files and for edge files and finally it sets comma as the delimiter to be used when storing the graph data:

```
FileGraphStoringConfig storingConfig = new
FileGraphStoringConfigBuilder(basePath) //
    .setNumPartitions(4) //
    .setInitialPartitionIndex(0) //
    .setVertexExtension(vertexExtension) //
    .setEdgeExtension(edgeExtension) //
    .setDelimiter(',') //
    .build();
```

You can also partition all tables equally using the `numPartitions` parameter. This implies that all tables are exported into the same number of files.

If you do not want to partition the tables equally, you can either create one `PartitionedGraphConfig` which contains for each provider a `FileGraphStoringConfig` (see [Table 15-22](#)) or we can use a version of `store()` that takes two maps of `FileGraphStoringConfigs`, one for the vertex tables and one for the edge tables.

For the first option, you can create for each vertex and edge table a `FileGraphStoringConfig` and put it into a `FileEntityProviderConfig` using `setStoringOptions` in the builder of `FileEntityProviderConfig`. The providers are then added to the `PartitionedGraphConfig` as edge and vertex providers using `addVertexProvider()` and `addEdgeProvider()` in the builder of `PartitionedGraphConfig`. Later you can use the `store()` method which takes the `PartitionedGraphConfig` as parameter.

The second option creates for every edge and vertex table a storing configuration, adds those into a vertex provider and an edge provider map and calls the corresponding `store()` method with these maps as parameters.

For example:

```
FileGraphStoringConfig vertexStoringConfig1 = new
FileGraphStoringConfigBuilder(basePath + "_vertexTable1") //
    .setNumPartitions(4) //
    .setInitialPartitionIndex(0) //
    .setVertexExtension(vertexExtension) //
    .setDelimiter(',') //
    .build();

FileGraphStoringConfig vertexStoringConfig2 = new
FileGraphStoringConfigBuilder(basePath + "_vertexTable2") //
    .setNumPartitions(4) //
    .setInitialPartitionIndex(0) //
    .setVertexExtension(vertexExtension) //
    .setDelimiter(',') //
    .build();

FileGraphStoringConfig edgeStoringConfig1 = new
FileGraphStoringConfigBuilder(basePath + "_edgeTable1") //
    .setNumPartitions(4) //
    .setInitialPartitionIndex(0) //
```



```
.setEdgeExtension(edgeExtension) //  
.setDelimiter(',') //  
.build();  
  
Map<String, FileGraphStoringConfig> vertexStoringConfigs = new HashMap<>();  
vertexStoringConfigs.put("vertexTable1", vertexStoringConfig1);  
vertexStoringConfigs.put("vertexTable2", vertexStoringConfig2);  
  
Map<String, FileGraphStoringConfig> edgeStoringConfigs = new HashMap<>();  
edgeStoringConfigs.put("edgeTable1", edgeStoringConfig);
```

Export into Multiple Files without FileGraphStoringConfig

If you only need to specify how many partitions are required and the base name to be used, it is simpler to use `store()` method by only specifying those parameters. Following this procedure, the graph server (PGX) will use defaults for the other fields. See [Table 15-22](#) for more information on default values.

Export into Multiple Files Using a Graph Configuration Object

An alternate way for exporting into multiple files is by creating a `FileGraphStoringConfig` and putting it into a `Graph Configuration` object using `setStoringOptions` in its builder, and then using the corresponding version of the `store()` method.

16

Log Management in the Graph Server (PGX)

The graph server (PGX) internally uses the SLF4J interface with Log4j as the default logger implementation.

- [Configuring Log4j Logging](#)

16.1 Configuring Log4j Logging

The default log4j logging configuration file is located in `/etc/oracle/graph/log4j2-server.xml`. This configuration file contains the target location for the logs in `/var/log/oracle/graph/`. Additionally, the rolling file appenders are also defined in this configuration file.

Note:

- Log4j is configured to roll the log files based on both log size (250 MB) and date.
- Log files are automatically saved in a compressed format in subdirectories, one directory per month. There can be multiple files on a given day.
- Also, each startup of the graph server(PGX) triggers a new log file.

The log4j configuration file is picked up automatically by the the graph server(PGX). To use this configuration in your java application, you can set the `log4j.configurationFile` system variable when launching the JVM:

```
java -Dlog4j.configurationFile=$PGX_HOME/conf/log4j2.xml ...
```

Changing Logging Level During a JShell Session

When connected to the graph server using JShell, you can use the `loglevel(String loggerName, String levelName)` function to quickly change the logging level of any logger. For example:

```
loglevel("oracle.pgx", "debug")
loglevel("ROOT", "info")
loglevel("org.apache.hadoop", "off")
```

Changing Slf4j Implementation

You can replace the log4j JARs in `$PGX_HOME/third-party` with your own slf4j implementation. You must only place your JAR files in `$PGX_HOME/third-party` and it will get wild-card included when the graph shell client is started.

Logging in a Web Application Server

The `graph-server-<version>-pgx<version>.war` file in the `oracle-graph-webapps-<version>.zip` download package contains the `log4j2.xml`. This file determines what should be logged in the web application running on the application server of your choice. The file is located in the folder `WEB-INF/classes` inside the `graph-server-<version>-pgx<version>.war` file. By default, only errors are logged. But you can change this file if you want more logging in your web server. You must restart the web server after you change the file, for the change to take effect.

Part III

Supplementary Information for Property Graph Support

This document has the following appendixes.

- [Handling Property Graphs Using a Two-Tables Schema](#)
For property graphs with relatively fixed, simple data structures, where you do not need the flexibility of `<graph_name>VT$` and `<graph_name>GE$` key/value data tables for vertices and edges, you can use a two-tables schema to achieve better run-time performance.
- [About Property Graph Data Formats](#)
Several graph formats are supported for property graph data.
- [Mapping Graph Server Roles to Default Privileges](#)
- [Disabling Transport Layer Security \(TLS\) in Graph Server](#)

A

Handling Property Graphs Using a Two-Tables Schema

For property graphs with relatively fixed, simple data structures, where you do not need the flexibility of `<graph_name>VT$` and `<graph_name>GE$` key/value data tables for vertices and edges, you can use a two-tables schema to achieve better run-time performance.



Note:

Support for the two-tables schema approach described in this topic has been deprecated and will probably be removed in a future release.

Instead, you are encouraged use the property graph schema approach to working with graph data, described in [Property Graph Schema Objects for Oracle Database](#).

The two-tables schema approach is a deprecated alternative to the recommended approach of using the property graph schema (described in [Property Graph Schema Objects for Oracle Database](#)).

- The property graph schema approach is designed mainly for heterogeneous and/or large graphs. When a graph model is used to present a dynamic application domain in which new relationships and possibly new data types for the same property name(s) are introduced and added to the graph model on the fly, using the property graph schema is recommended.

When a graph model is used to present a dynamic application domain in which new relationships and possibly new data types for the same property name(s) are introduced and added to the graph model on the fly, using the property graph schema is recommended.

- The two-tables schema approach is designed for homogenous graphs.

If a graph model represents an application domain where the set of relationships is already known and the total number of distinct relationships is relatively small (less than 1000), then the two-tables approach is a potential option. This situation usually happens when the original data source is from one or a set of existing relational tables or views.

An example of where the two-tables approach might be useful is if all nodes are employees of a specific organization, and each employee has a limited and fixed set of attributes and potential relationships. An example of where the two-tables approach would not be useful is if the nodes can be any individuals who can have different attributes and relationships, and where attributes and relationships can be dynamically added and altered.

In the flexible key/value approach (*not* two-tables), Oracle Spatial and Graph stores property graph data with a flexible schema: `<graph_name>VT$` for vertices and `<graph_name>GE$` for edges. In this schema, vertices and edges are stored using multiple rows where each row represents a key/value property associated with the vertex (or the edge) with a flexible data type, determined by the attribute `T` (type). This schema design can easily accommodate a

heterogeneous graph where vertices (edges) have different set of properties or data types of property values.

On the other hand, for a property graph with a homogeneous structure, you can store graph data using a two-tables schema. With this approach, each vertex is stored as a single row in a named vertex table, and each edge as a single row in a named edge table. This way, each column in the row corresponds to a property with a fixed data type. The in-memory analyst can then use this approach to construct and manage the in-memory graphs.

**Note:**

The two-tables approach is mainly for providing graph data for the in-memory analyst to existing Blueprints-based Java APIs, and text indexing does **not** work with the two-tables approach.

Graph data change tracking is only available when the property graph schema approach is used.

The following topics focus on how to create a property graph using a two-tables schema, as well as how to execute read and write operations over this data.

- [Preparing the Two-Tables Schema](#)
- [Storing Data in a Property Graph Using a Two-Tables Schema](#)
- [Reading Data from a Property Graph Using a Two-Tables Schema](#)

A.1 Preparing the Two-Tables Schema

`OraclePropertyGraphUtils.prepareTwoTablesGraphVertexTab` lets you customize the schema of a vertex table using a two-tables schema to store all the vertices in a graph. This operation requires a connection to an Oracle database, the table owner, the table name, and two arrays specifying the property names and their data types. By default, the table schema of the generated table includes the attribute `VID`, which represents the primary key of the table and is mapped to the vertex ID.

The following code snippet creates a vertex table using a two-tables schema. In this case, the generated table `employeesNodes` will include four attributes: `name`, `age`, `address`, and `SSN` (Social Security Number). The primary key of the vertex table is the generated attribute `VID`.

```
import oracle.pgx.common.types.PropertyType;
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[4]{ "name", "age", "address", "SSN" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.INTEGER);
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.STRING);

OraclePropertyGraphUtils.prepareTwoTablesGraphVertexTab(conn /*
Connection object */,
```

```

name */,
*/,
types */,

pg /* table owner */,
"employeesNodes" /* vertex table

propertyNames /* property names

propertyTypes /* property data

"pgts" /* table space */,
null /* storage options */,
true /* no logging */);

```

The preceding code produces a table schema as follows:

```

CREATE TABLE employeenodes
( VID number not null,
  NAME nvarchar2(15000),
  AGE integer,
  ADDRESS nvarchar2(15000),
  SSN nvarchar2(15000),
  CONSTRAINT employenodes_pk PRIMARY KEY (VID)
);

```

Similarly, `OraclePropertyGraphUtils.prepareTwoTablesGraphEdgeTab` lets you customize the schema of an edge table using a two-tables schema to store all the edges in a graph. This operation requires a connection to an Oracle database, the table owner, the table name, a two arrays specifying the property names and their data types. By default, the table schema of the generated table includes the following attributes: `EID`, which represents the primary key of the table and is mapped to the edge ID; `EL`, which is mapped to the edge label; and `SVID` and `DVID` for the source and destination vertex IDs, respectively.

The following code snippet creates an edge table using a two-tables schema. In this case, the generated table `organizationEdges` will include the attribute named `weight`. The primary key of the vertex table is the generated attribute `EID`, which is the default attribute of the table schema, mapped to the vertices' ID (long value) values.

```

import oracle.pgx.common.types.PropertyType;
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[1]{ "weight" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.DOUBLE);
OraclePropertyGraphUtils.prepareTwoTablesGraphEdgeTab(conn /* Connection
object */,

name */,
*/,
types */,

pg /* table owner */,
"organizationEdges" /* edge table

propertyNames /* property names

propertyTypes /* property data

"pgts" /* table space */,
null /* storage options */,
true /* no logging */);

```

The preceding code produces a table structure as follows:

```
CREATE TABLE organizationedges
( EID number not null,
  SVID number not null,
  DVID number not null,
  EL nvarchar2(3100),
  WEIGHT number,
  CONSTRAINT organizationedges_pk PRIMARY KEY (EID)
);
```

Note that if the table already exists, both `prepareTwoTablesGraphEdgeTab` and `prepareTwoTablesGraphEdgeTab` will truncate the table contents.

A.2 Storing Data in a Property Graph Using a Two-Tables Schema

To load a set of vertices into a vertex table using a two-tables schema, you can use the API `OraclePropertyGraphUtils.writeTwoTablesGraphVertexAndProperties`. This operation takes an array of `Iterable` (or `Iterator`) of `TinkerPop Blueprints Vertex` objects, and reads out the ID and the values for the properties defined in the vertex table schema. Based on this information, the vertex is later inserted as a new row in the vertex table. Note that if a vertex does not include a property defined in the schema, the value for that associated column is set to `NULL`.

The following code snippet creates a property graph `employeesGraphDAL` using the `OraclePropertyGraph` API, and loads two vertices and an edge. Then, it creates a vertex table `employeesNodes` using a two-tables schema and populates it with the data from the vertices in `employeesGraphDAL`. Note that the property `email` in the vertex `v1` is not loaded into the `employeesNode` table because it is not defined in the schema. Also, the property `SSN` for vertex `v2` is set `NULL` because it is not defined in the vertex.

```
// Create employeesGraphDAL
import oracle.pg.rdbms.*;
Oracle oracle = new Oracle(jdbcURL, username, password);
OraclePropertyGraph opgEmployees
    = OraclePropertyGraph.getInstance(oracle,
    "employeesGraphDAL");

// Create vertex v1 and assign it properties as key-value pairs
Vertex v1 = opgEmployees.addVertex(11);
v1.setProperty("age", Integer.valueOf(31));
v1.setProperty("name", "Alice");
v1.setProperty("address", "Main Street 12");
v1.setProperty("email", "alice@mymail.com");
v1.setProperty("SSN", "123456789");

Vertex v2 = opgEmployees.addVertex(21);
v2.setProperty("age", Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(11, v1, v2, "managerOf");
e1.setProperty("weight", 0.5d);
```



```

opgEmployees.commit();

// Prepare the vertex table using a Two Tables schema
import oracle.pgx.common.types.PropertyType;
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[4]{ "name", "age", "address", "SSN" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.INTEGER);
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.STRING);

Connection conn
    = opgEmployees.getOracle().clone().getConnection(); /* Clone the
connection
                                                    from the
property graph
                                                    instance */
OraclePropertyGraphUtils.prepareTwoTablesGraphVertexTab(conn /* Connection
object */,
                                                    pg /* table owner */,
                                                    "employeesNodes" /* vertex table
name */,
                                                    propertyNames /* property names
*/,
                                                    propertyTypes /* property data
types */,
                                                    "pgts" /* table space */,
                                                    null /* storage options */,
                                                    true /* no logging */);

// Get the vertices from the employeesDAL graph
Iterable<Vertex> vertices = opgEmployees.getVertices();

// Load the vertices into the vertex table using a Two-Tables schema
Connection[] conns = new Connection[1]; /* the connection array size defines
the
                                                    Degree of parallelism
(multithreading)
                                                    */
conns[1] = conn;
OraclePropertyGraphUtils.writeTwoTablesGraphVertexAndProperties(
                                                    conn /* Connectionobject */,
                                                    pg /* table owner */,
                                                    "employeesNodes" /* vertex table
name */,
                                                    1000 /* batch size*/,
                                                    new Iterable[] {vertices} /*
array of
                                                    vertex
iterables */);

```

To load a set of edges into an edge table using a two-tables schema, you can use the API `OraclePropertyGraphUtils.writeTwoTablesGraphEdgesAndProperties`. This operation

takes an array of Iterable (or Iterator) of Blueprints Edge objects, and reads out the ID, EL, SVID, DVID, and the values for the properties defined in the edge table schema. Based on this information, the edge is later inserted as a new row in the edge table. Note that if an edge does not include a property defined in the schema, the value for that given column is set to NULL.

The following code snippet creates a property graph `employeesGraphDAL` using the `OraclePropertyGraph` API, and loads two vertices and an edge. Then, it creates a vertex table `organizationEdges` using a two-tables schema, and populates it with the data from the edges in `employeesGraphDAL`.

```
// Create employeesGraphDAL
import oracle.pg.rdbms.*;
Oracle oracle = new Oracle(jdbcURL, username, password);
OraclePropertyGraph opgEmployees
    = OraclePropertyGraph.getInstance(oracle,
"employeesGraphDAL");

// Create vertex v1 and assign it properties as key-value pairs
Vertex v1 = opgEmployees.addVertex(11);
v1.setProperty("age", Integer.valueOf(31));
v1.setProperty("name", "Alice");
v1.setProperty("address", "Main Street 12");
v1.setProperty("email", "alice@mymail.com");
v1.setProperty("SSN", "123456789");

Vertex v2 = opgEmployees.addVertex(21);
v2.setProperty("age", Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(11, v1, v2, "managerOf");
e1.setProperty("weight", 0.5d);

opgEmployees.commit();

// Prepare the edge table using a Two Tables schema
import oracle.pg.common.types.PropertyType;
Connection conn
    = opgEmployees.getOracle().clone().getConnection(); /*
Clone the connection
                                                                    from
the property graph

instance */
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[1]{ "weight" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.DOUBLE);
OraclePropertyGraphUtils.prepareTwoTablesGraphEdgeTab(conn /*
Connection object */,
                                                                    pg /* table owner */,
                                                                    organizationEdges" /* edge
```

```

table name */,
                                     propertyNames /* property names
*/,
                                     propertyTypes /* property data
types */,
                                     "pgts" /* table space */,
                                     null /* storage options */,
                                     true /* no logging */);

// Get the edges from the employeesDAL graph
Iterator<Edge> edges = opgEmployees.getEdges().iterator();

// Load the edges into the edges table using a Two-Tables schema
Connection[] conns = new Connection[1]; /* the connection array size defines the
                                     Degree of parallelism
(multithreading)
                                     */
conns[1] = conn;
OraclePropertyGraphUtils.writeTwoTablesGraphVertexAndProperties(conn /*
Connection
object */,
                                     pg /* table owner */,
                                     "organizationEdges" /* edge
table
                                     name */,
                                     1000 /* batch size*/,
                                     new Iterator[] {edges} /* array
of
                                     iterator of edges
*/);

```

To optimize the performance of the storing operations, you can specify a set of flags and hints when calling the `writeTwoTablesGraph` APIs. These hints include:

- **DOP:** Degree of parallelism. The size of the connection array defines the degree of parallelism to use when loading the data. This determines the number of chunks to generate when reading the Iterables as well as the number of loader threads to use when loading the data into the table.
- **Batch Size:** An integer specifying the batch size to use for Oracle update statements in batching mode. A recommended batch size is 1000.

A.3 Reading Data from a Property Graph Using a Two-Tables Schema

To read a subset of vertices from a vertex table using a two-tables schema, you can use the API `OraclePropertyGraphUtils.readTwoTablesGraphVertexAndProperties`. This operation returns an array of `ResultSet` objects with all the rows found in the corresponding splits of the vertex table. Each `ResultSet` object in the array uses one of the connections provided to fetch the vertex rows from the corresponding split. The splits are determined by the specified number of total splits.

An integer ID (in the range of [0, N - 1]) is assigned to the splits in the vertex table with N splits. This way, the subset of splits queried will consist of those splits with ID value in the range between the start split ID and the start split ID plus the size of the connection array. If the sum is greater than the total number of splits, then the subset of splits queried will consist of those splits with ID in the range of [start split ID, N - 1].

The following code reads all vertices from a vertex table using a two-tables schema using a total of 1 split. Note that you can easily create an array of Blueprints Vertex Iterables by executing the API on `OraclePropertyGraph`. The vertices retrieved will include all the properties defined in the vertex table schema.

```
ResultSet[] rsAr = readTwoTablesGraphVertexAndProperties(conns,
                                                         "pg" /* table owner */,
                                                         "employeeNodes" /*
vertex table
                                                         name
*/,
                                                         1 /* Total Splits*/,
                                                         0 /* Start Split});

Iterable<Vertex>[] vertices = getVerticesPartitioned(rsAr /* ResultSet
array */,
                                                         true /* skip store
to cache */,
                                                         null /* vertex
filter
                                                         callback
*/,
                                                         null /*
optimization flag */);
```

To optimize reading performance, you can specify the list of property names to retrieve for each vertex read from the table.

The following code creates a property graph `employeesGraphDAL` using the `OraclePropertyGraph` API, and loads two vertices and an edge. Then, it creates a vertex table `employeeNodes` using a two-tables schema, and populates it with the data from the vertices in `employeesGraphDAL`. Finally, it reads the vertices out of the vertex table using only the name property.

```
// Create employeesGraphDAL
import oracle.pg.rdbms.*;
Oracle oracle = new Oracle(jdbcURL, username, password);
OraclePropertyGraph opgEmployees
    = OraclePropertyGraph.getInstance(oracle,
    "employeesGraphDAL");

// Create vertex v1 and assign it properties as key-value pairs
Vertex v1 = opgEmployees.addVertex(11);
v1.setProperty("age", Integer.valueOf(31));
v1.setProperty("name", "Alice");
v1.setProperty("address", "Main Street 12");
v1.setProperty("email", "alice@mymail.com");
v1.setProperty("SSN", "123456789");
```

```

Vertex v2 = opgEmployees.addVertex(21);
v2.setProperty("age", Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(11, v1, v2, "managerOf");
e1.setProperty("weight", 0.5d);

opgEmployees.commit();

// Prepare the vertex table using a Two Tables schema
import oracle.pgx.common.types.PropertyType;
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[4]{ "name", "age", "address", "SSN" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.INTEGER);
propertyType.add(PropertyType.STRING);
propertyType.add(PropertyType.STRING);

Connection conn
    = opgEmployees.getOracle().clone().getConnection(); /* Clone the
connection
                                                    from the
property graph
                                                    instance */
OraclePropertyGraphUtils.prepareTwoTablesGraphVertexTab(conn /* Connection
object */,
                                                    pg /* table owner */,
                                                    "employeesNodes" /* vertex table
name */,
                                                    propertyNames /* property names
*/,
                                                    propertyTypes /* property data
types */,
                                                    "pgts" /* table space */,
                                                    null /* storage options */,
                                                    true /* no logging */);

// Get the vertices from the employeesDAL graph
Iterable<Vertex> vertices = opgEmployees.getVertices();

// Load the vertices into the vertex table using a Two Tables schema
Connection[] conns = new Connection[1]; /* the connection array size defines the
Degree of parallelism
(multithreading)
*/
conns[1] = conn;
OraclePropertyGraphUtils.writeTwoTablesGraphVertexAndProperties(conn /*
Connection

```

```

object */,
                                pg /* table owner */,
                                "employeesNodes" /* vertex
table name */,
                                1000 /* batch size*/,
                                new Iterable[]
{vertices} /* array of
                                vertex
iterables */);

// Read the vertices (using only name property)
List<String> vPropertyNames = new ArrayList<String>();
vPropertyNames.add("name");
ResultSet[] rsAr = readTwoTablesGraphVertexAndProperties(conns,
                                "pg" /* table owner */,
                                "employeeNodes" /*
vertex table
                                name
*/,
                                vPropertyNames /* list
of property
names */,
                                1 /* Total Splits*/,
                                0 /* Start Split});

Iterable<Vertex>[] vertices = getVerticesPartitioned(rsAr /* ResultSet
array */,
                                true /* skip store
to cache */,
                                null /* vertex
filter
                                callback
*/,
                                null /*
optimization flag */);

for (int idx = 0; vertices.length; idx++) {
    Iterator<Vertex> it = vertices[idx].iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}

```

The preceding code produces output similar to the following:

```

Vertex ID 1 {name:str:Alice}
Vertex ID 2 {name:str:Bob}

```

To read a subset of edges from an edge table using a two-tables schema, you can use the API `OraclePropertyGraphUtils.readTwoTablesGraphEdgeAndProperties`. This operation returns an array of `ResultSet` objects with all the rows found in the corresponding splits of the vertex table. Each `ResultSet` object in the array uses one of the connections provided to fetch the vertex rows from the corresponding split. The splits are determined by the specified number of total splits.

Similar to what is done for reading vertices, an integer ID (in the range of [0, N - 1]) is assigned to the splits in the vertex table with N splits. The subset of splits queried will consist of those splits with ID value in the range between the start split ID and the start split ID plus the size of the connection array.

The following code creates a property graph `employeesGraphDAL` using the `OraclePropertyGraph` API, and loads two vertices and an edge. Then, it creates an edge table `organizationEdges` using a two-tables schema, and populates it with the data from the edges in `employeesGraphDAL`. Finally, it reads the edges out of table using only the name weight.

```
// Create employeesGraphDAL
import oracle.pg.rdbms.*;
Oracle oracle = new Oracle(jdbcURL, username, password);
OraclePropertyGraph opgEmployees
    = OraclePropertyGraph.getInstance(oracle,
"employeesGraphDAL");

// Create vertex v1 and assign it properties as key-value pairs
Vertex v1 = opgEmployees.addVertex(11);
v1.setProperty("age", Integer.valueOf(31));
v1.setProperty("name", "Alice");
v1.setProperty("address", "Main Street 12");
v1.setProperty("email", "alice@mymail.com");
v1.setProperty("SSN", "123456789");

Vertex v2 = opgEmployees.addVertex(21);
v2.setProperty("age", Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(11, v1, v2, "managerOf");
e1.setProperty("weight", 0.5d);

opgEmployees.commit();

// Prepare the edge table using a Two Tables schema
import oracle.pgx.common.types.PropertyType;
List<String> propertyNames = new ArrayList<String>();
propertyNames.addAll(new String[4]{ "weight" });

List<PropertyType> = new ArrayList<PropertyType>();
propertyType.add(PropertyType.DOUBLE);

Connection conn
    = opgEmployees.getOracle().clone().getConnection(); /* Clone the
connection
property graph
*/
OraclePropertyGraphUtils.prepareTwoTablesGraphEdgeTab(conn /* Connection
object */,
```

```

pg /* table owner */,
"organizationEdges" /* edge
table
name
*/,
propertyNames /* property
names */,
propertyTypes /* property
data types */,
"pgts" /* table space */,
null /* storage options */,
true /* no logging */);

// Get the edges from the employeesDAL graph
Iterable<Edge> edges = opgEmployees.getVertices();

// Load the vertices into the vertex table using a Two Tables schema
Connection[] conns = new Connection[1]; /* the connection array size
defines the
Degree of parallelism
(multithreading)
*/
conns[1] = conn;
OraclePropertyGraphUtils.writeTwoTablesGraphEdgeAndProperties(conn /*
Connection
object */,
pg /* table owner */,
"organizationEdges" /* edge
table name */,
1000 /* batch size*/,
new Iterable[] {edges} /*
array of
edge
iterables */);

// Read the edges (using only weight property)
List<String> ePropertyNames = new ArrayList<String>();
ePropertyNames.add("weight");
ResultSet[] rsAr = readTwoTablesGraphVertexAndProperties(conns,
"pg" /* table owner */,
"organizationEdges" /*
edge table
name
*/,
ePropertyNames /* list
of property
names
*/,
1 /* Total Splits*/,
0 /* Start Split);

Iterable<Edge>[] edges = getEdgesPartitioned(rsAr /* ResultSet array */,

```



```
cache */,                                     true /* skip store to
                                              null /* edge filter
                                              callback */,
                                              null /* optimization

flag */);

for (int idx = 0; edges.length; idx++) {
    Iterator<Edge> it = edges[idx].iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

The preceding code produces output similar to the following:

```
Edge ID 1 from Vertex ID 1 {} =[references]=> Vertex ID 2 {} edgeKV[{weight:dbl:0.5}]
```

B

About Property Graph Data Formats

Several graph formats are supported for property graph data.

- [GraphSON Data Format](#)
- [GraphML Data Format](#)
- [GML Data Format](#)
- [Oracle Flat File Format](#)

B.1 GraphSON Data Format

The GraphSON file format is based on JavaScript Object Notation (JSON) for describing graphs.

The example in this topic shows a GraphSON description of the property graph shown in [What Are Property Graphs?](#).

Example B-1 GraphSON Description of a Simple Property Graph

```
{
  "graph": {
    "mode": "NORMAL",
    "vertices": [
      {
        "name": "Alice",
        "age": 31,
        "_id": "1",
        "_type": "vertex"
      },
      {
        "name": "Bob",
        "age": 27,
        "_id": "2",
        "_type": "vertex"
      }
    ],
    "edges": [
      {
        "type": "friends",
        "_id": "3",
        "_type": "edge",
        "_outV": "1",
        "_inV": "2",
        "_label": "knows"
      }
    ]
  }
}
```

Related Topics

- [GraphSON Reader and Writer Library](#)

B.2 GraphML Data Format

The GraphML file format uses XML to describe graphs.

The example in this topic shows a GraphML description of the property graph shown in [What Are Property Graphs?](#).

Example B-2 GraphML Description of a Simple Property Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <key id="name" for="node" attr.name="name" attr.type="string"/>
  <key id="age" for="node" attr.name="age" attr.type="int"/>
  <key id="type" for="edge" attr.name="type" attr.type="string"/>
  <graph id="PG" edgedefault="directed">
    <node id="1">
      <data key="name">Alice</data>
      <data key="age">31</data>
    </node>
    <node id="2">
      <data key="name">Bob</data>
      <data key="age">27</data>
    </node>
    <edge id="3" source="1" target="2" label="knows">
      <data key="type">friends</data>
    </edge>
  </graph>
</graphml>
```

Related Topics

- [GraphML File Format](#)

B.3 GML Data Format

The Graph Modeling Language (GML) file format uses ASCII to describe graphs.



Note:

GML Data Format is not supported in Tinkerpop 3, and it has been deprecated in Tinkerpop 2.

The example in this topic shows a GML description of the property graph shown in [What Are Property Graphs?](#).

Example B-3 GML Description of a Simple Property Graph

```
graph [
  comment "Simple property graph"
  directed 1
  IsPlanar 1
  node [
    id 1
    label "1"
    name "Alice"
```

```

    age 31
  ]
node [
  id 2
  label "2"
  name "Bob"
  age 27
  ]
edge [
  source 1
  target 2
  label "knows"
  type "friends"
  ]
]

```

Methods are provided to import and export graphs from and into GML format.

The following fragments of code show how to import and export GML data. Note that these methods are deprecated and their use is discouraged:

```

// Get graph instance
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args, szGraphName);

// Import graph in GML format
String fileName = "./mygraph.gml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.importGML(opg, fileName, ps);

// Export graph into GML format
String fileName = "./mygraph.gml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.exportGML(opg, fileName, ps);

```

Related Topics

- [GML: A Portable Graph File Format" by Michael Himsolt](#)

B.4 Oracle Flat File Format

The Oracle flat file format exclusively describes property graphs. It is more concise and provides better data type support than the other file formats. The Oracle flat file format uses two files for a graph description, one for the vertices and one for edges. Commas separate the fields of the records.

Example B-4 Oracle Flat File Description of a Simple Property Graph

The following shows the Oracle flat files that describe the simple property graph example shown in [What Are Property Graphs?](#)

Vertex file:

```

1,name,1,Alice,,
1,age,2,,31,
2,name,1,Bob,,
2,age,2,,27,

```

Edge file:

```

1,1,2,knows,type,1,friends,,

```

The following shows the flat file description of the same graph for Tinkerpop 3, which has an additional field for storing the vertex label.

Vertex file:

```
1,name,1,Alice,,,person
1,age,2,,31,,person
2,name,1,Bob,,,person
2,age,2,,27,,person
```

Edge file:

```
3,1,2,knows,type,1,friends,,
```

Methods are provided to import and export graphs from and into Flat File format.

The following fragments of code show how to export a graph into Oracle Flat File Format. To import graphs, see [Parallel Loading of Graph Data](#).

```
// Get graph instance
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args, szGraphName);

// Export graph into Flat File Format
String vertexFileName = "./mygraph.opv";
String edgeFileName = "./mygraph.ope";
int dop = 2;
Boolean append = false;
OraclePropertyGraphUtils.exportFlatFiles(opg, vertexFileName, edgeFileName, dop, append);
```

Related Topics

- [Oracle Flat File Format Definition](#)
A property graph can be defined in two flat files, specifically description files for the vertices and edges.

C

Mapping Graph Server Roles to Default Privileges

Installing the PL/SQL packages of the Oracle Graph Server and Client distribution on the target Oracle Database, automatically creates the following roles and assigns the default permissions as shown in the following table:

Table C-1 Mapping Graph Server Roles to Default Privileges

Roles	Description	Permission
GRAPH_ADMINISTRATOR	User who performs operations on the in-memory graph server (PGX) using the Java API. (As compared to running start and stop operations as an OS user.)	PGX_SESSION_CREATE PGX_SERVER_GET_INFO PGX_SERVER_MANAGE
GRAPH_DEVELOPER	User who creates graphs, publishes graphs, modifies graphs, queries graphs, and views graphs using the Java API or SQLcl or the graph visualization application.	PGX_SESSION_CREATE PGX_SESSION_NEW_GRAPH PGX_SESSION_GET_PUBLISHED_GRAPH PGX_SESSION_MODIFY_MODEL PGX_SESSION_READ_MODEL
GRAPH_USER	User who queries graphs and views graphs Java API or SQLcl or the graph visualization application.	PGX_SESSION_CREATE PGX_SESSION_GET_PUBLISHED_GRAPH

D

Disabling Transport Layer Security (TLS) in Graph Server

For demonstration or evaluation purposes, it is possible to turn off transport layer security (TLS) of the graph server.

Caution:

This is **not** recommended for production. In a secure configuration, the server must always have TLS enabled.

The following instructions only apply if you installed the graph server via the RPM package.

Note:

If you deployed the graph server into your own web server (e.g Weblogic or Apache Tomcat), please refer to the manual of your web server for TLS configuration.

1. Edit `/etc/oracle/graph/server.conf` to change `enable_tls` to `false`.
2. Edit the `WEB-INF/web.xml` file inside the `WAR` file in `/opt/oracle/graph/graphviz` and configure cookies to be sent over non-secure connections by setting `<secure>>false</secure>` as follows:

```
<session-config>
  <tracking-mode>COOKIE</tracking-mode>
  <cookie-config>
    <secure>>false</secure>
  </cookie-config>
  ...
</session-config>
```

3. Additionally, replace `https` with `http` in the `pgx.base_url` property in the same `WEB-INF/web.xml` file. For example:

```
<context-param>
  <param-name>pgx.base_url</param-name>
  <param-value>http://localhost:7007</param-value>
</context-param>
```

4. Restart the server.

```
sudo systemctl restart pgx
```

The graph server now accepts connections over HTTP instead of HTTPS.

On Oracle Linux 7, you can execute the following script to perform the preceding four steps all at once:

```
echo "$(jq '.enable_tls = false' /etc/oracle/graph/server.conf)" > /etc/oracle/graph/server.conf
WAR=$(find /opt/oracle/graph/graphviz -name '*.war')
TMP=$(mktemp -d)
cd $TMP
unzip $WAR WEB-INF/web.xml
sed -i 's|<secure>true</secure>|<secure>false</secure>|' WEB-INF/web.xml
sed -i 's|https://|http://|' WEB-INF/web.xml
sudo zip $WAR WEB-INF/web.xml
rm -r $TMP
sudo systemctl restart pgx
```


Index

A

ANALYZE_PG procedure, [9-2](#)
automatic delta refresh, [4-50](#)

C

CF procedure, [9-4](#)
CF_CLEANUP procedure, [9-7](#)
CF_PREP procedure, [9-9](#)
CLEAR_PG procedure, [9-10](#)
CLEAR_PG_INDICES procedure, [9-11](#)
CLONE_GRAPH procedure, [9-11](#)
collaborative filtering, [9-4](#), [9-7](#), [9-9](#)
connected components
 finding, [9-32](#)
COUNT_TRIANGLE function, [9-12](#)
COUNT_TRIANGLE_CLEANUP procedure, [9-13](#)
COUNT_TRIANGLE_PREP procedure, [9-14](#)
COUNT_TRIANGLE_RENUM function, [9-16](#)
CREATE_EDGES_TEXT_IDX procedure, [9-17](#)
CREATE_PG procedure, [9-18](#)
CREATE_PG_SNAPSHOT_TAB procedure, [9-19](#)
CREATE_PG_TEXTIDX_TAB procedure, [9-21](#)
CREATE_STAT_TABLE procedure, [9-22](#)
CREATE_SUB_GRAPH procedure, [9-23](#)
CREATE_VERTICES_TEXT_IDX procedure,
 [9-24](#)

D

DROP_EDGES_TEXT_IDX procedure, [9-26](#)
DROP_PG procedure, [9-26](#)
DROP_PG_VIEW procedure, [9-27](#)
DROP_VERTICES_TEXT_IDX procedure, [9-27](#)

E

edge table statistics
 exporting, [9-30](#)
 importing, [9-51](#)
ESTIMATE_TRIANGLE_RENUM function, [9-28](#)
EXP_EDGE_TAB_STATS procedure, [9-30](#)
EXP_VERTEX_TAB_STATS procedure, [9-31](#)

F

FIND_CC_MAPPING_BASED procedure, [9-32](#)
FIND_CLUSTERS_CLEANUP procedure, [9-33](#)
FIND_CLUSTERS_PREP procedure, [9-34](#)
FIND_SP procedure, [9-36](#)
FIND_SP_CLEANUP procedure, [9-37](#)
FIND_SP_PREP procedure, [9-38](#)

G

geometries
 getting, [9-39](#), [9-41](#)
 getting from longitude and latitude, [9-44](#)
 WKT, [9-48](#), [9-49](#)
GET_BUILD_ID function, [9-39](#)
GET_GEOMETRY_FROM_V_COL function,
 [9-39](#)
GET_GEOMETRY_FROM_V_T_COLS function,
 [9-41](#)
GET_LATLONG_FROM_V_COL function, [9-42](#),
 [9-45](#)
GET_LATLONG_FROM_V_T_COLS function,
 [9-43](#)
GET_LONG_LAT_GEOMETRY function, [9-44](#)
GET_LONGLAT_FROM_V_T_COLS function,
 [9-46](#)
GET_SCN function, [9-47](#)
GET_VERSION function, [9-47](#)
GET_WKTGEOMETRY_FROM_V_COL function,
 [9-48](#)
GET_WKTGEOMETRY_FROM_V_T_COLS
 function, [9-49](#)
GRANT_ACCESS procedure, [9-50](#)

I

IMP_EDGE_TAB_STATS procedure, [9-51](#)
IMP_VERTEX_TAB_STATS procedure, [9-52](#)
in-memory Graph server (PGX), [4-1](#)

O

OPG_APIS package

- ANALYZE_PG, [9-2](#)
- CF, [9-4](#)
- CF_CLEANUP, [9-7](#)
- CF_PREP, [9-9](#)
- CLEAR_PG, [9-10](#)
- CLEAR_PG_INDICES, [9-11](#)
- CLONE_GRAPH, [9-11](#)
- COUNT_TRIANGLE, [9-12](#)
- COUNT_TRIANGLE_CLEANUP, [9-13](#)
- COUNT_TRIANGLE_PREP, [9-14](#)
- COUNT_TRIANGLE_RENUM, [9-16](#)
- CREATE_EDGES_TEXT_IDX, [9-17](#)
- CREATE_PG, [9-18](#)
- CREATE_PG_SNAPSHOT_TAB, [9-19](#)
- CREATE_PG_TEXTIDX_TAB, [9-21](#)
- CREATE_STAT_TABLE, [9-22](#)
- CREATE_SUB_GRAPH, [9-23](#)
- CREATE_VERTICES_TEXT_IDX, [9-24](#)
- DROP_EDGES_TEXT_IDX, [9-26](#)
- DROP_PG, [9-26](#)
- DROP_PG_VIEW, [9-27](#)
- DROP_VERTICES_TEXT_IDX, [9-27](#)
- ESTIMATE_TRIANGLE_RENUM, [9-28](#)
- EXP_EDGE_TAB_STATS, [9-30](#)
- EXP_VERTEX_TAB_STATS, [9-31](#)
- FIND_CC_MAPPING_BASED, [9-32](#)
- FIND_CLUSTERS_CLEANUP, [9-33](#)
- FIND_CLUSTERS_PREP, [9-34](#)
- FIND_SP, [9-36](#)
- FIND_SP_CLEANUP, [9-37](#)
- FIND_SP_PREP, [9-38](#)
- GET_BUILD_ID, [9-39](#)
- GET_GEOMETRY_FROM_V_COL, [9-39](#)
- GET_GEOMETRY_FROM_V_T_COLS, [9-41](#)
- GET_LATLONG_FROM_V_COL, [9-42](#), [9-45](#)
- GET_LATLONG_FROM_V_T_COLS, [9-43](#)
- GET_LONG_LAT_GEOMETRY, [9-44](#)
- GET_LONGLAT_FROM_V_T_COLS, [9-46](#)
- GET_SCN, [9-47](#)
- GET_VERSION, [9-47](#)
- GET_WKTGEOMETRY_FROM_V_COL, [9-48](#)
- GET_WKTGEOMETRY_FROM_V_T_COLS, [9-49](#)
- GRANT_ACCESS, [9-50](#)
- IMP_EDGE_TAB_STATS, [9-51](#)
- IMP_VERTEX_TAB_STATS, [9-52](#)
- PR, [9-54](#)
- PR_CLEANUP, [9-56](#)
- PR_PREP, [9-57](#)
- PREPARE_TEXT_INDEX, [9-58](#)
- reference information, [9-1](#)

- OPG_APIS package (*continued*)
 - RENAME_PG, [9-58](#)
 - SPARSIFY_GRAPH, [9-59](#)
 - SPARSIFY_GRAPH_CLEANUP, [9-61](#)
 - SPARSIFY_GRAPH_PREP, [9-62](#)
- OPG_GRAPHOP package
 - POPULATE_SKELETON_TAB, [10-1](#)
 - reference information, [10-1](#)

P

page rank

- calculating, [9-54](#)
- cleanup, [9-56](#)
- preparing to find, [9-57](#)

PGQL (Property Graph Query Language), [6-1](#)

PGX (in-memory Graph server), [4-1](#)

PgXML for Graphs, [8-1](#)

- DeepWalk Algorithm, [8-1](#)
- Pg2vec Algorithm, [8-33](#)
- Supervised GraphWise Algorithm, [8-11](#)
- Unsupervised GraphWise Algorithm, [8-25](#)

POPULATE_SKELETON_TAB procedure, [10-1](#)

PR procedure, [9-54](#)

PR_CLEANUP procedure, [9-56](#)

PR_PREP procedure, [9-57](#)

PREPARE_TEXT_INDEX procedure, [9-58](#)

property graph

- cleanup after sparsifying, [9-61](#)
- clearing (removing data from), [9-10](#)
- cloning, [9-11](#)
- collaborative filtering, [9-4](#), [9-7](#), [9-9](#)
- creating, [9-18](#)
- dropping, [9-26](#)
- dropping view definition, [9-27](#)
- preparing to sparsify, [9-62](#)
- removing text index metadata, [9-11](#)
- renaming, [9-58](#)
- sparsifying, [9-59](#)

property graph access privileges

- granting, [9-50](#)

Property Graph Query Language (PGQL), [6-1](#)

property graph statistics table

- creating, [9-22](#)

property graph support

- getting build ID, [9-39](#)
- getting SCN, [9-47](#)
- getting version, [9-47](#)

R

RENAME_PG procedure, [9-58](#)

REST Endpoints for Graph Visualization Application, [7-9](#)

- DELETE: Cancel a Query Execution, [7-14](#)

 REST Endpoints for Graph Visualization Application (*continued*)

- GET: Check a Query Completion, [7-14](#)
- GET: Get User, [7-13](#)
- GET: List Graphs, [7-10](#)
- GET: Retrieve a Query Result, [7-15](#)
- GET: Run a PGQL Query, [7-11](#)
- GET: Run a PGQL Query Asynchronously, [7-13](#)
- POST: Login, [7-10](#)

 S

- shortest path
 - cleanup, [9-37](#)
 - finding, [9-36](#)
 - preparing to find, [9-38](#)
- skeleton table
 - populating, [10-1](#)
- snapshot table
 - creating, [9-19](#)
- SPARSIFY_GRAPH procedure, [9-59](#)
- SPARSIFY_GRAPH_CLEANUP procedure, [9-61](#)
- SPARSIFY_GRAPH_PREP procedure, [9-62](#)
- statistics for property graph
 - analyzing, [9-2](#)
- subgraph
 - creating, [9-23](#)

 text index

- on property graph edge table, [9-17](#)
 - on property graph edge table (dropping), [9-26](#)
 - on property graph vertex table, [9-24](#)
 - on property graph vertex table (dropping), [9-27](#)
 - preparing, [9-58](#)
- text index table
- creating, [9-21](#)
- triangles
- cleanup after counting, [9-13](#)
 - counting, [9-12](#)
 - counting and renumbering vertices, [9-16](#)
 - estimating the number, [9-28](#)
 - preparing to count, [9-14](#)

 V

- vertex cluster mappings
 - preparing, [9-33](#), [9-34](#)
- vertex table statistics
 - exporting, [9-31](#)
 - importing, [9-52](#)