Oracle Graph

Property Graph Developer's Guide

20c
F15915-03
July 2020

Oracle Graph Property Graph Developer’s Guide, 20c

F15915-03

Copyright © 2016, 2020, Oracle and/or its affiliates.

Primary Author: Chuck Murray

Contributors: Melliyal Annamalai, Jorge Barba, Bill Beauregard, Hector Briseno, Hassan Chafi, Eugene Chong, Souripriya Das, Ana Estrada, 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, Steve Serra, Korbinian Schmid, Jane Tao, Oskar van Rest, 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 and Java 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

3 Using the In-Memory Graph Server (PGX)

Index

List of Figures

List of Tables

Preface

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

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 and Graph Developer's Guide*
- *Oracle Spatial and Graph RDF Knowledge Graph Developer's Guide*
- *Oracle Spatial and Graph GeoRaster Developer's Guide*
- *Oracle Spatial and Graph Topology Data Model and Network Data Model Graph 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.
italic	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

This topic contains the following.

Changes in Oracle Database Release 20c for Property Graph Support

The following changes apply to property graph support in Release 20c of Oracle Spatial and Graph.

New Features

Significant new features for this release include:

- GraphViz (graph visualization tool): Lightweight, single-page web application to visualize graphs.
- In-memory graph representation optimization for reduced memory usage and faster performance.
- User-defined functions (UDFs), for creating custom graph algorithms and extending product graph algorithms with Java or JavaScript syntax.
- Support for Autonomous Database
- Graph server authentication and authorization based on Oracle Database users and roles
- New Synchronizer API to keep partitioned graphs up-to-date with changes in the Oracle Database
- PGQL features:
  - Added a CREATE PROPERTY GRAPH statement to PGQL on Oracle Database for creating property graphs from existing tables
  - Added support for `CHEAPEST` and `TOP k CHEAPEST` path queries to PGQL on PGX
  - Implemented the PGQL 1.3 language specification, which can be found at https://pgql-lang.org/
  - Added support for INSERT, UPDATE, and DELETE queries
  - Added support for MATCH inside FROM and optional ON clauses
  - Added support for case insensitivity
  - Added support for quoted identifiers
  - Added support for subqueries after GROUP BY

  - Added a PGQL plugin for SQLcl
- New built-in algorithms:
  - Compute High Degree Vertices
  - Create Distance Index
  - Limited Shortest Path (Hop Distance)
  - All Reachable Vertices/Edges
  - Louvain (community detection)

Enhancements

Significant enhancements for this release include:

- Improved software packaging: The property graph features of Oracle Spatial and Graph will be available for delivery through Oracle Software Delivery Cloud Portal ("eDelivery") instead of by being installed in `$ORACLE_HOME/md/`. (See Property Graph Prerequisites.)
  - Split into client and server package.
  - Graph software no longer bundled with Oracle Database but released separately.
- Improved backwards compatibility with previous Oracle Database releases (see "Database Compatibility and Restrictions" in Property Graph Prerequisites).
- Improved shell for interactive use of Java API:
  - Earlier shells are unified under `$PG_HOME/bin/opg`.
  - Shell is now based on Java JShell.
  - The Groovy shell is deprecated.
- Improved error handling for data ingestion APIs.
- Improved performance to create new graphs or graph snapshots from `ChangeSet` changes in PGX.

Desupported

Desupported for this release:

- 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.

Deprecated

Deprecated for this release:

- 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;
  ```

1

Property Graph Support Overview

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

- About the Property Graph Feature of Oracle Database
  The Property Graph feature delivers advanced graph query and analytics capabilities in Oracle Database.
- Property Graph Prerequisites
  The requirements for using the Property Graph feature of Oracle Database are the following.
- Property Graph Features
  Graphs manage networks of linked data as vertices, edges, and properties of the vertices and edges.
- 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.
- 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 Server and Client supports the family of Oracle Autonomous Database.
- Migrating Property Graph Applications from Before Release 20
  If you are migrating from a previous version of Oracle Spatial and Graph to Release 20, you may need to make some changes to existing property graph-related applications.
- Quick Start: Interactively Analyze Graph Data
  This tutorial shows how you can quickly get started using property graph data.

1.1 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.

1.2 Property Graph Prerequisites

The requirements for using the Property Graph feature of Oracle Database are the following.

- Effective with Release 20, Oracle Graph Server and Client must be installed, as described later in this topic.
- `max_string_size` must be `enabled`.
- `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.

Subtopics:

Graph Server Installation

Starting with Release 20.1, Graph Server and Client will be available as a separate downloadable package. The libraries will not be available in $ORACLE_HOME.

For installing the Graph server, the prerequisites are:

- Oracle Linux 6 or 7 x64 or a similar Linux distribution such as RedHat
- Oracle JDK 8

The base installation steps for the Graph Server and Client are:

1. Download Oracle Graph Server and Client from Oracle Edelivery.
2. As root, install the RPM file using the rpm command line utility:

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

   Where `<version>` reflects the version that you downloaded. (For example: `oracle-graph-20.1.0.0.0.x86_64.rpm`)

   The `.rpm` file is the graph server.
3. As root, 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.

4. As <*graphuser*>, configure the server by modifying the files under /etc/oracle/graph.

> **Note:**
>
> For demonstration purposes only, you can edit `/etc/oracle/graph/pgx-server.log` to change `enable_tls` to `false`. This will allow you to start the server as described in the next step. This is **not** recommended for production. In a secure configuration, `enable_tls` should be set to `true`.

5. Ensure that authentication is enabled for database users that will connect to the graph server, as explained in PGX User Authentication and Authorization.
6. As <*graphuser*>, start the PGX server by executing `/opt/oracle/graph/pgx/bin/start-server`.

The PGX server is now ready to accept requests. Log files can be found in `/var/log/oracle/graph`.

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

- Interactively analyze property graphs locally using the Java Shell tool (*JShell*).
- 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 GraphViz in Oracle WebLogic Server).
- Deploy the graph server as a web application with Apache Tomcat (see Deploying to Apache Tomcat).

Graph Server Upgrade

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. For example:

```
rpm -U oracle-graph-20.2.0.0.0.x86_64.rpm
```

Graph Server Uninstallation

To uninstall the graph server, make sure the graph server is shut down, then run the following command.

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

Graph Client Installation

For installing the Graph client, the prerequisites are:

- A Unix-based operation system (such as Linux) or macOS or Microsoft Windows
- Oracle JDK 11

The base installation steps for the Graph client are:

1. Download Oracle Graph Client 20.1 from Oracle Edelivery.

2. Unzip the file into a directory of your choice.
3. Connect to a PGX server using Jshell. For example:

```
cd
        <client-install-dir>./bin/opg-jshell --base_url https://
<host>:7007
```

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

- Install the client into Apache Zeppelin.

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".

1.3 Property Graph Features

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
- 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
- Property Graph Sizing Recommendations

1.3.1 Property Graph Sizing Recommendations

The following are recommendations for property graph installation.

Table 1-1 Property Graph Sizing Recommendations

Graph Size	Recommended Physical Memory to be Dedicated	Recommended Number of CPU Processors
10 to 100M edges	Up to 14 GB RAM	2 to 4 processors, and up to 16 processors for more compute-intensive workloads
100M to 1B edges	14 GB to 100 GB RAM	4 to 12 processors, and up to 16 to 32 processors for more compute-intensive workloads
Over 1B edges	Over 100 GB RAM	12 to 32 processors, or more for especially compute-intensive workloads

1.4 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.

1.5 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
- 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

The same graph shell executable can be used in both local or remote modes.

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 automatically connects to a PGX instance (either remote or embedded depending on the `--base_url` command-line option) and creates a PGX session. If `--base_url` is not specified, a new local PGX instance and session are created.

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 can be found in `/opt/oracle/graph/bin` after server installation or `<INSTALL_DIR>/bin` after client installation. The shell may be launched by entering the following in your terminal:

```
./bin/opg-jshell
```

This starts the shell in embedded (local) mode, which means the graph functions are running in the client JVM.

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

```
opg-jshell-rdbms>
```

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 `opg-jshell` command:

```
./bin/opg-jshell --help
```

Embedded (Local) versus Remote Mode

The graph shell can be started in embedded (local) mode or remote mode. By default the shell starts in embedded mode, which means a local PGX instance is created running in the same JVM in which the shell is running.

Embedded mode is only supported in the Graph Server installation. Graph Client installations only support remote mode.

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/opg-jshell --pgx_conf path/to/my/pgx.conf
```

The graph shell can also be started in remote mode. In that case the shell connects to a PGX instance that runs in a separate JVM (possibly on a different machine). To launch the shell in remote mode, specify the base URL of the server with the `--base_url` option. For example:

```
./bin/opg-jshell --base_url https://myserver.com:7007
```

Batch Execution of Scripts

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

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

1.6 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.

1.6.1 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.

1.6.2 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 Reading Data from Oracle Database into Memory.

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).

1.7 Using Oracle Graph with the Autonomous Database

Oracle Graph Server and Client supports the family of Oracle Autonomous Database.

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.

1.7.1 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 Server, as explained in Property Graph Prerequisites.
2. Establish a JDBC connection, as described in the Oracle Autonomous Warehouse documentation.
   Note that the Oracle Graph Server installation already contains all the necessary JDBC client libraries for connecting to Autonomous Databases. You do not have to install them yourself. You only have to download the wallet, unzip it to a secure location, and then reference it when establishing the connection.

   For example:

   ```
   opg-jshell-rdbms> var jdbcUrl =
   "jdbc:oracle:thin:@db201901151442_low?TNS_ADMIN=/etc/wallet"
   opg-jshell-rdbms> var user = "hr"
   opg-jshell-rdbms> var pass = "ChangeMe1234#_"
   opg-jshell-rdbms> var conn = DriverManager.getConnection(jdbcUrl,
   user, pass)
   conn ==> oracle.jdbc.driver.T4CConnection@57e6cb01
   ```

3. Use the connection in your graph application.

1.7.2 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 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 Reading Data from Oracle Database into Memory 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

Then, configure the server JVM to reference the unzipped Oracle Wallet location before starting up by setting the JAVA_OPTS environment. For example:

```
export JAVA_OPTS="-Doracle.net.tns_admin=/etc/wallet -
Doracle.jdbc.fanEnabled=false"
cd /opt/oracle/graph
./pgx/bin/start-server --secret-store /etc/keystore.p12
```

If you start the PGX server using `systemd`, edit the service file at `/etc/systemd/system/pgx.service` and specify the environment variable the under the [Service] directive:

```
Environment="JAVA_OPTS=-Doracle.net.tns_admin=/etc/wallet"
```

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

After the file is edited, reload the changes using:

```
systemctl daemon-reload
```

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 starts the PGX server as follows:

  ```
  export JAVA_OPTS="-Doracle.net.tns_admin=/etc/wallet -
  Doracle.jdbc.fanEnabled=false"
  ```

```
cd /opt/oracle/graph
./pgx/bin/start-server --secret-store /etc/keystore.p12
```

The `/etc/wallet/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:@sombrero_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 Reading Data from Oracle Database into Memory. 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: https://weblogic.cafe/posts/atp-datasource/
  – 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 the reference them by sending a graph configuration file with the following connection properties:

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

1.8 Migrating Property Graph Applications from Before Release 20

If you are migrating from a previous version of Oracle Spatial and Graph to Release 20, you may need to make some changes to existing property graph-related applications.

Security-Related Changes

Oracle Graph Release 20.1 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 secrets to be stored in Java Keystore files**
  Previously, it was possible to write secrets in plain text into graph configuration files. Release 20.1 and later no longer allow that, and now require secrets to be stored in Java Keystore files instead, which can be referenced directly from graph configuration files. (See Reading Data from Oracle Database into Memory 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 whitelist**
  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`. Starting with Oracle Graph 20.1, if you set `allow_local_filesystem` to `true`, you must also specify a list of directories that are "whitelisted" 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. Starting in Oracle Graph 20.1, 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 encourage 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 20.1 works with older database versions. (See the "Database Compatibility and Restrictions" subtopic in Property Graph Prerequisites 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 Oracle Spatial and Graph

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

The packaging of the "graph" support in Oracle Spatial and Graph has been changed so that it is now installed separately from Oracle Database. (The "spatial" support in Oracle Spatial and Graph is **not** installed separately from Oracle Database.) After you have completed the graph support installation, completed preceding migration steps (if needed), and confirmed that everything is working well, it is recommended that you

remove the binaries of ***older*** Oracle Spatial and Graph installations from your Oracle Database installation by performing the following uninstall steps:

1. Make sure Spatial and Graph 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`

1.9 Quick Start: Interactively Analyze Graph Data

This tutorial shows how you can quickly get started using property graph data.

- Convert existing relational data into a graph.
- Query that data using PGQL.
- Run graph algorithms on that data and display results.

You can do all of this locally in a terminal without setting up any servers. All you need are:

- An installation of Oracle Graph Server (see Property Graph Prerequisites)
- Java 11 for the interactive analysis in this example. For Java downloads, see https://www.oracle.com/technetwork/java/javase/overview/index.html.
- Connection details to an Oracle Database 20c. (Other versions of Oracle Database will also work, but be aware of Property Graph Prerequisites.)
- Basic knowledge about how to run commands on Oracle Database (for example, using SQL*Plus or SQL Developer).

Major tasks for this tutorial:

- Set up the example data
- Start the shell
- Open a JDBC database connection
- Create a PGQL connection
- Write and execute the graph creation statement
- Run a few PGQL queries
- Load the graph into memory
- Execute algorithms and query the algorithm results
- Work with a remote graph server

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 server is installed, start the shell by as follows:

```
cd /opt/oracle/graph
./bin/opg-jshell
```

This starts the shell in embedded (local) mode, which means the graph functions are running in the client JVM.

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
```

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:

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

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
```

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:

```
opg-jshell-rdbms> conn.setAutoCommit(false)
opg-jshell-rdbms> 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:

```
opg-jshell-rdbms>
pgql.prepareStatement(Files.readString(Paths.get("<path>/
create.pgql"))).execute()
$16 ==> 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
opg-jshell-rdbms> 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
opg-jshell-rdbms> query.accept("select count(v) from hr match (v)")
+----------+
| count(v) |
+----------+
| 215      |
```

```
+----------+

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

// find the highest earning managers
opg-jshell> 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
opg-jshell> 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 |
+---------------+
```

Load the graph into memory

Next, load the graph into PGX. This will enable you to run a variety of different built-in algorithms on the graph and will also drastically improve query performance for larger graphs. 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.

> **Note:**
>
> Always use low-privilege read-only database user accounts for PGX, as explained in Security Best Practices with Graph Data.

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()
 .setJdbcUrl(jdbcUrl)
 .setUsername(user)
 .setPassword(pass)
 .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:

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

The session object is created for you automatically and points to a local PGX instance running in your shell.

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`:

```
opg-jshell-rdbms> 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:

```
opg-jshell> 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 |
+-----------------------------------------------------+
```

Work with a remote graph server

To launch the shell in remote mode, specify the URL of the server with the --base_url option. For example:

```
./bin/opg-jshell --base_url https://myserver.com:7007
```

If you want to share the graph with other client sessions or access the graph with the visualization tool, you cannot use the server in embedded mode. You must instead start the graph server (PGX) standalone by executing the following command:

```
$ /opt/oracle/graph/pgx/bin/start-server
```

and then connecting to the graph server with a shell in remote mode.

After you load the graph into the server, you must use the `publish()` API to make the graph visible to other sessions. For example:

```
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. (This application is described in Graph Visualization Application (GraphViz).)

2

Using Property Graphs in an Oracle Database Environment

This chapter provides conceptual and usage information about creating, storing, and working with property graph data in an Oracle Database environment.

- About Property Graphs
  Property graphs give you a different way of looking at your data.
- About Property Graph Data Formats
  Several graph formats are supported for property graph data.
- Property Graph Schema Objects for Oracle Database
  The property graph PL/SQL and Java APIs use special Oracle Database schema objects.
- 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).
- Using the Groovy-Based Shell with Property Graph Data
  The Oracle Graph property graph support includes a built-in Groovy-based shell (based on the original Gremlin Groovy shell script). With this command-line shell interface, you can explore the Java APIs.
- Using the In-Memory Analyst Zeppelin Interpreter with Oracle Database
  The in-memory analyst provides an interpreter implementation for Apache Zeppelin. This tutorial topic explains how to install the in-memory analyst interpreter into your local Zeppelin installation and to perform some simple operations.
- 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.

2.1 About 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?
- What Is Oracle Database Support for Property Graphs?

2.1.1 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

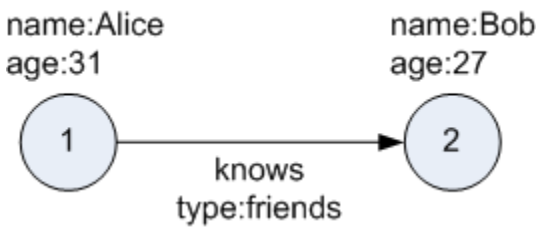
Each edge has a unique identifier and can have:

- 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 2-1 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

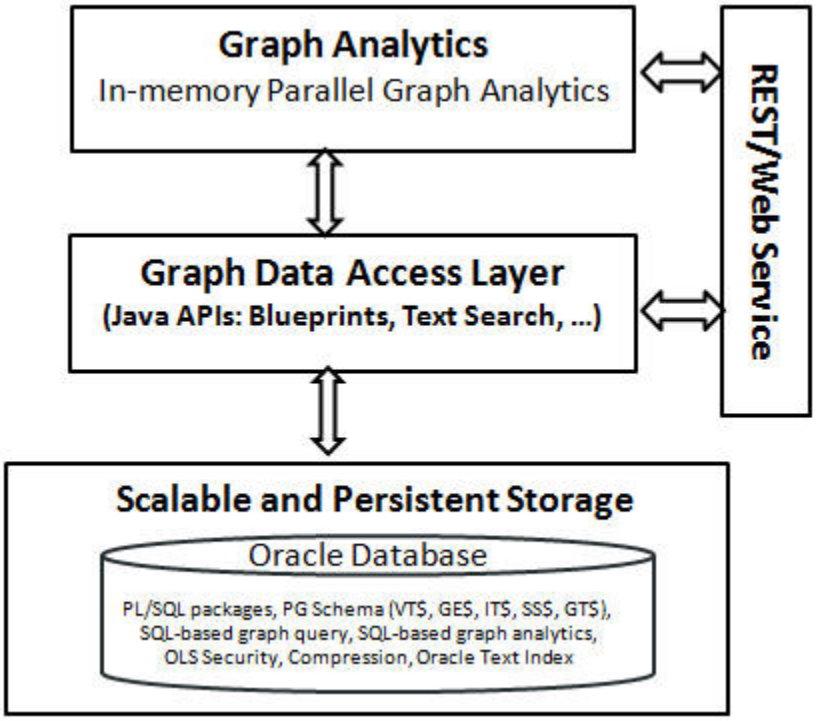
- Specifying Labels for Vertices

2.1.2 What Is Oracle Database Support for Property Graphs?

Property graphs are supported in Oracle Database, in addition to being supported for Big Data in Hadoop. This support consists of a set of PL/SQL packages, a data access layer, and an analytics layer.

The following figure provides an overview of the Oracle property graph architecture.

Figure 2-2 Oracle Property Graph Architecture



- In-Memory Graph Server (PGX)
- Data Access Layer
- Options for Property Graph Architecture

2.1.2.1 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), Dijsktra's shortest path, Hop Distance (Breadth-first search)
- Community Evaluation - Coefficient (Triangle Counting), Conductance, Modularity, Adamic-Adar counter.

2.1.2.2 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).

2.1.2.3 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)

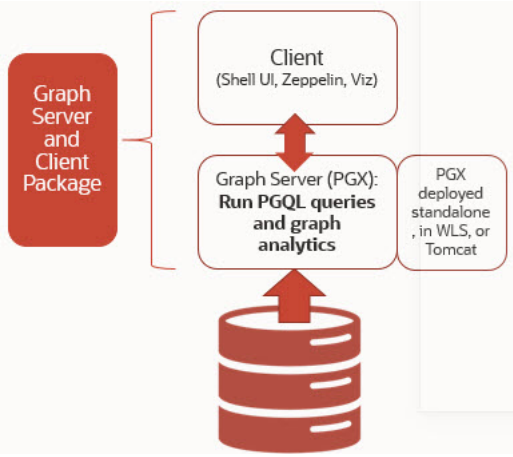
Both options let you use the Property Graph Query Language (PGQL).

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 2-3 Three-Tier Property Graph Architecture



Load the Graph into Oracle Database (2-Tier)

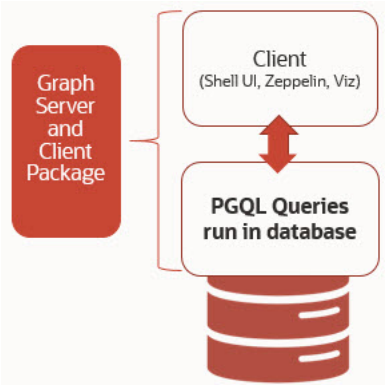
If you do not need to load the graph into the in-memory graph server, you can use another approach: create a property graph from data in relational tables, and store it in

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 2-4 Two-Tier Property Graph Architecture



2.2 About Property Graph Data Formats

Several graph formats are supported for property graph data.

- GraphML Data Format
- GraphSON Data Format
- GML Data Format
- Oracle Flat File Format

2.2.1 GraphML Data Format

The GraphML file format uses XML to describe graphs.

- The first example in this topic shows a GraphML description of the property graph shown in What Are Property Graphs?.
- The second example in this topic shows the GraphML description of the same graph for Tinkerpop 3. Notice the addition of vertex and edge labels referred as `labelV` and `labelE`, respectively.

Example 2-1 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>
```

Example 2-2 Tinkerpop 3 GraphML Description of a Simple Property Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
   <key id="labelV" for="node" attr.name="labelV" attr.type="string" />
   <key id="name" for="node" attr.name="name" attr.type="string" />
   <key id="age" for="node" attr.name="age" attr.type="int" />
   <key id="labelE" for="edge" attr.name="labelE" attr.type="string" />
   <key id="type" for="edge" attr.name="type" attr.type="string" />
   <graph id="PG" edgedefault="directed">
      <node id="1">
         <data key="labelV">person</data>
         <data key="name">Alice</data>
         <data key="age">31</data>
      </node>
      <node id="2">
         <data key="labelV">person</data>
         <data key="name">Bob</data>
         <data key="age">27</data>
      </node>
      <edge id="3" source="1" target="2">
         <data key="labelE">knows</data>
         <data key="type">friends</data>
      </edge>
   </graph>
</graphml>
```

Methods are provided to import and export graphs from and into GraphML format.

The following fragments of code show how to import and export GraphML data in Tinkerpop 2 and Tinkerpop 3 versions.

```
// Get graph instance
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(oracle, graphName);

// Import graph in GraphML format
String fileName = "./mygraph.graphml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.importGraphML(opg,fileName,ps);

// Export graph into GraphML format
String fileName = "./mygraph.graphml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.exportGraphML(opg,fileName,ps);

// Import graph into Tinkerpop 3 GraphML format
String fileName = "./mygraphT3.graphml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.importGraphMLTinkerpop3(opg,fileName,ps);
```

```
// Export graph into Tinkerpop 3 GraphML format
String fileName = "./mygraphT3.graphml";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.exportGraphMLTinkerpop3(opg,fileName,ps);
```

Related Topics

- GraphML File Format

2.2.2 GraphSON Data Format

The GraphSON file format is based on JavaScript Object Notation (JSON) for describing graphs.

- The first example in this topic shows a GraphSON description of the property graph shown in What Are Property Graphs?.
- The second example in this topic shows the GraphSON description of the same graph for Tinkerpop 3.

Example 2-3 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"
            }
        ]
    }
}
```

Example 2-4 GraphSON 3.0 Description of a Simple Property Graph

```
{"id":{"@type":"g:Int64","@value":1},"label":"person","outE":{"knows":[{"id":
{"@type":"g:Int64","@value":3},"inV":{"@type":"g:Int64","@value":2},"properties":
{"type":"friends"}}]},"properties":{"name":[{"id":
{"@type":"g:Int64","@value":66724076},"value":"Alice"}],"age":[{"id":
{"@type":"g:Int64","@value":96543},"value":{"@type":"g:Int32","@value":31}}]}}
{"id":{"@type":"g:Int64","@value":2},"label":"person","inE":
```

```
{"knows":[{"id":{"@type":"g:Int64","@value":3},"outV":
{"@type":"g:Int64","@value":1},"properties":{"type":"friends"}}]},"properties":
{"name":[{"id":{"@type":"g:Int64","@value":3440674},"value":"Bob"}],"age":[{"id":
{"@type":"g:Int64","@value":96540},"value":{"@type":"g:Int32","@value":27}}]}}
```

Methods are provided to import and export graphs from and into GraphSON format.

The following fragments of code show how to import and export GraphSON data in Tinkerpop 2 and Tinkerpop 3 versions. Note that the Tinkerpop 3 version has a “Tinkerpop3” suffix. This is to maintain backward compatibility.

```
// Get graph instance
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args, szGraphName);

// Import graph in GraphSON format
String fileName = "./mygraph.graphson";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.importGraphSON(opg,fileName,ps);

// Export graph into GraphSON format
String fileName = "./mygraph.graphson";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.exportGraphSON(opg,fileName,ps);

// Import graph into Tinkerpop 3 GraphSON format
String fileName = "./mygraphT3.graphson";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.importGraphSONTinkerpop3(opg,fileName,ps);

// Export graph into Tinkerpop 3 GraphSON format
String fileName = "./mygraphT3.graphson";
PrintStream ps = new PrintStream("./output");
OraclePropertyGraphUtils.exportGraphSONTinkerpop3(opg,fileName,ps);
```

Related Topics

- GraphSON Reader and Writer Library

2.2.3 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 2-5 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

2.2.4 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 2-6 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 tto 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,appe
nd);
```

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.

2.3 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).

> **Note:**
>
> An Oracle Partitioning license is required if you use the property graph schema. For performance and scalability, both VT$ and GE$ tables are hash partitioned based on IDs, and the number of partitions is customizable. The number of partitions should be a value that is power of 2 (2, 4, 8, 16, and so on). The partitions are named sequentially starting from "p1", so for a property graph created with 8 partitions, the set of partitions will be "p1", "p2", ..., "p8".

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

2.3.1 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, VTS, 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 myGraph$VT$
 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
 VTS                                  DATE
 VTE                                  DATE
 FE                                   NVARCHAR2(4000)


SQL> describe myGraph$GE$
 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
 VTS                                  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 VTS, 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 (see Representing Spatial Data in a Property Graph)

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 **GE$ 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 column reserved for future extensions.
- VTE, a timestamp with time zone column reserved for future extensionss.
- 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.

2.3.2 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)`

2.3.3 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.

2.4 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.

2.4.1 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 SESSION
CREATE TABLE
CREATE TYPE
```

2.5 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

2.5.1 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

2.5.1.1 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 classes into your Java program:

```
import oracle.pg.common.*;
import oracle.pg.text.*;
import oracle.pg.rdbms.*;
import oracle.pgx.config.*;
import oracle.pgx.common.types.*;
```

2.5.1.2 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.*;
```

2.5.2 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

2.5.2.1 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 chunksPiped 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. IIf 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 opgdl =
OraclePropertyGraphDataLoader.getInstance();

    opgdl.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 */);
```

2.5.2.2 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
                                  named pipe based splitting;
otherwise, regular file
                                  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 */);
```

2.5.2.3 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
                                                  (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 */,
                             48 /* 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 */,
                                szSQLLoaderPath /* SQL*Loader path: the
path to
                                                   bin/sqlldr*/,
                                true /* rebuild index */,
                                "pddl=t,pdml=t" /* options */);
```

2.5.3 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 partitionID
*/);
  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();
}
```

2.5.4 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++;
}
```

2.5.5 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);
```

2.5.6 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'sCollaborators
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 getAttributeKeyName(OracleEdgeBase e)
{
return "smithFeud";
}

/**
 * Define the property's value based on the in/out vertex role
 */
@Override
public Object getAttributeKeyValue(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 " ";
  }
}
```

2.5.7 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()));
}
```

2.5.8 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"; 0
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*/);
```

2.5.9 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();
  ...
```

2.5.10 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);
```

2.5.11 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");
```

2.5.12 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.

2.5.13 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<PgxConfig.Field, Object> confPgx = new HashMap<PgxConfig.Field,
Object>();
confPgx.put(PgxConfig.Field.ENABLE_GM_COMPILER, false);
confPgx.put(PgxConfig.Field.NUM_WORKERS_IO, dop);   //
confPgx.put(PgxConfig.Field.NUM_WORKERS_ANALYSIS, dop); // <= # of
physical cores
confPgx.put(PgxConfig.Field.NUM_WORKERS_FAST_TRACK_ANALYSIS, 2);
confPgx.put(PgxConfig.Field.SESSION_TASK_TIMEOUT_SECS, 0);  // no
timeout set
confPgx.put(PgxConfig.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);
```

2.5.14 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?

2.5.15 Building an In-Memory Graph

In addition to Reading Data from Oracle Database into Memory, 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(1l).setProperty("double-prop", 0.5)
v2 = builder.addVertex(2l).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")
```

2.5.16 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');
```

2.5.17 Executing PGQL Queries

You can execute PGQL queries directly against Oracle Database with the `PgqlStatement` and `PgqlPreparedStatement` interfaces. See Executing PGQL Queries Directly Against Oracle Database for details.

2.6 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

2.6.1 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

2.6.1.1 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);
```

2.6.2 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
```

2.6.3 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.

2.6.4 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

2.6.4.1 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);
```

2.6.5 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

2.6.5.1 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(100l);
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);
```

2.6.6 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.

2.6.7 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

2.6.7.1 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);
```

2.7 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.

2.7.1 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 userTS IDENTIFIED BY userTSpass;
GRANT connect, resource, create table, create view, create any
index TO userTS;
GRANT unlimited TABLESPACE to userTS;

CREATE USER userTS2 IDENTIFIED BY userTS2pass;
GRANT connect, resource, create table, create view, create any
index TO userTS2;
GRANT unlimited TABLESPACE to userTS2;

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",
"userTpassS");
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
```

2.8 Using the Groovy-Based Shell with Property Graph Data

The Oracle Graph property graph support includes a built-in Groovy-based shell (based on the original Gremlin Groovy shell script). With this command-line shell interface, you can explore the Java APIs.

To use this shell, you must first separately download and install Apache Groovy.

To start the shell, go to the `<graph client home>/bin/` directory. Included is the script `opg-groovy`.

The following example connects to an Oracle database, gets an instance of `OraclePropertyGraph` with graph name `myGraph`, loads some example graph data, and gets the list of vertices and edges.

```
$ sh ./opg-groovy

opg-rdbms> cfg =
cfg = GraphConfigBuilder.forPropertyGraphRdbms() \
.setJdbcUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl")\
.setUsername("scott").setPassword("<password>") \
.setName("connections") .setMaxNumConnections(2)\
.setLoadEdgeLabel(false) \
.addEdgeProperty("weight", PropertyType.DOUBLE, "1000000") \
.build();

opg-rdbms> opg = OraclePropertyGraph.getInstance(cfg);
==>oraclepropertygraph with name myGraph

opg-rdbms> opgdl = OraclePropertyGraphDataLoader.getInstance();
==>oracle.pg.nosql.OraclePropertyGraphDataLoader@576f1cad

opg-rdbms> opgdl.loadData(opg, new FileInputStream("../../data/
connections.opv"), new FileInputStream("../../data/connections.ope"), 4/*dop*/,
1000/*iBatchSize*/, true /*rebuildIndex*/, null /*szOptions*/); ==>null

opg-rdbms> opg.getVertices();
==>Vertex ID 5 {country:str:Italy, name:str:Pope Francis, occupation:str:pope,
religion:str:Catholicism, role:str:Catholic religion authority}
[... other output lines omitted for brevity ...]

opg-rdbms>  opg.getEdges();
==>Edge ID 1139 from Vertex ID 64 {country:str:United States, name:str:Jeff
Bezos, occupation:str:business man} =[leads]=> Vertex ID 37 {country:str:United
States, name:str:Amazon, type:str:online retailing} edgeKV[{weight:flo:1.0}]
[... other output lines omitted for brevity ...]
```

The following example customizes several configuration parameters for in-memory analytics. It connects to an Oracle database, gets an instance of `OraclePropertyGraph` with graph name `myGraph`, loads some example graph data, gets the list of vertices and edges, gets an in-memory analyst, and executes one of the built-in analytics, triangle counting.

```
$ sh ./opg-groovy
opg-rdbms>
opg-rdbms> dop=2;   // degree of parallelism
==>2
```

```
opg-rdbms> confPgx = new HashMap<PgxConfig.Field, Object>();
opg-rdbms> confPgx.put(PgxConfig.Field.ENABLE_GM_COMPILER, false);
==>null
opg-rdbms> confPgx.put(PgxConfig.Field.NUM_WORKERS_IO, dop);
==>null
opg-rdbms> confPgx.put(PgxConfig.Field.NUM_WORKERS_ANALYSIS, 3);
==>null
opg-rdbms> confPgx.put(PgxConfig.Field.NUM_WORKERS_FAST_TRACK_ANALYSIS, 2);
==>null
opg-rdbms> confPgx.put(PgxConfig.Field.SESSION_TASK_TIMEOUT_SECS, 0);
==>null
opg-rdbms> confPgx.put(PgxConfig.Field.SESSION_IDLE_TIMEOUT_SECS, 0);
==>null
opg-rdbms> instance = Pgx.getInstance()
==>null
opg-rdbms> instance.startEngine(confPgx)
==>null

opg-rdbms>
cfg = GraphConfigBuilder.forPropertyGraphRdbms() \
.setJdbcUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl")\
.setUsername("scott").setPassword("<password>") \
.setName("connections") .setMaxNumConnections(2)\
.setLoadEdgeLabel(false) \
.addEdgeProperty("weight", PropertyType.DOUBLE, "1000000") \
.build();
opg-rdbms> opg = OraclePropertyGraph.getInstance(cfg);
==>oraclepropertygraph with name myGraph

opg-rdbms> opgdl = OraclePropertyGraphDataLoader.getInstance();
==>oracle.pg.hbase.OraclePropertyGraphDataLoader@3451289b

opg-rdbms> opgdl.loadData(opg, "../../data/connections.opv", "../../data/
connections.ope", 4/*dop*/, 1000/*iBatchSize*/, true /*rebuildIndex*/, null /
*szOptions*/);
==>null

opg-rdbms> opg.getVertices();
==>Vertex ID 78 {country:str:United States, name:str:Hosain Rahman,
occupation:str:CEO of Jawbone}
...

opg-rdbms> opg.getEdges();
==>Edge ID 1139 from Vertex ID 64 {country:str:United States, name:str:Jeff
Bezos, occupation:str:business man} =[leads]=> Vertex ID 37 {country:str:United
States, name:str:Amazon, type:str:online retailing} edgeKV[{weight:flo:1.0}]
[... other output lines omitted for brevity ...]

opg-rdbms> session = Pgx.createSession("session-id-1");
opg-rdbms> g = session.readGraphWithProperties(cfg);
opg-rdbms> analyst = session.createAnalyst();

opg-rdbms>  triangles = analyst.countTriangles(false).get();
==>22
```

For detailed information about the Java APIs, see the Javadoc reference information.

2.9 Using the In-Memory Analyst Zeppelin Interpreter with Oracle Database

The in-memory analyst provides an interpreter implementation for Apache Zeppelin. This tutorial topic explains how to install the in-memory analyst interpreter into your local Zeppelin installation and to perform some simple operations.

Installing the Interpreter

The following steps were tested with Zeppelin version 0.8.2, and might have to be modified with newer versions.

1. If you have not already done so, download and install Apache Zeppelin.
2. If you have not already done so, download and install Apache Groovy 2.4.x.
3. Follow the official interpreter installation steps.
   a. Copy libraries:
      - Copy the libraries from the Oracle Graph Client for Apache Zeppelin package into `$ZEPPELIN_HOME/interpreter/pgx`.
      - Copy the libraries inside `$GROOVY_HOME/lib` into `$ZEPPELIN_HOME/interpreter/pgx`.
   b. Use the default `zeppelin-site` configuration from the template:
      `cp $ZEPPELIN_HOME/conf/zeppelin-site.xml.template $ZEPPELIN_HOME/conf/zeppelin-site.xml`
   c. Edit `$ZEPPELIN_HOME/conf/zeppelin-site.xml` and add the in-memory analyst Zeppelin interpreter class name `oracle.pgx.zeppelin.PgxInterpreter` to the `zeppelin.interpreters` property field.
   d. Restart Zeppelin.
   e. In the Zeppelin interpreter page, click the **+Create** button to add a new interpreter of interpreter group `pgx`.

      If you do not see a `pgx` interpreter group, try to remove `$ZEPPELIN_HOME/conf/interpreter.json` and restart Zeppelin.

Using the Interpreter

If you named the in-memory analyst interpreter `pgx`, you can send paragraphs to the in-memory analyst interpreter by starting the paragraphs with the `%pgx` directive, just as with any other interpreter.

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

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

- `PgqlResultSet` - the object returned by the `queryPgql("...")` 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.

2.10 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}]
```

2.11 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

2.11.1 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.

2.11.2 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 2-1 Edge File Record Format

Field Number	Name	Description
1	*edge_ID*	An integer that uniquely identifies the edge
2	*source_vertex_ID*	The *vertex_ID* of the outgoing tail of the edge.
3	*destination_vertex_ID*	The *vertex_ID* of the incoming head of the edge.
4	*edge_label*	The encoded label of the edge, which describes the relationship between the two vertices
5	*key_name*	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	*value_type*	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	*value*	The encoded, nonnull value of *key_name* when it is neither numeric nor timestamp (date)
8	*value*	The encoded, nonnull value of *key_name* when it is numeric

Table 2-1 (Cont.) Edge File Record Format

Field Number	Name	Description
9	*value*	The encoded, nonnull value of *key_name* 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 2-7 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,
```

2.11.3 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 2-2 Vertex File Record Format

Field Number	Name	Description
1	*vertex_ID*	An integer that uniquely identifies the vertex
2	*key_name*	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	*value_type*	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	*value*	The encoded, nonnull value of *key_name* when it is neither numeric nor date
5	*value*	The encoded, nonnull value of *key_name* when it is numeric

Table 2-2 (Cont.) Vertex File Record Format

Field Number	Name	Description
6	*value*	The encoded, nonnull value of *key_name* 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 `2015-03-26T00:00:00.000-05:00`: `SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSXXX"); encode(sdf.format((java.util.Date) value));`
7	*vertex_label*	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 2-8 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%20OSmith,,,person
1,birth%20year,2,,1961,,person
```

2.11.4 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 2-3 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

2.11.5 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,,,,
```

2.11.6 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 1000l will be applied to
vertex ID, use ctams to map RDBMS columns to attributes, set DOP to 8
OraclePropertyGraphUtils.convertRDBMSTable2OPV(conn, "EmployeeTab",
"empID", 1000l, ctams, 8, opvOS, (DataConverterListener) null);
```

> **Note:**
>
> The lowercase letter "l" as the last character in the offset value `1000l` 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 1000l 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 filein `.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.convertRDBMSTable2OP`**`E`** 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
10000l will be applied to edge ID, the source and destination vertices
of the edge are defined by columns “source" and “destination", offset
1000l 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", 10000l, “source", “destination", 1000l, true,
“relationType", ctams, 8, opeOS, (DataConverterListener) null);
```

> **Note:**
>
> The lowercase letter “l" as the last character in the offset value `10000l` 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 10000l is applied. Similarly, an offset 1000l is applied to the source and destination vertex IDs.

2.11.7 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,
```

3
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.

- PGX User Authentication and Authorization
  The Oracle Graph server (PGX) uses an Oracle database as identity manager by default.
- Reading Data from Oracle Database into Memory
  When data is in PGX (that is, when data has been read into memory), you can run any of the built-in algorithms against your data, even compile and execute your own custom algorithms and use PGQL to query results.
- 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.
- Configuring the In-Memory Analyst
  You can configure the in-memory analyst engine and its run-time behavior by assigning a single JSON file to the in-memory analyst at startup.
- 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 analyst 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 Analyst Server
  A preconfigured version of Apache Tomcat is bundled, which allows you to start the in-memory analyst server by running a script.

- 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.
- Connecting to the In-Memory Analyst Server
  After the property graph in-memory analyst is installed in a computer running Oracle Database -- or on a client system without Oracle Database server software as a web application on Apache Tomcat or Oracle WebLogic Server -- you can connect to the in-memory analyst server.
- Managing Property Graph Snapshots
  You can manage property graph snapshots.
- 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.

3.1 PGX User Authentication and Authorization

The Oracle Graph server (PGX) uses an Oracle database as identity manager by default.

This means that you log into the graph server using existing Oracle Database credentials (user name 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.

Basic Steps for Using an Oracle Database for Authentication

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 SYSDBA access (or ADMIN access for Autonomous Database) 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 SYSDBA (or ADMIN on Autonomous Database), create the following roles:

   ```
   CREATE ROLE graph_developer
   CREATE ROLE graph_administrator
   ```

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

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

   where <graphuser> is a user in the database.

7. 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.

- Prepare the Graph Server for Database Authentication
  Locate the `pgx.conf` file of your installation.
- Generate and Use a Token
  Generate and use a token for making authenticated remote requests to the graph server.
- Read Data from the Database
  If you have a valid authentication token, you can now read data from the database into the graph server without specifying any connection information in the graph configuration for as long as the token is valid.
- Token Expiration
  By default, tokens are valid for 4 hours. If the token expires, requests to the graph server will be rejected.
- Advanced Access Configuration
  You can customize the following fields in `pgx.conf` realm options to customize login behavior.
- Examples of Custom Authorization Rules
  You can define custom authorization rules for developers.
- 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.

3.1.1 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 pgx-webapp-20.3.0.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, this can be done using:

```
systemctl start pgx
```

3.1.2 Generate and Use a Token

Generate and use a token for making authenticated remote requests to the graph server.

If the graph server listens on `https://localhost:7007`, you can run the following command to sign in to the graph server as SCOTT:

```
curl -X POST -H 'Content-Type: application/json' -d '{"username":
"scott", "password": "<password>"}' https://localhost:7007/auth/token
```

The preceding example uses cURL to make an HTTP request to the server, you can use any HTTP client of your choice.

If the user exists and has one of the graph roles assigned, the server will reply with something like the following:

```
{
  "token_type": "Bearer",
  "expires_in": 3600,
  "access_token":
"eyJraWQiOiJEYXRhYmFzZVJlYWxtIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJwZ0FkbWluIiwicm9sZ
XMiOlsicmVzb3VyY2UiLCJjb25uZWN0IiwiZ3JhcGhfYWRtaW5pc3RyYXRvciJdLCJpc3MiOiJvcmFjbG
```

```
UucGcuaWRlbnRpdHkucmVzdC5BdXRoZW50aWNhdGlvblNlcnZpY2UiLCJleHAiOjE1OTAxMDU1MDV9.D1
4yGwvzW7zlyjxdiagknjB_wU3VSXnWKHFSYcLDkF2JclyMNE0MmtgJQ958BNFpvB-
ha0ODxn_H1mnIlk3Cq7aoLiXN9V2WoxpYPQSdTu1pU2cKo-NfKOJF_MaqnS-
USw0XozovqtrEsnaWid8uF8vAS0WHt0Wm8nTtijoe99K__tvDgpYQH3cqicERPlBMRov9oOg-
Rfuyg1o6CoGdgrNEMYG44RRJBXOBFCD15yJ2aUfMHU5fukAbh6aWmrkKbwwueUmgTdjhWlxooEwyF-
C_LjksVPba2M5zRX-WOC6Zp8Lqxr6uqhxR1W4XpuQLLaD2Vw4OwpP7M5AldsS2MQ"
}
```

You can now use the token to make authenticated remote requests to the graph server.

If you use the JShell client, you will be prompted by the shell to provide the token when you start the shell in remote mode. For example:

```
./bin/opg-jshell --base_url https://localhost:7007

Enter the authentication token: <token>
```

If you are using a Java client program, you can connect using the following:

```
import oracle.pgx.api.*

...

ServerInstance instance = Pgx.getInstance("https://localhost:7007",
"<token>");
PgxSession session = instance.createSession("my-session");

...
```

3.1.3 Read Data from the Database

If you have a valid authentication token, you can now read data from the database into the graph server without specifying any connection information in the graph configuration for as long as the token is valid.

Your database user must exist and have read access on the graph data in the database.

For example, the following graph configuration will read a property graph named `hr` into memory, authenticating as `scott/<password>` with the database:

```
GraphConfig config = GraphConfigBuilder.forPropertyGraphRdbms()
    .setName("hr")
    .addVertexProperty("FIRST_NAME", PropertyType.STRING)
    .addVertexProperty("LAST_NAME", PropertyType.STRING)
    .addVertexProperty("EMAIL", PropertyType.STRING)
    .addVertexProperty("CITY", PropertyType.STRING)
    .setPartitionWhileLoading(PartitionWhileLoading.BY_LABEL)
    .setLoadVertexLabels(true)
    .setLoadEdgeLabel(true)
    .build();
PgxGraph hr = session.readGraphWithProperties(config);
```

The following example is a graph configuration in JSON format that reads from relational tables into the graph server, without any connection information being provided in the configuration file itself:

```
{
    "name":"hr",
    "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":"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"
        }
    ]
}
```

For more information about how to read data from the database into the graph server, see Reading Data from Oracle Database into Memory.

3.1.4 Token Expiration

By default, tokens are valid for 4 hours. If the token expires, requests to the graph server will be rejected.

If that happens, you can generate a new token by logging in again and asking the server for a handle to your previous session by using the `ServerInstance#getSession("<session-id>")` API. For example:

```
opg> var sessionId = session.getId() // remember session ID in variable

opg> var graph = session.readGraphWithProperties(config) // fails
because token expired

// obtain new token (see above for example)
opg> var newToken = ...

// get reference to previous session back
opg> session = Pgx.getInstance(instance.getBaseUrl(),
newToken).getSession(sessionId)

opg> var graph = session.readGraphWithProperties(config) // works now
```

3.1.5 Advanced Access Configuration

You can customize the following fields in `pgx.conf` realm options to customize login behavior.

Table 3-1 Advanced Access Configuration Options

Field Name	Explanation	Default
`token_expiration_seconds`	After how many seconds the generated bearer token will expire.	14400 (4 hours)
`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

Table 3-1 (Cont.) Advanced Access Configuration Options

Field Name	Explanation	Default
`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

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

- Customizing Roles and Permissions
  By default, the graph server maps the following roles to the following permissions in `pgx.conf`.
- 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.

3.1.5.1 Customizing Roles and Permissions

By default, the graph server maps the following roles to the following permissions in `pgx.conf`.

```
"authorization": [{
  "pgx_role": "GRAPH_ADMINISTRATOR",
  "pgx_permissions": [{
    "grant": "PGX_SESSION_CREATE"
  }, {
    "grant": "PGX_SERVER_GET_INFO"
  }, {
    "grant": "PGX_SERVER_MANAGE"
  }]
}, {
  "pgx_role": "GRAPH_DEVELOPER",
  "pgx_permissions": [{
    "grant": "PGX_SESSION_CREATE"
  }, {
    "grant": "PGX_SESSION_NEW_GRAPH"
```

```
  }, {
    "grant": "PGX_SESSION_GET_PUBLISHED_GRAPH"
  }]
}]
```

You can fully customize this mapping by adding and removing roles and specifying permissions to which a role maps. You can also authorize individual users instead of roles. This topic includes examples of how to customize the permission mapping.

To change the authorization mappings, you can:

- Modify the `pgx.conf` file and then restart the server, or
- Do it at run time by using the `ServerInstance#updatePgxConfig()` API. You need the PGX_SERVER_MANAGE permission to do this. Note that using this API will not persist those changes.

3.1.5.2 Adding and Removing Roles

You can add new role permission mappings or remove existing mappings by modifying the authorization list.

For example:

```
"authorization": [{
  "pgx_role": "MY_CUSTOM_ROLE_1",
  "pgx_permissions": [...]
}, {
  "pgx_role": "MY_CUSTOM_ROLE_2",
  "pgx_permissions": [...]
}, {
  "pgx_role": "MY_CUSTOM_ROLE_3",
  "pgx_permissions": [...]
}
```

Note that role and user names in PGX case case-sensitive, whereas in the database they will be case-insensitive (if not quoted). This means that if you perform `CREATE ROLE my_custom_role_1` in the database, you will have to reference it in the `pgx.conf` file with `"pgx_role": "MY_CUSTOM_ROLE_1"`. On the other hand, if you perform `CREATE ROLE "my_custom_role_2"` in the database, you will have to reference it in the `pgx.conf` file with `"pgx_role": "my_custom_role_2"`.

3.1.5.3 Defining Permissions for Individual Users

In addition to defining permissions for roles, you can define permissions for individual users.

For example:

```
"authorization": [{
  "pgx_user": "SCOTT",
  "pgx_permissions": [...]
}, {
  "pgx_user": "JANE",
  "pgx_permissions": [...]
```

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

3.1.6 Examples of Custom Authorization Rules

You can define custom authorization rules for developers.

- Example 3-1
- Example 3-2
- Example 3-3
- Example 3-4

Example 3-1 Allowing Developers to Use Custom Graph Algorithms

To allow developers to compile custom graph algorithms (see Using Custom PGX Graph Algorithms, add the following static permission to the list of permissions:

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

Example 3-2 Allowing Developers to Publish Graphs

Allowing graph server users to publish graphs or share graphs with other users which originate from the Oracle Database breaks the database authorization model. If you work with graphs in the database, use GRANT statements in the database instead. See the OPG_APIS.GRANT_ACCESS API for examples how to do this for PG graphs. When reading from relational tables, use normal GRANT READ statements on tables.

To allow developers to publish graphs, add the following static permission to the list of permissions:

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

Publishing graphs alone does not give others access to the graph. You must also specify the type of access. There are three levels of permissions for graphs:

1. READ: allows to read the graph data via the PGX API or in PGQL queries, run Analyst or custom algorithms on a graph and create a subgraph or clone the given graph

2. EXPORT: export the graph via the PgxGraph#store() APIs. Includes READ permission. Please note that in addition to the EXPORT permission, users also need WRITE permission on a file system in order to export the graph.
3. MANAGE: publish the graph or snapshot, grant or revoke permissions on the graph. Includes the EXPORT 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 = Pgx.getInstance("<base-url>", "<auth-token-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 = Pgx.getInstance("<base-url>", "<auth-token-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 3-3 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. 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 3-4 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:

```
...
"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"
  },
...
```

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:

```
"location": "/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.

3.1.7 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
```

3.2 Reading Data from Oracle Database into Memory

When data is in PGX (that is, when data has been read into memory), you can run any of the built-in algorithms against your data, even compile and execute your own custom algorithms and use PGQL to query results.

Depending on your needs, there are two different approaches to how you can read data from the Oracle Database into PGX.

- **Graph database use case**

  You store your data as a graph in the database and manage that data in the database via graph APIs. You only need PGX as an accelerator for expensive queries or to run graph algorithms on the entire graph.

  For this use case, you should store the data in the property graph format in the Oracle Database (VT$ and GE$ tables), use PGQL on RDBMS to manage the data in the database and then read from that property graph format into PGX. You can also configure PGX to periodically fetch updates from the database automatically in the background to keep the data synchronized.

  Note that the use of PGX is optional in this use case. For some applications the capabilities available in the database only are sufficient.

- **Analytics-only use case**

  Your data is stored in relational form and you want to keep managing that data using standard PL/SQL. You are not interested in a "graph database" but still want to benefit from the analytical capabilities of PGX, which exploit the connectivity of your data for specific analytical use cases.

> **Note:**
>
> This topic applies to both user managed and Autonomous Databases. However, the examples shown are for user managed Databases. For extra configuration steps required for Autonomous Databases, see Using Oracle Graph with the Autonomous Database.

Subtopics:

- Store the database password in a keystore

- 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

Store the database password in a keystore

Regardless of your use case, 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 Read Data from the Database, 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

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 https://docs.oracle.com/cd/E56133_01/latest/reference/loader/database/pg-format.html.)

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"
                }
            ]
        }
    ]
}
```

Note about vertex and edge IDs:

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 default strategy to generate the vertex IDs is to use the keys provided during loading of the graph. In that case, each vertex should have a vertex key that is unique across all the types of vertices. For edges, by default no keys are required in the edge data, and edge IDs will be automatically generated by PGX. 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.

However, because it may cumbersome for partitioned graphs to define such identifiers, it is possible to disable that requirement for the vertices and/or edges by setting the `vertex_id_strategy` and `edge_id_strategy` graph configuration fields to the value `no_ids` as shown in the preceding example. When disabling vertex (resp. edge) IDs, the implication is that PGX will forbid the call to APIs using vertex (resp. edge) IDs, including the ones indicated previously.

If you want to call those APIs but do not have globally unique IDs in your relational tables, you can specify the `unstable_generated_ids` generation strategy, which generates new IDs for you. As the name suggests, there is no correlation to the original IDs in your input tables and there is no guarantee that those IDs are stable. Same as with the edge IDs, it is possible that loading the same input tables twice yields two different generated IDs for the same vertex.

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/opg-jshell --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:

  ```
  opg-jshell-rdbms> 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"
    }],
    ...
  }
  ```

  Now, when you start up the server using the start-server script, provide the path to the keystore file which will prompt for the keystore password before server startup. For example:

  ```
  ./pgx/bin/start-server --secret-store /etc/my-secrets/keystore.p12

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

- **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.

3.3 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 kept synchornized.
- 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.

Limitations for Synchronizing

The synchronizer API currently has the following limitations

- Only partitioned graph configurations with all providers being database tables are supported.
- Both the vertex and edge ID strategy must be set as follows:

  ```
  "vertex_id_strategy": "keys_as_ids",
  "edge_id_strategy": "keys_as_ids"
  ```

- Each edge/vertex provider must be configured to create a key mapping. In each provider block of the graph configuration, add the following loading section:

  ```
  "loading": {
    "create_key_mapping": true
  }
  ```

  This implies that vertices and edges must have globally unique ID columns.
- Each edge/vertex provider 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"
  }
  ```

For a detailed example, including some options, see the following topic.

- Example of Synchronizing
  As an example of performing synchronization, assume you have the following Oracle Database tables, PERSONS and FRIENDSHIPS.

3.3.1 Example of Synchronizing

As an example of performing synchronization, 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,
  INT_PROP INT 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,
  TS_PROP TIMESTAMP,
  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)
);
```

You add some sample data into these tables:

```
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'));
```

Synchronizing Using Connection Information in the Graph Configuration

You then want to synchronize using connection information in the graph configuration. You have the following sample graph configuration (`KeystoreGraphConfigExample`), which reads those tables as a graph:

```
{
  "name": "PeopleFriendships",
  "optimized_for": "updates",
  "edge_id_strategy": "keys_as_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"
```

```
  }
}
```

(In the preceding example, replace the values `<jdbc_url>`, `<username>`, and `<keystore_alias>` with the values for connecting to your database.)

Open the Oracle Property Graph JShell (be sure to register the keystore containing the database password when starting it), and load the graph into memory:

```
var pgxGraph = session.readGraphWithProperties("persons_graph.json");
```

The following output line shows that the example graph has four vertices and four edges:

```
pgxGraph ==> PgxGraph[name=PeopleFriendships,N=4,E=4,created=1594754376861]
```

Now, back in the database, insert a few new rows:

```
INSERT INTO PERSONS (NAME, BIRTHDATE, HEIGHT) VALUES
('Mariana',to_date('21/08/1996','DD/MM/YYYY'),1.65);
INSERT INTO PERSONS (NAME, BIRTHDATE, HEIGHT) VALUES
('Francisco',to_date('13/06/1963','DD/MM/YYYY'),1.75);
INSERT INTO FRIENDSHIPS (PERSON_A, PERSON_B, MEETING_DATE) VALUES (1,
6, to_date('13/06/2013','DD/MM/YYYY'));
COMMIT;
```

Back in JShell, you can now use the `FlashbackSynchronizer` API to automatically fetch and apply those changes:

```
var synchronizer = new
Synchronizer.Builder<FlashbackSynchronizer>().setType(FlashbackSynchroni
zer.class).setGraph(pgxGraph).build()
pgxGraph = synchronizer.sync()
```

As you can see from the output, the new vertices and edges have been applied:

```
pgxGraph ==> PgxGraph[name=PeopleFriendships,N=6,E=5,created=1594754376861]
```

Note that `pgxGraph = synchronizer.sync()` is equivalent to calling the following:

```
synchronizer.sync()
session.setSnapshot(pgxGraph, PgxSession.LATEST_SNAPSHOT)
```

Splitting the Fetching and Applying of Changes

The `synchronizer.sync()` invocation 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()
}
```

Synchronizing Using an Explicit Oracle Connection

The synchronizer API fetches the changes on the client side. That means the client needs to connect to the database. In the preceding example, it did that by reading the connection information available in the graph configuration of the loaded `PgxGraph` object. However, there can be situations in which connection information cannot be obtained from the `PgxGraph` object, such as when:

- The associated graph configuration does not contain any database connection information, and the graph was loaded using credentials of a logged in user; or
- The associated graph configuration contains a datasource ID corresponding to a connection stored on the server side.

In these cases, you can pass in an Oracle connection object building the synchronizer object to be used to fetch changes. For example (`ExampleGraphConfig.json`):

```
String jdbcUrl = "<JDBC URL>";
String username = "<USERNAME>";
String password = "<PASSWORD>";
Connection connection = DriverManager.getConnection(jdbcUrl, username,
password)
Synchronizer synchronizer = new
Synchronizer.Builder<FlashbackSynchronizer>()
    .setType(FlashbackSynchronizer.class)
    .setGraph(pgxGraph)
    .setConnection(connection)
    .build();
```

3.4 Configuring the In-Memory Analyst

You can configure the in-memory analyst engine and its run-time behavior by assigning a single JSON file to the in-memory analyst at startup.

This file can include the parameters shown in the following table. Some examples follow the table.

To specify the configuration file, see Specifying the Configuration File to the In-Memory Analyst.

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 3-2 Configuration Parameters for the In-Memory Analyst

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_local_filesystem	boolean	Allow loading from the local file system in client/server mode. Default is false. If set to true, additionally specify the property `datasource_dir_whitelist` to list the directories. The server can only read from the directories that are whitelisted.	false
allow_override_scheduling_information	boolean	If true, allow all users to override scheduling information like task weight, task priority, and number of threads	true

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
allowed_remote_loading_locations	array of string	Allow loading of graphs into the PGX engine from remote locations (http, https, ftp, ftps, s3, hdfs). Default is empty. Value supported is “*” (asterisk), meaning that all remote locations will be allowed. Note that pre-loaded graphs are loaded from any location, regardless of the value of this setting. WARNING: Specifying * (asterisk) should be done only if you want to explicitly allow users of the PGX remote interface to access files on the local file system.	[]
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 `preload_graphs` can have auto refresh enabled.	false
allowed_remote_loading_locations	array of string	*(This option may reduce security; use it only if you know what you are doing!)* 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.	[]
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

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
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
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
datasource_dir_whitelist	array of string	If `allow_local_filesystem` is set, the list of directories from which it is allowed to read files.	[]
dfs_threshold_large	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
enable_csrf_token_checks	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
enable_gm_compiler	boolean	*[relevant when profiling with solaris studio]* When enabled, label experiments using the 'er_label' command.	false

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
enable_shutdown_cleanup_hook	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
enterprise_scheduler_config	object	Configuration parameters for the enterprise scheduler.	null
enterprise_scheduler_flags	object	*[relevant for enterprise_scheduler]* Enterprise scheduler-specific settings.	null
explicit_spin_locks	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
graph_algorithm_language	enum[GM_LEGACY, GM, JAVA]	Front-end compiler to use.	gm
graph_validation_lever	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	enum[ALLLOW_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.	allow_inconsistencies
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

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
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)	2147483644
max_active_sessions	integer	Maximum number of sessions allowed to be active at a time.	1024
max_distinct_strings_per_pool	integer	*[only relevant if string_pooling_strategy is indexed]* Number of distinct strings per property after which to stop pooling. If the limit is reached, an exception is thrown.	65536
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. Note: 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.	<available-physical-memory>
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 sesssion get rejected. A negative value means no limit.	-1

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
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, enterprise_allocator]	The memory allocator to use.	basic_allocator
memory_cleanup_interval	integer	Memory cleanup interval in seconds.	600
ms_bfs_frontier_type_strategy	enum[auto_grow, short, int]	The type strategy to use for MS-BFS frontiers.	auto_grow
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
pooling_factor	number	*[only relevant if string_pooling_strategy is on_heap]* This value prevents the string pool to grow as big as the property size, which could render the pooling ineffective.	0.25

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
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.	[]
random_generator_strategy	enum[non_deterministic, deterministic]	Method of generating random numbers in PGX.	non_deterministic
random_seed	long	*[relevant for deterministic random number generator only]* Seed for the deterministic random number generator used in the in-memory analyst. The default is -24466691093057031.	-24466691093057031
release_memory_threshold	double	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 reached.	4096

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

Parameter	Type	Description	Default
scheduler	enum[basic_scheduler, enterprise_scheduler]	The scheduler to use. `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.	advanced_scheduler
session_idle_timeout_secs	integer	Timeout of idling sessions in seconds. Zero (0) means 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 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
spark_streams_interface	string	The name of an interface will be used for spark data communication.	null
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]	*[only relevant if use_string_pool is enabled]* The string pooling strategy to use.	on_heap

Table 3-2 (Cont.) Configuration Parameters for the In-Memory Analyst

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).	<system-tmp-dir>
udf_config_directory	string	Directory path containing UDF files.	null
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-CPUs>`

- `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-CPUs>`

Example 3-5 Minimal In-Memory Analyst Configuration

The following example causes the in-memory analyst 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 3-6 Two Pre-loaded Graphs

sets more fields and specifies two fixed graphs for loading into memory during 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
    }
```

```
  ]
}
```

- Specifying the Configuration File to the In-Memory Analyst

3.4.1 Specifying the Configuration File to the In-Memory Analyst

The in-memory analyst configuration file is parsed by the in-memory analyst 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 analyst or specify it programmatically. This topic identifies several ways to specify the file

Programmatically

All configuration fields exist as Java enums. Example:

```
Map<PgxConfig.Field, Object> pgxCfg = new HashMap<>();
pgxCfg.put(PgxConfig.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 analyst 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 analyst 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 Local Shell

To change how the shell configures the local in-memory analyst 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 analyst 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 analyst 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 ...
```

3.5 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 analyst 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 analyst 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 analyst 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 you write is whitelisted in the in-memory analyst 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 whitelist.)

```
opg-jshell> var graph =
session.createGraphBuilder().addVertex(1).addVertex(2).addVertex(3).addE
dge(1,2).addEdge(2,3).addEdge(3, 1).build()
graph ==> PgxGraph[name=anonymous_graph_1,N=3,E=3,created=1581623669674]
```

```
opg-jshell> analyst.pagerank(graph)
$3 ==> VertexProperty[name=pagerank,type=double,graph=anonymous_graph_1]

// Now save the state of this graph

opg-jshell> 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
opg-jshell> 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
opg-jshell> graphFromDisk.queryPgql("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.

```
-jshell> 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
opg-jshell> 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
opg-jshell> 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
opg-jshell> graphFromDisk.queryPgql("select x.pagerank
match (x)").print().close()
```

3.6 Executing Built-in Algorithms

The in-memory analyst contains a set of built-in algorithms that are available as Java APIs.

This topic describes the use of the in-memory analyst using Triangle Counting and PageRank analytics as examples.

- About the In-Memory Analyst

- Running the Triangle Counting Algorithm
- Running the PageRank Algorithm

3.6.1 About the In-Memory Analyst

The in-memory analyst 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);
```

3.6.2 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

```
opg> 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 in-memory analyst 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`.

3.6.3 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 analyst, 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

```
opg> rank = analyst.pagerank(graph, 0.001, 0.85, 100);
==> ...
opg> 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());
}
```

3.7 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 than appears in the following subtopics, see the PGX Algorithm specification at https://docs.oracle.com/cd/E56133_01/latest/PGX_Algorithm_Language_Specification.pdf.

- Writing a Custom PGX Algorithm
- Compiling and Running a PGX Algorithm
- Example Custom PGX Algorithm: PageRank

3.7.1 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. 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 can return a value (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

3.7.1.1 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);
```

3.7.1.2 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()`.

3.7.1.3 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)
);
```

3.7.2 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).
2. Create a session (either implicitly in the shell or explicitly in Java). For example:

   ```
   cd $PGX_HOME
   ./bin/opg
   ```

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)
   ```

3.7.3 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);
  }
}
```

3.8 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 Reading Data from Oracle Database into Memory.

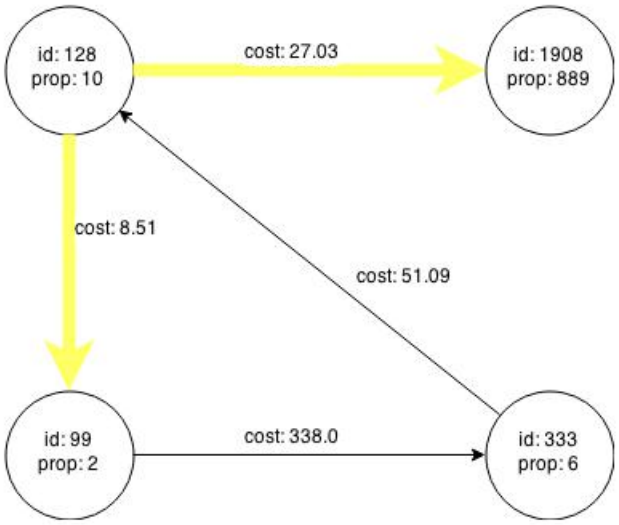
- About Filter Expressions
- Using a Simple Filter to Create a Subgraph
- Using a Complex Filter to Create a Subgraph
- Using a Vertex Set to Create a Bipartite Subgraph

3.8.1 About Filter Expressions

Filter expressions are expressions that are evaluated for each edge. The expression can define predicates that an edge must fulfil to be contained in the result, in this case a subgraph.

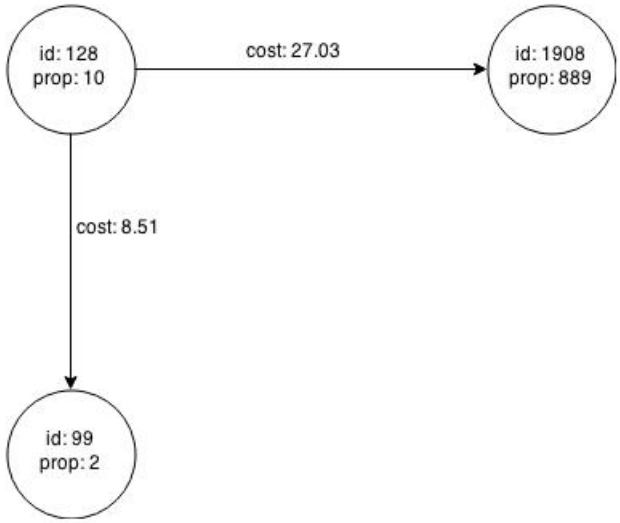
Consider an example graph that consists of four vertices (nodes) and four edges. For an edge to match the filter expression `src.prop == 10`, the source vertex `prop` property must equal 10. Two edges match that filter expression, as shown in the following figure.

Figure 3-1 Edges Matching src.prop == 10



The following figure shows the graph that results when the filter is applied. The filter excludes the edges associated with vertex 333, and the vertex itself.

Figure 3-2 Graph Created by the Simple Filter



Using filter expressions to select a single vertex or a set of vertices is difficult. For example, selecting only the vertex with the property value `10` is impossible, because the only way to match the vertex is to match an edge where `10` is either the source or destination property value. However, when you match an edge you automatically include the source vertex, destination vertex, and the edge itself in the result.

3.8.2 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"));
```

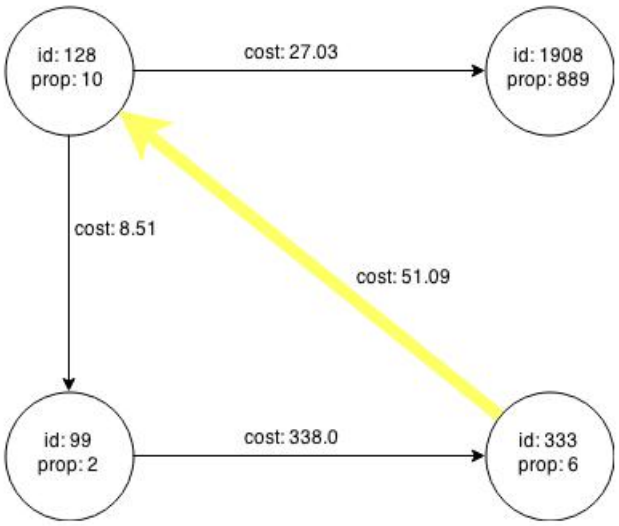
3.8.3 Using a Complex Filter to Create a Subgraph

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
```

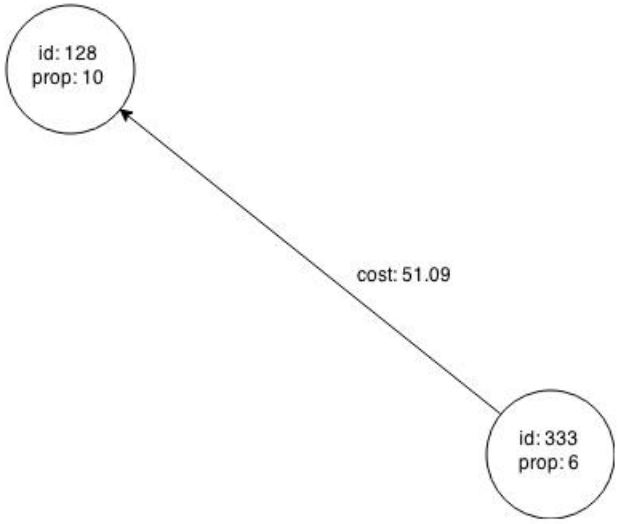
One edge in the sample graph matches this filter expression, as shown in the following figure.

Figure 3-3 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 vertixes 99 and 1908, and so excludes those vertices also.

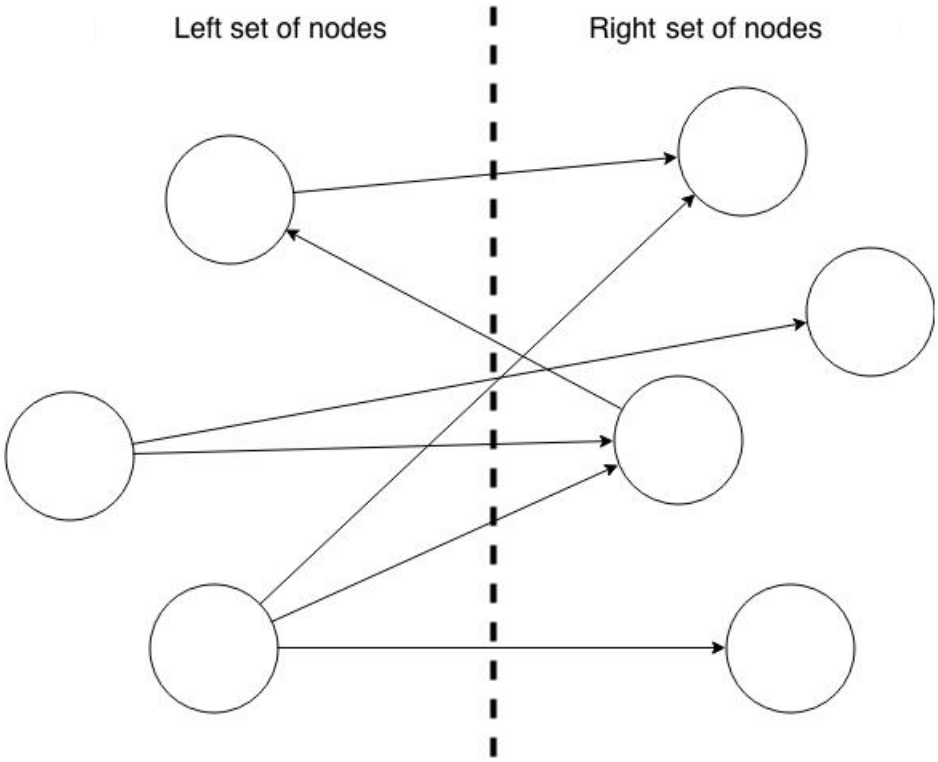
Figure 3-4 Graph Created by the outDegree Filter



3.8.4 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 analyst, 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

```
opg> s = graph.createVertexSet()
==> ...
opg> s.addAll([graph.getVertex(333), graph.getVertex(99)])
==> ...
opg> s.size()
==> 2
opg> bGraph = graph.bipartiteSubGraphFromLeftSet(s)
==> PGX Bipartite Graph named sample-sub-graph-4
```

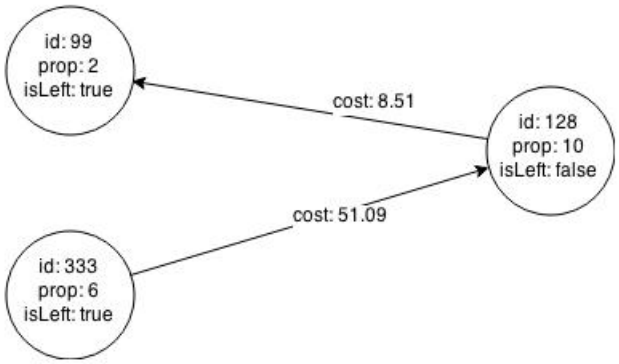
Using Java to Create a Bipartite Subgraph

```
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 analyst 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 also.

3.9 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 Analyst or a Java Application
- Checking Out a Specific Snapshot of the Graph
- Advanced Auto-Refresh Configuration

3.9.1 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 analyst configuration file (located in `$ORACLE_HOME/md/property_graph/pgx/conf/pgx.conf`):

```
{
  "allow_user_auto_refresh": true
}
```

3.9.2 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();
```

3.9.3 Reading the Graph Using the In-Memory Analyst or a Java Application

After creating the graph configuration, you can load the graph into the in-memory analyst using the regular APIs.

```
opg> 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.

3.9.4 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:

```
opg> 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 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 in-memory analyst shell:

```
opg> G.getNumVertices()
==> 5
opg> G.getNumEdges()
==> 5

opg> session.setSnapshot( G, 1453315122685 )
==> null

opg> G.getNumVertices()
==> 4
opg> 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:

```
opg> 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:

```
opg> 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.

3.9.5 Advanced Auto-Refresh Configuration

You can specify advanced options for auto-refresh configuration.

Internally, the in-memory analyst 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 analyst 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
  }
}
```

3.10 Starting the In-Memory Analyst Server

A preconfigured version of Apache Tomcat is bundled, which allows you to start the in-memory analyst server by running a script.

If you need to configure the server before starting it, see Configuring the In-Memory Analyst Server.

You can start the server by running the following script: `/opt/oracle/graph/pgx/bin/start-server`

Note that running the `start-server` script does not start the server as a daemon, and the terminal will not return until you stop the server (for example, by pressing Ctrl+C to interrupt the process). This also means that the server will stop running if you close the terminal in which you started the script.

PGX is integrated with `systemd` to run it as a Linux service in the background. To start the PGX server as a daemon process, use the following command (you must have root privileges):

```
systemctl start pgx
```

To stop the server, use:

```
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.

- Configuring the In-Memory Analyst Server

3.10.1 Configuring the In-Memory Analyst Server

You can configure the in-memory analyst server 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 3-3 Configuration Options for In-Memory Analyst Server

Option	Type	Description	Default
authorization	string	File that maps clients to roles for authorization.	server.auth.conf

Table 3-3 (Cont.) Configuration Options for In-Memory Analyst Server

Option	Type	Description	Default
ca_certs	array of string	List of trusted certificates (PEM format). If 'enable_tls' is set to false, this option has no effect.	[See information after this table.]
enable_client_authentication	boolean	If true, the client is authenticated during TLS handshake. See the TLS protocol for details. This flag does not have any effect if 'enable_tls' is false.	true
enable_tls	boolean	If true, the server enables transport layer security (TLS).	true
port	integer	Port that the PGX server should listen on	7007
server_cert	string	The path to the server certificate to be presented to TLS clients (PEM format). If 'enable_tls' is set to false, this option has no effect	null
server_private_key	string	the private key of the server (PKCS#8, PEM format). If 'enable_tls' is set to false, this option has no effect	null

The in-memory analyst web server 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,
  "server_cert": "certificates/server_certificate.pem",
  "server_private_key": "certificates/server_key.pem",
  "ca_certs": [ "certificates/ca_certificate.pem" ],
  "authorization": "auth/server.auth.conf",
  "enable_tls": true,
  "enable_client_authentication": true
}
```

The following is an example `server.auth.conf` configuration file: mapping client (applications) identified by their certificate DN string to roles:

```
{
  "authorization": [{
    "dn": "CN=Client, OU=Development, O=Oracle, L=Belmont, ST=California, C=US",
   "admin": false
  }, {
```

```
    "dn": "CN=Admin, OU=Development, O=Oracle, L=Belmont, ST=California, C=US",
   "admin": true
  }]
}
```

You can turn off client-side authentication or SSL/TLS authentication entirely in the server configuration. However, we recommend having two-way SSL/TLS enabled for any production usage.

3.11 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 and higher.

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 for Tomcat. It follows the naming pattern: `graph-server-<version>-pgx<version>-tomcat.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.
5. Copy the `.war` file into the Tomcat `webapps` directory. For example:

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

Related Topics

- The Tomcat documentation (select desired version)

3.12 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 for Weblogic server.
   a. For Weblogic Server version 12.1.x, use this web application archive: `graph-server-<version>-pgx<version>-wls121x.war`
   b. For Weblogic Server version 12.2.x, use this web application archive: `graph-server-<version>-pgx<version>-wls122x.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>
```

3.13 Connecting to the In-Memory Analyst Server

After the property graph in-memory analyst is installed in a computer running Oracle Database -- or on a client system without Oracle Database server software as a web application on Apache Tomcat or Oracle WebLogic Server -- you can connect to the in-memory analyst server.

- Connecting with the In-Memory Analyst Shell
- Connecting with Java
- Connecting with the PGX REST API

3.13.1 Connecting with the In-Memory Analyst Shell

The simplest way to connect to an in-memory analyst instance is to specify the base URL of the server. The following base URL can connect the SCOTT user to the local instance listening on port 8080:

```
http://scott:<password>@localhost:8080/pgx
```

To start the in-memory analyst shell with this base URL, you use the `--base_url` command line argument

```
cd $PGX_HOME
./bin/opg-jshell --base_url http://scott:<password>@localhost:8080/pgx
```

You can connect to a remote instance the same way. However, the in-memory analyst currently does not provide remote support for the Control API.

- About Logging HTTP Requests

3.13.1.1 About Logging HTTP Requests

The in-memory analyst 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:

```
opg> /loglevel oracle.pgx DEBUG
===> log level of oracle.pgx logger set to DEBUG
opg> session.readGraphWithProperties("sample_http.adj.json", "sample")
10:24:25,056 [main] DEBUG RemoteUtils - Requesting POST http://
scott:<password>@localhost:8080/pgx/core/session/session-shell-6nqg5dd/graph
HTTP/1.1 with payload {"graphName":"sample","graphConfig":{"uri":"http://
path.to.some.server/pgx/sample.adj","separator":" ","edge_props":
[{"type":"double","name":"cost"}],"node_props":
[{"type":"integer","name":"prop"}],"format":"adj_list"}}
10:24:25,088 [main] DEBUG RemoteUtils - received HTTP status 201
10:24:25,089 [main] DEBUG RemoteUtils - {"futureId":"87d54bed-bdf9-4601-98b7-
ef632ce31463"}
10:24:25,091 [pool-1-thread-3] DEBUG PgxRemoteFuture$1 - Requesting GET http://
scott:<password>@localhost:8080/pgx/future/session/session-shell-6nqg5dd/result/
87d54bed-bdf9-4601-98b7-ef632ce31463 HTTP/1.1
10:24:25,300 [pool-1-thread-3] DEBUG RemoteUtils - received HTTP status 200
10:24:25,301 [pool-1-thread-3] DEBUG RemoteUtils - {"stats":
{"loadingTimeMillis":0,"estimatedMemoryMegabytes":0,"numEdges":4,"numNodes":4},"g
raphName":"sample","nodeProperties":{"prop":"integer"},"edgeProperties":
{"cost":"double"}}
```

This example requires that the graph URI points to a file that the in-memory analyst server can access using HTTP or HDFS.

3.13.2 Connecting with Java

You can specify the base URL when you initialize the in-memory analyst using Java. An example is as follows. A URL to an in-memory analyst server is provided to the `getInMemAnalyst` API call.

```
import oracle.pg.rdbms.*;
import oracle.pgx.api.*;
```

```
PgRdbmsGraphConfigcfg =
GraphConfigBuilder.forPropertyGraphRdbms().setJdbcUrl("jdbc:oracle:thin:@127.0.0.
1:1521:orcl")
   .setUsername("scott").setPassword("<password>")  .setName("mygraph")
   .setMaxNumConnections(2) .setLoadEdgeLabel(false)
   .addVertexProperty("name", PropertyType.STRING, "default_name")
   .addEdgeProperty("weight", PropertyType.DOUBLE, "1000000")
   .build();OraclePropertyGraph opg = OraclePropertyGraph.getInstance(cfg);
ServerInstance remoteInstance = Pgx.getInstance("http://
scott:<password>@hostname:port/pgx");
PgxSession session = remoteInstance.createSession("my-session");

PgxGraph graph = session.readGraphWithProperties(opg.getConfig());
```

3.13.3 Connecting with the PGX REST API

You can connect to an in-memory analyst instance using the REST API PGX endpoints. This enables you to interact with the in-memory analyst in a language other than Java to implement your own client.

The examples in this topic assume that:

- Linux with curl is installed. curl is a simple command-line utility to interact with REST endpoints.)
- The PGX server is up and running on `http://localhost:7007`.
- The PGX server has authentication/authorization disabled; that is, `$ORACLE_HOME/md/property_graph/pgx/conf/server.conf` contains `"enable_tls": false`. (This is a non-default setting and **not recommended** for production).
- PGX allows reading graphs from the local file system; that is, `$ORACLE_HOME/md/property_graph/pgx/conf/pgx.conf` contains `"allow_local_filesystem": true`. (This is a non-default setting and **not recommended** for production).

For the Swagger specification, you can see a full list of supported endpoints in JSON by opening `http://localhost:7007/swagger.json` in your browser.

- Step 1: Obtain a CSRF token
- Step 2: Create a session
- Step 3: Read a graph
- Step 4: Create a property
- Step 5: Run the PageRank algorithm on the loaded graph
- Step 6: Execute a PGQL query

Step 1: Obtain a CSRF token

Request a CSRF token:

```
curl -v http://localhost:7007/token
```

The response will look like this:

```
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 7007 (#0)
> GET /token HTTP/1.1
> Host: localhost:7007
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 201
< SET-COOKIE: _csrf_token=9bf51c8f-1c75-455e-9b57-ec3ca1c63cc0;Version=1;
HttpOnly
< Content-Length: 0
```

As you can see in the response, this will set a cookie `_csrf_token` to a token value. `9bf51c8f-1c75-455e-9b57-ec3ca1c63cc0` is used as an example token for the following requests. For any write requests, PGX server requires the same token to be present in both cookie and payload.

Step 2: Create a session

To create a new session, send a JSON payload:

```
curl -v --cookie '_csrf_token=9bf51c8f-1c75-455e-9b57-ec3ca1c63cc0'
-H 'content-type: application/json' -X POST http://
localhost:7007/core/v1/sessions -d '{"source":"my-application",
"idleTimeout":0, "taskTimeout":0, "timeUnitName":"MILLISECONDS",
"_csrf_token":"9bf51c8f-1c75-455e-9b57-ec3ca1c63cc0"}'
```

Replace `my-application` with a value describing the application that you are running. This value can be used by server administrators to map sessions to their applications. Setting idle and task timeouts to `0` means the server will determine when the session and submitted tasks time out. You must provide the same CSRF token in both the cookie header and the JSON payload.

The response will look similar to the following:

```
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 7007 (#0)
> POST /core/v1/sessions HTTP/1.1
> Host: localhost:7007
> User-Agent: curl/7.47.0
> Accept: */*
> Cookie: _csrf_token=9bf51c8f-1c75-455e-9b57-ec3ca1c63cc0
> content-type: application/json
> Content-Length: 159
>
* upload completely sent off: 159 out of 159 bytes
< HTTP/1.1 201
< SET-COOKIE: SID=abae2811-6dd2-48b0-93a8-8436e078907d;Version=1; HttpOnly
< Content-Length: 0
```

The response sets a cookie to the session ID value that was created for us. Session ID `abae2811-6dd2-48b0-93a8-8436e078907d` is used as an example for subsequent requests.

Step 3: Read a graph

> **Note:**
>
> if you want to analyze a pre-loaded graph or a graph that is already published by another session, you can skip this step. All you need to access pre-loaded or published graphs is the name of the graph.

To read a graph, send the graph configuration as JSON to the server as shown in the following example (replace `<graph-config>` with the JSON representation of an actual PGX graph config).

```
curl -v -X POST --cookie '_csrf_token=9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0;SID=abae2811-6dd2-48b0-93a8-8436e078907d'  http://
localhost:7007/core/v1/loadGraph -H 'content-type:
application/json'  -d  '{"graphConfig":<graph-
config>,"graphName":null,"csrf_token":"9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0"}'
```

Here an example of a graph config that reads a property graph from the Oracle database:

```
{
  "format": "pg",
  "db_engine": "RDBMS",
  "jdbc_url":"jdbc:oracle:thin:@127.0.0.1:1521:orcl122",
  "username":"scott",
  "password":"tiger",
  "max_num_connections": 8,
  "name": "connections",
  "vertex_props": [
    {"name":"name", "type":"string"},
    {"name":"role", "type":"string"},
    {"name":"occupation", "type":"string"},
    {"name":"country", "type":"string"},
    {"name":"political party", "type":"string"},
    {"name":"religion", "type":"string"}
  ],
  "edge_props": [
    {"name":"weight", "type":"double", "default":"1"}
  ],
  "edge_label": true,
  "loading": {
    "load_edge_label": true
  }
}
```

Passing `"graphName": null` tells the server to generate a name.

The server will reply something like the following:

```
* upload completely sent off: 315 out of 315 bytes
< HTTP/1.1 202
< Location: http://localhost:7007/core/v1/futures/8a46ef65-01a9-4bd0-87d3-
ffe9dfd2ce3c/status
< Content-Type: application/json;charset=utf-8
< Content-Length: 51
< Date: Mon, 05 Nov 2018 17:22:22 GMT
<
* Connection #0 to host localhost left intact
{"futureId":"8a46ef65-01a9-4bd0-87d3-ffe9dfd2ce3c"}
```

About Asynchronous Requests

Most of the PGX REST endpoints are asynchronous. Instead of keeping the connection open until the result is ready, PGX server submits as task and immediately returns a future ID with status code 200, which then can be used by the client to periodically request the status of the task or request the result value once done.

From the preceding response, you can request the future status like this:

```
curl -v --cookie
'SID=abae2811-6dd2-48b0-93a8-8436e078907d'  http://localhost:7007/
core/v1/futures/8a46ef65-01a9-4bd0-87d3-ffe9dfd2ce3c/status
```

Which will return something like:

```
< HTTP/1.1 200
< Content-Type: application/json;charset=utf-8
< Content-Length: 730
< Date: Mon, 05 Nov 2018 17:35:19 GMT
<
* Connection #0 to host localhost left intact
{"id":"eb17f75b-e4c1-4a66-81a0-4ff0f8b4cb92","links":[{"href":"http://
localhost:7007/core/v1/futures/eb17f75b-e4c1-4a66-81a0-4ff0f8b4cb92/
status","rel":"self","method":"GET","interaction":["async-
polling"]},{"href":"http://localhost:7007/core/v1/futures/eb17f75b-
e4c1-4a66-81a0-4ff0f8b4cb92","rel":"abort","method":"DELETE","interaction":
["async-polling"]},{"href":"http://localhost:7007/
core/v1/futures/eb17f75b-e4c1-4a66-81a0-4ff0f8b4cb92/
status","rel":"canonical","method":"GET","interaction":["async-
polling"]},{"href":"http://localhost:7007/core/v1/futures/eb17f75b-
e4c1-4a66-81a0-4ff0f8b4cb92/value","rel":"related","method":"GET","interaction":
["async-polling"]}],"progress":"succeeded","completed":true,"intervalToPoll":1}
```

Besides the status (`succeeded` in this case), this output also includes links to cancel the task (`DELETE`) and to retrieve the result of the task once completed (`GET <future-id>/value`):

```
curl -X GET --cookie
'SID=abae2811-6dd2-48b0-93a8-8436e078907d' http://localhost:7007/
core/v1/futures/cdc15a38-3422-42a1-baf4-343c140cf95d/value
```

Which will return details about the loaded graph, including the name that was generated by the server (`sample`):

```
{"id":"sample","links":[{"href":"http://localhost:7007/core/v1/graphs/
sample","rel":"self","method":"GET","interaction":["async-polling"]},
{"href":"http://localhost:7007/core/v1/graphs/
```

```
sample","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"nodeProperties":{"prop1":{"id":"prop1","links":[{"href":"http://
localhost:7007/core/v1/graphs/sample/properties/
prop1","rel":"self","method":"GET","interaction":["async-polling"]},
{"href":"http://localhost:7007/core/v1/graphs/sample/properties/
prop1","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"dimension":0,"name":"prop1","entityType":"vertex","type":"integer","
transient":false}},"vertexLabels":null,"edgeLabel":null,"metaData":
{"id":null,"links":null,"numVertices":4,"numEdges":4,"memoryMb":0,"dataSourceVers
ion":"1536029578000","config":{"format":"adj_list","separator":" ","edge_props":
[{"type":"double","name":"cost"}],"error_handling":{},"vertex_props":
[{"type":"integer","name":"prop1"}],"vertex_uris":
["PATH_TO_FILE"],"vertex_id_type":"integer","loading":
{}},"creationRequestTimestamp":1541242100335,"creationTimestamp":1541242100774,"v
ertexIdType":"integer","edgeIdType":"long","directed":true},"graphName":"sample",
"edgeProperties":{"cost":{"id":"cost","links":[{"href":"http://localhost:7007/
core/v1/graphs/sample/properties/cost","rel":"self","method":"GET","interaction":
["async-polling"]},{"href":"http://localhost:7007/core/v1/graphs/sample/
properties/cost","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"dimension":0,"name":"cost","entityType":"edge","type":"double","tran
sient":false}},"ageMs":0,"transient":false}
```


For simplicity, the remaining steps omit the additional requests to request the status or value of asynchronous tasks.

Step 4: Create a property

Before you can run the PageRank algorithm on the loaded graph, you must create a vertex property of type DOUBLE on the graph, which can hold the computed ranking values:

```
curl -v -X POST --cookie '_csrf_token=9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0;SID=abae2811-6dd2-48b0-93a8-8436e078907d'  http://
localhost:7007/core/v1/graphs/sample/properties -H
'content-type: application/json'  -d
'{"entityType":"vertex","type":"double","name":"pagerank",
"hardName":false,"dimension":0,"_csrf_token":"9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0"}'
```

Requesting the result of the returned future will return something like:


```
{"id":"pagerank","links":[{"href":"http://localhost:7007/core/v1/graphs/sample/
properties/pagerank","rel":"self","method":"GET","interaction":["async-
polling"]},{"href":"http://localhost:7007/core/v1/graphs/sample/properties/
pagerank","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"dimension":0,"name":"pagerank","entityType":"vertex","type":"double"
,"transient":true}
```


Step 5: Run the PageRank algorithm on the loaded graph

The following example shows how to run an algorithm (PageRank in this case). The algorithm ID is part of the URL, and the parameters to be passed into the algorithm are part of the JSON payload:

```
curl -v -X POST --cookie '_csrf_token=9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0;SID=abae2811-6dd2-48b0-93a8-8436e078907d' http://
localhost:7007/core/v1/analyses/pgx_builtin_k1a_pagerank/run -H
'content-type: application/json' -d  '{"args":
```

```
[{"type":"GRAPH","value":"sample"},{"type":"DOUBLE_IN","value":0.001},
{"type":"DOUBLE_IN","value":0.85},{"type":"INT_IN","value":100},
{"type":"BOOL_IN","value":true},
{"type":"NODE_PROPERTY","value":"pagerank"}],"expectedReturnType":"void"
,"workloadCharacteristics":
["PARALLELISM.PARALLEL"],"_csrf_token":"9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0"}'
```

Once the future is completed, the result will look something like this:

```
{"success":true,"canceled":false,"exception":null,"returnValue":null,"executionTi
meMs":50}
```

Step 6: Execute a PGQL query

To query the results of the PageRank algorithm, you can run a PGQL query as shown in the following example:

```
curl -v -X POST --cookie '_csrf_token=9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0;SID=abae2811-6dd2-48b0-93a8-8436e078907d' http://
localhost:7007/core/v1/pgql/run -H 'content-type: application/json'
-d '{"pgqlQuery":"SELECT x.pagerank MATCH (x) WHERE
x.pagerank > 0","semantic":"HOMOMORPHISM", "schemaStrictnessMode":true,
"graphName" : "sample", "_csrf_token":"9bf51c8f-1c75-455e-9b57-
ec3ca1c63cc0"}'
```

The result is a set of links you can use to interact with the result set of the query:

```
{"id":"pgql_1","links":[{"href":"http://localhost:7007/core/v1/pgqlProxies/
pgql_1","rel":"self","method":"GET","interaction":["sync"]},{"href":"http://
localhost:7007/core/v1/pgqlResultProxies/pgql_1/
elements","rel":"related","method":"GET","interaction":["sync"]},{"href":"http://
localhost:7007/core/v1/pgqlResultProxies/pgql_1/
results","rel":"related","method":"GET","interaction":["sync"]},{"href":"http://
localhost:7007/core/v1/pgqlProxies/
pgql_1","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"exists":true,"graphName":"sample","resultSetId":"pgql_1","numResults
":4}
```

To request the first 2048 elements of the result set, send:

```
curl -X GET  --cookie 'SID=abae2811-6dd2-48b0-93a8-8436e078907d' http://
localhost:7007/core/v1/pgqlProxies/pgql_1/results?size=2048
```

The response looks something like this:

```
{"id":"/pgx/core/v1/pgqlProxies/pgql_1/results","links":[{"href":"http://
localhost:7007/core/v1/pgqlProxies/pgql_1/
results","rel":"self","method":"GET","interaction":["sync"]},{"href":"http://
localhost:7007/core/v1/pgqlProxies/pgql_1/
results","rel":"canonical","method":"GET","interaction":["async-
polling"]}],"count":4,"totalItems":4,"items":[[0.3081206521195582],
[0.21367103988538017],[0.21367103988538017],
[0.2645372681096815]],"hasMore":false,"offset":0,"limit":4,"showTotalResults":tru
e}
```

3.14 Managing Property Graph Snapshots

You can manage property graph snapshots.

> **Note:**
>
> Managing property graph snapshots is intended for advanced users.

You can persist different versions of a property graph as binary snapshots in the database. The binary snapshots represent a subgraph of graph data computed at runtime that may be needed for a future use. The snapshots can be read back later as input for the in-memory analytics, or as an output stream that can be used by the parallel property graph data loader.

You can **store** binary snapshots in the <graph_name>SS$ table of the property graph using the Java API `OraclePropertyGraphUtils.storeBinaryInMemoryGraphSnapshot`. This operation requires a connection to the Oracle database holding the property graph instance, the name of the graph and its owner, the ID of the snapshot, and an input stream from which the binary snapshot can be read. You can also specify the time stamp of the snapshot and the degree of parallelism to be used when storing the snapshot in the table.

You can **read** a stored binary snapshot using `oraclePropertyGraphUtils.readBinaryInMemGraphSnapshot`. This operation requires a connection to the Oracle database holding the property graph instance, the name of the graph and its owner, the ID of the snapshot to read, and an output stream where the binary file snapshot will be written into. You can also specify the degree of parallelism to be used when reading the snapshot binary-file from the table.

The following code snippet creates a property graph from the data file in Oracle Flat-file format, adds a new vertex, and exports the graph into an output stream using GraphML format. This output stream represents a binary file snapshot, and it is stored in the property graph snapshot table. Finally, this example reads back the file from the snapshot table and creates a second graph from its contents.

```
String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args,
szGraphName);
opgdl = OraclePropertyGraphDataLoader.getInstance();
opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
               "PDML=T,PDDL=T,NO_DUP=T,");

// Add a new vertex
Vertex v = opg.addVertex(Long.valueOf("1000"));
v.setProperty("name", "Alice");
opg.commit();

System.out.pritnln("Graph " + szGraphName + " total vertices: " +
                   opg.countVertices(dop));
System.out.pritnln("Graph " + szGraphName + " total edges: " +
```

```
                    opg.countEdges(dop));


// Get a snapshot of the current graph as a file in graphML format.
OutputStream os = new ByteArrayOutputStream();
OraclePropertyGraphUtils.exportGraphML(opg,
                                       os /* output stream */,
                                       System.out /* stream to show
progress */);

// Save the snapshot into the SS$ table
InputStream is = new ByteArrayInputStream(os.toByteArray());
OraclePropertyGraphUtils.storeBinaryInMemGraphSnapshot(szGraphName,
                                                       szGraphOwner /*
owner of the

property graph */,
                                                       conn /* database

connection */,
                                                       is,
                                                       (long) 1 /*
snapshot ID */,
                                                       1 /* dop */);
os.close();
is.close();

// Read the snapshot back from the SS$ table
OutputStream snapshotOS = new ByteArrayOutputStream();
OraclePropertyGraphUtils.readBinaryInMemGraphSnapshot(szGraphName,
                                                      szGraphOwner /*
owner of the

property graph */,
                                                      conn /* database

connection */,
                                                      new OutputStream[]
{snapshotOS},
                                                      (long) 1 /*

snapshot ID */,
                                                      1 /* dop */);

InputStream snapshotIS = new
ByteArrayInputStream(snapshotOS.toByteArray());
String szGraphNameSnapshot = szGraphName + "_snap";
OraclePropertyGraph opg =
OraclePropertyGraph.getInstance(args,szGraphNameSnapshot);

OraclePropertyGraphUtils.importGraphML(opg,
                                       snapshotIS /* input stream */,
                                       System.out /* stream to show
progress */);

snapshotOS.close();
snapshotIS.close();
```

```
System.out.pritnln("Graph " + szGraphNameSnapshot + " total vertices: "
+
                   opg.countVertices(dop));
System.out.pritnln("Graph " + szGraphNameSnapshot + " total edges: " +
                   opg.countEdges(dop));
```

The preceding example will produce output similar as the following:

```
Graph test total vertices: 79
Graph test total edges: 164
Graph test_snap total vertices: 79
Graph test_snap total edges: 164
```

3.15 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 the 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.

How to Use UDFs

The following simple example shows how to register a 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.package.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:

```
import oracle.pgx.algorithm.annotations.Udf;
....

@GraphAlgorithm
public class MyAlogrithm {
  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);
}
```

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 UDFs" subsection.)

Each user-defined function supports the fields shown in the following table.

Table 3-4 Fields for Each UDF

Field	Data Type	Description	Required?
function_name	string	Name of the function used as identifier in PGX	Required
language	enum[java, javascript]	Source language for he function (`java` or `javascript`)	Required
return_type	enum[boolean, integer, long, float, double, string]	Return type of the function	Required
arguments	array of object	Array of arguments. For each argument: type, argument name, required?	[]
implementation_reference	string	Reference to the function name on the classpath	null
namespace	string	Namespace of the function in PGX	null
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

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.

4.1 Simple Property Graph Queries

The examples in this topic query vertices, edges, and properties of the graph.

Example 4-1 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 4-2 Find an Edge with a Specified Edge ID

This example find 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 4-3 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 4-4 Get the Set of Property Keys Used

This example gets the set of property keys used for the vertices n 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 4-5 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.n 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 |       |       |  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 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
```

4.2 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 4-6 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 4-7 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) | 00:00:01 |       |       |
|   1 |  SORT ORDER BY                      |                 |     1 |    56
|     5  (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 4-8 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 (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------
-------------------------------------
|   0 | SELECT STATEMENT                    |                 |     1 |    56
|     5  (20)| 00:00:01 |       |       |
|   1 |  SORT ORDER BY                      |                 |     1 |    56
|     5  (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(INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
```

Example 4-9 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                   |                |       1 |      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>      <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)
```

Example 4-10 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.

4.3 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 4-11 Who Are a Person's Collaborators?

The following SQL ststement 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      | Pstart| Pstop |    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 4-12 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     | Pstart| Pstop |    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 4-13 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| CONNECTIONSGE$  |      5
|    125 |      2   (0)| 00:00:01 |     1 |     8 |  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 4-14 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  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ    |IN-OUT| PQ
Distrib |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
---
|   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     (0)| 00:00:01 |     1 |     8 |  Q1,00 | PCWP |        |
|  14 |        PARTITION HASH ITERATOR                        |
|     1 |       |     0     (0)| 00:00:01 |   KEY |   KEY |  Q1,00 | PCWP
|        |
|* 15 |         TABLE ACCESS BY LOCAL INDEX ROWID             | CONNECTIONSVT$
|       |       |        |                |   KEY |   KEY |  Q1,00 | PCWP
|        |
|* 16 |          INDEX UNIQUE SCAN                            | CONNECTIONSXQV$
|     1 |       |     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"))
```

4.4 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 4-15 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 4-16 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
|                             |   801 | 31239 |   147   (0)| 00:00:01 |
|       |        |      |            |
|   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)
|                            |       |       |            |          |
|       |  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
|                            |       |       |            |          |
|       |  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)
```

4.5 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 4-17 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)
```

4.6 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

4.6.1 Shortest Path Examples

The following examples demonstrate SQL-based shortest path analytics.

Example 4-18 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 4-19 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 1.000E+100
0         -1
        26          1          1 1.000E+100
0         -1
         8       1000          1 1.000E+100
0         -1
         3          1          1          2         52
0          0
        15          1          1 1.000E+100
0         -1
        21          1          1 1.000E+100
0         -1
        19          1          1 1.000E+100
0         -1
       ...
```

Example 4-20 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.
```

4.6.2 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.

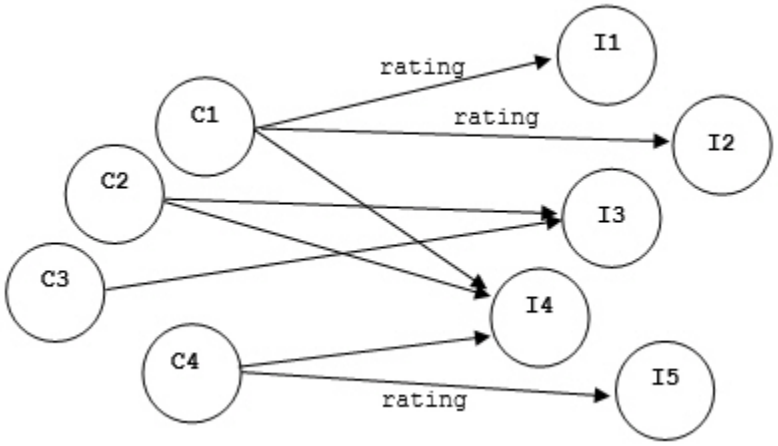
The following examples demonstrate SQL-based collaborative filtering analytics.

Example 4-21 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 4-1 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_l1,wt_r1,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_l1    "PHONESGE$$CFL157"
working table wt_r1    "PHONESGE$$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 4-22 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
```

f 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 4-23 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_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);

  -- 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_l1,wt_r1,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_l1,wt_r1,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        4.21764858
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        4.21764858
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
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.

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 Directly Against Oracle Database.

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 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).

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

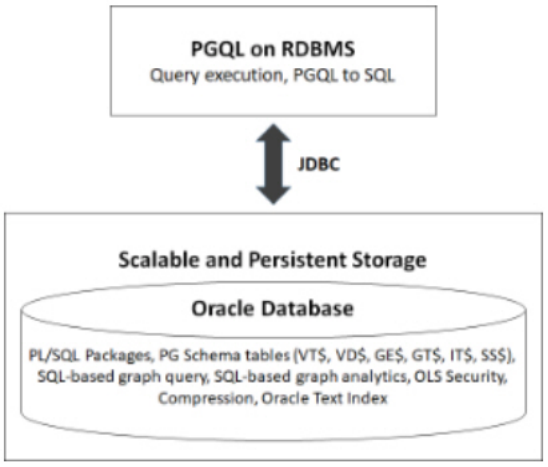
5.7 Executing PGQL Queries Directly Against Oracle Database

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.

This PGQL query execution flow is shown in the following figure.

Figure 5-1 PGQL on 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.

5.7.1 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
- IN and NOT IN predicates
- 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

5.7.1.1 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.

5.7.1.2 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 5-1 Type Casting Support in PGQL (From and To Types)

"to" type	from STRING	from INT	from LONG	from FLOAT	from DOUBLE	from BOOLEAN	from DATE	from TIMESTAMP	from TIMESTAMP 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
to BOOLEAN	Y	Y	Y	Y	Y	Y	N	N	N
to DATE	Y	N	N	N	N	N	Y	Y	Y
to TIMESTAMP	Y	N	N	N	N	N	Y	Y	Y
to TIMESTAMP 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)
```

5.7.1.3 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.

5.7.2 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 5-1 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 " +
          "    Key(id) " +
          "    Label \"people\" +
          "    PROPERTIES(name AS \"first_name\", dob AS \"birthday\")," +
          "  Hobby  " +
          "    Key(id) Label \"hobby\"  PROPERTIES(name AS \"name\")" +
          ")" +
          "EDGE TABLES (" +
          "  Hobbies" +
          "    SOURCE KEY(person) REFERENCES Person " +
          "    DESTINATION KEY(hobby) REFERENCES Hobby " +
          "    LABEL \"likes\" PROPERTIES (strength AS \"score\")" +
          ")";
        pgqlStmt.execute(pgql);

        // Execute a PGQL query to verify Graph creation
        pgql =
          "SELECT p.\"first_name\", p.\"birthday\", h.\"name\",
e.\"score\" " +
          "FROM MATCH (p:\"people\")-[e:\"likes\"]->(h:\"hobby\") ON " +
graph;
        rs = pgqlStmt.executeQuery(pgql, "");

        // Print the results
        rs.print();
      }
      finally {
        // close the sql statment
        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 `PgqlCreateExample1.java` is:

```
+-------------------------------------------------------+
|   first_name |   birthday            |   name |   score |
+-------------------------------------------------------+
| Alan         | 1995-05-25 17:00:00.0 | Music  | 30.0    |
| Claire       | 1967-11-29 16:00:00.0 | Music  | 20.0    |
| Ben          | 2007-02-14 16:00:00.0 | Sports | 10.0    |
| Alan         | 1995-05-25 17:00:00.0 | Sports | 20.0    |
+-------------------------------------------------------+
```

Example 5-2 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 statment
  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 |
+---------------------------------------+
```

5.7.3 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 5-3 PgqlDropExample1.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 PgqlDropExample1
{

  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 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 a PgqlStatement
      pgqlStmt = pgqlConn.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();
      }
    }
  }
}
```

5.7.4 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 20c, 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 5-18.)

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,
```

5.7.4.1 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 5-4 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, password);

      // Create a property graph
      opg = OraclePropertyGraph.getInstance(oracle, graph);
```

```
      // Clear any existing data
      opg.clearRepository();

      // Load data from opv and ope files
      OraclePropertyGraphDataLoader opgLoader =
OraclePropertyGraphDataLoader.getInstance();
      opgLoader.loadData(opg, vertexFile, edgeFile, 1);

      System.out.println("Vertices loaded:" + opg.countVertices());
      System.out.println("Edges loaded:" + opg.countEdges());

    }
    finally {
      // close the property graph
      if (opg != null) {
        opg.close();
      }
      // close oracle
      if (oracle != null) {
        oracle.dispose();
      }
    }
  }
}
```

`GraphLoaderExample.java` gives the following output for `test_graph`.

```
Vertices loaded:4
Edges loaded:17
```

Example 5-5 PgqlExample1.java

`PgqlExample1.java` executes a PGQL query and prints the query result. `PgqlConnection` is used to obtain a `PgqlStatement`. Next, it calls the `executeQuery` method of `PgqlStatement`, which returns a `PgqlResultSet` object. `PgqlResultSet` provides a `print()` method, which shows results in a tabular mode.

The `PgqlResultSet` and `PgqlStatement` objects should be closed after consuming the query result.

```
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 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 =
        "SELECT v.\"fname\" AS fname, v.\"lname\" AS lname, v.\"mval\"
AS mval "+
        "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 5-6 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 fiished 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 5-7 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);

      // Create a PgqlStatement
```

```
      ps = pgqlConn.createStatement();

      // Execute query to get a ResultSet object
      String pgql =
        "SELECT v.\"fname\" AS \"fname\", COUNT(v2) AS \"friendCnt\" "+
        "FROM MATCH (v)-[e:\"friendOf\"]->(v2) "+
        "GROUP BY v "+
        "ORDER BY \"friendCnt\" DESC";
      rs = ps.executeQuery(pgql, "");

      // Print 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();
      }
    }
  }
}
```

`PgqlExample3.java` gives the following output for `test_graph` (which can be loaded using `GraphLoaderExample.java` code).

```
+------------------+
| fname | friendCnt |
+------------------+
| John  | 2         |
| Bill  | 1         |
| Ray   | 1         |
| Susan | 1         |
+------------------+
```

Example 5-8 PgqlExample4.java

`PgqlExample4.java` shows a PGQL path query.

```
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 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();
      }
```

```
      // close the statement
      if (ps != null) {
        ps.close();
      }
      // close the connection
      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

`PgqlExample4.java` gives the following output for `test_graph`(which can be loaded using `GraphLoaderExample.java` code).

```
+--------+
| FRIEND |
+--------+
| Susan  |
| Bill   |
| Ray    |
+--------+
```

5.7.4.2 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

5.7.4.2.1 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 5-9 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 5-10 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   |
+----------------+
```

5.7.4.2.2 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), "");
```

5.7.4.3 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 5-11 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  |
+----------------------+
```

5.7.4.4 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 5-12 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",
T0$0.SVID AS "V1$ID",
T0$1.T AS "FNAME1$T",
T0$1.V AS "FNAME1$V",
T0$1.VN AS "FNAME1$VN",
T0$1.VT AS "FNAME1$VT",
n'E' AS "E$IT",
T0$0.EID AS "E$ID",
T0$0.T AS "SINCE$T",
T0$0.V AS "SINCE$V",
T0$0.VN AS "SINCE$VN",
T0$0.VT AS "SINCE$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(+)
) 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'fname' ) R
  WHERE L.VID = R.VID(+)
) T0$1
WHERE T0$0.SVID=T0$1.VID AND
(T0$0.EL = n'knows' AND T0$0.EL IS NOT NULL)
```

Example 5-13 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 --
```

5.7.4.5 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 5-2 PGQL Translation and Execution Options

Option	Default	Explict Argument	Options Flag	JVM Argument
Degree of parallelism	0	parallel	none	none
Timeout	unlimited	timeout	none	none

Table 5-2 (Cont.) PGQL Translation and Execution Options

Option	Default	Explict Argument	Options Flag	JVM Argument
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

5.7.4.5.1 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 5-14 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 "+
```

```
        "FROM MATCH (v1)-[:\"friendOf\"]->(v2)";
      rs = ps.executeQuery(pgql /* query string */,
                           100  /* timeout (sec): 0 is default and
implies no timeout */,
                           2    /* parallel: 1 is default */,
                           6    /* dynamic sampling: 2 is default */,
                           50   /* max results: -1 is default and
implies no limit */,
                           ""   /* options */);

      // 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();
      }
    }
  }
}
```

`PgqlExample10.java` gives the following output for `test_graph` (which can be loaded using GraphLoaderExample.java code).

```
+-----------------+
| FNAME1 | FNAME2 |
+-----------------+
| Ray    | Susan  |
| John   | Susan  |
| Bill   | John   |
| Susan  | John   |
| John   | Bill   |
+-----------------+
```

5.7.4.5.2 Using the GT$ Skeleton Table

The property graph relational schema defines a GT$ skeleton table that stores a single row for each edge in the graph, no matter how many properties an edge has. This skeleton table is populated by default so that PGQL query execution can take advantage of the GT$ table and avoid sorting operations on the GE$ table in many cases, which gives a significant performance improvement.

You can add `"USE_GT_TAB=F"` to the `options` argument of `executeQuery` and `translateQuery` or use `-Doracle.pg.rdbms.pgql.useGtTab=false` in the Java command line to turn off GT$ table usage.

Example 5-15 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)";
```

```
      // Get the SQL translation with GT table
      PgqlSqlQueryTrans sqlTrans = ps.translateQuery(pgql,"");

      // Print SQL translation
      System.out.println("-- SQL Translation with GT Table
----------------------");
      System.out.println(sqlTrans.getSqlTranslation());

      // Get the SQL translation without GT table
      sqlTrans = ps.translateQuery(pgql,"USE_GT_TAB=F");

      // Print SQL translation
      System.out.println("-- SQL Translation without GT Table
-------------------------");
      System.out.println(sqlTrans.getSqlTranslation());

    }
    finally {
      // close the statement
      if (ps != null) {
        ps.close();
      }
      // close the connection
      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

`PgqlExample11.java` gives the following output for `test_graph` (which can be loaded using `GraphLoaderExample.java` code).

```
-- SQL Translation with GT Table ----------------------
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)$VT"
FROM "SCOTT".TEST_GRAPHGT$ T0$0
-- SQL Translation without GT Table -------------------------
SELECT 7 AS "id(v1)$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)$VT"
FROM (SELECT DISTINCT EID, SVID, DVID,EL FROM "SCOTT".TEST_GRAPHGE$) T0$0
```

5.7.4.5.3 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 5-16 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 */,
                          " USE_DIST_RW=T " /* options */);
      System.out.println("-- DISTINCT RW Path Translation
--------------------");
      System.out.println(sqlTrans.getSqlTranslation()+"\n");

      // get SQL translation with CONNECT BY
      sqlTrans =
        ps.translateQuery(pgql /* query string */,
                          2    /* parallel: default is 1 */,
                          2    /* dynamic sampling: default is 2 */,
                          -1   /* max results: -1 implies no limit */,
                          " USE_RW=F " /* options */);
      System.out.println("-- CONNECT BY Path Translation
--------------------");
      System.out.println(sqlTrans.getSqlTranslation()+"\n");
    }
    finally {
      // close the statement
      if (ps != null) {
        ps.close();
      }
      // close the connection
      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

`PgqlExample12.java` gives the following output for `test_graph` (which can be loaded using `GraphLoaderExample.java` code).

```
-- Default 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 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 5-17 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   |
+-----------------+
```

5.7.4.5.4 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 5-18 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);
        System.out.println(v.toString());
      }
      // 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();
      }
      // close the property graph
      if (opg != null) {
        opg.close();
      }
      // close oracle
      if (oracle != null) {
        oracle.dispose();
      }
    }
  }
}
```

The output for `PgqlExample14.java` (which can be loaded using `GraphLoaderExample.java` code) is:

```
-- Results for edge query -----------------
+------------------+
| id(e) | label(e) |
+------------------+
| 6     | knows    |
| 11    | knows    |
| 10    | knows    |
| 5     | knows    |
| 4     | knows    |
| 13    | knows    |
| 9     | knows    |
| 12    | knows    |
| 8     | knows    |
| 7     | knows    |
| 14    | knows    |
| 15    | knows    |
+------------------+
-- Vertex objects retrieved from vertex query --
Vertex ID 3 [NULL] {}
Vertex ID 0 [NULL] {}
Vertex ID 2 [NULL] {}
Vertex ID 1 [NULL] {}
-- Vertex objects retrieved from vertex query with setPartial(true) --
Vertex ID 3 [NULL] {bval:bol:false, fname:str:Susan, lname:str:Blue,
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}
```

5.7.4.6 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 5-19 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 |
+---------------+
```

5.7.4.7 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 5-20 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$ |
-------------------------------------------------------------------------
```

5.7.5 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 5-21 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
(e3.\"_ora_id\" = 3, e3.since = DATE '1999-01-03') ";
      ps.execute(pgql, /*  query string  */
                  "", /* query options  */
                  ""  /* modify options */);

      // Execute a query to verify insertion
      pgql =
          "  SELECT id(p1) AS id1, p1.fname AS person1, id(p2) as id2,
p2.fname AS person2, id(e) as e, e.since "+
          "    FROM MATCH (p1)-[e:knows]->(p2) "+
          "ORDER BY id1, id2";
      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 `PgqlExample17.java` is:

```
+-----------------------------------------------------------+
| ID1 | PERSON1 | ID2 | PERSON2 | E | SINCE                 |
+-----------------------------------------------------------+
| 1   | Jake    | 2   | Amy     | 1 | 2003-04-20 17:00:00.0 |
| 1   | Jake    | 3   | Erik    | 2 | 2010-02-09 16:00:00.0 |
| 3   | Erik    | 4   | Jane    | 3 | 1999-01-02 16:00:00.0 |
+-----------------------------------------------------------+
```

For more examples of INSERT statement, see the relevant section of the PGQL specification here.

Example 5-22 PgqlExample18.java (Update)

`PgqlExample18.java` updates several properties of vertices and edges that are matched in the FROM clause of an UPDATE statement.

```
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 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 =
          "SELECT p1.fname AS fname1, p1.lname AS lname1, p1.age AS
age1, "+
          "       p2.fname AS fname2, p2.lname AS lname2, p2.age AS
age2, e.strength "+
          "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 `PgqlExample18.java` applied on a graph where `PgqlExample17.java` has been previously executed is:

```
+-------------------------------------------------------------------+
| FNAME1 | LNAME1 | AGE1   | FNAME2 | LNAME2 | AGE2   | STRENGTH |
+-------------------------------------------------------------------+
| Jake   | Red    | 47     | Amy    | White  | 29     | 100      |
| Jake   | Red    | 47     | Erik   | <null> | <null> | <null>   |
| Erik   | <null> | <null> | Jane   | <null> | <null> | <null>   |
+-------------------------------------------------------------------+
```

For more examples of UPDATE statement, see the relevant section of the PGQL specification here.

Example 5-23 PgqlExample19.java (Delete)

`PgqlExample19.java` deletes edges that are matched in the FROM clause of a DELETE statement.

```
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 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 5-24 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

5.7.5.1 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 5-3 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

5.7.5.1.1 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 5-25 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();

      // Delete all the edges in the graph
      String pgql =
        "DELETE e "+
        "  FROM MATCH () -[e]-> ()";
      ps.execute(pgql, /*  query string  */
                 "",   /* query options  */
                 "AUTO_COMMIT=F"  /* modify options */);

      // Execute a query to verify deletion
      pgql =
          "SELECT COUNT(e) "+
          "  FROM MATCH () -[e]-> ()";
      rs = ps.executeQuery(pgql, "");

      // Print the results
      System.out.println("Number of edges after deletion:");
```

```
      rs.print();
      rs.close();

      // Rollback the changes. This is possible because
      // AUTO_COMMIT=F flag was used in execute
      conn.rollback();

      // Execute a query to verify rollback
      pgql =
          "SELECT COUNT(e) "+
          "  FROM MATCH () -[e]-> ()";
      rs = ps.executeQuery(pgql, "");

      // Print the results
      System.out.println("Number of edges after rollback:");
      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();
      }
    }
  }
}
```

`PgqlExample21.java` gives the following output for a graph with one edge:

```
Number of edges after deletion:
+----------+
| COUNT(e) |
+----------+
| 0        |
+----------+
Number of edges after rollback:
+----------+
| COUNT(e) |
+----------+
| 1        |
+----------+
```

5.7.5.1.2 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` of 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 5-26 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();

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();
```

```
      // Delete all the vertices with output edges
      // This will throw an error
      String pgql =
        "DELETE v "+
        "  FROM MATCH (v) -[e]-> ()";
      ps.execute(pgql, /*  query string  */
                 "",   /* query options  */
                 "DELETE_CASCADE=F"  /* modify options */);
    }
    catch (PgqlToSqlException ex){
      System.out.println("Error in execution: " + ex.getMessage());
    }
    finally {
      // close the statement
      if (ps != null) {
        ps.close();
      }
      // close the connection
      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

`PgqlExample22.java` gives the following output for a graph with at least one edge:

```
Error in execution: Attempting to delete vertices with incoming/outgoing edges.
Drop edges first or turn on DELETE_CASCADE option
```

5.7.6 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
Graph Visualization Application (GraphViz)

The Graph Visualization application (GraphViz) enables interactive exploration and visualization of property graphs.

- About the Graph Visualization Application (GraphViz)
  GraphViz is a single-page web application that works with the in-memory graph analytics server.
- Deploying GraphViz
  For deploying GraphViz, $PG_HOME should be set and should point to the directory: `/opt/oracle/graph`
- Using GraphViz
  The principal points of entry for the GraphViz application are the query editor and the graph lists.

6.1 About the Graph Visualization Application (GraphViz)

GraphViz is a single-page web application that works with the in-memory graph analytics server.

The in-memory graph analytics server can be deployed in embedded mode or in Apache Tomcat or Oracle Weblogic Server. GraphViz 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.

GraphViz works with the in-memory analytics server. It can visualize graphs that are 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.

6.2 Deploying GraphViz

For deploying GraphViz, $PG_HOME should be set and should point to the directory: `/opt/oracle/graph`

- Deploying GraphViz for Demo and Test Environments
  Use this approach only in a trusted environment to test or demonstrate GraphViz.
- Deploying GraphViz in Oracle WebLogic Server
  The following instructions are for deploying GraphViz in Oracle WebLogic Server 12.2.1.x.

6.2.1 Deploying GraphViz for Demo and Test Environments

Use this approach only in a trusted environment to test or demonstrate GraphViz.

> **Note:**
>
> Disabling PGX authentication will allow anyone to connect to PGX and open the visualization UI without any authentication. For a secure deployment with authentication enabled, deploy in Oracle WebLogic Server.

1. Set `enable_tls` to `false` in `/etc/oracle/graph/server.conf`.
2. List the graphs to preload when starting the in-memory analytics server by listing their configuration files in `/etc/oracle/graph/pgx.conf`. For example:

```
"preload_graphs": [{
        "name": "my-graph",
        "path": "/scratch/data/graphs/my-graph.json"
      }]
```

3. Start the server: `/opt/oracle/graph/pgx/bin/start-server`
4. Open `http://localhost:7007/ui` in your browser.

6.2.2 Deploying GraphViz in Oracle WebLogic Server

The following instructions are for deploying GraphViz in Oracle WebLogic Server 12.2.1.x.

1. Start WebLogic Server.

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

2. Enable tunneling.
   In order to be able to deploy the GraphViz 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**.
3. Create groups and users that can access the GraphViz user interface. Go to the admin console (by default on `http://localhost:7001/console`), select **Security Realms** (left panel) > myrealm (main panel) > **Users and Groups** (top panel); and create a new group `GraphAnalysts` with at least one user. Any user that is part of the GraphAnalysts group will be able to access the graph visualization application.
4. Configure the GraphViz deployment descriptor.
   Modify the `WEB-INF/web.xml` file inside of the GraphViz WAR file. After you install the Oracle Graph Server package, the WAR file is located at: `/opt/oracle/graph/graphviz/pgviz-webapp-<<version>>.war`
   - Extract the WAR file in order to directly modify the file contents. For example, to extract:

```
unzip /opt/oracle/graph/graphviz/pgviz-webapp-*.war -d /tmp/
pgviz/
```

- Edit the `web.xml` descriptor using any file editor of your choice. For example:

  ```
  nano /tmp/pgviz/WEB-INF/web.xml
  ```

To configure GraphViz to communicate with the secure PGX deployment:

a. Locate the `graphviz.driver.class` context parameter. If applicable, set the value to `oracle.pgx.graphviz.driver.PgxDriver`. For example:

```
<context-param>
    <param-name>graphviz.driver.class</param-name>
    <param-value>oracle.pgx.graphviz.driver.PgxDriver</param-
value>
</context-param>
```

b. Locate the `pgx.base_url` context parameter. Modify the value to match your secure PGX deployment endpoint. Use the correct FQDN or IP address, along with the correct port. Be sure to specify **`https`** instead of `http` as the protocol. Example of the format:

```
<context-param>
    <param-name>pgx.base_url</param-name>
    <param-value>https://<<fqdn-or-ip>>:<<port>></param-value>
</context-param>
```

To configure GraphViz to communicate with PGQL on Oracle Database

a. Create a JDBC data source in Weblogic Server. (See the Weblogic Server documentation for the configuration steps: https://docs.oracle.com/en/middleware/fusion-middleware/weblogic-server/12.2.1.4/jdbca/jdbc_datasources.html.)

b. Locate the `graphviz.driver.class` context parameter. If applicable, set the value to `oracle.pg.rdbms.PgqlDriver`. For example:

```
<context-param>
    <param-name>graphviz.driver.class</param-name>
    <param-value>oracle.pg.rdbms.PgqlDriver</param-value>
</context-param>
```

c. Set the context parameter `graphviz.driver.rdbms.datasource_id` referencing the name of the data source you created as value. For example:

```
<context-param>
    <param-name>graphviz.driver.rdbms.datasource_id</param-name>
    <param-value>myContainerDataSource</param-value>
</context-param>
```

Replace `myContainerDataSource` with the name of the WebLogic Server JDBC data source you created in step a.

5. Configure web application authentication.

Configure web application authentication. Enable the security constraints by adding the following lines to the `<web-app>` element in the `web.xml` configuration file (the same file as in the preceding step).

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>freePages</web-resource-name>
        <url-pattern>/login/*</url-pattern>
        <url-pattern>/login/css/*</url-pattern>
        <url-pattern>/login/img/*</url-pattern>
        <url-pattern>/error_pages/*</url-pattern>
    </web-resource-collection>
</security-constraint>
<security-constraint>
    <display-name>SecurityCheck</display-name>
    <web-resource-collection>
        <web-resource-name>SecurityCheck</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>analysts</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>myrealm</realm-name>
    <form-login-config>
        <form-login-page>/login/login.html</form-login-page>
        <form-error-page>/login/login_failed.html</form-error-page>
    </form-login-config>
</login-config>
<security-role>
    <role-name>analysts</role-name>
</security-role>
<error-page>
    <error-code>403</error-code>
    <location>/login/login_failed.html</location>
</error-page>
```

In the `/tmp/pgviz/WEB-INF/weblogic.xml` descriptor file, map the app security configuration role to the Weblogic group you created in step 3. You can specify any user or group of the default security realm with the `<principal-name>` tag.

```
<security-role-assignment>
    <role-name>webuser</role-name>
    <principal-name>myGroup</principal-name>
</security-role-assignment>
```

6. (**Optional**) Register the PGX certificates with WebLogic Server. **Note: this step is only necessary if you want to use GraphViz to communicate with a PGX server.**Ensure that you have the following:

   - An HTTPS endpoint of securely deployed PGX server (using mutual TLS)
   - Access to authorized client certificate and truststore (in a password protected JKS format) to authenticate with PGX

   Assuming that the client certificate identifying Weblogic as a trusted PGX client is already added to a keystore, you need to configure WebLogic Server to use your keystore when making outbound HTTPS connections to PGX. In addition, you need to configure Weblogic Server to trust the PGX server certificate if it is not signed by an authority that is trusted by WebLogic Server by default.

   Go to the admin console (by default on `http://localhost:7001/console`), then select **Environment** (left panel) > **Servers** (left panel). On the main panel, click on the server that will run the graph visualization application. Then select the Server Start" tab in the second row of the "Configuration" tab. In the "Arguments" text area, specify the following system properties:

   ```
   -Djavax.net.ssl.trustStore=<<path-to-truststore>>
         -Djavax.net.ssl.keyStore=<<path-to-keystore>>
         -Djavax.net.ssl.keyStorePassword=<<keystore-password>>
   ```

   Where:

   a. `<<path-to-truststore>>` is your local file path to the trust store keystore (JKS format) file which contains the necessary certificates to establish trust with the PGX server.

   b. `<<path-to-truststore>><<path-to-keystore>>` is your local file path to the keystore (JKS format) file which contains the client certificate for Weblogic Server trusted by PGX.

   c. `<<keystore-password>>`is the keystore password for the keystore file.

   Then click **Save**.

7. Repackage the WAR file.
   Because you decompressed the WAR file to edit the `web.xml` and `weblogic.xml` files, repackage it before deploying the application.

   ```
   # create a backup of the original file
   mv /opt/oracle/graph/graphviz/pgviz-webapp-<<version>>.war ~/pgviz-
   webapp-<<version>>.war.bkp
   cd /tmp/pgviz/
   jar -cvf $PG_HOME/pgviz/pgviz-webapp-<<version>>.war *
   ```

8. Deploy the repackaged WAR file.
   To deploy the repackaged 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 /opt/
   oracle/graph/graphviz/pgviz-webapp-<<version>>.war
   ```

To undeploy again, you can use the following command:

```
java weblogic.Deployer -adminurl <<admin-console-url>> -username
<<admin-user>> -password <<admin-password>> -name /opt/oracle/graph/
graphviz/pgviz-webapp-<<version>>.war -undeploy
```

9. Enable SSL for inbound Weblogic Server connections. (See the Weblogic Server documentation for how to enable encyrpted connections.)
10. Test the deployment.
    To test the deployment, navigate in your browser to: `https://<<fqdn-ip>>:<<port>>/ui`

    The browser should prompt for user and password. After login, the GraphViz user interface (UI) will be displayed and the graphs from PGX will be retrieved.

    - If the authentication fails or if nothing is returned, there is probably a problem with the security realm. Check that the realm exists, as well as the user or group.
    - If the UI is loaded but the call to retrieve the graphs fails, there is probably an error with the certificates. List the certificates stored in the truststore, and make sure there is a signed certificate for PGX.

6.3 Using GraphViz

The principal points of entry for the GraphViz application are the query editor and the graph lists.

When you start GraphViz, 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 GraphViz.)

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 6-1 Query Visualization

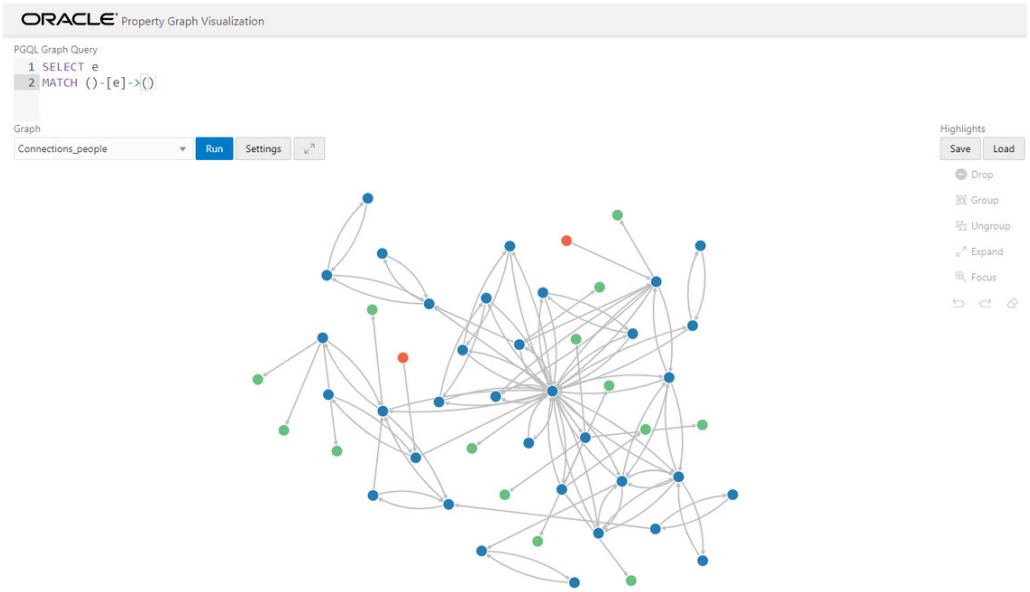
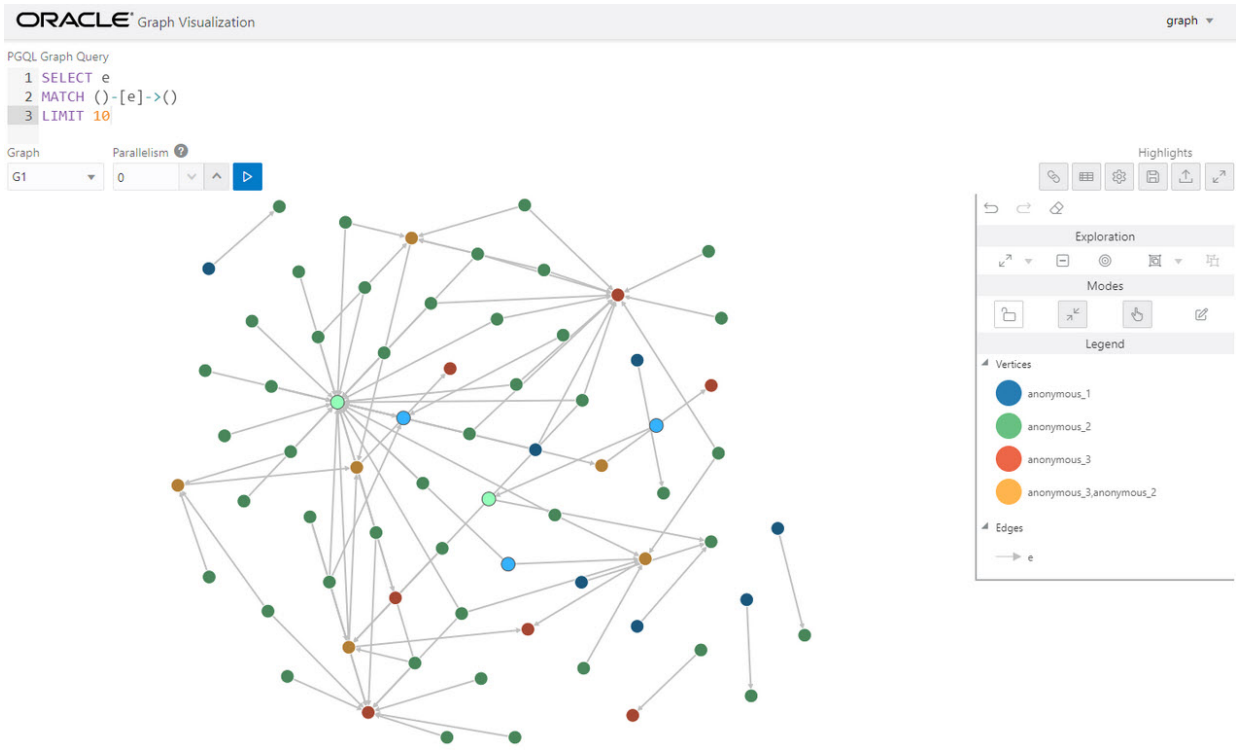


Figure 6-2 Query Visualization



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.

- GraphViz Degree of Parallelism
  The Parallelism input field lets you customize the degree of parallelism that is being used to execute the given query.
- GraphViz Modes
  The buttons on the right let you switch between two modes: Graph Manipulation and Zoom/Move.
- GraphViz Settings
  You can click the **Settings** gear icon to display the GraphViz settings window.
- 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 GraphViz
  You can provide GraphViz input data through URL parameters instead of using the form fields of the user interface.

6.3.1 GraphViz Degree of Parallelism

The Parallelism input field lets you customize the degree of parallelism that is being used to execute the given query.

By default, it is set to 0 (zero), which means the default value of the underlying query execution engine is being used. If you specify a value greater than 0, the underlying query execution engine uses your specified degree of parallelism.

However, if the underlying engine is in-memory graph server (PGX), the PGX server must be configured to have the enterprise scheduler enabled; otherwise, parallelism values greater than 0 will result in an error.

6.3.2 GraphViz 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.

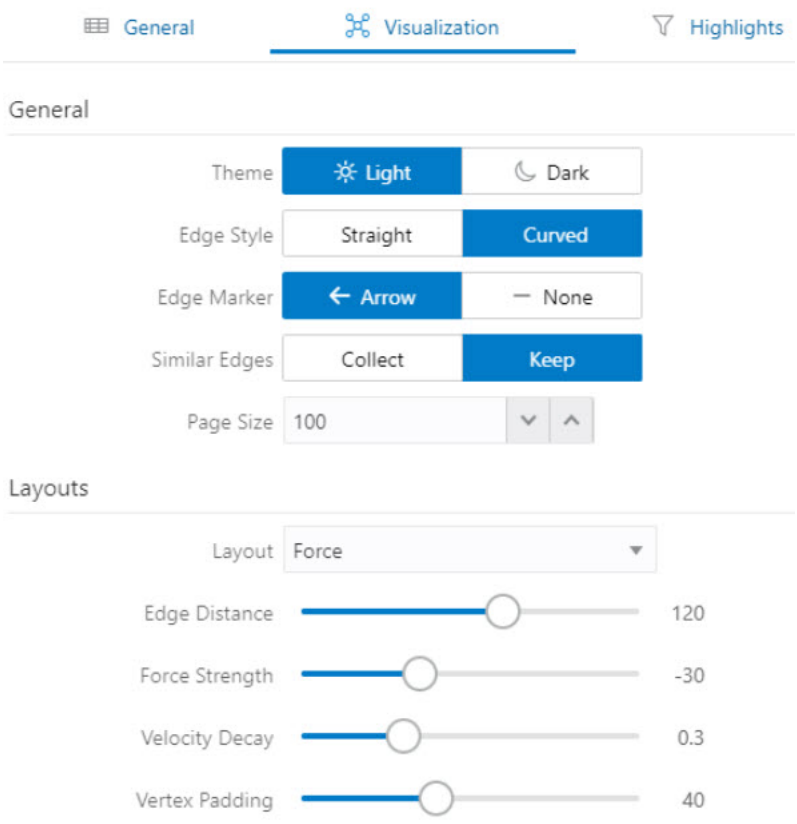
An additional mode, called **Sticky** mode, lets you cancel the action of dragging the nodes around.

6.3.3 GraphViz Settings

You can click the **Settings** gear icon to display the GraphViz 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 6-3 GraphViz Settings Window



The **General tab** includes the following:

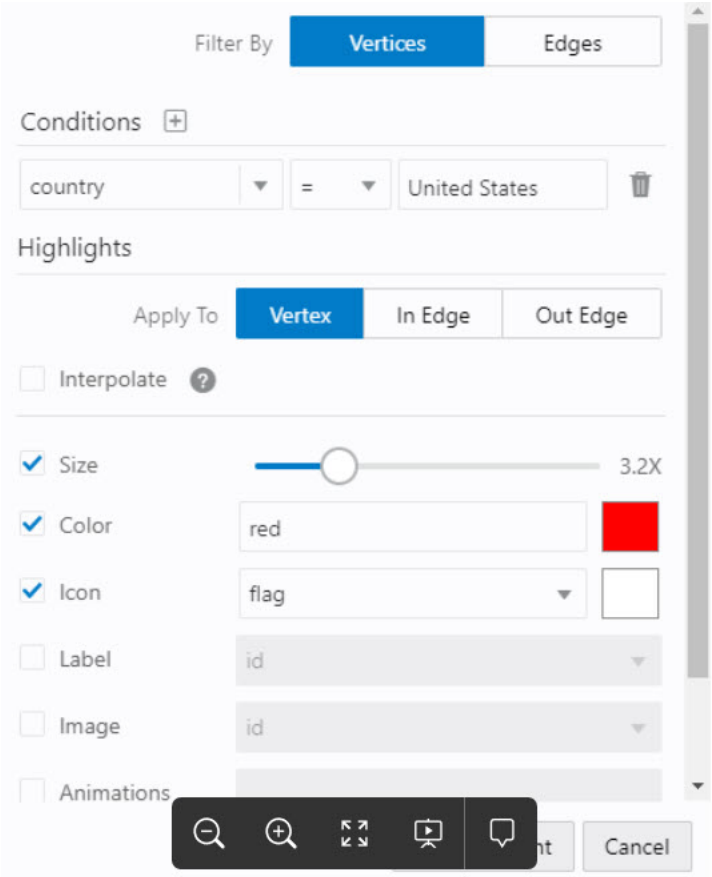
- **Number of hops**: The configurable number of hops for the expand and focus actions.
- **Truncate label**: Truncates the label if it exceeds the maximum length.
- **Max. visible label length**: Maximum length before truncating.
- **Show Label On Hover**: Controls whether the label is shown on hover.
- **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 6-4 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.

6.3.4 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.

6.3.5 Using URL Parameters to Control GraphViz

You can provide GraphViz input data through URL parameters instead of using the form fields of the user interface.

If you supply the parameters in the URL, the GraphViz 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 6-1 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

Spatial Support in Property Graphs

The property graph support in the Oracle Spatial and Graph option is integrated with the spatial support.

The integration has the following aspects: representing spatial data in a property Graph, creating a spatial index on that spatial data, and querying that spatial data.

7.1 Representing Spatial Data in a Property Graph

Spatial data can be used as values of vertex properties and edge properties.

For example, an entity can have a point (longitude/latitude) as the value of a property named *location*. As another example, an edge may have a polygon as the value of a property, and this property can represent the location at which this link (relationship) was established.

The following shows some example syntax for encoding spatial data in a property graph.

- Point: `'-122.230 37.560'`
- Point: `'POINT(-122.241 37.567)'`
- Point with SRID specified: `'srid/8307 POINT(-122.246 37.572)'`
- Polygon: `'POLYGON((-83.6  34.1, -83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))'`
- Polygon with SRID specified: `'srid/8307 POLYGON((-83.6 34.1, -83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6 34.1))'`
- Line string: `'LINESTRING (30 10, 10 30, 40 40)'`
- Multiline string: `'MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))'`

Assume a test property graph named `test`. The following statements add a set of vertices with coordinates (longitude and latitude) spacified for each.

```
insert into testVT$(vid, k, t, v) values(100, 'geoloc', 20, '-122.230
37.560');
insert into testVT$(vid, k, t, v) values(101, 'geoloc', 20, '-122.231
37.561');
insert into testVT$(vid, k, t, v) values(102, 'geoloc', 20, '-122.236
37.562914');
insert into testVT$(vid, k, t, v) values(103, 'geoloc', 20, '-122.241
37.567');
```

```
insert into testVT$(vid, k, t, v) values(104, 'geoloc', 20, '-122.246
37.572');
insert into testVT$(vid, k, t, v) values(105, 'geoloc', 20, '-122.251
37.577');
insert into testVT$(vid, k, t, v) values(200, 'geoloc', 20, '-122.256
37.582');
insert into testVT$(vid, k, t, v) values(201, 'geoloc', 20, '-122.261
37.587');
```

The Spatial data in the property graph can be used to construct SDO_GEOMETRY objects. For example, the OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS function can be used to read spatial data from the V column for all T of a specified value (such as 20), and return SDO_GEOMETRY objects. This function attempts to parse the value as coordinates if the value appears to be two numbers, and it uses the SDO_GEOMETRY constructor if the value is not a simple point. Finally, if a SRID is provided, it uses the SDO_CS_TRANSFORM procedure to transform using the given coordinate system.

The following example uses the OPG_APIS.GET_GEOMETRY_FROM_V_T_COLS function to get geometries from the `test` property graph. It includes some of the output.

```
SQL> select vid, k, opg_apis.get_geometry_from_v_t_cols
      from testVT$
        order by vid, k;
   . . .
       100 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.23,
37.56, NULL), NULL, NULL)
       101 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.231,
37.561, NULL), NULL, NULL)
       102 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.236,
37.562914, NULL), NULL, NULL)
       103 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.241,
37.567, NULL), NULL, NULL)
   . . .
```

You can generate SDO_GEOMETRY objects from WKT literals. The following example inserts WKT literals, and then uses the OPG_APIS.GET_WKTGEOMETRY_FROM_V_T_COLS function to construct SDO_GEOMETRY objects from the V, T columns.

```
truncate table testGE$;
truncate table testVT$;
insert into testVT$(vid, k, t, v) values(101, 'geoloc', 20,
'POLYGON((-83.6  34.1, -83.6 34.3, -83.4 34.3, -83.4 34.1, -83.6
34.1))');
insert into testVT$(vid, k, t, v) values(103, 'geoloc', 20,
'POINT(-122.241 37.567)');
insert into testVT$(vid, k, t, v) values(105, 'geoloc', 20,
'POINT(-122.251 37.577)');
insert into testVT$(vid, k, t, v) values(200, 'geoloc', 20,
'MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30
10))');
insert into testVT$(vid, k, t, v) values(201, 'geoloc', 20, 'LINESTRING
```

```
(30 10, 10 30, 40 40)');

prompt show the geometry info
SQL> select vid, k, opg_apis.get_wktgeometry_from_v_t_cols(v,t)
       from testVT$
      order by vid, k;
   . . .
       101 geoloc SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1,
1003, 1), SDO_ORDINATE_ARRAY(-83.6, 34.1, -83.6, 34.3, -83.4, 34.3,
-83.4, 34.1, -83.6, 34.1))
       103 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.241,
37.567, NULL), NULL, NULL)
       105 geoloc SDO_GEOMETRY(2001, 8307, SDO_POINT_TYPE(-122.251,
37.577, NULL), NULL, NULL)
       200 geoloc SDO_GEOMETRY(2006, 8307, NULL, SDO_ELEM_INFO_ARRAY(1,
2, 1, 7, 2, 1), SDO_ORDINATE_ARRAY(10, 10, 20, 20, 10, 40, 40, 40, 30,
30, 40, 20, 30, 10))
       201 geoloc SDO_GEOMETRY(2002, 8307, NULL, SDO_ELEM_INFO_ARRAY(1,
2, 1), SDO_ORDINATE_ARRAY(30, 10, 10, 30, 40, 40))
```

7.2 Creating a Spatial Index on Property Graph Data

After adding spatial data to a property graph, you can use OPG_APIS package subprograms to construct SDO_GEOMETRY objects, and then you can create a function-based spatial index on the vertices (VT$) or the edges (VT$) table.

Using the example property graph named `test`, the following statements add the necessary metadata and create a function-based spatial index.

```
SQL> -- In the schema that owns the property graph TEST:
SQL> --
SQL> insert into user_sdo_geom_metadata values('TESTVT$',
 'mdsys.opg_apis.get_geometry_from_v_t_cols(v,t)',
 sdo_dim_array(
   sdo_dim_element('Longitude', -180, 180, 0.005),
   sdo_dim_element('Latitude', -90, 90, 0.005)), 8307);

commit;

SQL> -- Create a function-based spatial index
SQL> create index testVTXGEO$
       on testVT$(mdsys.opg_apis.get_geometry_from_v_t_cols(v, t))
       indextype is mdsys.spatial_index_v2
       parameters ('tablespace=USERS')
       parallel 4
       local;
```

(To create a spatial index on your own property graph, replace the graph name `test` with the name of your graph.)

If the WKT literals are used in the V
column, then replace *mdsys.opg_apis.get_**geometry**_from_v_t_cols* with *mdsys.opg_apis.get_**wktgeometry**_from_v_t_cols* in the preceding two SQL statements.

Note that the preceding SQL spatial index creation steps are wrapped in convenient Java methods in the `OraclePropertyGraph` class defined in the `oracle.pg.rdbms` package:

```
  /**
   * This API creates a default Spatial index on edges. It assumes that
   * the mdsys.opg_apis.get_geometry_from_v_t_cols(v,t) PL/SQL is going to be
used
   * to create a function-based Spatial index. In addition, it adds a predefined
   * value into user_sdo_geom_metadata. To customize, please refer to the dev
   * guide for adding a row to user_sdo_geom_metadata and then creating a
   * Spatial index manually.
   * Note that, a DDL will be executed so expect an implict commit. If you
   * have changes that do not want to be persisted, run a rollback before calling
   * this method.
   * @param dop degree of parallelism used to create the Spatial index
   */
  public void createDefaultSpatialIndexOnEdges(int dop);

  /**
   * This API creates a default Spatial index on vertices. It assumes that
   * the mdsys.opg_apis.get_geometry_from_v_t_cols(v,t) PL/SQL is going to be
used
   * to create a function-based Spatial index. In addition, it adds a predefined
   * value into user_sdo_geom_metadata. To customize, please refer to the dev
   * guide for adding a row to user_sdo_geom_metadata and then creating a
   * Spatial index manually.
   * Note that a DDL will be executed so expect an implict commit. If you
   * have changes that do not want to be persisted, run a rollback before calling
   * this method.
   * @param dop degree of parallelism used to create the Spatial index
   */
  public void createDefaultSpatialIndexOnVertices(int dop);
```

7.3 Querying Spatial Data in a Property Graph

Oracle Spatial and Graph geospatial query functions can be applied to spatial data in a property graph. This topic provides some examples.

Note that a query based on spatial information can be combined with navigation and pattern matching.

The following example finds entities (vertices) that are within a specified distance (here, 1 mile) of a location (point geometry).

```
SQL> -- use SDO_WITHIN_DISTANCE to filter vertices
SQL>  select vid, k, t, v
        from testvt$
       where
sdo_within_distance(mdsys.opg_apis.get_geometry_from_v_t_cols(v, t),
            mdsys.sdo_geometry(2001, 8307,
mdsys.sdo_point_type(-122.23, 37.56, null), null, null),
            'distance=1 unit=mile') = 'TRUE'
       order by vid, k;
```

The output and execution plan may include the following. Notice that a newly created domain indexTESTVTXGEO$ is used in the execution.

```
       100 geoloc       20  -122.230 37.560
       101 geoloc       20  -122.231 37.561
       ..  ...              ...

-------------------------------------------------------------------------------
-------------------------------------------------------------------
| Id  | Operation                               | Name        | Rows  | Bytes |
Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |             |     1 | 18176
|     2  (50)| 00:00:01 |       |       |        |      |            |
|   1 |  PX COORDINATOR                         |             |       |
|           |          |       |       |        |      |            |
|   2 |   PX SEND QC (ORDER)                    | :TQ10001    |     1 | 18176
|     2  (50)| 00:00:01 |       |       |  Q1,01 | P->S | QC (ORDER) |
|   3 |    SORT ORDER BY                        |             |     1 | 18176
|     2  (50)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE                          |             |     1 | 18176
|     1   (0)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   5 |      PX SEND RANGE                      | :TQ10000    |     1 | 18176
|     1   (0)| 00:00:01 |       |       |  Q1,00 | P->P | RANGE      |
|   6 |       PX PARTITION HASH ALL             |             |     1 | 18176
|     1   (0)| 00:00:01 |     1 |     8 |  Q1,00 | PCWC |            |
|*  7 |        TABLE ACCESS BY LOCAL INDEX ROWID| TESTVT$     |     1 | 18176
|     1   (0)| 00:00:01 |     1 |     8 |  Q1,00 | PCWP |            |
|*  8 |         DOMAIN INDEX (SEL: 0.000000 %)  | TESTVTXGEO$ |       |
|     1   (0)| 00:00:01 |       |       |  Q1,00 |      |            |
-------------------------------------------------------------------------------
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   7 - filter(INTERNAL_FUNCTION("K") AND INTERNAL_FUNCTION("V"))
   8 -
access("MDSYS"."SDO_WITHIN_DISTANCE"("OPG_APIS"."GET_GEOMETRY_FROM_V_T_COLS"("V",
"T"),"MDSYS"."SDO_GEOMETRY"(2001,8307,"MDSYS"."SDO_P
             OINT_TYPE"((-122.23),37.56,NULL),NULL,NULL),'distance=1
unit=mile')='TRUE')
```

The following example sorts entities (vertices) based on their distance from a location.

```
-- Sort based on distance in miles
SQL> select vid, dist from (
       select vid, k, t, v,

sdo_geom.sdo_distance(mdsys.opg_apis.get_geometry_from_v_t_cols(v, t),
              mdsys.sdo_geometry(2001, 8307,
mdsys.sdo_point_type(-122.23, 37.56, null), null, null), 1.0,
'unit=mile') dist
         from testvt$
        where t = 20
      ) order by dist asc
;
```

The output and execution plan may include the following.

```
        ...
       101  .088148935
       102  .385863422
       103  .773127682
       104 1.2068052
       105 1.64421947
       200 2.08301065
       ...

-------------------------------------------------------------------------------
--------------
| Id  | Operation           | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
Pstart| Pstop |
-------------------------------------------------------------------------------
--------------
|   0 | SELECT STATEMENT    |         |     1 | 15062 |  1366   (1)| 00:00:01
|       |       |
|   1 |  SORT ORDER BY      |         |     1 | 15062 |  1366   (1)| 00:00:01
|       |       |
|   2 |   PARTITION HASH ALL|         |     1 | 15062 |  1365   (1)| 00:00:01
|     1 |     8 |
|*  3 |    TABLE ACCESS FULL| TESTVT$ |     1 | 15062 |  1365   (1)| 00:00:01
|     1 |     8 |
-------------------------------------------------------------------------------
--------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("T"=20 AND INTERNAL_FUNCTION("V"))
```

8

OPG_APIS Package Subprograms

The OPG_APIS 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.

8.1 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);
```

8.2 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_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
 opg_apis.cf_prep(edge_tab_name,wt_l,wt_r,wt_l1,wt_r1,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_l1,wt_r1,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_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';
  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_l1,wt_r1,wt_i,wt_ld,wt_rd,tablespace,options);
END;
/
```

8.3 OPG_APIS.CF_CLEANUP

Format

```
OPG_APIS.CF_CLEANUP(
     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

Preforms 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_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.

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 previous 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;
/
```

8.4 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

Preforms 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_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.

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 genenerated 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_l1 varchar2(32);
  wt_r1 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_l1,wt_r1,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_l1 ' || wt_l1);
  dbms_output.put_line(' wt_r1 ' || wt_r1);
  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;
/
```

8.5 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');
```

8.6 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');
```

8.7 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 aleady 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');
```

8.8 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;
/
```

8.9 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;
/
```

8.10 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;
/
```

8.11 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;
/
```

8.12 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);
```

8.13 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' creqtes the schema tables with an INMEMORYclause.
- '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');
```

8.14 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');
```

8.15 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');
```

8.16 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

tblapace
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);
```

8.17 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);
```

8.18 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);
```

8.19 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);
```

8.20 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');
```

8.21 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');
```

8.22 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);
```

8.23 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 `num_sub_ptns*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/(num_sub_ptns*num_sub_ptns)` of the graph; so to estimate the total number of triangles in the graph, you can multiply the result by `num_sub_ptns*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;
/
```

8.24 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 `'SYNOPSES'` (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
```

8.25 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 `'SYNOPSES'` (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
```

8.26 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 umber 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;
/
```

8.27 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);
```

8.28 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;
/
```

8.29 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
```

8.30 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);
```

8.31 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"
```

8.32 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
```

8.33 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))
```

8.34 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))
```

8.35 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)
```

8.36 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)
```

8.37 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)
```

8.38 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)
```

8.39 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)
```

8.40 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
```

8.41 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
```

8.42 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))
```

8.43 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))
```

8.44 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 on e 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
```

8.45 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  BOOLEAN DEFAULT FALSE,
     force              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 `'SYNOPSES'` (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');
```

8.46 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 `'SYNOPSES'` (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');
```

8.47 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 od 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
```

8.48 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);
```

8.49 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"
```

8.50 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();
```

8.51 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');
```

8.52 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
```

8.53 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;
/
```

8.54 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"
```

9
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.

9.1 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');
```

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.

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 */,
                                                        pg /* table owner */,
                                                        "employeesNodes" /* vertex
table name */,
                                                        propertyNames /* property
names */,
                                                        propertyTypes /* property
data types */,
                                                        "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 */,
                                                        pg /* table owner */,
                                                        organizationEdges" /* edge
table name */,
                                                        propertyNames /* property
names */,
                                                        propertyTypes /* property
data types */,
                                                        "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(1l);
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(2l);
v2.setProperty("age",  Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");
```

```
// Add edge e1
Edge e1 = opgEmployees.addEdge(1l, 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(1l);
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(2l);
v2.setProperty("age",  Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(1l, 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;
       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 `employeNodes` 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(1l);
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(2l);
v2.setProperty("age",  Integer.valueOf(27));
v2.setProperty("name", "Bob");
v2.setProperty("adress", "Sesame Street 334");

// Add edge e1
Edge e1 = opgEmployees.addEdge(1l, 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(1l);
       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(2l);
       v2.setProperty("age",  Integer.valueOf(27));
       v2.setProperty("name", "Bob");
       v2.setProperty("adress", "Sesame Street 334");


       // Add edge e1
       Edge e1 = opgEmployees.addEdge(1l, 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
                                                                    from
the property graph

instance */
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 */,
                                                             true /* skip store
to cache */,
                                                             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}]
```

Index

O

P

R

S

T

V