Oracle Spatial and Graph

Property Graph Developer's Guide

Release 12.2

E75735-09

May 2017

ORACLE

Oracle Spatial and Graph Property Graph Developer's Guide, Release 12.2

E75735-09

Primary Author: Chuck Murray

Contributors: Bill Beauregard, Hector Briseno, Hassan Chafi, Zazhil Herena, Sungpack Hong, Roberto Infante, Hugo Labra, Gabriela Montiel-Moreno, Siva Ravada, Carlos Reyes, Korbinian Schmid, Jane Tao, Zhe (Alan) Wu

Contents

3 Using the In-Memory Analyst (PGX)

4 SQL-Based Property Graph Query and Analytics

5 OPG_APIS Package Subprograms

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 Semantic 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.

1

Spatial and Graph Property Graph Support Overview

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

1.1 About Property Graph Support

Oracle Spatial and Graph delivers advanced spatial and graph analytic capabilities in Oracle Database.

The property graph features support graph operations, indexing, queries, search, and in-memory analytics.

No special configuration operations are required. The requirements for using the property graph features are:

- Oracle Spatial and Graph must be installed.
- `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.
- Java 8 or higher must be installed.

  However, there is no need for a separate installation because JDK 8 is installed under $ORACLE_HOME. Just make sure that the correct version of Java is used to run property graph-based applications.

1.2 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, cyber security, 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 Tinkerpop Blueprints, and Java graph APIs
- Text search and query through integration with Apache Lucene, SolrCloud, and Oracle Text
- Scripting languages support for Groovy and Python
- A parallel, in-memory graph analytics engine
- A fast, scalable suite of social network analysis functions that include ranking, centrality, recommender, community detection, path finding
- Parallel bulk load and export of property graph data in Oracle-defined flat files format
- Manageability through a Groovy-based console to execute Java and Tinkerpop Gremlin APIs

Property Graph Sizing Recommendations

1.2.1 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

1.3 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.

Representing Spatial Data in a Property Graph

Creating a Spatial Index on Property Graph Data

Querying Spatial Data in a Property Graph

1.3.1 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))
```

1.3.2 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);
```

1.3.3 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"))
```

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.

<graph_name>GE$ key/value data tables for vertices and edges, you can use a two-tables schema to achieve better run-time performance.

Oracle Flat File Format Definition
: A property graph can be defined in two flat files, specifically description files for the vertices and edges.

Example Python User Interface
: The example Python scripts in $ORACLE_HOME/md/property_graph/pyopg/ can used with Oracle Spatial and Graph Property Graph, and you may want to change and enhance them (or copies of them) to suit your needs.

2.1 About Property Graphs

Property graphs allow an easy association of properties (key-value pairs) with graph vertices and edges, and they enable analytical operations based on relationships across a massive set of data.

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:

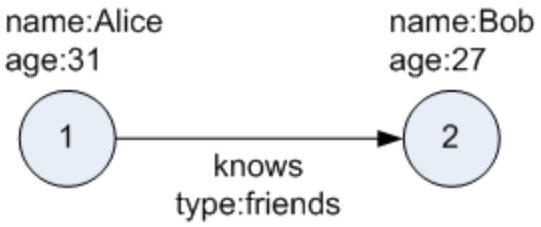
- A set of outgoing edges
- A set of incoming edges
- A collection of properties

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

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



Standards are not available for Big Data Spatial and Graph property graph data model, but it is similar to the W3C standards-based Resource Description Framework (RDF) graph data model. The property graph data model is simpler and much less precise than RDF. These differences make it 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:

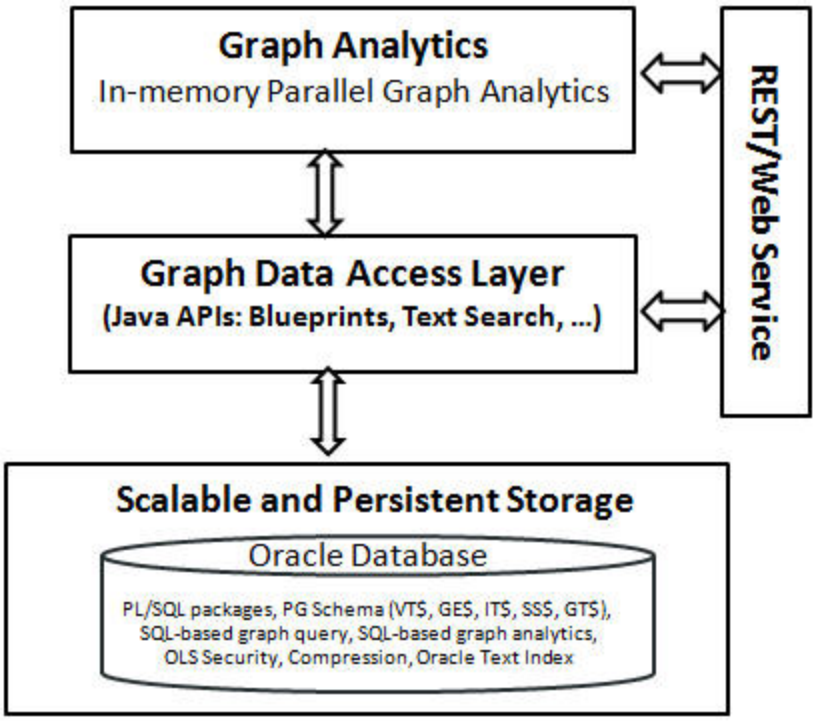
Specifying Labels for Vertices

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 Analyst

Data Access Layer

Storage Management

2.1.2.1 In-Memory Analyst

The in-memory analyst layer enables you to analyze property graphs using parallel in-memory execution. It provides over 35 analytic functions, including path calculation, ranking, community detection, and recommendations.

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. The Java APIs include an implementation of TinkerPop Blueprints graph interfaces for the property graph data model. The APIs also integrate with the Apache Lucene and Apache SolrCloud, which are widely-adopted open-source text indexing and search engines.

2.1.2.3 Storage Management

Property graphs are stored in Oracle Database. Tables are used internally to model the vertices and edges of property graphs.

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 example in this topic shows a GraphML description of the property graph shown in What Are Property Graphs?.

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>
```

Related Topics:

The GraphML File Format

2.2.2 GraphSON Data Format

The GraphSON file format is based on JavaScript Object Notation (JSON) for describing graphs. The example in this topic shows a GraphSON description of the property graph shown in What Are Property Graphs?.

Example 2-2 GraphSON Description of a Simple Property Graph

```
{
    "graph": {
        "mode":"NORMAL",
        "vertices": [
            {
                "name": "Alice",
                "age": 31,
                "_id": "1",
                "_type": "vertex"
            },
            {
                "name": "Bob",
                "age": 27,
                "_id": "2",
                "_type": "vertex"
            }
```

```
        ],
        "edges": [
            {
                "type": "friends",
                "_id": "3",
                "_type": "edge",
                "_outV": "1",
                "_inV": "2",
                "_label": "knows"
            }
        ]
    }
}
```

Related Topics:

GraphSON Reader and Writer Library

2.2.3 GML Data Format

The Graph Modeling Language (GML) file format uses ASCII to describe graphs. The example in this topic shows a GML description of the property graph shown in What Are Property Graphs?.

Example 2-3 GML Description of a Simple Property Graph

```
graph [
   comment "Simple property graph"
   directed 1
   IsPlanar 1
   node [
      id 1
      label "1"
      name "Alice"
      age 31
        ]
   node [
      id 2
      label "2"
      name "Bob"
      age 27
        ]
   edge [
      source 1
      target 2
      label "knows"
      type "friends"
        ]
      ]
```

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 2-4 Oracle Flat File Description of a Simple Property Graph

The following shows the Oracle flat files that describe the simple property graph example shown in What Are Property Graphs?.

Vertex file:

```
1,name,1,Alice,,
1,age,2,,31,
2,name,1,Bob,,
2,age,2,,27,
```

Edge file:

```
1,1,2,knows,type,1,friends,,
```

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.

Oracle Spatial and Graph lets store, query, manipulate, and query property graph data in Oracle Database. For example, to create a property graph named myGraph, you can use either the Java APIs (oracle.pg.rdbms.OraclePropertyGraph) or the PL/SQL APIs (MDSYS.OPG_APIS package).

With the PL/SQL API:

```
BEGIN
     opg_apis.create_pg(
           'myGraph',
           dop => 4,             -- degree of parallelism
           num_hash_ptns => 8,   -- number of hash partitions used to store the graph
           tbs => 'USERS',       -- tablespace
           options => 'COMPRESS=T'
           );
END;
/
```

With the Java API:

```
  cfg = GraphConfigBuilder
            .forPropertyGraphRdbms()
            .setJdbcUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl")
            .setUsername("<your_user_name>")
            .setPassword("<your_password>")
            .setName("myGraph")
            .setMaxNumConnections(8)
            .setLoadEdgeLabel(false)
            .build();

  OraclePropertyGraph opg = OraclePropertyGraph.getInstance(cfg);
```

After the property graph `myGraph` 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.

For simplicity, only simple graph names are allowed, and they are case insensitive.

Additional internal tables are created with SS$, IT$, and GT$ suffixes, to store graph snapshots, text index metadata, and graph skeleton (topological structure), respectively.

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)
```

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 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.

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".

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`).

Default Indexes on Vertex (VT$) and Edge (GE$) Tables

Flexibility in the Property Graph Schema

2.3.1 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 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)`

2.3.2 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:

- Additional indexes can be added on demand.
- 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. Create your Java programs, using the classes provided in the Java API.

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.

2.5 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.

2.5.1 Overview of the Java APIs

The Java APIs that you can use for property graphs include the following:

2.5.1.1 Oracle Spatial and Graph Property Graph Java APIs

Oracle Spatial and Graph property graph support provides database-specific APIs for Oracle Database. The data access layer API (`oracle.pg.*`) implements TinkerPop Blueprints APIs, text search, and indexing for property graphs stored in 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.*;
```

Also include TinkerPop Blueprints Java APIs.

2.5.1.2 TinkerPop Blueprints Java APIs

TinkerPop Blueprints supports the property graph data model. The API provides utilities for manipulating graphs, which you use primarily through the Spatial and Graph property graph data access layer Java APIs.

To use the Blueprints APIs, import the classes into your Java program:

```
import com.tinkerpop.blueprints.Vertex;
import com.tinkerpop.blueprints.Edge;
```

Related Topics:

Blueprints: A Property Graph Model Interface API

2.5.1.3 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 = "db12c"; /*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 = "db12c"; /*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 = "db12c"; /* 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 Barack Obama'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: Barack Obama
Vertex v = opg.getVertices("name", "Barack Obama").iterator().next();

// EdgeFilterCallback to retrieve all collaborators of Barack Obama
// from the United States who are not politicians
CollaboratorsFilterCallback cefc = CollaboratorsFilterCallback.getInstance(npvfc, v);

Iterable<<Edge> obamaCollabs = opg.getEdges((String[])null /* Match any
of the properties */,
cefc /* Match the
EdgeFilterCallback */
);
Iterator<<Edge> iter = obamaCollabs.iterator();

System.out.println("\n\n--------Collaborators of Barack Obama from " +
 " the US and non-politician\n\n");
long countV = 0;
while (iter.hasNext()) {
Edge edge = iter.next(); // get the edge
// check if obama 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: Barack Obama
Vertex v = opg.getVertices("name", "Barack Obama").iterator().next();

// EdgeFilterCallback to retrieve all collaborators of Barack Obama
// from the United States who are not politicians
CollaboratorsFilterCallback cefc = CollaboratorsFilterCallback.getInstance(npvfc, v);

opg.setEdgeFilterCallback(cefc);

Iterable<Edge> obamaCollabs = opg.getEdges();
Iterator<Edge> iter = obamaCollabs.iterator();
```

```
System.out.println("\n\n--------Collaborators of Barack Obama from " +
 " the US and non-politician\n\n");
long countV = 0;
while (iter.hasNext()) {
Edge edge = iter.next(); // get the edge
// check if obama 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 `obamaCollaborator` attribute associated only with Barack Obama collaborators. The value of this property is specified based on the role of the collaborators.

```
private static class CollaboratorsVertexOpCallback
implements VertexOpCallback
{
private OracleVertexBase m_obama;
private List<Vertex> m_obamaCollaborators;

public CollaboratorsVertexOpCallback(OraclePropertyGraph opg)
{
// Get a list of Barack Obama'sCollaborators
m_obama = (OracleVertexBase) opg.getVertices("name",
 "Barack Obama")
.iterator().next();

Iterable<Vertex> iter = m_obama.getVertices(Direction.BOTH,
"collaborates");
```

```
m_obamaCollaborators = 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 Barack
 * Obama
*/
@Override
public boolean needOp(OracleVertexBase v)
{
return m_obamaCollaborators != null &&
 m_obamaCollaborators.contains(v);
}

@Override
public String getAttributeKeyName(OracleVertexBase v)
{
return "obamaCollaborator";
}

/**
 * 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 `obamaFeud` attribute associated only with Barack Obama feuds. The value of this property is specified based on the role of the collaborators.

```
private static class FeudsEdgeOpCallback
implements EdgeOpCallback
{
private OracleVertexBase m_obama;
private List<Edge> m_obamaFeuds;

public FeudsEdgeOpCallback(OraclePropertyGraph opg)
{
// Get a list of Barack Obama's feuds
m_obama = (OracleVertexBase) opg.getVertices("name",
 "Barack Obama")
.iterator().next();

Iterable<Vertex> iter = m_obama.getVertices(Direction.BOTH,
"feuds");
m_obamaFeuds = 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 Barack Obama's
 * feuds
*/
@Override
public boolean needOp(OracleEdgeBase e)
{
return m_obamaFeuds != null && m_obamaFeuds.contains(e);
}

@Override
public String getAttributeKeyName(OracleEdgeBase e)
{
return "obamaFeud";
}

/**
 * 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_obama.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 Obama based on their role
opg.addAttributeToAllVertices(cvoc, true /** Skip store to Cache */, dop);

// Look up for all collaborators of Obama
Iterable<Vertex> collaborators = opg.getVertices("obamaCollaborator", "political");
System.out.println("Political collaborators of Barack Obama " +
getVerticesAsString(collaborators));

collaborators = opg.getVertices("obamaCollaborator", "business");
System.out.println("Business collaborators of Barack Obama " +
getVerticesAsString(collaborators));

// Add an attribute to all people having a feud with Barack Obama 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 Obama
Iterable<Edge> feuds = opg.getEdges("obamaFeud", "political");
System.out.println("\n\nPolitical feuds of Barack Obama " + getEdgesAsString(feuds));

feuds = opg.getEdges("obamaFeud", "business");
System.out.println("Business feuds of Barack Obama " +
getEdgesAsString(feuds));
```

The following code fragment defines an implementation of `VertexOpCallback` that can be used to remove vertices having value philanthropy for attribute `obamaCollaborator`, then call the API `removeAttributeFromAllVertices`; It also defines an implementation of `EdgeOpCallback` that can be used to remove edges having value business for attribute `obamaFeud`, then call the API `removeAttributeFromAllEdges`.

```
System.out.println("\n\nRemove 'obamaCollaborator' property from all the" +
 "philanthropy collaborators");
PhilanthropyCollaboratorsVertexOpCallback pvoc =
PhilanthropyCollaboratorsVertexOpCallback.getInstance();

opg.removeAttributeFromAllVertices(pvoc);

System.out.println("\n\nRemove 'obamaFeud' property from all the" + "business
feuds");
BusinessFeudsEdgeOpCallback beoc = BusinessFeudsEdgeOpCallback.getInstance();

opg.removeAttributeFromAllEdges(beoc);

/**
 * Implementation of a EdgeOpCallback to remove the "obamaCollaborators"
 * property from all people collaborating with Barack Obama 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
   * obamaCollaborator is Philanthropy
   */
  @Override
  public boolean needOp(OracleVertexBase v)
  {
    String type = v.getProperty("obamaCollaborator");
    return type != null && type.equals("philanthropy");
  }

  @Override
  public String getAttributeKeyName(OracleVertexBase v)
  {
    return "obamaCollaborator";
  }

  /**
   * 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 "obamaFeud" property
 * from all connections in a feud with Barack Obama 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 obamaFeud is
   * business
   */
  @Override
  public boolean needOp(OracleEdgeBase e)
  {
    String type = e.getProperty("obamaFeud");
    return type != null && type.equals("business");
  }

 @Override
 public String getAttributeKeyName(OracleEdgeBase e)
 {
   return "obamaFeud";
 }

 /**
  * 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 = "db12c"; /*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_hos
t>:<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(",")
  .build();
```

Related Topics:

What Are Property Graphs?

2.5.15 Building an In-Memory Graph

In addition to Reading Graph Data 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');
```

2.6 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.

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 the available open-source search engines Apache Lucene and SolrCloud, and 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.

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());
```

If you want to modify the initial configuration of a text index, you may need first to drop the existing graph and recreate the index using the new configuration.

- Configuring Text Indexes Using the Apache Lucene Search Engine
- Configuring Text Indexes Using the SolrCloud Search Engine
- Configuring Text Indexes Using Oracle Text

Configuring Text Indexes Using the Apache Lucene Search Engine

A text index using Apache Lucene Search engine uses a `LuceneIndexParameters` configuration object. The configuration parameters for indexes using a Lucene Search engine include:

- **Number of directories:** an integer specifying the number of Apache Lucene directories to use for the automatic index. Using multiple directories provides storage and performance scalability. The default value is set to 1.
- **Batch Size:** an integer specifying the batch size to use for document batching in Apache Lucene. The default batch size used is 10000.
- **Commit Batch Size:** an integer specifying the number of document to add into the Apache Lucene index before a commit operation is executed. The default commit batch size used is 500000.
- **Data type handling flag:** a Boolean specifying if Apache Lucene data types handling is enabled. Enabling data types handling fasten up lookups over numeric and date time data types.
- **Directory names:** a string array specifying the base path location where the Apache Lucene directories will be created.

The following code fragment creates the configuration for a text index using Apache Lucene Search Engine with a physical directory.

```
OracleIndexParameters indexParams =
      OracleIndexParameters.buildFS(4, 4, 10000, 50000, true,
                                    "/home/data/text-index");
```

Oracle Spatial and Graph extends the Apache Lucene capabilities by using a DBFS directory to store all Apache Lucene directories and data files.

Configuring Text Indexes Using the SolrCloud Search Engine

A text index using SolrCloud Search engine uses a `SolrIndexParameters` object behind the scenes to identify the SolrCloud host name, the number of shards, and replication factor used during the index construction. The configuration parameters for indexes using a SolrCloud Search engine include:

- **Configuration name:** the name of the Zookeeper directory where the SolrCloud configuration files for Oracle Property Graph are stored, such as *opgconfig*. The configuration files include the required field's schema (schema.xml) and storage settings (`solrconfig.xml`).
- **Server URL:** the SolrCloud server URL used to connect to the SolrCloud service, such as*http://localhost:2181/solr*
- **SolrCloud Node Set:** the host names of the nodes in the SolrCloud service where the collection's shards will be stored, such as *node01:8983_solr,node02:8983_solr,node03:8983_solr*. If the value is set to null, then the collection will be created using all the SolrCloud nodes available in the service.
- **Zookeeper Timeout**: a positive integer representing the timeout (in seconds) used to wait for a Zookeeper connection.
- **Number of shards:** the number of shards to create for the text index collection. If the SolrCloud configuration is using an HDFS directory, the number of shards

must not exceed the number of SolrCloud nodes specified in the SolrCloud node set.

- **Replication factor:** the replication factor used in the SolrCloud collection. The default value is set to *1*.
- **Maximum shards per node:** the maximum number of shards that can be created on each SolrCloud node. Note that this value must not be smaller than number of shards/# of nodes in the SolrCloud Node set.
- **DOP:** the degree of parallelism to use when reading the vertices (or edges) from the property graph and indexing the K/V pairs. The default value is set to *1*.
- **Batch Size:** an integer specifying the batch size to use for document batching in Apache SolrCloud. The default batch size used is 10000.
- **Commit Batch Size:** an integer specifying the number of document to add into the Apache SolrCloud index before a commit operation is executed. The default commit batch size used is 500000.
- **Write timeout:** the timeout (in seconds) used to wait for an index operation to be completed. If the index operation was unsuccessful due to a communication error, the operation will be tried out again until the timeout is reached or the operation completes.

The following code fragment creates the configuration for a text index using SolrCloud.

```
String configName = "opgconfig";
String solrServerUrl = "nodea:2181/solr"
String solrNodeSet = "nodea:8983_solr,nodeb:8983_solr," +
                     "nodec:8983_solr,noded:8983_solr";

int zkTimeout = 15;
int numShards = 4;
int replicationFactor = 1;
int maxShardsPerNode = 1;

OracleIndexParameters indexParams =
                OracleIndexParameters.buildSolr(configName,
                                        solrServerUrl,
                                        solrNodeSet,
                                        zkTimeout,
                                        numShards,
                                        replicationFactor,
                                        maxShardsPerNode,
                                        4,
                                        10000,
                                        500000,
                                        15);
```

When using SolrCloud, you must first load a collection's configuration for the text indexes into Apache Zookeeper, as described in Uploading a Collection's SolrCloud Configuration to Zookeeper.

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`.

The supplied example ExampleRDBMS6 creates a property graph from an input file, creates an automatic text index on vertices, and executes some text search queries using Apache Lucene.

The following code fragment creates an automatic index over an existing property graph's vertices with these property keys: name, role, religion, and country. The automatic text index will be stored under four subdirectories under the `/home/data/text-index` directory. Apache Lucene data types handling is enabled. This example uses a DOP (parallelism) of 4 for re-indexing tasks.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(…);

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
    opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

// Create an automatic index using Apache Lucene engine.
// Specify Index Directory parameters (number of directories,
// number of connections to database, batch size, commit size,
// enable datatypes, location)
OracleIndexParameters indexParams =
     OracleIndexParameters.buildFS(4, 4, 10000, 50000, true,
             "/home/data/text-index ");
opg.setDefaultIndexParameters(indexParams);

// specify indexed keys
String[] indexedKeys = new String[4];
indexedKeys[0] = "name";
indexedKeys[1] = "role";
indexedKeys[2] = "religion";
indexedKeys[3] = "country";
```

```
// Create auto indexing on above properties for all vertices
opg.createKeyIndex(indexedKeys, Vertex.class);
```

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:Barack Obama`. 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. For details on text search syntax using Apache Lucene, see https://lucene.apache.org/core/2_9_4/queryparsersyntax.html.

```
// Find all vertices with name Barack Obama.
    Iterator<Vertices> vertices = opg.getVertices("name", "Barack Obama").iterator();
    System.out.println("----- Vertices with name Barack Obama -----");
    countV = 0;
    while (vertices.hasNext()) {
      System.out.println(vertices.next());
      countV++;
    }
    System.out.println("Vertices found: " + countV);

   // Find all vertices with name including keyword "Obama"
   // Wildcard searching is supported.
    boolean useWildcard = true;
    Iterator<Vertices> vertices = opg.getVertices("name", "*Obama*").iterator();
    System.out.println("----- Vertices with name *Obama* -----");
    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 Barack Obama-----
Vertex ID 1 {name:str:Barack Obama, role:str:political authority, occupation:str:
44th president of United States of America, country:str:United States, political
party:str:Democratic, religion:str:Christianity}
Vertices found: 1

----- Vertices with name *Obama* -----
Vertex ID 1 {name:str:Barack Obama, role:str:political authority, occupation:str:
44th president of United States of America, country:str:United States, political
party:str:Democratic, religion:str:Christianity}
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.

The example code in this topic creates a manual text index on edges, puts some data into the index, and executes some text search queries using Apache SolrCloud.

When using SolrCloud, you must first load a collection's configuration for the text indexes into Apache Zookeeper, as described in Uploading a Collection's SolrCloud Configuration to Zookeeper.

The following code fragment creates a manual text index over an existing property graph using four shards, one shard per node, and a replication factor of 1. The number of shards corresponds to the number of nodes in the SolrCloud cluster.

```
 String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
    opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

// Create a manual text index using SolrCloud// Specify Index Directory parameters:
configuration name, Solr Server URL, Solr Node set,
// replication factor, zookeeper timeout (secs),
// maximum number of shards per node,
   // number of connections to database, batch size, commit size,
         // write timeout (in secs)
             String configName = "opgconfig";
             String solrServerUrl = "nodea:2181/solr"
             String solrNodeSet = "nodea:8983_solr,nodeb:8983_solr," +
                                  "nodec:8983_solr,noded:8983_solr";

         int zkTimeout = 15;
         int numShards = 4;
         int replicationFactor = 1;
         int maxShardsPerNode = 1;

OracleIndexParameters indexParams =
```

```
                OracleIndexParameters.buildSolr(configName,
                                        solrServerUrl,
                                        solrNodeSet,
                                        zkTimeout,
                                        numShards,
                                        replicationFactor,
                                        maxShardsPerNode,
                                        4,
                                        10000,
                                        500000,
                                        15);
opg.setDefaultIndexParameters(indexParams);


// Create manual indexing on above properties for all vertices
OracleIndex<Edge> index = ((OracleIndex<Edge>) opg.createIndex("myIdx", Edge.class));

Vertex v1 = opg.getVertices("name", "Barack Obama").iterator().next();

Iterator<Edge> edges
               = v1.getEdges(Direction.OUT, "collaborates").iterator();

         while (edges.hasNext()) {
            Edge edge = edges.next();
            Vertex vIn = edge.getVertex(Direction.IN);
            index.put("collaboratesWith", vIn.getProperty("name"), edge);
         }
```

The next code fragment executes a query over the manual index to get all edges with the key/value pair c`ollaboratesWith:Beyonce`. Additionally, wildcards search can be supported by specifying the parameter `useWildCards` in the get API call.

```
// Find all edges with collaboratesWith Beyonce.
   // Wildcard searching is supported using true parameter.
    edges = index.get("collaboratesWith", "Beyonce").iterator();
    System.out.println("----- Edges with name Beyonce -----");
    countE = 0;
    while (edges.hasNext()) {
      System.out.println(edges.next());
      countE++;
    }
    System.out.println("Edges found: "+ countE);

// Find all vertices with name including Bey*.
   // Wildcard searching is supported using true parameter.
    edges = index.get("collaboratesWith", "*Bey*", true).iterator();
    System.out.println("----- Edges with collaboratesWith Bey* -----");
    countE = 0;
    while (edges.hasNext()) {
      System.out.println(edges.next());
      countE++;
    }
    System.out.println("Edges found: " + countE);
```

The preceding code example produces output like the following:

```
----- Edges with name Beyonce -----
Edge ID 1000 from Vertex ID 1 {country:str:United States, name:str:Barack Obama,
occupation:str:44th president of United States of America, political
party:str:Democratic, religion:str:Christianity, role:str:political authority}
=[collaborates]=> Vertex ID 2 {country:str:United States, music genre:str:pop soul ,
```

```
name:str:Beyonce, role:str:singer actress} edgeKV[{weight:flo:1.0}]
Edges found: 1

----- Edges with collaboratesWith Bey* -----
Edge ID 1000 from Vertex ID 1 {country:str:United States, name:str:Barack Obama,
occupation:str:44th president of United States of America, political
party:str:Democratic, religion:str:Christianity, role:str:political authority}
=[collaborates]=> Vertex ID 2 {country:str:United States, music genre:str:pop soul ,
name:str:Beyonce, role:str:singer actress} edgeKV[{weight:flo:1.0}]
Edges found: 1
```

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 Apache Lucene
- Executing Search Queries Over a Text Index Using SolrCloud
- Executing Search Queries Over a Text Index Using Oracle Text

Executing Search Queries Over a Text Index Using Apache Lucene

The following code fragment creates an automatic index using Apache Lucene, and executes a query over the text index by specifying a particular key/value pair.

```
// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
               OraclePropertyGraphDataLoader.getInstance();

opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
               "PDML=T,PDDL=T,NO_DUP=T,");

// Create an automatic index using Apache Lucene engine.
// Specify Index Directory parameters (number of directories,
// number of connections to database, batch size, commit size,
// enable datatypes, location)
OracleIndexParameters indexParams =
            OracleIndexParameters.buildFS(4, 4, 10000, 50000, true,
                                          "/home/data/text-index ");
opg.setDefaultIndexParameters(indexParams);

// Create manual indexing on above properties for all vertices
OracleIndex<Edge> index = ((OracleIndex<Edge>) opg.createIndex("myIdx", Edge.class));

Vertex v1 = opg.getVertices("name", "Barack Obama").iterator().next();

Iterator<Edge> edges
               = v1.getEdges(Direction.OUT, "collaborates").iterator();

while (edges.hasNext()) {
  Edge edge = edges.next();
  Vertex vIn = edge.getVertex(Direction.IN);
  index.put("collaboratesWith", vIn.getProperty("name"), edge);
  index.put("country", vIn.getProperty("country"), edge);
}

// Wildcard searching is supported using true parameter.
Iterator<Edge> edges = index.get("country", "United States").iterator();
```

```
System.out.println("----- Edges with query: " + queryExpr + " -----");
long countE = 0;
while (edges.hasNext()) {
  System.out.println(edges.next());
  countE++;
}
System.out.println("Edges found: "+ countE);
```

In this case, the text index will produce a search query out of the key and value objects. If the `useWildcards` flag is not specified or enabled, then results retrieved will include only exact matches. If the value object is a numeric or date time value, the produced query will be an inclusive range query where the lower and upper limit is defined by the value. Only numeric or date time matches will be retrieved.

On the other hand, if the value is a string then all matching key/value pairs will be retrieved no matter their data type. The resulting text query of this type of queries is a Boolean query with a set of optional search terms, one for each supported data type. Further details on data type handling can be found in Handling Data Types.

This way, the previous code produces a query expression *country1:"United States" OR country9:"United States" OR … OR countryE:"United States"* (if Lucene's data type handling is enabled), or*country:"1United States" OR country:"2United States" OR … OR country:"EUnited States"* (if Lucene's data type handling is disabled).

Using a String value object with wildcards enabled requires that the value be written using Apache Lucene syntax. For details on text search syntax using Apache Lucene, see https://lucene.apache.org/core/2_9_4/queryparsersyntax.htmlhttps://lucene.apache.org/core/2_9_4/queryparsersyntax.html.

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 using a single key/value pair with String data type only. The following code produces a query expression *country1:"United States"* (if Lucene's data type handling is enabled), or *country:"1United States"* (if Lucene's data type handling is disabled).

```
// Wildcard searching is supported using true parameter.
Iterator<Edge> edges = index.get("country", "United States", true,
String.class).iterator();

System.out.println("----- Edges with query: " + queryExpr + " -----");
long countE = 0;
while (edges.hasNext()) {
  System.out.println(edges.next());
  countE++;
}
System.out.println("Edges found: "+ countE);
```

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. Oracle Spatial and Graph provides a set of utilities to help users write their own Lucene text search queries using the query syntax and data type identifiers required by the automatic and manual text indexes.

The method `buildSearchTerm(key, value, dtClass)` in `LuceneIndex` creates a query expression of the form *field:query_expr* by adding the data type identifier to the key (or value) and transforming the value into the required string representation based on the given data type and Apache Lucene's data type handling configuration.

The following code fragment uses the `buildSearchTerm` method to produce a query expression *country1:United** (if Lucene's data type handling is enabled), or *country:1United** (if Lucene's data type handling is disabled) used in the previous examples:

```
String szQueryStrCountry = index.buildSearchTerm("country",
                                  "United*", String.class);
```

To deal with the key and values as individual objects to construct a different Lucene Query like a `WildcardQuery` using the required syntax, the methods `appendDatatypesSuffixToKey(key, dtClass)` and `appendDatatypesSuffixToValue(value, dtClass)` in `LuceneIndex` will append the appropriate data type identifiers and transform the value into the required Lucene string representation based on the given data type.

The following code fragment uses the `appendDatatypesSuffixToKey` method to generate the field name required in a Lucene text query. If Lucene's data type handling is enabled, the string returned will append the String data type identifier as a suffix of the key (`country1`). In any other case, the retrieved string will be the original key (`country`).

```
String key = index.appendDatatypesSuffixToKey("country", String.class);
```

The next code fragment uses the `appendDatatypesSuffixToValue` method to generate the query body expression required in a Lucene text query. If Lucene's data type handling is disabled, the string returned will append the String data type identifier as a prefix of the key (`1United*`). In all other cases, the string returned will be the string representation of the value (`United*`).

```
String value = index.appendDatatypesSuffixToValue("United*", String.class);
```

`LuceneIndex` also supports generating a Term object using the method `buildSearchTermObject(key, value, dtClass)`. Term objects are commonly used among different type of Lucene Query objects to constrain the fields and values of the documents to be retrieved. The following code fragment shows how to create a Wildcard Query object using the `buildSearchTermObject` method.

```
Term term = index.buildSearchTermObject("country", "United*", String.class);
Query query = new WildcardQuery(term);
```

Executing Search Queries Over a Text Index Using SolrCloud

The following code fragment creates an automatic index using SolrCloud, and executes a query over the text index by specifying a particular key/value pair.

```
// Create a manual text index using SolrCloud// Specify Index Directory parameters:
configuration name, Solr Server URL, Solr Node set,
// replication factor, zookeeper timeout (secs),
// maximum number of shards per node,
// number of connections to database, batch size, commit size,
// write timeout (in secs)
String configName = "opgconfig";
String solrServerUrl = "nodea:2181/solr"
String solrNodeSet = "nodea:8983_solr,nodeb:8983_solr," +
                     "nodec:8983_solr,noded:8983_solr";

int zkTimeout = 15;
int numShards = 4;
int replicationFactor = 1;
int maxShardsPerNode = 1;

OracleIndexParameters indexParams =
```

```
                    OracleIndexParameters.buildSolr(configName,
                                                    solrServerUrl,
                                                    solrNodeSet,
                                                    zkTimeout,
                                                    numShards,
                                                    replicationFactor,
                                                    maxShardsPerNode,
                                                    4,
                                                    10000,
                                                    500000,
                                                    15);
opg.setDefaultIndexParameters(indexParams);

// specify indexed keys
String[] indexedKeys = new String[4];
indexedKeys[0] = "name";
indexedKeys[1] = "role";
indexedKeys[2] = "religion";
indexedKeys[3] = "country";

// Create auto indexing on above properties for all vertices
opg.createKeyIndex(indexedKeys, Vertex.class);

// Create manual indexing on above properties for all vertices
OracleIndex<Vertex> index = ((OracleIndex<Vertex>) opg.getAutoIndex(Vertex.class);

Iterator<Vertex> vertices = index.get("country", "United States").iterator();
System.out.println("----- Vertices with query: " + queryExpr + " -----");
countV = 0;
while (vertices.hasNext()) {
  System.out.println(vertices.next());
  countV++;
}
System.out.println("Vertices found: "+ countV);
```

In this case, the text index will produce a search query out of the value object. If the `useWildcards` flag is not specified or enabled, then results retrieved will include only exact matches. If the value object is a numeric or date time value, the produced query will be an inclusive range query where the lower and upper limit is defined by the value. Only numeric or date time matches will be retrieved.

On the other hand, if the value is a string then all matching key/value pairs will be retrieved no matter their data type. The resulting text query of this type of queries is a Boolean query with a set of optional search terms, one for each supported data type. Further details on data type handling can be found in Handling Data Types.

This way, the previous code produces a query expression *country_str:"United States" OR country_ser:"United States" OR … OR country_json:"United States"*.

Using a String value object with wildcards enabled requires that the value be written using Apache Lucene Syntax. For details on text search syntax using Apache Lucene, see https://lucene.apache.org/core/2_9_4/queryparsersyntax.htmlhttps://lucene.apache.org/core/2_9_4/queryparsersyntax.html.

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 using a single key/value pair with String data type only. The code produces a query expression *country_str:"United States"*.

```
// Wildcard searching is supported using true parameter.
Iterator<Edge> edges = index.get("country", "United States", true,
```

```
String.class).iterator();
    System.out.println("----- Edges with query: " + queryExpr + " -----");
    countE = 0;
    while (edges.hasNext()) {
      System.out.println(edges.next());
      countE++;
    }
    System.out.println("Edges found: "+ countE);
```

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. Oracle Spatial and Graph provides a set of utilities to help users write their own Lucene text search queries using the query syntax and data type identifiers required by the automatic and manual text indexes.

The method `buildSearchTerm(key, value, dtClass)` in `SolrIndex` creates a query expression of the form *field:query_expr* by adding the data type identifier to the key (or value) and transforming the value into the required string representation using the data type formats required by the index.

The following code fragment uses the `buildSearchTerm` method to produce a query expression *country_str:United** used in the previous example:

```
String szQueryStrCountry = index.buildSearchTerm("country",
                                  "United*", String.class);
```

To deal with the key and values as individual objects to construct a different SolrCloud query like a `WildcardQuery` using the required syntax, the methods `appendDatatypesSuffixToKey(key, dtClass)` and `appendDatatypesSuffixToValue(value, dtClass)` in `SolrIndex` will append the appropriate data type identifiers and transform the key and value into the required SolrCloud string representation based on the given data type.

The following code fragment uses the `appendDatatypesSuffixToKey` method to generate the field name required in a Lucene text query. If Lucene's data type handling is enabled, the string returned will append the String data type identifier as a suffix of the key (`country_str`).

```
String key = index.appendDatatypesSuffixToKey("country", String.class);
```

The next code fragment uses the `appendDatatypesSuffixToValue` method to generate the query body expression required in a Lucene text query. If Lucene's data type handling is disabled, the string returned will append the String data type identifier as a prefix of the key (`1United*`). In all other cases, the string returned will be the string representation of the value (`United*`).

```
String key = index.appendDatatypesSuffixToKey("country", String.class);
```

The next code fragment uses the `appendDatatypesSuffixToValue` method to generate the query body expression required in a `SolrCloud` text query. The string returned will be the string representation of the value (`United*`).

```
String value = index.appendDatatypesSuffixToValue("United*", String.class);
```

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("Obama" /* 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 *Obama*.

```
// Create the text query object for Oracle Text
OracleTextQueryObject otqo
```

```
                = OracleTextQueryObject.getInstance("Obama" /* 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. The following topics describe how to append this prefix/suffix for Apache Lucene and SolrCloud.

Appending Data Type Identifiers on Apache Lucene

Appending Data Type Identifiers on SolrCloud

Handling Data Types on Oracle Text

2.6.5.1 Appending Data Type Identifiers on Apache Lucene

When Lucene's data types handling is enabled, you must append the proper data type identifier as a suffix to the key in the query expression. This can be done by executing a `String.concat()` operation to the key. If Lucene's data types handling is disabled, you must insert the data type identifier as a prefix in the value String. The following table shows the data type identifiers available for text indexing using Apache Lucene (see also the Javadoc for `LuceneIndex`).

Table 2-1 Apache Lucene Data Type Identifiers

Lucene Data Type Identifier	Description
TYPE_DT_STRING	String
TYPE_DT_BOOL	Boolean
TYPE_DT_DATE	Date

Table 2-1 (Cont.) Apache Lucene Data Type Identifiers

Lucene Data Type Identifier	Description
TYPE_DT_FLOAT	Float
TYPE_DT_DOUBLE	Double
TYPE_DT_LONG	Long
TYPE_DT_CHAR	Character
TYPE_DT_SHORT	Short
TYPE_DT_BYTE	Byte
TYPE_DT_SPATIAL	Spatial
TYPE_DT_INTEGER	Integer
TYPE_DT_SERIALIZABLE	Serializable

The following code fragment creates a manual index on edges using Lucene's data type handling, adds data, and later executes a query over the manual index to get all edges with the key/value pair `collaboratesWith:Beyonce AND country1:United*` using wildcards.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(…);

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
    opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

// Specify Index Directory parameters (number of directories,
   // number of connections to database, batch size, commit size,
         // enable datatypes, location)
OracleIndexParameters indexParams =
     OracleIndexParameters.buildFS(4, 4, 10000, 50000, true,
            "/home/data/text-index ");
opg.setDefaultIndexParameters(indexParams);
// Create manual indexing on above properties for all edges
OracleIndex<Edge> index = ((OracleIndex<Edge>) opg.createIndex("myIdx", Edge.class));

Vertex v1 = opg.getVertices("name", "Barack Obama").iterator().next();

Iterator<Edge> edges
               = v1.getEdges(Direction.OUT, "collaborates").iterator();

          while (edges.hasNext()) {
             Edge edge = edges.next();
             Vertex vIn = edge.getVertex(Direction.IN);
             index.put("collaboratesWith", vIn.getProperty("name"), edge);
             index.put("country", vIn.getProperty("country"), edge);
          }
```

```
// Wildcard searching is supported using true parameter.
    String key = "country";
    key =
key.concat(String.valueOf(oracle.pg.text.lucene.LuceneIndex.TYPE_DT_STRING));

    String queryExpr = "Beyonce AND " + key + ":United*";
    edges = index.get("collaboratesWith", queryExpr, true /
*UseWildcard*/).iterator();
    System.out.println("----- Edges with query: " + queryExpr + " -----");
    countE = 0;
    while (edges.hasNext()) {
      System.out.println(edges.next());
      countE++;
    }
    System.out.println("Edges found: "+ countE);
```

The preceding code example might produce output like the following:

```
----- Edges with name Beyonce AND country1:United* -----
Edge ID 1000 from Vertex ID 1 {country:str:United States, name:str:Barack Obama,
occupation:str:44th president of United States of America, political
party:str:Democratic, religion:str:Christianity, role:str:political authority}
=[collaborates]=> Vertex ID 2 {country:str:United States, music genre:str:pop soul ,
name:str:Beyonce, role:str:singer actress} edgeKV[{weight:flo:1.0}]
Edges found: 1
```

The following code fragment creates an automatic index on vertices, disables Lucene's data type handling, adds data, and later executes a query over the manual index from a previous example to get all vertices with the key/value pair `country:United* AND role:1*political*` using wildcards.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(…);

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
    opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

// Create an automatic index using Apache Lucene engine.
// Specify Index Directory parameters (number of directories,
   // number of connections to database, batch size, commit size,
         // enable datatypes, location)
OracleIndexParameters indexParams =
     OracleIndexParameters.buildFS(4, 4, 10000, 50000, false, "/ home/data/text-
index ");
opg.setDefaultIndexParameters(indexParams);

// specify indexed keys
String[] indexedKeys = new String[4];
indexedKeys[0] = "name";
indexedKeys[1] = "role";
indexedKeys[2] = "religion";
indexedKeys[3] = "country";

// Create auto indexing on above properties for all vertices
opg.createKeyIndex(indexedKeys, Vertex.class);
```

```
// Wildcard searching is supported using true parameter.
    String value = "*political*";
    value = String.valueOf(LuceneIndex.TYPE_DT_STRING) + value;
String queryExpr = "United* AND role:" + value;


vertices = opg.getVertices("country", queryExpr,  true /*useWildcard*/).iterator();
    System.out.println("----- Vertices with query: " + queryExpr + " -----");
    countV = 0;
    while (vertices.hasNext()) {
      System.out.println(vertices.next());
      countV++;
    }
    System.out.println("Vertices found: " + countV);
```

The preceding code example might produce output like the following:

```
----- Vertices with query: United* and role:1*political* -----
Vertex ID 30 {name:str:Jerry Brown, role:str:political authority, occupation:str:
34th and 39th governor of California, country:str:United States, political
party:str:Democratic, religion:str:roman catholicism}
Vertex ID 24 {name:str:Edward Snowden, role:str:political authority,
occupation:str:system administrator, country:str:United States,
religion:str:buddhism}
Vertex ID 22 {name:str:John Kerry, role:str:political authority, country:str:United
States, political party:str:Democratic, occupation:str:68th United States Secretary
of State, religion:str:Catholicism}
Vertex ID 21 {name:str:Hillary Clinton, role:str:political authority,
country:str:United States, political party:str:Democratic, occupation:str:67th
United States Secretary of State, religion:str:Methodism}
Vertex ID 19 {name:str:Kirsten Gillibrand, role:str:political authority,
country:str:United States, political party:str:Democratic, occupation:str:junior
United States Senator from New York, religion:str:Methodism}
Vertex ID 13 {name:str:Ertharin Cousin, role:str:political authority,
country:str:United States, political party:str:Democratic}
Vertex ID 11 {name:str:Eric Holder, role:str:political authority, country:str:United
States, political party:str:Democratic, occupation:str:United States Deputy Attorney
General}
Vertex ID 1 {name:str:Barack Obama, role:str:political authority, occupation:str:
44th president of United States of America, country:str:United States, political
party:str:Democratic, religion:str:Christianity}
Vertices found: 8
```

2.6.5.2 Appending Data Type Identifiers on SolrCloud

For Boolean operations on SolrCloud text indexes, you must append the proper data type identifier as suffix to the key in the query expression. This can be done by executing a `String.concat()` operation to the key. The following trable shows the data type identifiers available for text indexing using SolrCloud (see the Javadoc for `SolrIndex`).

Table 2-2 SolrCloud Data Type Identifiers

Solr Data Type Identifier	Description
TYPE_DT_STRING	String
TYPE_DT_BOOL	Boolean
TYPE_DT_DATE	Date

Table 2-2 (Cont.) SolrCloud Data Type Identifiers

Solr Data Type Identifier	Description
TYPE_DT_FLOAT	Float
TYPE_DT_DOUBLE	Double
TYPE_DT_INTEGER	Integer
TYPE_DT_LONG	Long
TYPE_DT_CHAR	Character
TYPE_DT_SHORT	Short
TYPE_DT_BYTE	Byte
TYPE_DT_SPATIAL	Spatial
TYPE_DT_SERIALIZABLE	Serializable

The following code fragment creates a manual index on edges using SolrCloud, adds data, and later executes a query over the manual index to get all edges with the key/value pair `collaboratesWith:Beyonce AND country1:United*` using wildcards.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(args,
                                                          szGraphName);

String szOPVFile = "../../data/connections.opv";
String szOPEFile = "../../data/connections.ope";

// Do a parallel data loading
OraclePropertyGraphDataLoader opgdl =
OraclePropertyGraphDataLoader.getInstance();
    opgdl.loadData(opg, szOPVFile, szOPEFile, 2 /* dop */, 1000, true,
"PDML=T,PDDL=T,NO_DUP=T,");

// Create a manual text index using SolrCloud// Specify Index Directory parameters:
configuration name, Solr Server URL, Solr Node set,
// replication factor, zookeeper timeout (secs),
// maximum number of shards per node,
   // number of connections to database, batch size, commit size,
         // write timeout (in secs)
             String configName = "opgconfig";
             String solrServerUrl = "nodea:2181/solr"
             String solrNodeSet = "nodea:8983_solr,nodeb:8983_solr," +
                                  "nodec:8983_solr,noded:8983_solr";

         int zkTimeout = 15;
         int numShards = 4;
         int replicationFactor = 1;
         int maxShardsPerNode = 1;

OracleIndexParameters indexParams =
                OracleIndexParameters.buildSolr(configName,
                                        solrServerUrl,
                                        solrNodeSet,
                                        zkTimeout,
```

```
                                        numShards,
                                        replicationFactor,
                                        maxShardsPerNode,
                                        4,
                                        10000,
                                        500000,
                                        15);
opg.setDefaultIndexParameters(indexParams);

// Create manual indexing on above properties for all vertices
OracleIndex<Edge> index = ((OracleIndex<Edge>) opg.createIndex("myIdx", Edge.class));

Vertex v1 = opg.getVertices("name", "Barack Obama").iterator().next();

Iterator<Edge> edges
               = v1.getEdges(Direction.OUT, "collaborates").iterator();

          while (edges.hasNext()) {
             Edge edge = edges.next();
             Vertex vIn = edge.getVertex(Direction.IN);
             index.put("collaboratesWith", vIn.getProperty("name"), edge);
             index.put("country", vIn.getProperty("country"), edge);
          }

// Wildcard searching is supported using true parameter.
    String key = "country";
    key = key.concat(oracle.pg.text.solr.SolrIndex.TYPE_DT_STRING);

    String queryExpr = "Beyonce AND " + key + ":United*";
    edges = index.get("collaboratesWith", queryExpr, true /**
UseWildcard*/).iterator();
    System.out.println("----- Edges with query: " + queryExpr + " -----");
    countE = 0;
    while (edges.hasNext()) {
      System.out.println(edges.next());
      countE++;
    }
    System.out.println("Edges found: "+ countE);
```

The preceding code example might produce output like the following:

```
----- Edges with name Beyonce AND country_str:United* -----
Edge ID 1000 from Vertex ID 1 {country:str:United States, name:str:Barack Obama,
occupation:str:44th president of United States of America, political
party:str:Democratic, religion:str:Christianity, role:str:political authority}
=[collaborates]=> Vertex ID 2 {country:str:United States, music genre:str:pop soul ,
name:str:Beyonce, role:str:singer actress} edgeKV[{weight:flo:1.0}]
Edges found: 1
```

2.6.5.3 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 Uploading a Collection's SolrCloud Configuration to Zookeeper

Before using SolrCloud text indexes on Oracle Spatial and Graph property graphs, you must upload a collection's configuration to Zookeeper. This can be done using the ZkCli tool from one of the SolrCloud cluster nodes.

A predefined collection configuration directory can be found in `dal/opg-solr-config` under the installation home. The following shows an example of how to upload the `PropertyGraph` configuration directory.

1. Copy `$ORACLE_HOME/md/property_graph/dal/opg-solr-config.zip` into the `/tmp` directory on one of the Solr cluster nodes. For example:

   ```
   scp –r $ORACLE_HOME/md/property_graph/dal/opg-solr-config.zip user@solr-node:/tmp
   ```

2. Execute following commands such as the following, using the ZkCli tool on the same node:

   ```
   cd /tmp
   unzip opg-solr-config.zip
   $SOLR_HOME/bin/zkcli.sh -zkhost 127.0.0.1:2181/solr -cmd upconfig –confname
   opgconfig -confdir /tmp/opg-solr-config
   ```

2.6.7 Updating Configuration Settings on Text Indexes for Property Graph Data

Oracle's property graph support manages manual and automatic text indexes through integration with Apache Lucene, SolrCloud, and 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. The

following code fragment creates two automatic Apache Lucene-based indexes (on vertices and edges) over an existing property graph, disables them, and re-creates them to use SolrCloud.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
      …);

// Create an automatic index using Apache Lucene.
// Specify Index Directory parameters (number of directories,
// number of connections to database, batch size, commit size,
// enable datatypes, location)
OracleIndexParameters luceneIndexParams =
     OracleIndexParameters.buildFS(4, 4, 10000, 50000, true,
             "/home/oracle/text-index ");

// Specify indexed keys
String[] indexedKeys = new String[4];
indexedKeys[0] = "name";
indexedKeys[1] = "role";
indexedKeys[2] = "religion";
indexedKeys[3] = "country";

// Create auto indexing on above properties for all vertices
opg.createKeyIndex(indexedKeys, Vertex.class, luceneIndexParams.getParameters());

// Create auto indexing on weight for all edges
opg.createKeyIndex("weight", Edge.class, luceneIndexParams.getParameters());

// Disable auto indexes to change parameters
opg.getOracleIndexManager().disableVertexAutoIndexer();
opg.getOracleIndexManager().disableEdgeAutoIndexer();


// Recreate text indexes using SolrCloud
// Specify Index Directory parameters: configuration name, Solr Server URL, Solr
Node set,
// replication factor, zookeeper timeout (secs),
// maximum number of shards per node,
// number of connections to database, batch size, commit size,
// write timeout (in secs)
String configName = "opgconfig";
String solrServerUrl = "nodea:2181/solr";
String solrNodeSet = "nodea:8983_solr,nodeb:8983_solr," +
   "nodec:8983_solr,noded:8983_solr";

int zkTimeout = 15;
int numShards = 4;
int replicationFactor = 1;
int maxShardsPerNode = 1;

OracleIndexParameters solrIndexParams =
OracleIndexParameters.buildSolr(configName,
                                solrServerUrl,
                                solrNodeSet,
                                zkTimeout,
                                numShards,
                                replicationFactor,
                                maxShardsPerNode,
                                4,
                                10000,
                                500000,
```

```
                                                    15);

// Create auto indexing on above properties for all vertices
opg.createKeyIndex(indexedKeys, Vertex.class, solrIndexParams.getParameters());

// Create auto indexing on weight for all edges
opg.createKeyIndex("weight", Edge.class, solrIndexParams.getParameters());
```

2.6.8 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 querying is an optimized solution taking advantage of the distribution of the data in the index among shards in SolrCloud (or subdirectories in Apache Lucene), so each one is queried using separate index connection. This involves multiple threads and connections to SolrCloud (or Apache Lucene) search engines to increase performance on read operations and retrieve multiple elements from the index. Note that this approach will not rank the matching results based on their score.

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 Query Using Apache Lucene
- Parallel Text Search Using SoltCloud
- Parallel Text Search Using Oracle Text

Parallel Text Query Using Apache Lucene

You can use parallel text query using Apache Lucene by calling the method `getPartitioned` in `LuceneIndex`, specifying an array of connections to set of subdirectories (`SearcherManager` objects), the key/value pair to search, and the starting subdirectory ID. Each connection needs to be linked to the appropriate subdirectory, as each subdirectory is independent of the rest of the subdirectories in the index.

The following code fragment generates an automatic text index using the Apache Lucene Search engine, and executes a parallel text query. The number of calls to the `getPartitioned` method in the `LuceneIndex` class is controlled by the total number of subdirectories and the number of connections used.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
 …);

// Create an automatic index
OracleIndexParameters indexParams
= OracleIndexParameters.buildFS(dop /* number of directories */,
dop /* number of connections
used when indexing */,
10000 /* batch size before commit*/,
500000 /* commit size before Lucene commit*/,
true /* enable datatypes */,
"./lucene-index" /* index location */);
```

```
opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name property for all vertices
System.out.println("Create automatic index on name for vertices");
opg.createKeyIndex("name", Vertex.class);

// Get the LuceneIndex object
SearcherManager[] conns = new SearcherManager[dop];
LuceneIndex<Vertex> index = (LuceneIndex<Vertex>) opg.getAutoIndex(Vertex.class);

long lCount = 0;
for (int split = 0; split < index.getTotalShards();
 split += conns.length) {
// Gets a connection object from subdirectory split to
//(split + conns.length)
for (int idx = 0; idx < conns.length; idx++) {
conns[idx] = index.getOracleSearcherManager(idx + split);
}

// Gets elements from split to split + conns.length
Iterable<Vertex>[] iterAr
= index.getPartitioned(conns /* connections */,
 "name"/* key */,
 "*" /* value */,
 true /* wildcards */,
 split /* start split ID */);

lCount = countFromIterables(iterAr); /* Consume iterables in parallel */

// Do not close the connections to the subdirectories after completion,
// because those connections are used by the LuceneIndex object itself.
}

// Count all vertices
System.out.println("Vertices found using parallel query: " + lCount);
```

Parallel Text Search Using SolrCloud

You can use parallel text query using SolrCloud by calling the method `getPartitioned` in `SolrIndex`, specifying an array of connections to SolrCloud (`CloudSolrServer` objects), the key/value pair to search, and the starting shard ID.

The following code fragment generates an automatic text index using the SolrCloud Search engine and executes a parallel text query. The number of calls to the `getPartitioned` method in the `SolrIndex` class is controlled by the total number of shards in the index and the number of connections used.

```
OraclePropertyGraph opg = OraclePropertyGraph.getInstance(…);

String configName = "opgconfig";
String solrServerUrl = args[4];//"localhost:2181/solr"
String solrNodeSet = args[5]; //"localhost:8983_solr";

int zkTimeout = 15; // zookeeper timeout in seconds
int numShards = Integer.parseInt(args[6]); // number of shards in the index
int replicationFactor = 1; // replication factor
int maxShardsPerNode = 1; // maximum number of shards per node

// Create an automatic index using SolrCloud
OracleIndexParameters indexParams =
 OracleIndexParameters.buildSolr(configName,
 solrServerUrl,
```

```
 solrNodeSet,
 zkTimeout /* zookeeper timeout in seconds */,
 numShards /* total number of shards */,
 replicationFactor /* Replication factor */,
 maxShardsPerNode /* maximum number of shardsper node*/,
 4 /* dop used for scan */,
 10000 /* batch size before commit*/,
 500000 /* commit size before SolrCloud commit*/,
 15 /* write timeout in seconds */);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name property for all vertices
System.out.println("Create automatic index on name for vertices");
opg.createKeyIndex("name", Vertex.class);

// Get the SolrIndex object
SearcherManager[] conns = new SearcherManager[dop];
SolrIndex<Vertex> index = (SolrIndex<Vertex>) opg.getAutoIndex(Vertex.class);

// Open an array of connections to handle connections to SolrCloud needed for
parallel text search
CloudSolrServer[] conns = new CloudSolrServer[dop];

for (int idx = 0; idx < conns.length; idx++) {
conns[idx] = index.getCloudSolrServer(15 /* write timeout in
secs*/);
}

// Iterate to cover all the shards 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 */,
 "*" /* value */,
 true /* wildcards */,
 split /* start split ID */);

lCount = countFromIterables(iterAr); /* Consume iterables in parallel */
}

// Close the connections to the sub-directories after completed
for (int idx = 0; idx < conns.length; idx++) {
conns[idx].shutdown();
}

// Count results
System.out.println("Vertices found using parallel query: " + lCount);
```

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("Obama" /* 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);
```

2.6.9 Using Native Query Objects on Text Indexes for Property Graph Data

Using Query objects directly is for advanced users, enabling them to take full advantage of the underlying query capabilities of the text search engine (Apache Lucene or SolrCloud). For example, you can add constraints to text searches, such as adding a boost to the matching scores and adding sorting clauses.

Using text searches with Query objects will produce an Iterable object holding all the vertices (or edges) with an attribute (or set of attributes) matching the text query while satisfying the constraints. This approach will automatically rank the results based on their matching score.

To build the clauses in the query body, you may need to consider the data type used by the key/value pair to be matched, as well as the configuration of the search engine used. For more information about building a search term, see Handling Data Types.

Using Native Query Objects with Apache Lucene

You can use native query objects using Apache Lucene by calling the method `get(Query)` in `LuceneIndex`. You can also use parallel text query with native query objects by calling the method `getPartitioned(SearcherManager[], Query, int)` in `LuceneIndex` specifying an array of connections to a set of subdirectories (`SearcherManager` objects), the Lucene query object, and the starting subdirectory ID. Each connection must be linked to the appropriate subdirectory, because each subdirectory is independent of the rest of the subdirectories in the index.

The following code fragment generates an automatic text index using Apache Lucene Search engine, creates a Lucene Query, and executes a parallel text query. The number of calls to the `getPartitioned` method in the `LuceneIndex` class is controlled by the total number of subdirectories and the number of connections used.

```
import oracle.pg.text.lucene.LuceneIndex;
import org.apache.lucene.search.*;
import org.apache.lucene.index.*;

...

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
 …);

// Create an automatic index
OracleIndexParameters indexParams = OracleIndexParameters.buildFS(dop /* number of
directories */,
dop /* number of connections
used when indexing */,
10000 /* batch size before commit*/,
500000 /* commit size before Lucene commit*/,
true /* enable datatypes */,
"./lucene-index" /* index location */);
```

```
opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name and country properties for all vertices
System.out.println("Create automatic index on name and country for vertices");
String[] indexedKeys = new String[2];
indexedKeys[0]="name";
indexedKeys[1]="country";
opg.createKeyIndex(indexedKeys, Vertex.class);

// Get the LuceneIndex object
LuceneIndex<Vertex> index = (LuceneIndex<Vertex>) opg.getAutoIndex(Vertex.class);

// Search first for Key name with property value Beyon* using only string
//data types
Term term = index.buildSearchTermObject("name", "Beyo*", String.class);
Query queryBey = new WildcardQuery(term);

// Add another condition to query all the vertices whose country is
//"United States"
String key = index.appendDatatypesSuffixToKey("country", String.class);
String value = index.appendDatatypesSuffixToValue("United States", String.class);

Query queryCountry = new PhraseQuery();
StringTokenizer st = new StringTokenizer(value);
while (st.hasMoreTokens()) {
  queryCountry.add(new Term(key, st.nextToken()));
};

//Concatenate queries
BooleanQuery bQuery = new BooleanQuery();
bQuery.add(queryBey, BooleanClause.Occur.MUST);
bQuery.add(queryCountry, BooleanClause.Occur.MUST);

long lCount = 0;
SearcherManager[] conns = new SearcherManager[dop];
for (int split = 0; split < index.getTotalShards(); split += conns.length) {
  // Gets a connection object from subdirectory split to
  //(split + conns.length). Skip the cache so we clone the connection and
  // avoid using the connection used by the index.
  for (int idx = 0; idx < conns.length; idx++) {
    conns[idx] = index.getOracleSearcherManager(idx + split,
                                      true /* skip looking in the
cache*/
);
  }

  // Gets elements from split to split + conns.length
  Iterable<Vertex>[] iterAr = index.getPartitioned(conns /* connections */,
                                                   bQuery,
                                                   split /* start split ID */);

  lCount = countFromIterables(iterAr); /* Consume iterables in parallel */

  // Do not close the connections to the sub-directories after completed,
  // as those connections are used by the index itself

}

// Count all vertices
System.out.println("Vertices found using parallel query: " + lCount);
```

Using Native Query Objects withSolrCloud

You can directly use native query objects against SolrCloud by calling the method `get(SolrQuery)` in `SolrIndex`. You can also use parallel text query with native query objects by calling the method `getPartitioned(CloudSolrServer[],SolrQuery,int)` in `SolrIndex` specifying an array of connections to SolrCloud (`CloudSolrServer` objects), the `SolrQuery` object, and the starting shard ID.

The following code fragment generates an automatic text index using the Apache SolrCloud Search engine, creates a `SolrQuery` object, and executes a parallel text query. The number of calls to the `getPartitioned` method in the `SolrIndex` class is controlled by the total number of subdirectories and the number of connections used.

```
import oracle.pg.text.solr.*;
import org.apache.solr.client.solrj.*;

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
 …);

String configName = "opgconfig";
String solrServerUrl = args[4];//"localhost:2181/solr"
String solrNodeSet = args[5]; //"localhost:8983_solr";

int zkTimeout = 15; // zookeeper timeout in seconds
int numShards = Integer.parseInt(args[6]); // number of shards in the index
int replicationFactor = 1; // replication factor
int maxShardsPerNode = 1; // maximum number of shards per node

// Create an automatic index using SolrCloud
OracleIndexParameters indexParams =
 OracleIndexParameters.buildSolr(configName,
 solrServerUrl,
 solrNodeSet,
 zkTimeout          /* zookeeper timeout in seconds */,
 numShards          /* total number of shards */,
 replicationFactor  /* Replication factor */,
 maxShardsPerNode   /* maximum number of shardsper node*/,
 4                  /* dop used for scan */,
 10000              /* batch size before commit*/,
 500000             /* commit size before SolrCloud commit*/,
 15                 /* write timeout in seconds */
);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name property for all vertices
System.out.println("Create automatic index on name and country for vertices");
String[] indexedKeys = new String[2];
indexedKeys[0]="name";
indexedKeys[1]="country";
opg.createKeyIndex(indexedKeys, Vertex.class);

// Get the SolrIndex object
SolrIndex<Vertex> index = (SolrIndex<Vertex>) opg.getAutoIndex(Vertex.class);

// Search first for Key name with property value Beyon* using only string
//data types
String szQueryStrBey = index.buildSearchTerm("name", "Beyo*", String.class);
String key = index.appendDatatypesSuffixToKey("country", String.class);
```

```
String value = index.appendDatatypesSuffixToValue("United States", String.class);

String szQueryStrCountry = key + ":" + value;
Solrquery query = new SolrQuery(szQueryStrBey + " AND " + szQueryStrCountry);

//Query using get operation
index.get(query);

// Open an array of connections to handle connections to SolrCloud needed
// for parallel text search
CloudSolrServer[] conns = new CloudSolrServer[dop];

for (int idx = 0; idx < conns.length; idx++) {
conns[idx] = index.getCloudSolrServer(15 /* write timeout in
secs*/);
}

// Iterate to cover all the shards 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 */,
 query,
 split /* start split ID */);

lCount = countFromIterables(iterAr); /* Consume iterables in parallel */
}

// Close the connections to SolCloud after completion
for (int idx = 0; idx < conns.length; idx++) {
  conns[idx].shutdown();
}

// Count results
System.out.println("Vertices found using parallel query: " + lCount);
```

2.6.10 Using Native Query Results on Text Indexes for Property Graph Data

Applying native query results directly into property graph data enables users to take full advantage of the querying capabilities of the text search engine (Apache Lucene or SolrCloud). This way, users can execute different type of queries (like Faceted queries) on the text engine and parse the retrieved results into vertices (or edges) objects.

Using text searches with Query results will produce an `Iterable` object holding all the vertices (or edges) from the given result object. This approach will automatically rank the results based on their result set order.

To execute the search queries directly into Apache Lucene or SolrCloud index, you may need to consider the data type used by the key/value pair to be matched, as well as the configuration of the search engine used. For more information about building a search term, see Handling Data Types.

- Using Native Query Results with Apache Lucene
- Using Native Query Results with SolrCloud

Using Native Query Results with Apache Lucene

You can use native query results using Apache Lucene by calling the method `get(TopDocs) in LuceneIndex`. A `TopDocs` object provides a set of

Documents matching a text search query over a specific Apache Lucene directory. `LuceneIndex` will produce an `Iterable` object holding all the vertices (or edges) from the documents found in the `TopDocs` object.

Oracle property graph text indexes that use Apache Lucene are created using multiple Apache Lucene directories. Indexed vertices and edges are spread among the directories to enhance storage scalability and query performance. If you need to execute a query against all the data in the property graph's text index, you must execute the query against each Apache Lucene directory. You can easily get the `IndexSearcher` object associated to a directory by using the API `getOracleSearcher in LuceneIndex`.

The following code fragment generates an automatic text index using Apache Lucene Search engine, creates a Lucene Query and executes it against an `IndexSearcher` object to get a `TopDocs` object. Later, an `Iterable` of vertices is created from the result object.

```
import oracle.pg.text.lucene.LuceneIndex;
import org.apache.lucene.search.*;
import org.apache.lucene.index.*;

...

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
 …);

// Create an automatic index
OracleIndexParameters indexParams = OracleIndexParameters.buildFS(dop /* number of
directories */,
dop /* number of connections
used when indexing */,
10000 /* batch size before commit*/,
500000 /* commit size before Lucene commit*/,
true /* enable datatypes */,
"./lucene-index" /* index location */);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name and country properties for all vertices
System.out.println("Create automatic index on name and country for vertices");
String[] indexedKeys = new String[2];
indexedKeys[0]="name";
indexedKeys[1]="country";
opg.createKeyIndex(indexedKeys, Vertex.class);

// Get the LuceneIndex object
LuceneIndex<Vertex> index = (LuceneIndex<Vertex>) opg.getAutoIndex(Vertex.class);

// Search first for Key name with property value Beyon* using only string
//data types
Term term = index.buildSearchTermObject("name", "Beyo*", String.class);
Query queryBey = new WildcardQuery(term);

// Add another condition to query all the vertices whose country is
//"United States"
String key = index.appendDatatypesSuffixToKey("country", String.class);
String value = index.appendDatatypesSuffixToValue("United States", String.class);

Query queryCountry = new PhraseQuery();
StringTokenizer st = new StringTokenizer(value);
while (st.hasMoreTokens()) {
```

```
  queryCountry.add(new Term(key, st.nextToken()));
};

//Concatenate queries
BooleanQuery bQuery = new BooleanQuery();
bQuery.add(queryBey, BooleanClause.Occur.MUST);
bQuery.add(queryCountry, BooleanClause.Occur.MUST);

// Get the IndexSearcher object needed to execute the query.
// The index searcher object is mapped to a single Apache Lucene directory
SearcherManager searcherMgr =
        index.getOracleSearcherManager(0, true /* skip looking in the cache*/);
IndexSearcher indexSearcher = searcherMgr.acquire();
// search for the first 1000 results in the current index directory 0
TopDocs docs = index.search(bQuery, 1000);

long lCount = 0;
Iterable<Vertex> it = index.get(docs);

while (it.hasNext()) {
  System.out.println(it.next());
  lCount++;
}
System.out.println("Vertices found: "+ lCount);
```

Using Native Query Results with SolrCloud

You can use native query results using SolrCloud by calling the method `get(QueryResponse)` in `SolrIndex`. A `QueryResponse` object provides a set of Documents matching a text search query over a specific SolrCloud collection. `SolrIndex` will produce an `Iterable` object holding all the vertices (or edges) from the documents found in the `QueryResponse` object.

The following code fragment generates an automatic text index using the Apache SolrCloud Search engine, creates a `SolrQuery` object, and executes it against a `CloudSolrServer` object to get a `QueryResponse` object. Later, an `Iterable` of vertices is created from the result object.

```
import oracle.pg.text.solr.*;
import org.apache.solr.client.solrj.*;

OraclePropertyGraph opg = OraclePropertyGraph.getInstance(
 …);

String configName = "opgconfig";
String solrServerUrl = args[4];//"localhost:2181/solr"
String solrNodeSet = args[5]; //"localhost:8983_solr";

int zkTimeout = 15; // zookeeper timeout in seconds
int numShards = Integer.parseInt(args[6]); // number of shards in the index
int replicationFactor = 1; // replication factor
int maxShardsPerNode = 1; // maximum number of shards per node

// Create an automatic index using SolrCloud
OracleIndexParameters indexParams =
 OracleIndexParameters.buildSolr(configName,
 solrServerUrl,
 solrNodeSet,
 zkTimeout          /* zookeeper timeout in seconds */,
 numShards          /* total number of shards */,
 replicationFactor  /* Replication factor */,
```

```
 maxShardsPerNode   /* maximum number of shardsper node*/,
 4                  /* dop used for scan */,
 10000              /* batch size before commit*/,
 500000             /* commit size before SolrCloud commit*/,
 15                 /* write timeout in seconds */
);

opg.setDefaultIndexParameters(indexParams);

// Create auto indexing on name property for all vertices
System.out.println("Create automatic index on name and country for vertices");
String[] indexedKeys = new String[2];
indexedKeys[0]="name";
indexedKeys[1]="country";
opg.createKeyIndex(indexedKeys, Vertex.class);

// Get the SolrIndex object
SolrIndex<Vertex> index = (SolrIndex<Vertex>) opg.getAutoIndex(Vertex.class);

// Search first for Key name with property value Beyon* using only string
//data types
String szQueryStrBey = index.buildSearchTerm("name", "Beyo*", String.class);
String key = index.appendDatatypesSuffixToKey("country", String.class);
String value = index.appendDatatypesSuffixToValue("United States", String.class);

String szQueryStrCountry = key + ":" + value;
Solrquery query = new SolrQuery(szQueryStrBey + " AND " + szQueryStrCountry);

CloudSolrServer conn = index.getCloudSolrServer(15 /* write timeout in
secs*/);

//Query using get operation
QueryResponse qr = conn.query(query, SolrRequest.METHOD.POST);
Iterable<Vertex> it = index.get(qr);

long lCount = 0;

while (it.hasNext()) {
  System.out.println(it.next());
  lCount++;
}

System.out.println("Vertices found: "+ lCount);
```

2.7 Access Control for Property Graph Data (Graph-Level and OLS)

The property graph feature in Oracle Spatial and 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/welcome1;

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 Shell with Property Graph Data

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

To start the Groovy shell, go to the `dal/groovy` directory under the installation home (`$ORACLE_HOME/md/property_graph by default` by default). For example:

```
cd $ORACLE_HOME/md/property_graph/dal/groovy/
```

Included is the script `gremlin-opg-rdbms.sh` for connecting to an Oracle database.

Note: To run some gremlin traversal examples, you must first do the following import operation:

```
import com.tinkerpop.pipes.util.structures.*;
```

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 ./gremlin-rdbms.sh

opg-rdbms> cfg =
cfg = GraphConfigBuilder.forPropertyGraphRdbms() \
.setJdbcUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl")\
.setUsername("scott").setPassword("tiger") \
.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 ./gremlin-opg-rdbms.sh
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("tiger") \
.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 Creating Property Graph Views on an RDF Graph

With Oracle Spatial and 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}]
```

2.10 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.

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.

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

2.10.1 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.

2.10.2 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.

2.10.3 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}]
```

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.

A property graph can be defined in two flat files, specifically description files for the vertices and edges.

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 2-3 Edge File Record Format

Field Number	Name	Description
1	*edge_ID*	An integer that uniquely identifies the edge

Table 2-3 (Cont.) Edge File Record Format

Field Number	Name	Description
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
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.)

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 six fields separated by commas. Each record must contain five commas to delimit all fields, whether or not they have values:

vertex_ID, key_name, value_type, value, value, value

The following table describes the fields composing a vertex file record.

Table 2-4 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 2-4 (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));`

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.)

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 2-5 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.convertRDBMSTable2OPE` and its Javadoc information are as follows:

```
/**
* conn: is an connect instance to the Oracle relational database
* rdbmsTableName: name of the RDBMS table to be converted
* eidColName is the name of an column in RDBMS table to be treated as edge ID
* lEIDOffset is the offset will be applied to the edge ID
* svidColName is the name of an column in RDBMS table to be treated as source vertex
ID of the edge
* dvidColName is the name of an column in RDBMS table to be treated as destination
vertex ID of the edge
* lVIDOffset is the offset will be applied to the vertex ID
* bHasEdgeLabelCol a Boolean flag represents if the given RDBMS table has a column
for edge labels; if true, use value of column elColName as the edge label;
otherwise, use the constant string elColName as the edge label
* elColName is the name of an column in RDBMS table to be treated as edge labels
* ctams defines how to map columns in the RDBMS table to the attributes
* dop degree of parallelism
* dcl an instance of DataConverterListener to report the progress and control the
behavior when errors happen
*/
OraclePropertyGraphUtils.convertRDBMSTable2OPE(
        Connection conn,
        String rdbmsTableName,
        String eidColName,
        long lEIDOffset,
        String svidColName,
        String dvidColName,
        long lVIDOffset,
        boolean bHasEdgeLabelCol,
        String elColName,
        ColumnToAttrMapping[] ctams,
        int dop,
        OutputStream opeOS,
        DataConverterListener dcl);
```

The following code snippet converts this table into an Oracle-defined edge file (`.ope`):

```
// location of the output file
String ope = "./EmpRelationTab.ope";
OutputStream opeOS = new FileOutputStream(ope);
// an array of ColumnToAttrMapping objects; each object defines how to map a column
in the RDBMS table to an attribute of the edge in an Oracle Property Graph.
ColumnToAttrMapping[] ctams = new ColumnToAttrMapping[1];
// map column "startDate" to attribute "since" of type Date
ctams[0] = ColumnToAttrMapping.getInstance(“startDate", “since",Date.class);
// convert RDBMS table “EmpRelationTab" into ope file “./EmpRelationTab.opv", column
“relationID" is the edge ID column, offset 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,
```

2.12 Example Python User Interface

The example Python scripts in `$ORACLE_HOME/md/property_graph/pyopg/` can used with Oracle Spatial and Graph Property Graph, and you may want to change and enhance them (or copies of them) to suit your needs.

Note:

Names do not appear in the vertex or edge files, but are provided here to simplify field references.

To invoke the user interface to run the examples, use the script `pyopg.sh`.

The examples include the following:

- Example 1: Connect to an Oracle database and perform a simple check of the number of vertices and edges. To run it:

```
sh ./pyopg.sh

connectRDBMS("connections", "jdbc:oracle:thin:@127.0.0.1:1521:orcl", "scott",
"tiger");print "vertices", countV()
print "edges", countE()
```

  In the preceding example, `mygraph` is the name of the graph stored in the Oracle database, and the remaining arguments are the connection information to access the Oracle database. They must be customized for your environment.

- Example 2: Connect to an Oracle database and run a few analytical functions. To run it:

```
connectRDBMS("connections", "jdbc:oracle:thin:@127.0.0.1:1521:orcl", "scott",
"tiger");
print "vertices", countV()
print "edges", countE()

import pprint

analyzer = analyst()
print "# triangles in the graph", analyzer.countTriangles()

graph_communities = [{"commid":i.getName(),"size":i.size()} for i in
analyzer.communities().iterator()]

import pandas as pd
import numpy as np

community_frame = pd.DataFrame(graph_communities)
community_frame[:5]

import matplotlib as mpl
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16,12));
community_frame["size"].plot(kind="bar", title="Communities and Sizes")
ax.set_xticklabels(community_frame.index);
plt.show()
```

The preceding example connects to an Oracle database, prints basic information about the vertices and edges, get an in memory analyst, computes the number of triangles, performs community detection, and finally plots out in a bar chart communities and their sizes.

For detailed information about this example Python interface, see the following directory under the installation home:

```
$ORACLE_HOME/md/property_graph/pyopg/doc/
```

3

Using the In-Memory Analyst (PGX)

The in-memory analyst feature of Oracle Spatial and Graph supports a set of analytical functions.

This chapter provides examples using the in-memory analyst (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:

Reading a Graph into Memory
: This topic provides an example of reading graph interactively into memory using the shell interface.

Reading Custom Graph Data
: You can read your own custom graph data.

Storing Graph Data on Disk
: After reading a graph into memory using either Java or the Shell, you can store it on disk in different formats. You can then use the stored graph data as input to the in-memory analyst at a later time.

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

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 underlying property graph in the database.

Deploying to Jetty
: You can deploy the in-memory analyst to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic Server. This example shows how to deploy the in-memory analyst as a web application with Eclipse Jetty.

Deploying to Apache Tomcat
: You can deploy the in-memory analyst to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic. This example shows how to deploy In-Memory Analytics as a web application with Apache Tomcat.

Deploying to Oracle WebLogic Server
: You can deploy the in-memory analysts to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic Server. This example shows how to deploy the in-memory analyst 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 Hadoop cluster -- or on a client system without Hadoop as a web application on Eclipse Jetty, Apache Tomcat, or Oracle WebLogic Server -- you can connect to the in-memory analyst server.

Managing Property Graph Snapshots
: Oracle Spatial and Graph Property Graph lets you manage property graph snapshots.

3.1 Reading a Graph into Memory

This topic provides an example of reading graph interactively into memory using the shell interface.

These are the major steps:

Connecting to an In-Memory Analyst Server Instance

Using the Shell Help

Providing Graph Metadata in a Configuration File

Reading Graph Data into Memory

3.1.1 Connecting to an In-Memory Analyst Server Instance

To start the in-memory analyst shell:

1. Open a terminal session on the system where property graph support is installed.
2. Either start a local (embedded) in-memory analyst instance or connect to a remote in-memory analyst instance

   - Java example of starting a local (embedded) instance:

   ```
   import java.util.Map;
   import java.util.HashMap;
   import oracle.pgx.api.*;
   import oracle.pgx.config.PgxConfig.Field;

   String url = Pgx.EMBEDDED_URL; // local JVM
   ServerInstance instance = Pgx.getInstance(url);
   instance.startEngine(); // will use default configuration
   PgxSession session = instance.createSession("test");
   ```

   - Java example of connecting to a remote instance:

   ```
   import java.util.Map;
   import java.util.HashMap;
   import oracle.pgx.api.*;
   import oracle.pgx.config.PgxConfig.Field;

   String url = "http://my-server.com:8080/pgx" // replace with base URL of
   your setup
   ServerInstance instance = Pgx.getInstance(url);
   PgxSession session = instance.createSession("test");
   ```

3. In the shell, enter the following commands, but select only one of the commands to start or connect to the desired type of instance:

```
export PGX_HOME=$ORACLE_HOME/md/property_graph/pgx
cd $PGX_HOME
./bin/pgx --help
./bin/pgx --version

# start embedded shell
./bin/pgx

# start remote shell
./bin/pgx --base_url http://my-server.com:8080/pgx
```

   For the embedded shell, the output should be similar to the following:

```
10:43:46,666 [main] INFO Ctrl$2 - >>> PGX engine running.
pgx>
```

4. Optionally, show the predefined variables:

```
pgx> instance
==> PGX Server Instance running on embedded mode
pgx> session
==> PGX session pgxShell registered at PGX Server Instance running on embedded
mode
pgx> analyst
==> Analyst for PGX session pgxShell registered at PGX Server Instance running on
embedded mode
pgx>
```

   Examples in some other topics assume that the instance and session variables have been set as shown here.

If the in-memory analyst software is installed correctly, you will see an engine-running log message and the in-memory analyst shell prompt (`pgx>`):

The variables `instance`, `session`, and `analyst` are ready to use.

In the preceding example in this topic, the shell started a local instance because the `pgx` command did not specify a remote URL.

3.1.2 Using the Shell Help

The in-memory analyst shell provides a help system, which you access using the `:help` command.

3.1.3 Providing Graph Metadata in a Configuration File

This topic presents an example of providing graph metadata in a configuration file. Follow these steps oo create a directory and some example files.

1. Create a directory to hold the example files that you will create. For example:

```
mkdir -p ${ORACLE_HOME}/md/property_graph/examples/pgx/graphs/
```

2. In that directory, create a text file named `sample.adj.json` with the following content for the graph configuration file. This configuration file describes how the in-memory analyst reads the graph.

```
{
  "uri": "sample.adj",
  "format": "adj_list",
  "node_props": [{
    "name": "prop",
    "type": "integer"
  }],
  "edge_props": [{
    "name": "cost",
    "type": "double"
  }],
  "separator": " "
}
```

3. In the same directory, create a text file named `sample.adj` with the following content for the graph data:

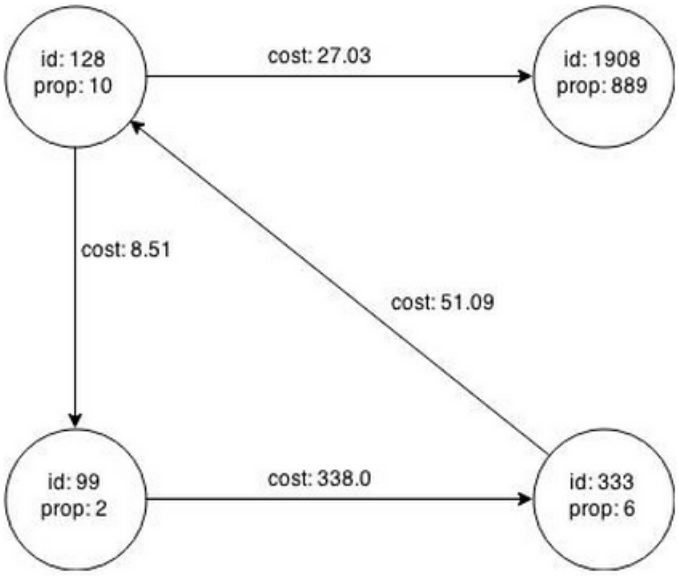
```
128 10 1908 27.03 99 8.51
99 2 333 338.0
1908 889
333 6 128 51.09
```

In the configuration file, the `uri` field provides the location of the graph data. This path resolves relative to the parent directory of the configuration file. When the in-memory analyst loads the graph, it searches for a file named `sample.adj` containing the graph data.

The other fields in the configuration file indicate that the graph data is provided in adjacency list format, and consists of one node property of type `integer` and one edge property of type `double`.

The following figure shows a property graph created from the data:

Figure 3-1 Property Graph Rendered by sample.adj Data



3.1.4 Reading Graph Data into Memory

To read a graph into memory, you must pass the following information:

- The path to the graph configuration file that specifies the graph metadata

- A unique alphanumeric name that you can use to reference the graph

  An error results if you previously loaded a different graph with the same name.

Example: Using the Shell to Read a Graph

```
pgx> graph = session.readGraphWithProperties("<ORACLE_HOME>/md/property_graph/
examples/pgx/graphs/sample.adj.json", "sample");
==> PGX Graph named sample bound to PGX session pgxShell ...
pgx> graph.getNumVertices()
==> 4
```

Example: Using Java to Read a Graph

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

PgxGraph graph = session.readGraphWithProperties("<ORACLE_HOME>/md/property_graph/
examples/pgx/graphs/sample.adj.json");
```

The following topics contain additional examples of reading a property graph into memory:

Read a Graph Stored in Oracle Database into Memory

Read a Graph Stored in the Local File System into Memory

3.1.4.1 Read a Graph Stored in Oracle Database into Memory

To read a property graph stored in Oracle Database, you can create a JSON based configuration file as follows. Note that the hosts, store name, graph name, and other information must be customized for your own setup.

```
% cat /tmp/my_graph_oracle.json
{"loading":{"load_edge_label":false},
"vertex_props":[
{"default":"default_name","name":"name","type":"string"}
],
"password":"<YOUR_PASSWORD>",
"db_engine":"RDBMS",
"max_num_connections":8,
"username":"scott",
"error_handling":
{},"format":"pg","jdbc_url":"jdbc:oracle:thin:@127.0.0.1:1521:<SID>",
"name":"connections",
"edge_props":[
{"default":"1000000","name":"cost","type":"double"}
]
}
```

Then, read the configuration file into memory. The following example snippet reads the file into memory, generates an undirected graph (named `U`) from the original data, and counts the number of triangles.

```
pgx> g = session.readGraphWithProperties("/tmp/my_graph_oracle.json", "connections")
pgx> analyst.countTriangles(g, false)
==> 8
```

3.1.4.2 Read a Graph Stored in the Local File System into Memory

The following command uses the configuration file from Providing Graph Metadata in a Configuration File and the name `my-graph`:

```
pgx> g = session.readGraphWithProperties<ORACLE_HOME>/md/property_graph/examples/pgx/
graphs/sample.adj.json", "my-graph")
```

3.2 Reading Custom Graph Data

You can read your own custom graph data.

This example creates a graph, alters it, and shows how to read it properly. This graph uses the adjacency list format, but the in-memory analyst supports several graph formats.

The main steps are:

Creating a Simple Graph File

Adding a Vertex Property

Using Strings as Vertex Identifiers

Adding an Edge Property

3.2.1 Creating a Simple Graph File

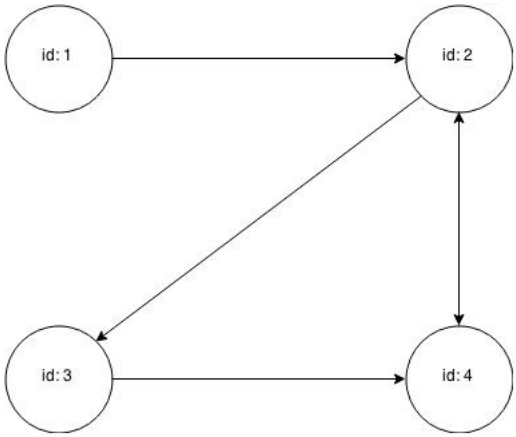
This example creates a small, simple graph in adjacency list format with no vertex or edge properties. Each line contains the vertex (node) ID, followed by the vertex IDs to which its outgoing edges point:

```
1 2
2 3 4
3 4
4 2
```

In this list, a single space separates the individual tokens. The in-memory analyst supports other separators, which you can specify in the graph configuration file.

The following figure shows the data rendered as a property graph with 4 vertices and 5 edges. (There are two edges between vertex 2 and vertex 4, each pointing in a direction opposite form the other.)

Figure 3-2 Simple Custom Property Graph



Reading a graph into the in-memory analyst requires a graph configuration. You can provide the graph configuration using either of these methods:

- Write the configuration settings in JSON format into a file

- Using a Java `GraphConfigBuilder` object.

The following examples show both methods.

JSON Configuration

```
{
    "uri": "graph.adj",
    "format":"adj_list",
    "separator":" "
}
```

Java Configuration

```
import oracle.pgx.config.FileGraphConfig;
import oracle.pgx.config.Format;
import oracle.pgx.config.GraphConfigBuilder;
FileGraphConfig config = GraphConfigBuilder
   .forFileFormat(Format.ADJ_LIST)
   .setUri("graph.adj")
   .setSeparator(" ")
   .build();
```

3.2.2 Adding a Vertex Property

The graph in Creating a Simple Graph File consists of vertices and edges, without vertex or edge properties. Vertex properties are positioned directly after the source vertex ID in each line. The graph data would look like this if you added a `double` vertex (node) property with values 0.1, 2.0, 0.3, and 4.56789 to the graph:

```
1 0.1 2
2 2.0 3 4
3 0.3 4
4 4.56789 2
```

Note:

The in-memory analyst supports only homogeneous graphs, in which all vertices have the same number and type of properties.

For the in-memory analyst to read the modified data file, you must add a vertex (node) property in the configuration file or the builder code. The following examples provide a descriptive name for the property and set the type to `double`.

JSON Configuration

```
{
    "uri": "graph.adj",
    "format":"adj_list",
    "separator":" ",
    "node_props":[{
        "name":"double-prop",
        "type":"double"
    }]
}
```

Java Configuration

```
import oracle.pgx.common.types.PropertyType;
import oracle.pgx.config.FileGraphConfig;
import oracle.pgx.config.Format;
```

```
import oracle.pgx.config.GraphConfigBuilder;

FileGraphConfig config = GraphConfigBuilder.forFileFormat(Format.ADJ_LIST)
    .setUri("graph.adj")
    .setSeparator(" ")
    .addNodeProperty("double-prop", PropertyType.DOUBLE)
    .build();
```

3.2.3 Using Strings as Vertex Identifiers

The previous examples used `integer` vertex (node) IDs. The default in In-Memory Analytics is `integer` vertex IDs, but you can define a graph to use `string` vertex IDs instead.

This data file uses "node 1", "node 2", and so forth instead of just the digit:

```
"node 1" 0.1 "node 2"
"node 2" 2.0 "node 3" "node 4"
"node 3" 0.3 "node 4"
"node 4" 4.56789 "node 2"
```

Again, you must modify the graph configuration to match the data file:

JSON Configuration

```
{
    "uri": "graph.adj",
    "format":"adj_list",
    "separator":" ",
    "node_props":[{
        "name":"double-prop",
        "type":"double"
    }],
    "node_id_type":"string"
}
```

Java Configuration

```
import oracle.pgx.common.types.IdType;
import oracle.pgx.common.types.PropertyType;
import oracle.pgx.config.FileGraphConfig;
import oracle.pgx.config.Format;
import oracle.pgx.config.GraphConfigBuilder;

FileGraphConfig config = GraphConfigBuilder.forFileFormat(Format.ADJ_LIST)
    .setUri("graph.adj")
    .setSeparator(" ")
    .addNodeProperty("double-prop", PropertyType.DOUBLE)
    .setNodeIdType(IdType.STRING)
    .build();
```

Note:

`string` vertex IDs consume much more memory than `integer` vertex IDs.

Any single or double quotes inside the string must be escaped with a backslash (\).

Newlines (\n) inside strings are not supported.

3.2.4 Adding an Edge Property

This example adds an edge property of type `string` to the graph. The edge properties are positioned after the destination vertex (node) ID.

```
"node1" 0.1 "node2" "edge_prop_1_2"
"node2" 2.0 "node3" "edge_prop_2_3" "node4" "edge_prop_2_4"
"node3" 0.3 "node4" "edge_prop_3_4"
"node4" 4.56789 "node2" "edge_prop_4_2"
```

The graph configuration must match the data file:

JSON Configuration

```
{
    "uri": "graph.adj",
    "format":"adj_list",
    "separator":" ",
    "node_props":[{
        "name":"double-prop",
        "type":"double"
    }],
    "node_id_type":"string",
     "edge_props":[{
        "name":"edge-prop",
        "type":"string"
    }]
}
```

Java Configuration

```
import oracle.pgx.common.types.IdType;
import oracle.pgx.common.types.PropertyType;
import oracle.pgx.config.FileGraphConfig;
import oracle.pgx.config.Format;
import oracle.pgx.config.GraphConfigBuilder;

FileGraphConfig config = GraphConfigBuilder.forFileFormat(Format.ADJ_LIST)
    .setUri("graph.adj")
    .setSeparator(" ")
    .addNodeProperty("double-prop", PropertyType.DOUBLE)
    .setNodeIdType(IdType.STRING)
    .addEdgeProperty("edge-prop", PropertyType.STRING)
    .build();
```

3.3 Storing Graph Data on Disk

After reading a graph into memory using either Java or the Shell, you can store it on disk in different formats. You can then use the stored graph data as input to the in-memory analyst at a later time.

Storing graphs over HTTP/REST is currently not supported.

The options include:

Storing the Results of Analysis in a Vertex Property

Storing a Graph in Edge-List Format on Disk

3.3.1 Storing the Results of Analysis in a Vertex Property

This example reads a graph into memory and analyzes it using the Pagerank algorithm. This analysis creates a new vertex property to store the PageRank values.

Using the Shell to Run PageRank

```
pgx> g = session.readGraphWithProperties("<ORACLE_HOME>/md/property_graph/
examples/pgx/graphs/sample.adj.json", "my-graph")
==> ...
pgx> rank = analyst.pagerank(g, 0.001, 0.85, 100)
```

Using Java to Run PageRank

```
PgxGraph g = session.readGraphWithProperties("<ORACLE_HOME>/md /property_graph/
examples/pgx/graphs/sample.adj.json", "my-graph");
VertexProperty<Integer, Double> rank = session.createAnalyst().pagerank(g, 0.001,
0.85, 100);
```

3.3.2 Storing a Graph in Edge-List Format on Disk

This example stores the graph, the result of the Pagerank analysis, and all original edge properties as a file in edge-list format on disk.

To store a graph, you must specify:

- The graph format
- A path where the file will be stored
- The properties to be stored. Specify VertexProperty.ALL or EdgeProperty.ALL to store all properties, or VertexProperty.NONE or EdgePropery.NONE to store no properties. To specify individual properties, pass in the VertexProperty or EdgeProperty objects you want to store.
- A flag that indicates whether to overwrite an existing file with the same name

The following examples store the graph data in `/tmp/sample_pagerank.elist`, with the `/tmp/sample_pagerank.elist.json` configuration file. The return value is the graph configuration for the stored file. You can use it to read the graph again.

Using the Shell to Store a Graph

```
pgx> config = g.store(Format.EDGE_LIST, "/tmp/sample_pagerank.elist", [rank],
EdgeProperty.ALL, false)
==> {"uri":"/tmp/sample_pagerank.elist","edge_props":
[{"type":"double","name":"cost"}],"vertex_id_type":"integer","loading":
{},"format":"edge_list","attributes":{},"vertex_props":
[{"type":"double","name":"pagerank"}],"error_handling":{}}
```

Using Java to Store a Graph

```
import oracle.pgx.api.*;
import oracle.pgx.config.*;

FileGraphConfig config = g.store(Format.EDGE_LIST, "/tmp/sample_pagerank.elist",
Collections.singletonList(rank), EdgeProperty.ALL, false);
```

3.4 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.

3.4.1 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);
```

3.4.2 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

```
pgx> 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`.

3.4.3 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 session private. You can create transient properties by calling `createVertexProperty` and `createEdgeProperty` on `PgxGraph` objects.

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

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

3.5 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 Graph Data into Memory.

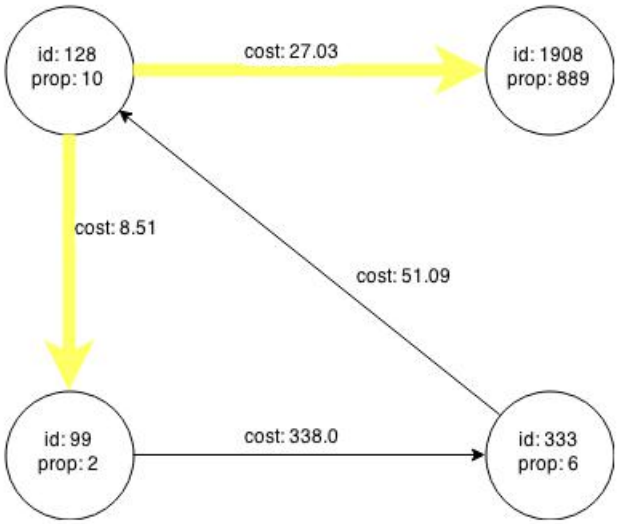
About Filter Expressions

3.5.1 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.

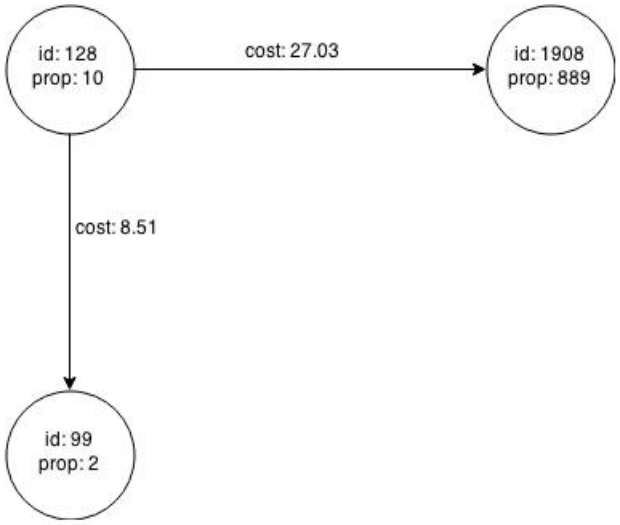
Consider the graph in Providing Graph Metadata in a Configuration File, which consists of four vertices (nodes) and four edges. For an edge to match the filter expression `src.prop == 10`, the source vertex `prop` property must equal 10. Two edges match that filter expression, as shown in the following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

Figure 3-3 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 3-4 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.

3.5.2 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"));
```

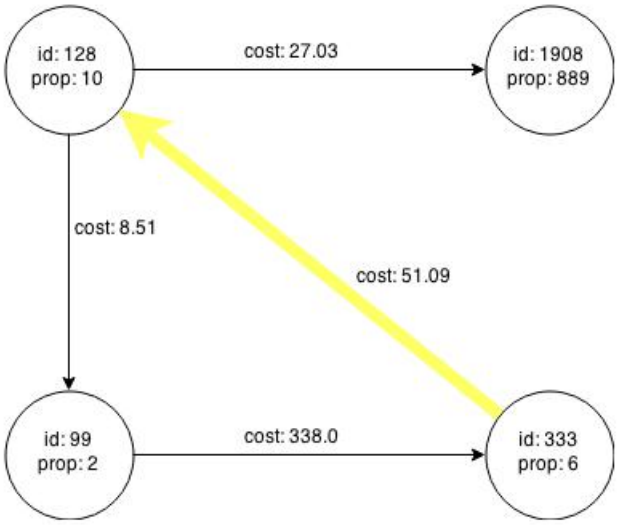
3.5.3 Using a Complex Filter to Create a Subgraph

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
```

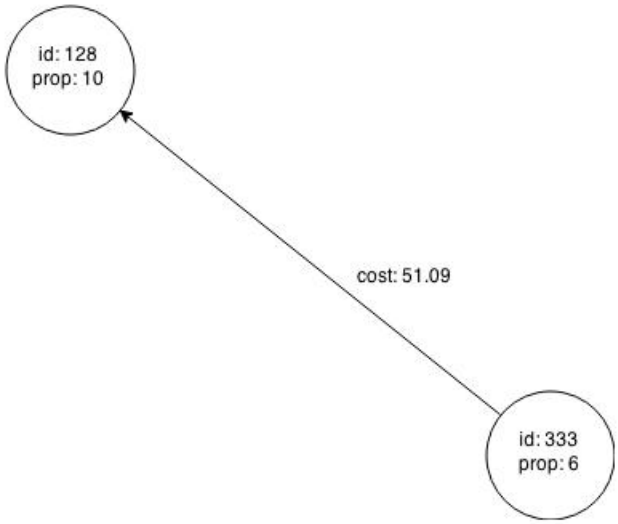
One edge in the sample graph matches this filter expression, as shown in the following figure.

Figure 3-5 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.

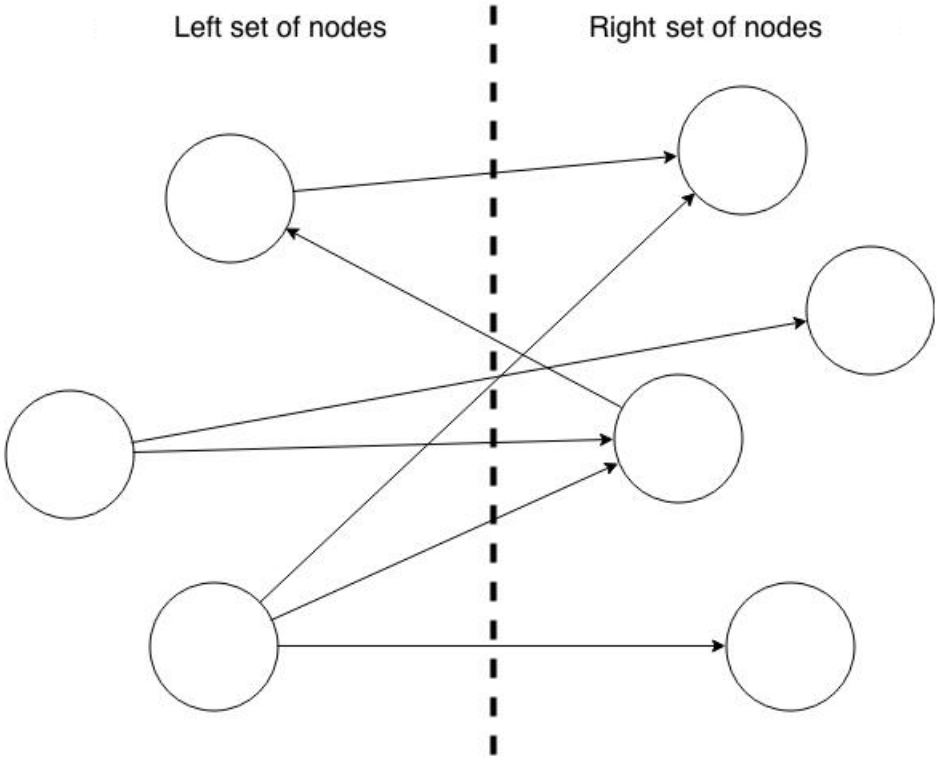
Figure 3-6 Graph Created by the outDegree Filter



3.5.4 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 created in Providing Graph Metadata in a Configuration File. They create a vertex collection and fill it with the vertices for the left side.

Using the Shell to Create a Bipartite Subgraph

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

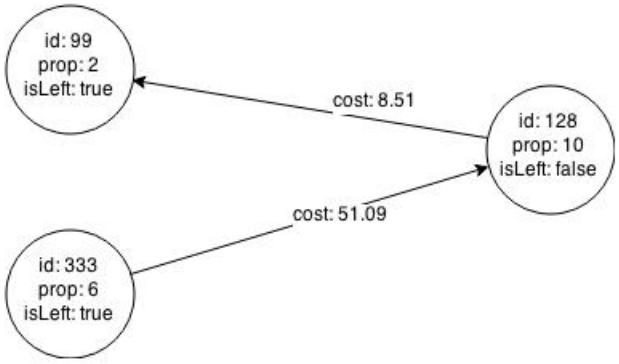
Using Java to Create a Bipartite Subgraph

```
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.

3.6 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 underlying property graph in the database.

3.6.1 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
}
```

3.6.2 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": "tiger",
  "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("tiger")
  .setName("my_graph")
  .addVertexProperty("prop", PropertyType.INTEGER)
  .addEdgeProperty("cost", PropertyType.DOUBLE)
  .setAutoRefresh(true)
  .setUpdateIntervalSec(60)
  .build();
```

3.6.3 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.

```
pgx> 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.

3.6.4 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:

```
pgx> session.getAvailableSnapshots(G)
==> GraphMetaData [getNumVertices()=4, getNumEdges()=4, memoryMb=0,
dataSourceVersion=1453315103000, creationRequestTimestamp=1453315122669 (2016-01-20
10:38:42.669), creationTimestamp=1453315122685 (2016-01-20 10:38:42.685),
vertexIdType=integer, edgeIdType=long]
==> GraphMetaData [getNumVertices()=5, getNumEdges()=5, memoryMb=3,
dataSourceVersion=1452083654000, creationRequestTimestamp=1453314938744 (2016-01-20
10:35:38.744), creationTimestamp=1453314938833 (2016-01-20 10:35:38.833),
vertexIdType=integer, edgeIdType=long]
```

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:

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

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

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

```
pgx> 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:

```
pgx> 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.

3.6.5 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
  }
}
```

3.7 Deploying to Jetty

You can deploy the in-memory analyst to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic Server. This example shows how to deploy the in-memory analyst as a web application with Eclipse Jetty.

1. Copy the in-memory analyst web application archive (WAR) file into the Jetty `webapps` directory:

   ```
   export PGX_HOME=$ORACLE_HOME/md/property_graph/pgx
   cp $PGX_HOME/server/webapp/pgx-webapp-<version>.war $JETTY_HOME/webapps/pgx.war
   ```

   Note that there are two `.war` files in the `pgx/server` directory. The `.war` file with a name containing "wls" is for WebLogic Server. The other war file is for other J2EE containers, including Jetty.

2. Set up a security realm within Jetty that specifies where it can find the user names and passwords. To add the most basic security realm, which reads the credentials from a file, add this snippet to `$JETTY_HOME/etc/jetty.xml`:

   ```
   <Call name="addBean">
     <Arg>
       <New class="org.eclipse.jetty.security.HashLoginService">
         <Set name="name">PGX-Realm</Set>
                 <Set name="config">
                   etc/realm.properties
                 </Set>
         <Set name="refreshInterval">0</Set>
       </New>
     </Arg>
   </Call>
   ```

This snippet instructs Jetty to use the simplest, in-memory login service it supports, the `HashLoginService`. This service uses a configuration file that stores the user names, passwords, and roles.

3. Add the users to `$JETTY_HOME/etc/realm.properties` in the following format:

   ```
   username: password, role
   ```

   For example, this line adds user `SCOTT`, with password `TIGER` and the `USER` role.

   ```
   scott: tiger, USER
   ```

4. Ensure that port 8080 is not already in use, and then start Jetty:

   ```
   cd $JETTY_HOME
   java -jar start.jar
   ```

5. Verify that Jetty is working, using the appropriate credentials for your installation:

   ```
   cd $PGX_HOME
   ./bin/pgx --base_url http://scott:tiger@localhost:8080/pgx
   ```

6. (Optional) Modify the in-memory analyst configuration files.

   The configuration file (`pgx.conf`) and the logging parameters (`log4j.xml`) for the in-memory analyst engine are in the WAR file under `WEB-INF/classes`. Restart the server to enable the changes.

About the Authentication Mechanism

Related Topics:

The Jetty documentation for configuration and use

3.7.1 About the Authentication Mechanism

The in-memory analyst web deployment uses `BASIC Auth` by default. You should change to a more secure authentication mechanism for a production deployment.

To change the authentication mechanism, modify the `security-constraint` element of the `web.xml` deployment descriptor in the web application archive (WAR) file.

3.8 Deploying to Apache Tomcat

You can deploy the in-memory analyst to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic. This example shows how to deploy In-Memory Analytics as a web application with Apache Tomcat.

The in-memory analyst ships with `BASIC Auth` enabled, which requires a security realm. Tomcat supports many different types of realms. This example configures the simplest one, `MemoryRealm`. See the Tomcat Realm Configuration How-to for information about the other types.

1. Copy the in-memory analyst WAR file into the Tomcat `webapps` directory. For example:

   ```
   cp $PGX_HOME/webapp/pgx-webapp-<VERSION>.war $CATALINA_HOME/webapps/pgx.war
   ```

2. Open `$CATALINA_HOME/conf/server.xml` in an editor and add the following realm class declaration under the `<Engine>` element:

```
<Realm className="org.apache.catalina.realm.MemoryRealm" />
```

3. Open `CATALINA_HOME/conf/tomcat-users.xml` in an editor and define a user for the `USER` role. Replace `scott` and `tiger` in this example with an appropriate user name and password:

```
<role rolename="USER" />
<user username="scott" password="tiger" roles="USER" />
```

4. Ensure that port 8080 is not already in use.

5. Start Tomcat:

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

6. Verify that Tomcat is working:

```
cd $PGX_HOME
./bin/pgx --base_url http://scott:tiger@localhost:8080/pgx
```

Note:

Oracle recommends `BASIC Auth` only for testing. Use stronger authentication mechanisms for all other types of deployments.

Related Topics:

The Tomcat documentation (select desired version)

3.9 Deploying to Oracle WebLogic Server

You can deploy the in-memory analysts to Eclipse Jetty, Apache Tomcat, or Oracle WebLogic Server. This example shows how to deploy the in-memory analyst as a web application with Oracle WebLogic Server.

Installing Oracle WebLogic Server

Deploying the In-Memory Analyst

Verifying That the Server Works

3.9.1 Installing Oracle WebLogic Server

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

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

3.9.2 Deploying the In-Memory Analyst

To deploy the in-memory analyst to Oracle WebLogic, use commands like the following. Substitute your administrative credentials and WAR file for the values shown in this example:

```
. $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 $PGX_HOME/server/pgx-webapp-wls.war
```

If the script runs successfully, you will see a message like this one:

```
Target state: deploy completed on Server myserver
```

3.9.3 Verifying That the Server Works

Verify that you can connect to the server by entering a command in the following format:

```
$PGX_HOME/bin/pgx --base_url http://scott:<password>@localhost:7001/pgx
```

3.10 Connecting to the In-Memory Analyst Server

After the property graph in-memory analyst is installed in a Hadoop cluster -- or on a client system without Hadoop as a web application on Eclipse Jetty, 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 an HTTP Request

3.10.1 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:tiger@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/pgx --base_url http://scott:tiger@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

3.10.1.1 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:

```
pgx> :loglevel oracle.pgx DEBUG
===> log level of oracle.pgx logger set to DEBUG
pgx> session.readGraphWithProperties("sample_http.adj.json", "sample")
10:24:25,056 [main] DEBUG RemoteUtils - Requesting POST http://scott:tiger@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:tiger@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},"graphName":"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.

3.10.2 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:@127.0.0.1:15
21:orcl")
   .setUsername("scott").setPassword("tiger")  .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:tiger@hostname:port/
pgx");
PgxSession session = remoteInstance.createSession("my-session");

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

3.10.3 Connecting with an HTTP Request

The in-memory analyst shell uses HTTP requests to communicate with the in-memory analyst server. You can use the same HTTP endpoints directly or use them to write your own client library.

This example uses HTTP to call `create session`:

```
HTTP POST 'http://scott:tiger@localhost:8080/pgx/core/session' with payload
'{"source":"shell"}'
Response: {"sessionId":"session-shell-42v3b9n7"}
```

The call to `create session` returns a session identifier. Most HTTP calls return an in-memory analyst UUID, which identifies the resource that holds the result of the request. Many in-memory analyst requests take a while to complete, but you can obtain a handle to the result immediately. Using that handle, an `HTTP GET` call to a special endpoint provides the result of the request (or block, if the request is not complete).

Most interactions with the in-memory analyst with HTTP look like this example:

```
// any request, with some payload
HTTP POST 'http://scott:tiger@localhost:8080/pgx/core/session/session-shell-42v3b9n7/
```

```
graph' with payload '{"graphName":"sample","graphConfig":{"edge_props":
[{"type":"double","name":"cost"}],"format":"adj_list","separator":" ","node_props":
[{"type":"integer","name":"prop"}],"uri":"http://path.to.some.server/pgx/
sample.adj"}}'
Response: {"futureId":"15fc72e9-42e9-4527-9a31-bd20eb0adafb"}

// get the result using the in-memory analyst future UUID.
HTTP GET 'http://scott:tiger@localhost:8080/pgx/future/session/session-
shell-42v3b9n7/result/15fc72e9-42e9-4527-9a31-bd20eb0adafb'
Response: {"stats":{"loadingTimeMillis":0,"estimatedMemoryMegabytes":0,"numNodes":
4,"numEdges":4},"graphName":"sample","nodeProperties":
{"prop":"integer"},"edgeProperties":{"cost":"double"}}
```

3.11 Managing Property Graph Snapshots

Oracle Spatial and Graph Property Graph lets you manage property graph snapshots.

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.

Note:

Managing property graph snapshots is intended for advanced users.

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
```

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.

Property Graph Query Language (PGQL)
: PGQL is a SQL-like query language for property graph data structures that consist of *nodes* that are connected to other nodes by *edges*, each of which can have key-value pairs (properties) associated with them.

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        Barack Obama
     1 occupation  44th president of United States of America
     ...
```

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 unlike the text index capabilities provided by either Apache Lucene or Apache SolrCloud, 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 "Obama".

```
SQL> select vid, k, t, v
       from connectionsVT$
      where t=1
        and contains(v, 'Obama', 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  Barack Obama

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",'Obama',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 searchs 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.

SQL-based analytics are especially useful if:

- The computation is not bound by the physical memory.
- There is no need to move the graph data outside the database.

- Computation is easily done against the latest snapshot of a property graph.
- There is no need to 'sync up' (synchronize) data inside and outside the database.

However, when a graph (or a subgraph) fits in memory, then running analytics provided by the in-memory analyst usually provides better better performance than using SQL-based analytics. Because many of the analytics implementation require using intermediate data structures, most SQL- (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

Example 4-18 Shortest Path

Consider shortest path, for example. Internally, Oracle Database uses the bidirectional Dijkstra algorithm. The following code snippet shows an end-to-end flow.

```
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 inmemory or advanced compression, you can pre-create the preceding working table and pass the name in.

Example 4-19 Create Working Table and Perform Analytics

The following statements first create a working table with the same column structure and basic compression enabled, then pass it to the computation.

```
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 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.

```
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.
```

4.7 Property Graph Query Language (PGQL)

PGQL is a SQL-like query language for property graph data structures that consist of *nodes* that are connected to other nodes 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.

Topology Constraints with PGQL

Constraints are Directional with PGQL

Vertex and Edge Labels with PGQL

Regular Path Queries with PGQL

Aggregation and Sorting with PGQL

Related Topics:

PGQL Specification

4.7.1 Topology Constraints with PGQL

Pattern matching is done using *topology constraints*, which describe a pattern of connections between nodes in the graph, Value constraints (similar to their SQL equivalents) let you further constrain matches by specifying properties that those connections and nodes must have.

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 host1.id(), host2.id(), host3.id()
WHERE                                                              /* choose what
to return */
    (host1) -[c1 WITH toPort = 22 and opened = true]-> (host2)      /* topology must
match this pattern */
      -[connection2 WITH toPort = 22 and opened = true]-> (host3),
    connection1.bytes > 300,                                       /* meaningful
amount of data was exchanged */
    connection2.bytes > 300,
    connection1.start < connection2.start,                         /* second
```

```
connection within time-frame of first */
    connection2.start + connection2.duration < connection1.start +
connection1.duration
GROUP BY host1.id(), host2.id(), host3.id()                          /* aggregate
multiple matching connections */
ORDER BY DESC(connection1.when)                                     /* reverse sort
chronologically */
```

4.7.2 Constraints are Directional with PGQL

A topological constraint 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
WHERE                              /* note the arrow directions below */
  (p1:person WITH name = 'April') -[:likes]-> (friend) <-[:likes]- (p2:person WITH
name = 'Chris'),
  friend.dob > p1.dob AND friend.dob > p2.dob
ORDER BY friend.dob DESC
```

4.7.3 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 WHERE (p:person) -[e:likes]-> (m:movie WITH title='Star Wars'),
  (p) -[e:likes]-> (m:movie WITH title='Avatar')
```

4.7.4 Regular Path Queries with PGQL

Regular path queries allow a pattern to be reused. The following example finds all of the common ancestors of Mario and Luigi.

```
PATH has_parent := () -[:has_father|has_mother]-> ()
SELECT ancestor.name
WHERE
  (:Person WITH name = 'Mario') -/:has_parent*/-> (ancestor:Person),
  (:Person WITH name = 'Luigi') -/:has_parent*/-> (ancestor)
```

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.

4.7.5 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.

5

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.

5.1 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);
```

5.2 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');
```

5.3 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');
```

5.4 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');
```

5.5 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;
/
```

5.6 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;
/
```

5.7 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;
/
```

5.8 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;
/
```

5.9 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);
```

5.10 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');
```

5.11 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');
```

5.12 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');
```

5.13 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);
```

5.14 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);
```

5.15 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);
```

5.16 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);
```

5.17 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');
```

5.18 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');
```

5.19 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);
```

5.20 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;
/
```

5.21 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
```

5.22 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
```

5.23 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;
/
```

5.24 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);
```

5.25 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;
/
```

5.26 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
```

5.27 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);
```

5.28 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"
```

5.29 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
```

5.30 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))
```

5.31 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,-83.434.3,-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))
```

5.32 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)
```

5.33 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)
```

5.34 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)
```

5.35 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)
```

5.36 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)
```

5.37 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.

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
```

5.38 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
```

5.39 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))
```

5.40 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))
```

5.41 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
```

5.42 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');
```

5.43 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');
```

5.44 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
```

5.45 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);
```

5.46 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"
```

5.47 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();
```

5.48 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');
```

5.49 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
```

5.50 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;
/
```

5.51 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"
```

6

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.

6.1 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');
```

Index

P

R

S

T

V