Oracle® Spatial

Topology and Network Data Models Developer's Guide

11*g* Release 2 (11.2)

E11831-01

July 2009

Provides usage and reference information about the topology data model and network data model capabilities of Oracle Spatial.

ORACLE®

Oracle Spatial Topology and Network Data Models Developer's Guide, 11*g* Release 2 (11.2)

E11831-01

Primary Author: Chuck Murray

Contributors: Ning An, Betsy George, Huiling Gong, Siva Ravada, Jack Wang

Contents

2 Editing Topologies

3 SDO_TOPO Package Subprograms

4 SDO_TOPO_MAP Package Subprograms

6 SDO_NET Package Subprograms

7 SDO_NET_MEM Package Subprograms

Index

List of Examples

List of Figures

List of Tables

Preface

Oracle Spatial Topology and Network Data Models Developer's Guide provides usage and reference information about the topology data model and network data model of Oracle Spatial, which is often referred to as just Spatial.

Audience

This guide is intended for those who need to use the Spatial topology or network data model to work with data about nodes, edges, and faces in a topology or nodes, links, and paths in a network.

You are assumed to be familiar with the main Spatial concepts, data types, and operations, as documented in *Oracle Spatial Developer's Guide*.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request

process. Information about TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html, and a list of phone numbers is available at http://www.fcc.gov/cgb/dro/trsphonebk.html.

Related Documents

For more information, see *Oracle Spatial Developer's Guide*.

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.

What's New in the Topology and Network Data Models?

This section describes new and changed features for Oracle Spatial topology and network data models for Oracle Database Release 11.

Release 11.2 Changes

This section describes new and changed features of the Oracle Spatial topology and network data models for Oracle Database 11*g* Release 1 (11.1).

New Network Analysis Functions in the Java API

The `NetworkAnalyst` class in `oracle.spatial.network.lod` contains many new methods. This class is the single entry point to all the network analysis operations supported by the network data model load on demand (LOD) approach to analysis. The new methods include the following:

- Hierarchical shortest path analysis (`NetworkAnalyst.shortestPathHierarchical`) supports hierarchical shortest path analysis for networks with multiple link levels.
- Shortest path A* analysis(`NetworkAnalyst.shortestPathAStar`) supports the A* shortest path algorithm with a user-defined heuristic cost function. It provides better performance than the Dijkstra algorithm because fewer nodes are explored.
- Network buffer zones (`NetworkAnalyst.networkBuffer`), also called zones of influence, operation on networks. Includes coverage and cost information.
- Traveling salesperson analysis(`NetworkAnalyst.tsp`) supports the minimum cost tour that includes all given nodes. Supports both closed and open tours (start and/or end are fixed).
- Minimum cost spanning tree analysis (`NetworkAnalyst.mcst`) finds the links with the sum of minimum link costs to connect all nodes.
- Minimum cost polygon generation (`NetworkAnalyst.withinCostPolygon`) provides a spatial representation (polygon) based on the minimum cost network coverage. For example, the Drive Time polygon is one such representation when travel time is used as link cost.
- K shortest paths analysis (`NetworkAnalyst.kShortestPaths`) finds the *k* shortest paths between two nodes.

Reference information (Javadoc) on all supported packages and classes is in *Oracle Spatial Java API Reference*. The network data model Java API is introduced in Section 5.11.2.

Power Law (Scale-Free) Network Partitioning

The new SDO_NET.LOGICAL_POWERLAW_PARTITION procedure (documented in Chapter 6) partitions a logical power law (also called scale-free) network, and stores the information in the partition table. (In a power law network, the node degree values contain the power law information.)

New Functions for Spatial Networks

The following new functions are available for use with spatial (including LRS) networks:

- SDO_NET.GET_PERCENTAGE returns the percentage of the distance along a link's line string geometry of a point geometry.
- SDO_NET.GET_PT returns the point geometry that is a specified percentage of the distance along a link's line string geometry.

These functions are documented in Chapter 6.

Other New Subprograms

SDO_NET.GENERATE_NODE_LEVELS (documented in Chapter 6) generates node levels for a specified hierarchical (multilevel) network, and stores the information in a table.

Changes to Existing Subprograms

Significant changes to existing subprograms (documented in Chapter 6) include the following:

- `regenerate_node_levels` is added as an input parameter for SDO_NET.GENERATE_PARTITION_BLOBS.

Network Metadata View Changes

The NODE_LEVEL_TABLE_NAME column is added to the USER_SDO_NETWORK_METADATA and ALL_SDO_NETWORK_METADATA views, which are described in Section 5.10.1.

Release 11.1 Changes

This section describes new and changed features of the Oracle Spatial topology and network data models for Oracle Database 11*g* Release 1 (11.1).

Load on Demand Analysis and Network Partitioning

To perform network analysis, you now have the option of partitioning the network and having needed partitions automatically loaded on demand during network analysis. This approach supplements the previous in-memory approach for network editing and analysis.

Note: The in-memory approach will be deprecated in the next release of Spatial. Future development will enhance the load on demand approach, which you are encouraged use instead of the in-memory approach.

With the load on demand approach, you first partition the network into partitions of manageable size, and you use the load on demand Java API to perform analysis, which automatically loads and unloads needed partitions during analysis. This approach enables Spatial to perform network analysis on very large networks without memory limitation becoming a significant factor.

Load on demand analysis and network partitioning are described in Section 5.7.

Subpaths

A subpath is partial path along a path, created either as a result of a network analysis operation or explicitly by a user. The start and end points of a subpath are defined as link indexes and the percentage of the distance to the next node in the path. Subpaths are explained and illustrated in Section 5.3.1.

User Data Support

You can define your own data in the user data metadata view, and Spatial will manage the user data as well as the connectivity information. Each user data entry is stored as a column in node, link, path, and subpath tables. Spatial currently supports simple data types such as VARCHAR2, INTEGER, NUMBER, and SDO_GEOMETRY.

See the information about the `include_user_data` parameter for the SDO_NET.GENERATE_PARTITION_BLOB, SDO_NET.GENERATE_PARTITION_BLOBS, and SDO_NET.GET_PARTITION_SIZE subprograms.

XML API for Network Analysis

You can use the network data model XML API to perform network analysis. Web service requests are supported through Oracle Spatial Web services, which are described in *Oracle Spatial Developer's Guide*.

The XML schema of the network data model XML API is described in the following:

`$ORACLE_HOME/md/doc/sdondmxml.zip`

Part I

Topology Data Model

This document has two main parts:

- Part I provides conceptual, usage, and reference information about the topology data model of Oracle Spatial.
- Part II provides conceptual, usage, and reference information about the network data model of Oracle Spatial.

Part I contains the following chapters:

1

Topology Data Model Overview

The topology data model of Oracle Spatial lets you work with data about nodes, edges, and faces in a topology. For example, United States Census geographic data is provided in terms of nodes, chains, and polygons, and this data can be represented using the Spatial topology data model. You can store information about topological elements and geometry layers in Oracle Spatial tables and metadata views. You can then perform certain Spatial operations referencing the topological elements, for example, finding which chains (such as streets) have any spatial interaction with a specific polygon entity (such as a park).

This chapter describes the Spatial data structures and data types that support the topology data model, and what you need to do to populate and manipulate the structures. You can use this information to write a program to convert your topological data into formats usable with Spatial.

Note: Although this chapter discusses some topology terms as they relate to Oracle Spatial, it assumes that you are familiar with basic topology concepts.

It also assumes that you are familiar with the main Spatial concepts, data types, and operations, as documented in *Oracle Spatial Developer's Guide*.

This chapter contains the following major sections:

- Section 1.13, "README File for Spatial and Related Features"

1.1 Main Steps in Using Topology Data

This section summarizes the main steps for working with topology data in Oracle Spatial. It refers to important concepts, structures, and operations that are described in detail in other sections.

The specific main steps depend on which of two basic approaches you follow, which depend on the kind of data you will use to build the topology:

- If you have data about the edges, nodes, and faces (but not spatial geometry data), follow the steps in Section 1.1.1, "Using a Topology Built from Topology Data".
- If you will build the topology from Spatial geometries that will become topology features, follow the steps in Section 1.1.2, "Using a Topology Built from Spatial Geometries".

You can use the topology data model PL/SQL and Java APIs to update the topology (for example, to change the data about an edge, node, or face). The PL/SQL API for most editing operations is the SDO_TOPO_MAP package, which is documented in Chapter 4. The Java API is described in Section 1.8.2.

1.1.1 Using a Topology Built from Topology Data

The main steps for working with a topology built from topology data are as follows:

1. Create the topology, using the SDO_TOPO.CREATE_TOPOLOGY procedure. This causes the <topology-name>_EDGE$, <topology-name>_NODE$, <topology-name>_FACE$, and <topology-name>_HISTORY$ tables to be created. (These tables are described in Section 1.5.1, Section 1.5.2, Section 1.5.3, and Section 1.5.5, respectively.)
2. Load topology data into the node, edge, and face tables created in Step 1. This is typically done using a bulk-load utility, but it can be done using SQL INSERT statements.
3. Create a feature table for each type of topology geometry layer in the topology. For example, a city data topology might have separate feature tables for land parcels, streets, and traffic signs.
4. Associate the feature tables with the topology, using the SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER procedure for each feature table. This causes the <topology-name>_RELATION$ table to be created. (This table is described in Section 1.5.4.)
5. Initialize topology metadata, using the SDO_TOPO.INITIALIZE_METADATA procedure. (This procedure also creates spatial indexes on the <topology-name>_EDGE$, <topology-name>_NODE$, and <topology-name>_FACE$ tables, and additional B-tree indexes on the <topology-name>_EDGE$ and <topology-name>_NODE$ tables.)
6. Load the feature tables using the SDO_TOPO_GEOMETRY constructor. (This constructor is described in Section 1.6.2.)
7. Query the topology data (for example, using one of topology operators described in Section 1.8.1).
8. Optionally, edit topology data using the PL/SQL or Java application programming interfaces (APIs).

Section 1.12.1 contains a PL/SQL example that performs these main steps.

1.1.2 Using a Topology Built from Spatial Geometries

To build a topology from spatial geometries, you must first perform the standard operations for preparing data for use with Oracle Spatial, as described in *Oracle Spatial Developer's Guide*:

1. Create the spatial tables.
2. Update the spatial metadata (USER_SDO_GEOM_METADATA view).
3. Load data into the spatial tables.
4. Validate the spatial data.
5. Create the spatial indexes.

The main steps for working with a topology built from Oracle Spatial geometries are as follows:

1. Create the topology, using the SDO_TOPO.CREATE_TOPOLOGY procedure. This causes the <topology-name>_EDGE$, <topology-name>_NODE$, <topology-name>_FACE$, and <topology-name>_HISTORY$ tables to be created. (These tables are described in Section 1.5.1, Section 1.5.2, Section 1.5.3, and Section 1.5.5, respectively.)
2. Create the universe face (F0, defined in Section 1.2).
3. Create a feature table for each type of topology geometry layer in the topology. For example, a city data topology might have separate feature tables for land parcels, streets, and traffic signs.
4. Associate the feature tables with the topology, using the SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER procedure for each feature table. This causes the <topology-name>_RELATION$ table to be created. (This table is described in Section 1.5.4.)
5. Create a TopoMap object and load the whole topology into cache.
6. Load the feature tables, inserting data from the spatial tables and using the SDO_TOPO_MAP.CREATE_FEATURE function.
7. Initialize topology metadata, using the SDO_TOPO.INITIALIZE_METADATA procedure. (This procedure also creates spatial indexes on the <topology-name>_EDGE$, <topology-name>_NODE$, and <topology-name>_FACE$ tables, and additional B-tree indexes on the <topology-name>_EDGE$ and <topology-name>_NODE$ tables.)
8. Query the topology data (using one of topology operators described in Section 1.8.1).
9. Optionally, edit topology data using the PL/SQL or Java application programming interfaces (APIs).

Section 1.12.2 contains a PL/SQL example that performs these main steps.

1.2 Topology Data Model Concepts

Topology is a branch of mathematics concerned with objects in space. Topological relationships include such relationships as *contains*, *inside*, *covers*, *covered by*, *touch*, and *overlap with boundaries intersecting*. Topological relationships remain constant when the

coordinate space is deformed, such as by twisting or stretching. (Examples of relationships that are not topological include *length of*, *distance between*, and *area of*.)

The basic elements in a topology are its nodes, edges, and faces.

A **node**, represented by a point, can be isolated or it can be used to bound edges. Two or more edges meet at a non-isolated node. A node has a coordinate pair associated with it that describes the spatial location for that node. Examples of geographic entities that might be represented as nodes include start and end points of streets, places of historical interest, and airports (if the map scale is sufficiently large).

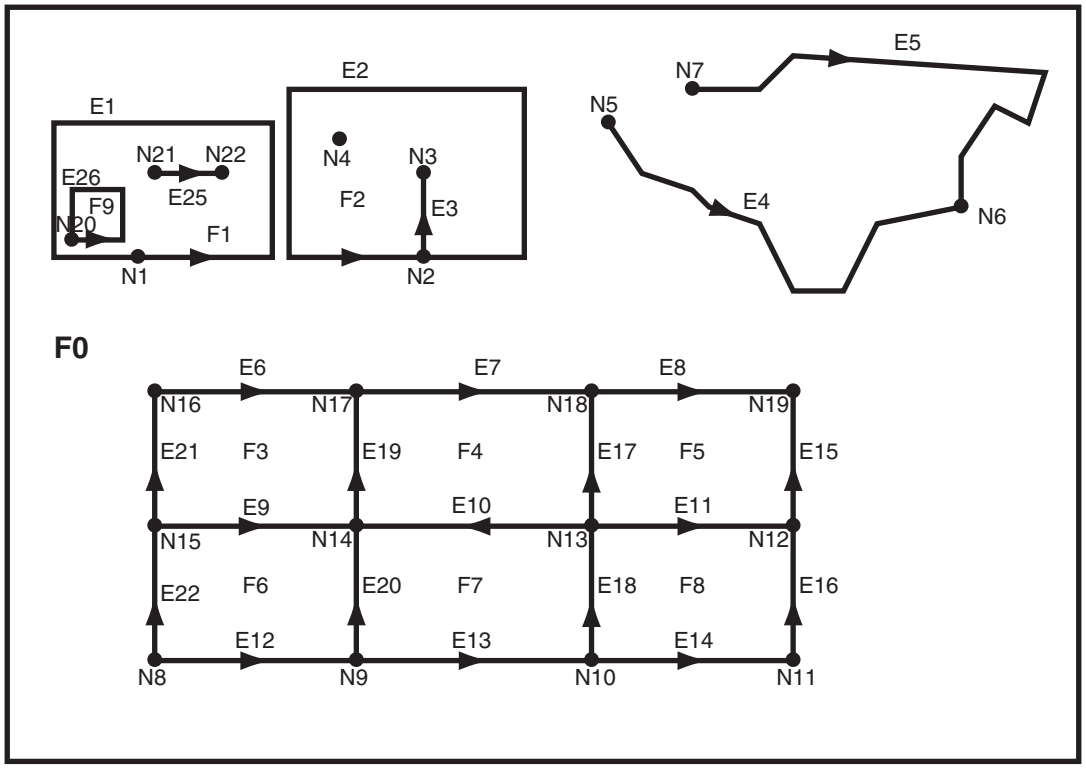
An **edge** is bounded by two nodes: the start (origin) node and the end (terminal) node. An edge has an associated geometric object, usually a coordinate string that describes the spatial representation of the edge. An edge may have several vertices making up a line string. (Circular arcs are not supported for topologies.) Examples of geographic entities that might be represented as edges include segments of streets and rivers.

The order of the coordinates gives a **direction** to an edge, and direction is important in determining topological relationships. The positive direction agrees with the orientation of the underlying edge, and the negative direction reverses this orientation. Each orientation of an edge is referred to as a **directed edge**, and each directed edge is the mirror image of its other directed edge. The start node of the positive directed edge is the end node of the negative directed edge. An edge also lies between two faces and has references to both of them. Each directed edge contains a reference to the next edge in the contiguous perimeter of the face on its left side.

A **face**, corresponding to a polygon, has a reference to one directed edge of its outer boundary. If any island nodes or island edges are present, the face also has a reference to one directed edge on the boundary of each island. Examples of geographic entities that might be represented as faces include parks, lakes, counties, and states.

Figure 1–1 shows a simplified topology containing nodes, edges, and faces. The arrowheads on each edge indicate the positive direction of the edge (or, more precisely, the orientation of the underlying line string or curve geometry for positive direction of the edge).

Figure 1–1 Simplified Topology



Notes on Figure 1–1:

- *E* elements (E1, E2, and so on) are edges, *F* elements (F0, F1, and so on) are faces, and *N* elements (N1, N2, and so on) are nodes.
- **F0** (face zero) is created for every topology. It is the universe face containing everything else in the topology. There is no geometry associated with the universe face. F0 has the face ID value of -1 (negative 1).
- There is a node created for every point geometry and for every start and end node of an edge. For example, face F1 has only an edge (a closed edge), E1, that has the same node as the start and end nodes (N1). F1 also has edge E25, with start node N21 and end node N22.
- An **isolated node** (also called an **island node**) is a node that is isolated in a face. For example, node N4 is an isolated node in face F2.
- An **isolated edge** (also called an **island edge**) is an edge that is isolated in a face. For example, edge E25 is an isolated edge in face F1.
- A **loop edge** is an edge that has the same node as its start node and end node. For example, edge E1 is a loop edge starting and ending at node N1.
- An edge cannot have an isolated (island) node on it. The edge can be broken up into two edges by adding a node on the edge. For example, if there was originally a single edge between nodes N16 and N18, adding node N17 resulted in two edges: E6 and E7.
- Information about the topological relationships is stored in special edge, face, and node information tables. For example, the edge information table contains the following information about edges E9 and E10. (Note the direction of the

arrowheads for each edge.) The next and previous edges are based on the left and right faces of the edge.

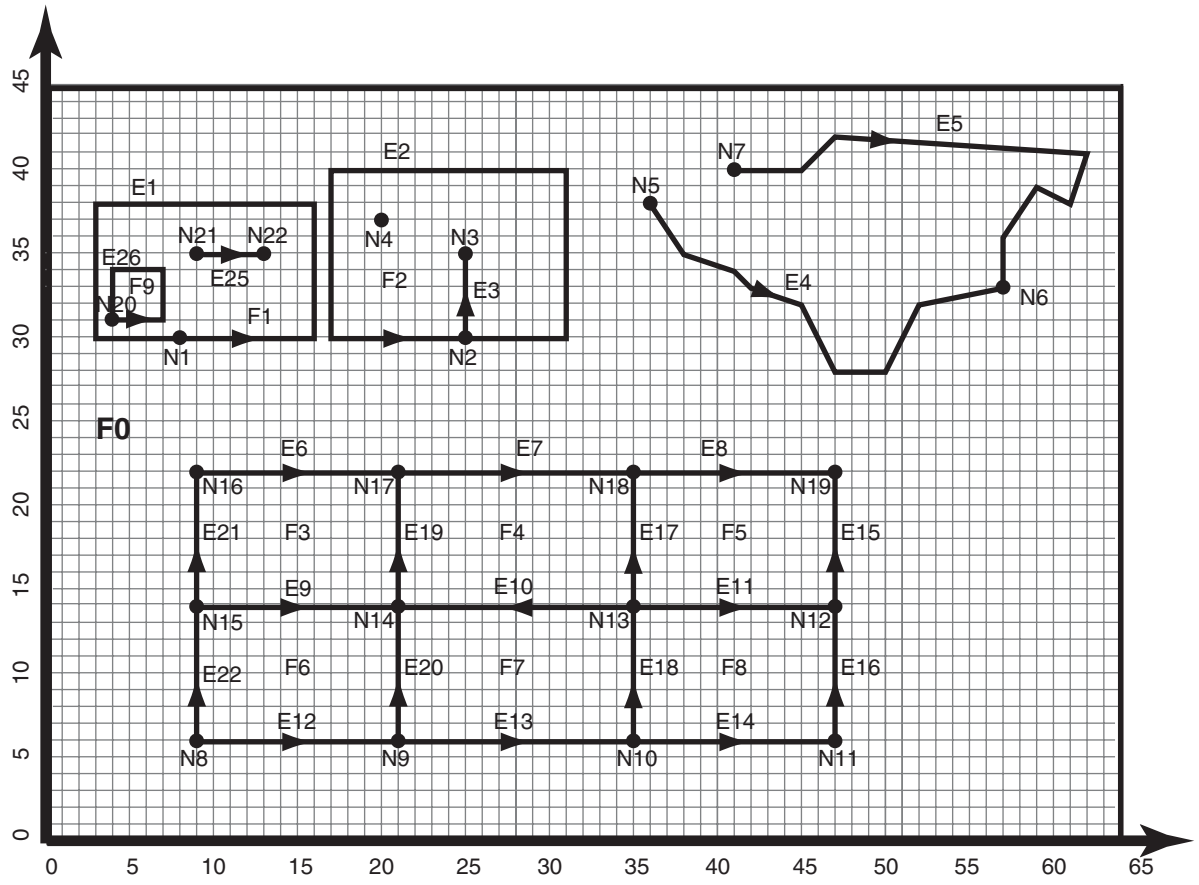
For edge E9, the start node is N15 and the end node is N14, the next left edge is E19 and the previous left edge is -E21, the next right edge is -E22 and the previous right edge is E20, the left face is F3 and the right face is F6.

For edge E10, the start node is N13 and the end node is N14, the next left edge is -E20 and the previous left edge is E18, the next right edge is E17 and the previous right edge is -E19, the left face is F7 and the right face is F4.

For additional examples of edge-related data, including an illustration and explanations, see Section 1.5.1.

Figure 1–2 shows the same topology illustrated in Figure 1–1, but it adds a grid and unit numbers along the x-axis and y-axis. Figure 1–2 is useful for understanding the output of some of the examples in Chapter 3 and Chapter 4.

Figure 1–2 Simplified Topology, with Grid Lines and Unit Numbers



1.2.1 Tolerance in the Topology Data Model

Tolerance is used to associate a level of precision with spatial data. **Tolerance** reflects the *distance that two points can be apart and still be considered the same* (for example, to accommodate rounding errors). The tolerance value must be a positive number greater than zero.

However, in the topology data model, tolerance can have two meanings depending on the operation being performed: one meaning is the traditional Oracle Spatial definition of tolerance, and the other is a fixed tolerance value of 10E-15.

- The tolerance value specified in the call to the SDO_TOPO.CREATE_TOPOLOGY procedure refers to the traditional Oracle Spatial definition, as explained in *Oracle Spatial Developer's Guide*. This value is used when indexes are created in the node, edge, and face tables, and when spatial operators are used to query these tables.
- The tolerance value used for internal computations (for example, finding edge intersections) during topology editing operations is always 10E-15 (based on Java double precision arithmetic). This value is used during the validation checks performed by the SDO_TOPO_MAP.VALIDATE_TOPO_MAP and SDO_TOPO_MAP.VALIDATE_TOPOLOGY functions.

Thus, for example, an edge geometry that is considered valid by the SDO_TOPO_MAP.VALIDATE_TOPO_MAP or SDO_TOPO_MAP.VALIDATE_TOPOLOGY function might not be valid if that geometry is passed to the SDO_GEOM.VALIDATE_GEOMETRY_WITH_CONTEXT function.

1.3 Topology Geometries and Layers

A **topology geometry** (also referred to as a **feature**) is a spatial representation of a real world object. For example, *Main Street* and *Walden State Park* might be the names of topology geometries. The geometry is stored as a set of **topological elements** (nodes, edges, and faces), which are sometimes also referred to as *primitives*. Each topology geometry has a unique ID (assigned by Spatial when records are imported or loaded) associated with it.

A **topology geometry layer** consists of topology geometries, usually of a specific topology geometry type, although it can be a collection of multiple types (see Section 1.3.2 for information about collection layers). For example, *Streets* might be the topology geometry layer that includes the *Main Street* topology geometry, and *State Parks* might be the topology geometry layer that includes the *Walden State Park* topology geometry. Each topology geometry layer has a unique ID (assigned by Spatial) associated with it. The data for each topology geometry layer is stored in a **feature table**. For example, a feature table named CITY_STREETS might contain information about all topology geometries (individual roads or streets) in the *Streets* topology geometry layer.

Each topology geometry (feature) is defined as an object of type SDO_TOPO_GEOMETRY (described in Section 1.6.1), which identifies the topology geometry type, topology geometry ID, topology geometry layer ID, and topology ID for the topology.

Topology metadata is automatically maintained by Spatial in the USER_SDO_TOPO_METADATA and ALL_SDO_TOPO_METADATA views, which are described in Section 1.7.2. The USER_SDO_TOPO_INFO and ALL_SDO_TOPO_INFO views (described in Section 1.7.1) contain a subset of this topology metadata.

1.3.1 Features

Often, there are fewer features in a topology than there are topological elements (nodes, edges, and faces). For example, a road feature may consist of many edges, an area feature such as a park may consist of many faces, and some nodes may not be associated with point features. Figure 1–3 shows point, line, and area features associated with the topology that was shown in Figure 1–1 in Section 1.2.

Figure 1–3 Features in a Topology

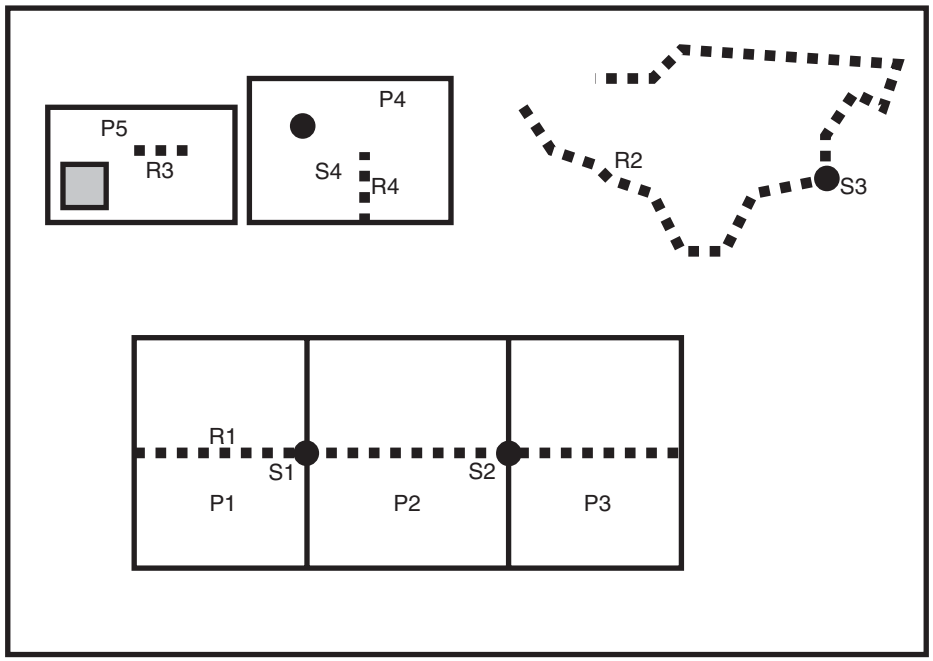


Figure 1–3 shows the following kinds of features in the topology:

- Point features (traffic signs), shown as dark circles: `S1`, `S2`, `S3`, and `S4`
- Linear features (roads or streets), shown as dashed lines: `R1`, `R2`, `R3`, and `R4`
- Area features (land parcels), shown as rectangles: `P1`, `P2`, `P3`, `P4`, and `P5`

  Land parcel `P5` does not include the shaded area within its area. (Specifically, `P5` includes face `F1` but not face `F9`. These faces are shown in Figure 1–1 in Section 1.2.)

Example 1–12 in Section 1.12.1 defines these features.

1.3.2 Collection Layers

A **collection layer** is a topology geometry layer that can contain topological elements of different topology geometry types. For example, using the `CITY_DATA` topology from the examples in Section 1.12, you could create a collection layer to contain specific land parcel, city street, and traffic sign elements.

To create a collection layer, follow essentially the same steps for creating other types of layers. Create a feature table for the layer, as in the following example:

```
CREATE TABLE collected_features ( -- Selected heterogeneous features
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);
```

Associate the feature table with the topology, specifying `COLLECTION` for the `topo_geometry_layer_type` parameter in the call to the SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER procedure, as in the following example:

```
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', COLLECTED_FEATURES',
'FEATURE', 'COLLECTION');
```

To load the feature table for the collection layer, insert the necessary rows, as shown in Example 1–1.

Example 1–1 Loading the Feature Table for a Collection Layer

```
-- Take R5 from the CITY_STREETS layer.
INSERT INTO collected_features VALUES(
  'C_R5',
  SDO_TOPO_GEOMETRY('CITY_DATA',
    2,  -- tg_type = line/multiline
    4,  -- tg_layer_id
    SDO_TOPO_OBJECT_ARRAY(
      SDO_TOPO_OBJECT(20, 2),
      SDO_TOPO_OBJECT(-9, 2)))
);

-- Take S3 from the TRAFFIC_SIGNS layer.
INSERT INTO collected_features VALUES(
  'C_S3',
  SDO_TOPO_GEOMETRY('CITY_DATA',
    1,  -- tg_type = point/multipoint
    4,  -- topo layer id
    SDO_TOPO_OBJECT_ARRAY(
       SDO_TOPO_OBJECT(6, 1)))
);

-- Take P3 from the LAND_PARCELS layer.
INSERT INTO collected_features VALUES(
  'C_P3',
  SDO_TOPO_GEOMETRY('CITY_DATA',
    3,  -- tg_type = (multi)polygon
    4,
    SDO_TOPO_OBJECT_ARRAY(
      SDO_TOPO_OBJECT(5, 3),
      SDO_TOPO_OBJECT(8, 3)))
);

-- Create a collection from a polygon and a point.
INSERT INTO collected_features VALUES(
  'C1',
  SDO_TOPO_GEOMETRY('CITY_DATA',
    4,  -- tg_type = collection
    4,
    SDO_TOPO_OBJECT_ARRAY(
      SDO_TOPO_OBJECT(5, 3),
      SDO_TOPO_OBJECT(6, 1)))
);

-- Create a collection from a polygon and a line.
INSERT INTO collected_features VALUES(
  'C2',
  SDO_TOPO_GEOMETRY('CITY_DATA',
    4,  -- tg_type = collection
    4,
    SDO_TOPO_OBJECT_ARRAY(
      SDO_TOPO_OBJECT(8, 3),
      SDO_TOPO_OBJECT(10, 2)))
);

-- Create a collection from a line and a point.
```

```
INSERT INTO collected_features VALUES(
  'C3',
  SDO_TOPO_GEOMETRY('CITY_DATA',
     4,  -- tg_type = collection
     4,
     SDO_TOPO_OBJECT_ARRAY(
       SDO_TOPO_OBJECT(-5, 2),
       SDO_TOPO_OBJECT(10, 1)))
);
```

1.4 Topology Geometry Layer Hierarchy

In some topologies, the topology geometry layers (feature layers) have one or more parent-child relationships in a **topology hierarchy**. That is, the layer at the topmost level consists of features in its child layer at the next level down in the hierarchy; the child layer might consist of features in its child layer at the next layer farther down; and so on. For example, a land use topology might have the following topology geometry layers at different levels of hierarchy:

- States at the highest level, which consists of features from its child layer, Counties
- Counties at the next level down, which consists of features from its child layer, Tracts
- Tracts at the next level down, which consists of features from its child layer, Block Groups
- Block Groups at the next level down, which consists of features from its child layer, Land Parcels
- Land Parcels at the lowest level of the hierarchy

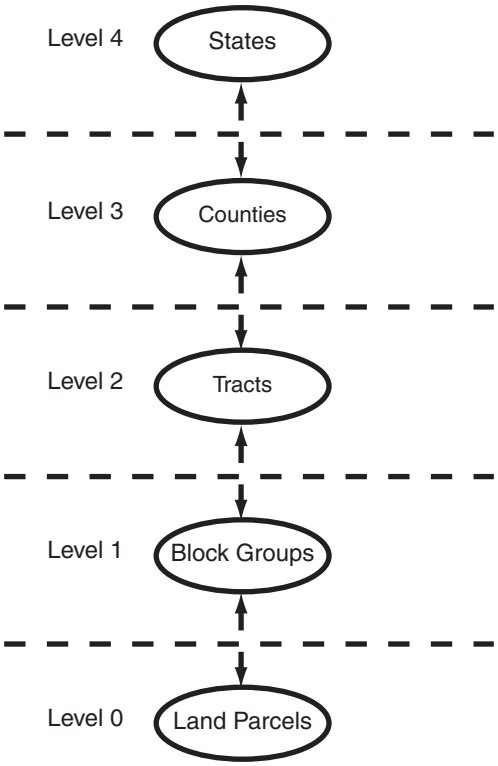
If the topology geometry layers in a topology have this hierarchical relationship, it is far more efficient if you model the layers as hierarchical than if you specify all topology geometry layers at a single level (that is, with no hierarchy). For example, it is more efficient to construct SDO_TOPO_GEOMETRY objects for counties by specifying only the tracts in the county than by specifying all land parcels in all block groups in all tracts in the county.

The lowest level (for the topology geometry layer containing the smallest kinds of features) in a hierarchy is level 0, and successive higher levels are numbered 1, 2, and so on. Topology geometry layers at adjacent levels of a hierarchy have a parent-child relationship. Each topology geometry layer at the higher level is the **parent layer** for one layer at the lower level, which is its **child layer**. A parent layer can have only one child layer, but a child layer can have one or more parent layers. Using the preceding example, the Counties layer can have only one child layer, Tracts; however, the Tracts layer could have parent layers named Counties and Water Districts.

Note: Topology geometry layer hierarchy is somewhat similar to network hierarchy, which is described in Section 5.5; however, there are significant differences, and you should not confuse the two. For example, the lowest topology geometry layer hierarchy level is 0, and the lowest network hierarchy level is 1; and in a topology geometry layer hierarchy each parent must have one child and each child can have many parents, while in a network hierarchy each parent can have many children and each child must have one parent.

Figure 1–4 shows the preceding example topology geometry layer hierarchy. Each level of the hierarchy shows the level number and the topology geometry layer in that level.

Figure 1–4 Topology Geometry Layer Hierarchy



To model topology geometry layers as hierarchical, specify the child layer in the `child_layer_id` parameter when you call the SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER procedure to add a parent topology geometry layer to the topology. Add the lowest-level (level 0) topology geometry layer first; then add the level 1 layer, specifying the level 0 layer as its child; then add the level 2 layer, specifying the level 1 layer as its child; and so on. Example 1–2 shows five topology geometry layers being added so that the 5-level hierarchy is established.

Example 1–2 Modeling a Topology Geometry Layer Hierarchy

```
-- Create the topology. (Null SRID in this example.)
EXECUTE SDO_TOPO.CREATE_TOPOLOGY('LAND_USE_HIER', 0.00005);

-- Create feature tables.
CREATE TABLE land_parcels ( -- Land parcels (selected faces)
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE block_groups (
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE tracts (
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);
```

```
CREATE TABLE counties (
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE states (
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

-- (Other steps not shown here, such as populating the feature tables
-- and initializing the metadata.)
. . .
-- Associate feature tables with the topology; include hierarchy information.

DECLARE
  land_parcels_id NUMBER;
  block_groups_id NUMBER;
  tracts_id NUMBER;
  counties_id NUMBER;
BEGIN
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('LAND_USE_HIER', 'LAND_PARCELS',
  'FEATURE','POLYGON');
SELECT tg_layer_id INTO land_parcels_id FROM user_sdo_topo_info
  WHERE topology = 'LAND_USE_HIER' AND table_name = 'LAND_PARCELS';
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('LAND_USE_HIER', 'BLOCK_GROUPS',
  'FEATURE','POLYGON', NULL, land_parcels_id);
SELECT tg_layer_id INTO block_groups_id FROM user_sdo_topo_info
  WHERE topology = 'LAND_USE_HIER' AND table_name = 'BLOCK_GROUPS';
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('LAND_USE_HIER', 'TRACTS',
  'FEATURE','POLYGON', NULL, block_groups_id);
SELECT tg_layer_id INTO tracts_id FROM user_sdo_topo_info
  WHERE topology = 'LAND_USE_HIER' AND table_name = 'TRACTS';
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('LAND_USE_HIER', 'COUNTIES',
  'FEATURE','POLYGON', NULL, tracts_id);
SELECT tg_layer_id INTO counties_id FROM user_sdo_topo_info
  WHERE topology = 'LAND_USE_HIER' AND table_name = 'COUNTIES';
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('LAND_USE_HIER', 'STATES',
  'FEATURE','POLYGON', NULL, counties_id);
END;
/
```

Within each level above level 0, each layer can contain features built from features at the next lower level (as is done in Example 1–2), features built from topological elements (faces, nodes, edges), or a combination of these. For example, a tracts layer can contain tracts built from block groups or tracts built from faces, or both. However, each feature within the layer must be built only either from features from the next lower level or from topological elements. For example, a specific tract can consist of block groups or it can consist of faces, but it cannot consist of a combination of block groups and faces.

To insert or update topology geometry objects in feature tables for the levels in a hierarchy, use the appropriate forms of the SDO_TOPO_GEOMETRY constructor. Feature tables are described in Section 1.3, and SDO_TOPO_GEOMETRY constructors are described in Section 1.6.2.

Note: The TOPO_ID and TOPO_TYPE attributes in the relationship information table have special meanings when applied to parent layers in a topology with a topology geometry layer hierarchy. See the explanations of these attributes in Table 1–5 in Section 1.5.4.

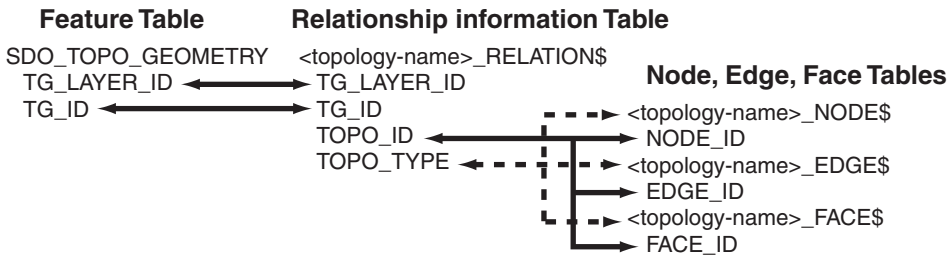
1.5 Topology Data Model Tables

To use the Spatial topology capabilities, you must first insert data into special edge, node, and face tables, which are created by Spatial when you create a topology. The edge, node, and face tables are described in Section 1.5.1, Section 1.5.2, and Section 1.5.3, respectively.

Spatial automatically maintains a relationship information (<topology-name>_RELATION$) table for each topology, which is created the first time that a feature table is associated with a topology (that is, at the first call to the SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER procedure that specifies the topology). The relationship information table is described in Section 1.5.4.

Figure 1–5 shows the role of the relationship information table in connecting information in a feature table with information in its associated node, edge, or face table.

Figure 1–5 Mapping Between Feature Tables and Topology Tables



As shown in Figure 1–5, the mapping between feature tables and the topology node, edge, and face tables occurs through the <topology-name>_RELATION$ table. In particular:

- Each feature table includes a column of type SDO_TOPO_GEOMETRY. This type includes a TG_LAYER_ID attribute (the unique ID assigned by Oracle Spatial when the layer is created), as well as a TG_ID attribute (the unique ID assigned to each feature in a layer). The values in these two columns have corresponding values in the TG_LAYER_ID and TG_ID columns in the <topology-name>_RELATION$ table.
- Each feature has one or more rows in the <topology-name>_RELATION$ table.
- Given the TG_LAYER_ID and TG_ID values for a feature, the set of nodes, faces, and edges associated with the feature can be determined by matching the TOPO_ID value (the node, edge, or face ID) in the <topology-name>_RELATION$ table with the corresponding ID value in the <topology-name>_NODE$, <topology-name>_EDGE$, or <topology-name>_FACE$ table.

The following considerations apply to schema, table, and column names that are stored in any Oracle Spatial metadata views. For example, these considerations apply to the names of edge, node, face, relationship, and history information tables, and to

the names of any columns in these tables and schemas for these tables that are stored in the topology metadata views described in Section 1.7.

- The name must contain only letters, numbers, and underscores. For example, the name cannot contain a space ( ), an apostrophe ('), a quotation mark ("), or a comma (,).
- All letters in the names are converted to uppercase before the names are stored in metadata views or before the tables are accessed. This conversion also applies to any schema name specified with the table name.

1.5.1 Edge Information Table

You must store information about the edges in a topology in the <topology-name>_EDGE$ table, where <*topology-name*> is the name of the topology as specified in the call to the SDO_TOPO.CREATE_TOPOLOGY procedure. Each edge information table has the columns shown in Table 1–1.

Table 1–1 Columns in the <topology-name>_EDGE$ Table

Column Name	Data Type	Description
EDGE_ID	NUMBER	Unique ID number for this edge
START_NODE_ID	NUMBER	ID number of the start node for this edge
END_NODE_ID	NUMBER	ID number of the end node for this edge
NEXT_LEFT_EDGE_ID	NUMBER	ID number (signed) of the next left edge for this edge
PREV_LEFT_EDGE_ID	NUMBER	ID number (signed) of the previous left edge for this edge
NEXT_RIGHT_EDGE_ID	NUMBER	ID number (signed) of the next right edge for this edge
PREV_RIGHT_EDGE_ID	NUMBER	ID number (signed) of the previous right edge for this edge
LEFT_FACE_ID	NUMBER	ID number of the left face for this edge
RIGHT_FACE_ID	NUMBER	ID number of the right face for this edge
GEOMETRY	SDO_GEOMETRY	Geometry object (line string) representing this edge, listing the coordinates in the natural order for the positive directed edge

The NEXT_LEFT_EDGE_ID and NEXT_RIGHT_EDGE_ID values refer to the next directed edges in the counterclockwise delineation of the perimeters of the left and right faces, respectively. The PREV_LEFT_EDGE_ID and PREV_RIGHT_EDGE_ID values refer to the previous directed edges in the counterclockwise delineation of the perimeters of the left and right faces, respectively. The LEFT_FACE_ID value refers to the face to the left of the positive directed edge, and the RIGHT_FACE_ID value refers to the face to the left of the negative directed edge. For any numeric ID value, the sign indicates which orientation of the target edge is being referred to.

Figure 1–6 shows nodes, edges, and faces that illustrate the relationships among the various ID columns in the edge information table. (In Figure 1–6, thick lines show the edges, and thin lines with arrowheads show the direction of each edge.)

Figure 1–6 Nodes, Edges, and Faces

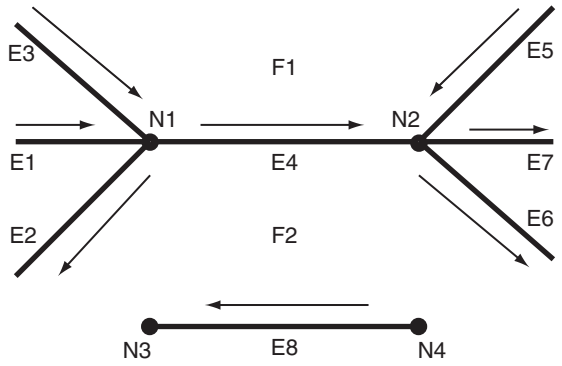


Table 1–2 shows the ID column values in the edge information table for edges E4 and E8 in Figure 1–6. (For clarity, Table 1–2 shows ID column values with alphabetical characters, such as E4 and N1; however, the ID columns actually contain numeric values only, specifically the numeric ID value associated with each named object.)

Table 1–2 Edge Table ID Column Values

EDGE_ ID	START_ NODE_ ID	END_ NODE_ ID	NEXT_ LEFT_ EDGE_ ID	PREV_ LEFT_ EDGE_ ID	NEXT_ RIGHT_ EDGE_ ID	PREV_ RIGHT_ EDGE_ ID	LEFT_ FACE_ ID	RIGHT_ FACE_ ID
E4	N1	N2	-E5	E3	E2	-E6	F1	F2
E8	N4	N3	-E8	-E8	E8	E8	F2	F2

In Figure 1–6 and Table 1–2:

- The start node and end node for edge E4 are N1 and N2, respectively. The next left edge for edge E4 is E5, but its direction is the opposite of edge E4, and therefore the next left edge for E4 is stored as -E5 (negative E5).
- The previous left edge for edge E4 is E3, and because it has the same direction as edge E4, the previous left edge for E4 is stored as E3.
- The next right face is determined using the negative directed edge of E4. This can be viewed as reversing the edge direction and taking the next left edge and previous left edge. In this case, the next right edge is E2 and the previous right edge is -E6 (the direction of edge E6 is opposite the negative direction of edge E4). For edge E4, the left face is F1 and the right face is F2.
- Edges E1 and E7 are neither leftmost nor rightmost edges with respect to edge E4, and therefore they do not appear in the edge table row associated with edge E4.

1.5.2 Node Information Table

You must store information about the nodes in a topology in the <topology-name>_ NODE$ table, where <*topology-name*> is the name of the topology as specified in the call to the SDO_TOPO.CREATE_TOPOLOGY procedure. Each node information table has the columns shown in Table 1–3.

Table 1–3 Columns in the <topology-name>_NODE$ Table

Column Name	Data Type	Description
NODE_ID	NUMBER	Unique ID number for this node
EDGE_ID	NUMBER	ID number (signed) of the edge (if any) associated with this node
FACE_ID	NUMBER	ID number of the face (if any) associated with this node
GEOMETRY	SDO_GEOMETRY	Geometry object (point) representing this node

For each node, the EDGE_ID or FACE_ID value (but not both) must be null:

- If the EDGE_ID value is null, the node is an isolated node (that is, isolated in a face).
- If the FACE_ID value is null, the node is not an isolated node, but rather the start node or end node of an edge.

1.5.3 Face Information Table

You must store information about the faces in a topology in the <topology-name>_FACE$ table, where <*topology-name*> is the name of the topology as specified in the call to the SDO_TOPO.CREATE_TOPOLOGY procedure. Each face information table has the columns shown in Table 1–4.

Table 1–4 Columns in the <topology-name>_FACE$ Table

Column Name	Data Type	Description
FACE_ID	NUMBER	Unique ID number for this face
BOUNDARY_EDGE_ID	NUMBER	ID number of the boundary edge for this face. The sign of this number (which is ignored for use as a key) indicates which orientation is being used for this boundary component (positive numbers indicate the left of the edge, and negative numbers indicate the right of the edge).
ISLAND_EDGE_ID_LIST	SDO_LIST_TYPE	Island edges (if any) in this face. (The SDO_LIST_TYPE type is described in Section 1.6.6.)
ISLAND_NODE_ID_LIST	SDO_LIST_TYPE	Island nodes (if any) in this face. (The SDO_LIST_TYPE type is described in Section 1.6.6.)
MBR_GEOMETRY	SDO_GEOMETRY	Minimum bounding rectangle (MBR) that encloses this face. (This is required, except for the universe face.) The MBR must be stored as an optimized rectangle (a rectangle in which only the lower-left and the upper-right corners are specified). The SDO_TOPO.INITIALIZE_METADATA procedure creates a spatial index on this column.

1.5.4 Relationship Information Table

As you work with topological elements, Spatial automatically maintains information about each object in <topology-name>_RELATION$ tables, where <*topology-name*> is

the name of the topology and there is one such table for each topology. Each row in the table uniquely identifies a topology geometry with respect to its topology geometry layer and topology. Each relationship information table has the columns shown in Table 1–5.

Table 1–5 Columns in the <topology-name>_RELATION$ Table

Column Name	Data Type	Description
TG_LAYER_ID	NUMBER	ID number of the topology geometry layer to which the topology geometry belongs
TG_ID	NUMBER	ID number of the topology geometry
TOPO_ID	NUMBER	For a topology that does not have a topology geometry layer hierarchy: ID number of a topological element in the topology geometry For a topology that has a topology geometry layer hierarchy: Reserved for Oracle use
TOPO_TYPE	NUMBER	For a topology that does not have a topology geometry layer hierarchy: 1 = node, 2 = edge, 3 = face For a topology that has a topology geometry layer hierarchy: Reserved for Oracle use
TOPO_ATTRIBUTE	VARCHAR2	Reserved for Oracle use

1.5.5 History Information Table

The history information table for a topology contains information about editing operations that are not recorded in other information tables. Thus, the history information table is not a comprehensive record of topology modifications. Instead, it contains rows for node, edge, or face editing operations only when one or more feature tables are associated with the topology and any of the following conditions are met:

- An existing face or edge is split as a result of the operation.
- A single face or edge is created by merging two faces or two edges as a result of the operation.

Spatial automatically maintains information about these operations in <topology-name>_HISTORY$ tables, where <*topology-name*> is the name of the topology and there is one such table for each topology. Each row in the table uniquely identifies an editing operation on a topological element, although an editing operation (such as using the SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY function) can add multiple rows. (Topology editing is discussed in Chapter 2.) Each history information table has the columns shown in Table 1–6.

Table 1–6 Columns in the <topology-name>_HISTORY$ Table

Column Name	Data Type	Description
TOPO_TX_ID	NUMBER	ID number of the transaction that was started by a call to the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure or to the loadWindow or loadTopology Java method. Each transaction can consist of several editing operations. You can get the transaction ID number for the current updatable TopoMap object by calling the SDO_TOPO_MAP.GET_TOPO_TRANSACTION_ID function.
TOPO_SEQUENCE	NUMBER	Sequence number assigned to an editing operation within the transaction

Table 1–6 (Cont.) Columns in the <topology-name>_HISTORY$ Table

Column Name	Data Type	Description
TOPOLOGY	VARCHAR2	ID of the topology containing the objects being edited
TOPO_ID	NUMBER	ID number of a topological element in the topology geometry
TOPO_TYPE	NUMBER	Type of topological element: 1 = node, 2 = edge, 3 = face
TOPO_OP	VARCHAR2	Type of editing operation that was performed on the topological element: `I` for insert or `D` for delete
PARENT_ID	NUMBER	For an insert operation, the ID of the parent topological element from which the current topological element is derived; for a delete operation, the ID of the resulting topological element

Consider the following examples:

- Adding a node to break edge E2, generating edge E3: The TOPO_ID value of the new edge is the ID of E3, the TOPO_TYPE value is 2, the PARENT_ID value is the ID of E2, and the TOPO_OP value is `I`.
- Deleting a node to merge edges E6 and E7, resulting in E7: The TOPO_ID value is the ID of E6, the TOPO_TYPE value is 2, the PARENT_ID value is the ID of E7, and the TOPO_OP value is `D`.

To further illustrate the effect of editing operations on the history information table, a test procedure was created to perform various editing operations on a simple topology, and to examine the effect on the history information table for the topology. The procedure performed these main steps:

1. It created and initialized a non-geodetic topology with a universe face, and added a line feature layer and an area feature layer to the topology.
2. It created a rectangular area by adding four isolated nodes and four edges connecting the isolated nodes. This caused a face (consisting of the rectangle) to be created, and it caused one row to be added to the history information table: an insert operation for the new face, whose parent is the universe face.

   The following statement shows the history information table row added by this insertion:

```
SELECT topo_id, topo_type, topo_op, parent_id
  FROM hist_test_history$ ORDER BY topo_tx_id, topo_sequence, topology;

   TOPO_ID  TOPO_TYPE TOP  PARENT_ID
---------- ---------- --- ----------
         1          3 I           -1

1 row selected.
```

3. It split the rectangular face into two smaller rectangular faces (side-by-side) by adding two nodes and a vertical edge connecting these nodes, which caused two edges (the top and bottom edges) and the face to be split. Three rows were added to the history information table: an insert operation for each of the two new edges (with the parent of each new edge being the existing edge that was split), and an insert operation for the new face (whose parent is the original rectangular face that was split).

   The following statement shows the history information table rows added thus far. The rows added by this step are shown in bold:

```
SELECT topo_id, topo_type, topo_op, parent_id
  FROM hist_test_history$ ORDER BY topo_tx_id, topo_sequence, topology;

   TOPO_ID  TOPO_TYPE TOP  PARENT_ID
---------- ---------- --- ----------
         1          3 I           -1
         6          2 I            2
         7          2 I            4
         2          3 I            1

4 rows selected.
```

4. It added a diagonal edge to small rectangular face on the left (using the existing nodes), and it removed the vertical edge that was added in Step 3. Two rows were added to the history information table: an insert operation for the new face created as a result of the edge addition (with the parent of each new face being the small rectangular face on the left that was split), and a delete operation as a result of the edge removal (with the resulting face taking its topological object ID from one of the "parent" faces that were merged).

   The following statement shows the history information table rows added thus far. The rows added by this step are shown in bold:

```
SELECT topo_id, topo_type, topo_op, parent_id
  FROM hist_test_history$ ORDER BY topo_tx_id, topo_sequence, topology;

   TOPO_ID  TOPO_TYPE TOP  PARENT_ID
---------- ---------- --- ----------
         1          3 I           -1
         6          2 I            2
         7          2 I            4
         2          3 I            1
         3          3 I            2
         1          3 D            2

6 rows selected.
```

1.6 Topology Data Types

The main data type associated with the topology data model is SDO_TOPO_GEOMETRY, which describes a topology geometry. The SDO_TOPO_GEOMETRY type has several constructors and member functions. This section describes the topology model types, constructors, and member functions.

1.6.1 SDO_TOPO_GEOMETRY Type

The description of a topology geometry is stored in a single row, in a single column of object type SDO_TOPO_GEOMETRY in a user-defined table. The object type SDO_TOPO_GEOMETRY is defined as:

```
CREATE TYPE sdo_topo_geometry AS OBJECT
  (tg_type      NUMBER,
   tg_id        NUMBER,
   tg_layer_id  NUMBER,
   topology_id  NUMBER);
```

The SDO_TOPO_GEOMETRY type has the attributes shown in Table 1–7.

Table 1–7 SDO_TOPO_GEOMETRY Type Attributes

Attribute	Explanation
TG_TYPE	Type of topology geometry: 1 = point or multipoint, 2 = line string or multiline string, 3 = polygon or multipolygon, 4 = heterogeneous collection
TG_ID	Unique ID number (generated by Spatial) for the topology geometry
TG_LAYER_ID	ID number for the topology geometry layer to which the topology geometry belongs. (This number is generated by Spatial, and it is unique within the topology geometry layer.)
TOPOLOGY_ID	Unique ID number (generated by Spatial) for the topology

Each topology geometry in a topology is uniquely identified by the combination of its TG_ID and TG_LAYER_ID values.

You can use an attribute name in a query on an object of SDO_TOPO_GEOMETRY. Example 1–3 shows SELECT statements that query each attribute of the FEATURE column of the CITY_STREETS table, which is defined in Example 1–12 in Section 1.12.

Example 1–3 SDO_TOPO_GEOMETRY Attributes in Queries

```
SELECT s.feature.tg_type FROM city_streets s;
SELECT s.feature.tg_id FROM city_streets s;
SELECT s.feature.tg_layer_id FROM city_streets s;
SELECT s.feature.topology_id FROM city_streets s;
```

1.6.2 SDO_TOPO_GEOMETRY Constructors

The SDO_TOPO_GEOMETRY type has constructors for inserting and updating topology geometry objects. The constructors can be classified into two types, depending on the kind of objects they use:

- Constructors that specify the lowest-level topological elements (nodes, edges, and faces). These constructors have at least one attribute of type SDO_TOPO_OBJECT_ARRAY and no attributes of type SDO_TGL_OBJECT_ARRAY.
- Constructors that specify elements in the child level. These constructors have at least one attribute of type SDO_TGL_OBJECT_ARRAY and no attributes of type SDO_TOPO_OBJECT_ARRAY.

To insert and update topology geometry objects when the topology does not have a topology geometry layer hierarchy or when the operation affects the lowest level (level 0) in the hierarchy, you must use constructors that specify the lowest-level topological elements (nodes, edges, and faces). (Topology geometry layer hierarchy is explained in Section 1.4.)

To insert and update topology geometry objects when the topology has a topology geometry layer hierarchy and the operation affects a level other than the lowest in the hierarchy, you can use either or both types of constructor. That is, for each topology geometry object to be inserted or updated, you can use either of the following:

- To insert and update a topology geometry object consisting of the lowest-level topological elements (for example, to create a tract from faces), use the format that has at least one attribute of type SDO_TOPO_OBJECT_ARRAY and no attributes of type SDO_TGL_OBJECT_ARRAY.
- To insert and update a topology geometry object consisting of features at the next lower level (for example, create a tract from block groups), use the format that has

at least one attribute of type SDO_TGL_OBJECT_ARRAY and no attributes of type SDO_TOPO_OBJECT_ARRAY.

This section describes the available SDO_TOPO_GEOMETRY constructors.

Note: An additional SDO_TOPO_GEOMETRY constructor with the same attributes as the type definition (`tg_type, tg_id, tg_layer_id, topology_id`) is for Oracle internal use only.

1.6.2.1 Constructors for Insert Operations: Specifying Topological Elements

The SDO_TOPO_GEOMETRY type has the following constructors for insert operations in which you specify topological elements (faces, nodes, or edges). You must use one of these formats to create new topology geometry objects when the topology does not have a topology geometry layer hierarchy or when the operation affects the lowest level (level 0) in the hierarchy, and you can use one of these formats to create new topology geometry objects when the operation affects a level higher than level 0 in the hierarchy:

```
SDO_TOPO_GEOMETRY (topology     VARCHAR2,
                   tg_type      NUMBER,
                   tg_layer_id  NUMBER,
                   topo_ids     SDO_TOPO_OBJECT_ARRAY)

SDO_TOPO_GEOMETRY (topology      VARCHAR2,
                   table_name    VARCHAR2,
                   column_name   VARCHAR2,
                   tg_type       NUMBER,
                   topo_ids      SDO_TOPO_OBJECT_ARRAY)
```

The SDO_TOPO_OBJECT_ARRAY type is defined as a VARRAY of SDO_TOPO_OBJECT objects.

The SDO_TOPO_OBJECT type has the following two attributes:

```
(topo_id NUMBER, topo_type NUMBER)
```

The TG_TYPE and TOPO_IDS attribute values must be within the range of values from the <topology-name>_RELATION$ table (described in Section 1.5.4) for the specified topology.

Example 1–4 shows two SDO_TOPO_GEOMETRY constructors, one in each format. Each constructor inserts a topology geometry into the LAND_PARCELS table, which is defined in Example 1–12 in Section 1.12.

Example 1–4 INSERT Using Constructor with SDO_TOPO_OBJECT_ARRAY

```
INSERT INTO land_parcels VALUES ('P1', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 3), -- face_id = 3
      SDO_TOPO_OBJECT (6, 3))) -- face_id = 6
);

INSERT INTO land_parcels VALUES ('P1A', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
```

```
    'LAND_PARCELS', -- Table name
    'FEATURE', -- Column name
    3, -- Topology geometry type (polygon/multipolygon)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 3), -- face_id = 3
      SDO_TOPO_OBJECT (6, 3))) -- face_id = 6
);
```

1.6.2.2 Constructors for Insert Operations: Specifying Lower-Level Features

The SDO_TOPO_GEOMETRY type has the following constructors for insert operations in which you specify features in the next lower level of the hierarchy. You can use one of these formats to create new topology geometry objects when the operation affects a level higher than level 0 in the hierarchy:

```
SDO_TOPO_GEOMETRY (topology     VARCHAR2,
                   tg_type      NUMBER,
                   tg_layer_id  NUMBER,
                   topo_ids     SDO_TGL_OBJECT_ARRAY)

SDO_TOPO_GEOMETRY (topology      VARCHAR2,
                   table_name    VARCHAR2,
                   column_name   VARCHAR2,
                   tg_type       NUMBER,
                   topo_ids      SDO_TGL_OBJECT_ARRAY)
```

The SDO_TGL_OBJECT_ARRAY type is defined as a VARRAY of SDO_TGL_OBJECT objects.

The SDO_TGL_OBJECT type has the following two attributes:

```
(tgl_id NUMBER, tg_id NUMBER)
```

Example 1–5 shows an SDO_TOPO_GEOMETRY constructor that inserts a row into the BLOCK_GROUPS table, which is the feature table for the Block Groups level in the topology geometry layer hierarchy. The Block Groups level is the parent of the Land Parcels level at the bottom of the hierarchy.

Example 1–5 INSERT Using Constructor with SDO_TGL_OBJECT_ARRAY

```
INSERT INTO block_groups VALUES ('BG1', -- Feature name
  SDO_TOPO_GEOMETRY('LAND_USE_HIER',
    3, -- Topology geometry type (polygon/multipolygon)
    2, -- TG_LAYER_ID for block groups (from ALL_SDO_TOPO_METADATA)
    SDO_TGL_OBJECT_ARRAY (
      SDO_TGL_OBJECT (1, 1), -- land parcel ID = 1
      SDO_TGL_OBJECT (1, 2))) -- land parcel ID = 2
);
```

1.6.2.3 Constructors for Update Operations: Specifying Topological Elements

The SDO_TOPO_GEOMETRY type has the following constructors for update operations in which you specify topological elements (faces, nodes, or edges). You must use one of these formats to update topology geometry objects when the topology does not have a topology geometry layer hierarchy or when the operation affects the lowest level (level 0) in the hierarchy, and you can use one of these formats to update topology geometry objects when the operation affects a level higher than level 0 in the hierarchy:

```
SDO_TOPO_GEOMETRY (topology         VARCHAR2,
                   tg_type          NUMBER,
```

```
                    tg_layer_id      NUMBER,
                    add_topo_ids     SDO_TOPO_OBJECT_ARRAY,
                    delete_topo_ids  SDO_TOPO_OBJECT_ARRAY)

SDO_TOPO_GEOMETRY (topology         VARCHAR2,
                    table_name       VARCHAR2,
                    column_name      VARCHAR2,
                    tg_type          NUMBER,
                    add_topo_ids     SDO_TOPO_OBJECT_ARRAY,
                    delete_topo_ids  SDO_TOPO_OBJECT_ARRAY)
```

For example, you could use one of these constructor formats to add an edge to a linear feature or to remove an obsolete edge from a feature.

The SDO_TOPO_OBJECT_ARRAY type definition and the requirements for the TG_TYPE and TOPO_IDS attribute values are as described in Section 1.6.2.1.

You can specify values for both the ADD_TOPO_IDS and DELETE_TOPO_IDS attributes, or you can specify values for one attribute and specify the other as null; however, you cannot specify null values for both ADD_TOPO_IDS and DELETE_TOPO_IDS.

Example 1–6 shows two SDO_TOPO_GEOMETRY constructors, one in each format. Each constructor removes two faces from the CITY_DATA topology in the LAND_PARCELS table, which is defined in Example 1–12 in Section 1.12.

Example 1–6 UPDATE Using Constructor with SDO_TOPO_OBJECT_ARRAY

```
UPDATE land_parcels l SET l.feature = SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    NULL, -- No topological elements to be added
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 3), -- face_id = 3
      SDO_TOPO_OBJECT (6, 3))) -- face_id = 6
WHERE l.feature_name = 'P1';

UPDATE land_parcels l SET l.feature = SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    'LAND_PARCELS', -- Table name
    'FEATURE', -- Column name
    3, -- Topology geometry type (polygon/multipolygon)
    NULL, -- No topological elements to be added
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 3), -- face_id = 3
      SDO_TOPO_OBJECT (6, 3))) -- face_id = 6
WHERE l.feature_name = 'P1A';
```

1.6.2.4 Constructors for Update Operations: Specifying Lower-Level Features

The SDO_TOPO_GEOMETRY type has the following constructors for update operations in which you specify features in the next lower level of the hierarchy. You can use one of these formats to update topology geometry objects when the operation affects a level higher than level 0 in the hierarchy:

```
SDO_TOPO_GEOMETRY (topology         VARCHAR2,
                    tg_type          NUMBER,
                    tg_layer_id      NUMBER,
                    add_topo_ids     SDO_TGL_OBJECT_ARRAY,
                    delete_topo_ids  SDO_TGL_OBJECT_ARRAY)
```

```
SDO_TOPO_GEOMETRY (topology         VARCHAR2,
                   table_name       VARCHAR2,
                   column_name      VARCHAR2,
                   tg_type          NUMBER,
                   add_topo_ids     SDO_TGL_OBJECT_ARRAY,
                   delete_topo_ids  SDO_TGL_OBJECT_ARRAY)
```

For example, you could use one of these constructor formats to add an edge to a linear feature or to remove an obsolete edge from a feature.

The SDO_TGL_OBJECT_ARRAY type definition and the requirements for its attribute values are as described in Section 1.6.2.2.

You can specify values for both the ADD_TOPO_IDS and DELETE_TOPO_IDS attributes, or you can specify values for one attribute and specify the other as null; however, you cannot specify null values for both ADD_TOPO_IDS and DELETE_TOPO_IDS.

Example 1–7 shows two SDO_TOPO_GEOMETRY constructors, one in each format. Each constructor deletes the land parcel with the ID value of 2 from a feature (named `BG1` in the first format and `BG1A` in the second format, though each feature has the same definition) from the `CITY_DATA` topology in the BLOCK_GROUPS table, which is the feature table for the Block Groups level in the topology geometry layer hierarchy. The Block Groups level is the parent of the Land Parcels level at the bottom of the hierarchy.

Example 1–7 UPDATE Using Constructor with SDO_TGL_OBJECT_ARRAY

```
UPDATE block_groups b SET b.feature = SDO_TOPO_GEOMETRY(
  'LAND_USE_HIER',
  3, -- Topology geometry type (polygon/multipolygon)
  2, -- TG_LAYER_ID for block groups (from ALL_SDO_TOPO_METADATA)
  null, -- No IDs to add
  SDO_TGL_OBJECT_ARRAY (
    SDO_TGL_OBJECT (1, 2)) -- land parcel ID = 2
  )
WHERE b.feature_name = 'BG1';

UPDATE block_groups b SET b.feature = SDO_TOPO_GEOMETRY(
  'LAND_USE_HIER',
  'BLOCK_GROUPS', -- Feature table
  'FEATURE', -- Feature column
  3, -- Topology geometry type (polygon/multipolygon)
  null, -- No IDs to add
  SDO_TGL_OBJECT_ARRAY (
    SDO_TGL_OBJECT (1, 2)) -- land parcel ID = 2
  )
WHERE b.feature_name = 'BG1A';
```

1.6.3 GET_GEOMETRY Member Function

The SDO_TOPO_GEOMETRY type has a member function GET_GEOMETRY, which you can use to return the SDO_GEOMETRY object for the topology geometry object.

Example 1–8 uses the GET_GEOMETRY member function to return the SDO_GEOMETRY object for the topology geometry object associated with the land parcel named `P1`.

Example 1–8 GET_GEOMETRY Member Function

```
SELECT l.feature_name, l.feature.get_geometry()
  FROM land_parcels l WHERE l.feature_name = 'P1';

FEATURE_NAME
------------------------------
L.FEATURE.GET_GEOMETRY()(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO,
--------------------------------------------------------------------------------
P1
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 3, 1), SDO_ORDINATE_ARRAY(
21, 14, 21, 22, 9, 22, 9, 14, 9, 6, 21, 6, 21, 14))
```

1.6.4 GET_TGL_OBJECTS Member Function

The SDO_TOPO_GEOMETRY type has a member function GET_TGL_OBJECTS, which you can use to return the SDO_TOPO_OBJECT_ARRAY object for a topology geometry object in a geometry layer with a hierarchy level greater than 0 (zero) in a topology with a topology geometry layer hierarchy. (If the layer is at hierarchy level 0 or is in a topology that does not have a topology geometry layer hierarchy, this method returns a null value.)

The SDO_TGL_OBJECT_ARRAY type is described in Section 1.6.2.2.

Example 1–9 uses the GET_TGL_OBJECTS member function to return the SDO_TOPO_OBJECT_ARRAY object for the topology geometry object associated with the block group named `BG2`.

Example 1–9 GET_TGL_OBJECTS Member Function

```
SELECT bg.feature_name, bg.feature.get_tgl_objects()
  FROM block_groups bg WHERE bg.feature_name = 'BG2';

FEATURE_NAME
------------------------------
BG.FEATURE.GET_TGL_OBJECTS()(TGL_ID, TG_ID)
--------------------------------------------------------------------------------
BG2
SDO_TGL_OBJECT_ARRAY(SDO_TGL_OBJECT(1, 3), SDO_TGL_OBJECT(1, 4))
```

1.6.5 GET_TOPO_ELEMENTS Member Function

The SDO_TOPO_GEOMETRY type has a member function GET_TOPO_ELEMENTS, which you can use to return the SDO_TOPO_OBJECT_ARRAY object for the topology geometry object.

The SDO_TOPO_OBJECT_ARRAY type is described in Section 1.6.2.1.

Example 1–8 uses the GET_TOPO_ELEMENTS member function to return the SDO_TOPO_OBJECT_ARRAY object for the topology geometry object associated with the land parcel named `P1`.

Example 1–10 GET_TOPO_ELEMENTS Member Function

```
SELECT l.feature_name, l.feature.get_topo_elements()
  FROM land_parcels l WHERE l.feature_name = 'P1';

FEATURE_NAME
------------------------------
L.FEATURE.GET_TOPO_ELEMENTS()(TOPO_ID, TOPO_TYPE)
--------------------------------------------------------------------------------
```

```
P1
SDO_TOPO_OBJECT_ARRAY(SDO_TOPO_OBJECT(3, 3), SDO_TOPO_OBJECT(6, 3))
```

1.6.6 SDO_LIST_TYPE Type

The SDO_LIST_TYPE type is used to store the EDGE_ID values of island edges and NODE_ID values of island nodes in a face. The SDO_LIST_TYPE type is defined as:

```
CREATE TYPE sdo_list_type as VARRAY(2147483647) OF NUMBER;
```

1.6.7 SDO_EDGE_ARRAY and SDO_NUMBER_ARRAY Types

The SDO_EDGE_ARRAY type is used to specify the coordinates of attached edges affected by a node move operation. The SDO_EDGE_ARRAY type is defined as:

```
CREATE TYPE sdo_edge_array as VARRAY(1000000) OF MDSYS.SDO_NUMBER_ARRAY;
```

The SDO_NUMBER_ARRAY type is a general-purpose type used by Spatial for arrays. The SDO_NUMBER_ARRAY type is defined as:

```
CREATE TYPE sdo_number_array as VARRAY(1048576) OF NUMBER;
```

1.7 Topology Metadata Views

There are two sets of topology metadata views for each schema (user): xxx_SDO_TOPO_INFO and xxx_SDO_TOPO_METADATA, where *xxx* can be USER or ALL. These views are read-only to users; they are created and maintained by Spatial.

The xxx_SDO_TOPO_METADATA views contain the most detailed information, and each xxx_SDO_TOPO_INFO view contains a subset of the information in its corresponding xxx_SDO_TOPO_METADATA view.

1.7.1 xxx_SDO_TOPO_INFO Views

The following views contain basic information about topologies:

- USER_SDO_TOPO_INFO contains topology information for all feature tables owned by the user.
- ALL_SDO_TOPO_INFO contains topology information for all feature tables on which the user has SELECT permission.

The USER_SDO_TOPO_INFO and ALL_SDO_TOPO_INFO views contain the same columns, as shown Table 1–8. (The columns are listed in their order in the view definition.)

Table 1–8 Columns in the xxx_SDO_TOPO_INFO Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2	Owner of the topology
TOPOLOGY	VARCHAR2	Name of the topology
TOPOLOGY_ID	NUMBER	ID number of the topology
TOLERANCE	NUMBER	Tolerance value associated with topology geometries in the topology. (Tolerance is explained in Section 1.2.1.)

Table 1–8 (Cont.) Columns in the xxx_SDO_TOPO_INFO Views

Column Name	Data Type	Purpose
SRID	NUMBER	Coordinate system (spatial reference system) associated with all topology geometry layers in the topology. Is null if no coordinate system is associated; otherwise, it contains a value from the SRID column of the MDSYS.CS_SRS table (described in *Oracle Spatial Developer's Guide*).
TABLE_SCHEMA	VARCHAR2	Name of the schema that owns the table containing the topology geometry layer column
TABLE_NAME	VARCHAR2	Name of the table containing the topology geometry layer column
COLUMN_NAME	VARCHAR2	Name of the column containing the topology geometry layer data
TG_LAYER_ID	NUMBER	ID number of the topology geometry layer
TG_LAYER_TYPE	VARCHAR2	Contains one of the following: POINT, LINE, CURVE, POLYGON, or COLLECTION. (LINE and CURVE have the same meaning.)
TG_LAYER_LEVEL	NUMBER	Hierarchy level number of this topology geometry layer. (Topology geometry layer hierarchy is explained in Section 1.4.)
CHILD_LAYER_ID	NUMBER	ID number of the topology geometry layer that is the child layer of this layer in the topology geometry layer hierarchy. Null if this layer has no child layer or if the topology does not have a topology geometry layer hierarchy. (Topology geometry layer hierarchy is explained in Section 1.4.)
DIGITS_RIGHT_OF_DECIMAL	NUMBER	Number of digits permitted to the right of the decimal point in the expression of any coordinate position when features are added to an existing topology. All incoming features (those passed as arguments to the `addLinearGeometry`, `addPolygonGeometry`, or `addPointGeometry` method in the Java API or the equivalent PL/SQL subprograms) are automatically snapped (truncated) to the number of digits right of the decimal. Default: 16.

1.7.2 xxx_SDO_TOPO_METADATA Views

The following views contain detailed information about topologies:

- USER_SDO_TOPO_METADATA contains topology information for all tables owned by the user.
- ALL_SDO_TOPO_METADATA contains topology information for all tables on which the user has SELECT permission.

The USER_SDO_TOPO_METADATA and ALL_SDO_TOPO_METADATA views contain the same columns, as shown Table 1–9. (The columns are listed in their order in the view definition.)

Table 1–9 Columns in the xxx_SDO_TOPO_METADATA Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2	Owner of the topology

Table 1–9 (Cont.) Columns in the xxx_SDO_TOPO_METADATA Views

Column Name	Data Type	Purpose
TOPOLOGY	VARCHAR2	Name of the topology
TOPOLOGY_ID	NUMBER	ID number of the topology
TOLERANCE	NUMBER	Tolerance value associated with topology geometries in the topology. (Tolerance is explained in Section 1.2.1.)
SRID	NUMBER	Coordinate system (spatial reference system) associated with all topology geometry layers in the topology. Is null if no coordinate system is associated; otherwise, contains a value from the SRID column of the MDSYS.CS_SRS table (described in *Oracle Spatial Developer's Guide*).
TABLE_SCHEMA	VARCHAR2	Name of the schema that owns the table containing the topology geometry layer column
TABLE_NAME	VARCHAR2	Name of the table containing the topology geometry layer column
COLUMN_NAME	VARCHAR2	Name of the column containing the topology geometry layer data
TG_LAYER_ID	NUMBER	ID number of the topology geometry layer
TG_LAYER_TYPE	VARCHAR2	Contains one of the following: POINT, LINE, CURVE, POLYGON, or COLLECTION. (LINE and CURVE have the same meaning.)
TG_LAYER_LEVEL	NUMBER	Hierarchy level number of this topology geometry layer. (Topology geometry layer hierarchy is explained in Section 1.4.)
CHILD_LAYER_ID	NUMBER	ID number of the topology geometry layer that is the child layer of this layer in the topology geometry layer hierarchy. Null if this layer has no child layer or if the topology does not have a geometry layer hierarchy. (Topology geometry layer hierarchy is explained in Section 1.4.)
NODE_SEQUENCE	VARCHAR2	Name of the sequence containing the next available node ID number
EDGE_SEQUENCE	VARCHAR2	Name of the sequence containing the next available edge ID number
FACE_SEQUENCE	VARCHAR2	Name of the sequence containing the next available face ID number
TG_SEQUENCE	VARCHAR2	Name of the sequence containing the next available topology geometry ID number
DIGITS_RIGHT_OF_DECIMAL	NUMBER	Number of digits permitted to the right of the decimal point in the expression of any coordinate position when features are added to an existing topology. All incoming features (those passed as arguments to the `addLinearGeometry`, `addPolygonGeometry`, or `addPointGeometry` method in the Java API or the equivalent PL/SQL subprograms) are automatically snapped (truncated) to the number of digits right of the decimal. Default: 16

1.8 Topology Application Programming Interface

The topology data model application programming interface (API) consists of the following:

- PL/SQL functions and procedures in the SDO_TOPO package (described in Chapter 3) and the SDO_TOPO_MAP package (described in Chapter 4)
- PL/SQL topology operators (described in Section 1.8.1)
- Java API (described in Section 1.8.2)

1.8.1 Topology Operators

With the topology data model PL/SQL API, you can use the Oracle Spatial operators, except for the following:

- SDO_RELATE (but you can use the SDO_RELATE convenience operators that do not use the `mask` parameter)
- SDO_NN
- SDO_NN_DISTANCE
- SDO_WITHIN_DISTANCE

To use spatial operators with the topology data model, you must understand the usage and reference information about Spatial operators, which are documented in *Oracle Spatial Developer's Guide*. This section describes only additional information or differences that apply to using spatial operators with topologies. Otherwise, unless this section specifies otherwise, the operator-related information in *Oracle Spatial Developer's Guide* applies to the use of operators with topology data.

When you use spatial operators with topologies, the formats of the first two parameters can be any one of the following:

- Two topology geometry objects (type SDO_TOPO_GEOMETRY)

  For example, the following statement finds all city streets features that have any interaction with a land parcel feature named P3. (This example uses definitions and data from Section 1.12.1.)

```
SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_ANYINTERACT (c.feature, l.feature) = 'TRUE';

FEATURE_NAME
------------------------------
R1
```

- A topology geometry object (type SDO_TOPO_GEOMETRY) as the first parameter and a spatial geometry (type SDO_GEOMETRY) as the second parameter

  For example, the following statement finds all city streets features that have any interaction with a geometry object that happens to be a polygon identical to the boundary of the land parcel feature named P3. (This example uses definitions and data from Section 1.12.2.)

```
SELECT c.feature_name FROM city_streets c
  WHERE SDO_ANYINTERACT (c.feature,
    SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1),
      SDO_ORDINATE_ARRAY(35,6, 47,6, 47,14, 47,22, 35,22, 35,14, 35,6))) =
'TRUE';
```

```
FEATURE_NAME
------------------------------
R1
```

- A topology geometry object (type SDO_TOPO_GEOMETRY) as the first parameter and a topology object array object (type SDO_TOPO_OBJECT_ARRAY) as the second parameter

  For example, the following statement finds all city streets features that have any interaction with an SDO_TOPO_OBJECT_ARRAY object that happens to be identical to the land parcel feature named P3. (This example uses definitions and data from Section 1.12.2.)

```
SELECT c.feature_name FROM city_streets c WHERE
  SDO_ANYINTERACT (c.feature,
   SDO_TOPO_OBJECT_ARRAY (SDO_TOPO_OBJECT (5, 3), SDO_TOPO_OBJECT (8, 3)))
   = 'TRUE';

FEATURE_NAME
------------------------------
R1
```

Example 1–11 shows different topology operators checking for a specific relationship between city streets features and the land parcel named P3. The first statement shows the SDO_FILTER operator, and the remaining statements show the SDO_RELATE convenience operators that include the "mask" in the operator name. With the convenience operators in this example, only SDO_ANYINTERACT, SDO_OVERLAPBDYINTERSECT, and SDO_OVERLAPS return any resulting feature data. (As Figure 1–3 in Section 1.3.1 shows, the only street feature to have any interaction with land parcel P3 is R1.) All statements in Example 1–11 use the format where the first two parameters are topology geometry objects.

Example 1–11 Topology Operators

```
-- SDO_FILTER
SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_FILTER (c.feature, l.feature) = 'TRUE';

FEATURE_NAME
------------------------------
R1

-- SDO_RELATE convenience operators
SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_ANYINTERACT (c.feature, l.feature) = 'TRUE';

FEATURE_NAME
------------------------------
R1

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_CONTAINS (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
```

```
    SDO_COVEREDBY (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_COVERS (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_EQUAL (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_INSIDE (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_ON (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_OVERLAPBDYINTERSECT (c.feature, l.feature) = 'TRUE';

FEATURE_NAME
------------------------------
R1

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_OVERLAPBDYDISJOINT (c.feature, l.feature) = 'TRUE';

no rows selected

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_OVERLAPS (c.feature, l.feature) = 'TRUE';

FEATURE_NAME
------------------------------
R1

SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
    SDO_TOUCH (c.feature, l.feature) = 'TRUE';

no rows selected
```

See also the usage notes for the SDO_TOPO.RELATE function in Chapter 3.

1.8.2 Topology Data Model Java Interface

The Java client interface for the topology data model consists of the following classes:

- `TopoMap`: class that stores edges, nodes, and faces, and provides methods for adding and deleting elements while maintaining topological consistency both in the cache and in the underlying database tables
- `Edge`: class for an edge
- `Face`: class for a face
- `Node`: class for a node
- `Point2DD`: class for a point
- `CompGeom`: class for static computational geometry methods
- `InvalidTopoOperationException`: class for the invalid topology operation exception
- `TopoValidationException`: class for the topology validation failure exception
- `TopoEntityNotFoundException`: class for the entity not found exception
- `TopoDataException`: class for the invalid input exception

For detailed reference information about the topology data model classes, as well as some usage information about the Java API, see the Javadoc-generated API documentation: open `index.html` in a directory that includes the path `sdotopo/doc/javadoc`.

1.9 Exporting and Importing Topology Data

You can export a topology from one database and import it into a new topology with the same name, structures, and data in another database, as long as the target database does not already contain a topology with the same name as the exported topology. To export topology data from one database and import it into another database, follow the steps in this section.

In the database with the topology data to be exported, perform the following actions:

1. Connect to the database as the owner of the topology.
2. Execute the SDO_TOPO.PREPARE_FOR_EXPORT procedure (documented in Chapter 3), to create the topology export information table, with a name in the format <topology-name>_EXP$. (This table contains the same columns as the USER_SDO_TOPO_INFO and ALL_SDO_TOPO_INFO views. These columns are described in Table 1–8 in Section 1.7.1.)

   For example, preparing the sample `CITY_DATA` topology for export creates the CITY_DATA_EXP$ table.
3. Export all tables related to the topology, including the feature tables and the <topology-name>_EDGE$, <topology-name>_FACE$, <topology-name>_HISTORY$, <topology-name>_NODE$, <topology-name>_RELATION$, and <topology-name>_EXP$ tables. The names of feature tables (if they exist) are stored in the topology metadata.

   This creates a file with the extension `.dmp` (for example, `city_data.dmp`).

In the database into which to import the topology data, perform the following actions:

1. Connect to the target database, that is, the database in which to create a topology with the same name, structures, and data as the topology exported from the source

database. Connect as the user for the schema that is to own the topology to be created.

2. Ensure that the target database does not already contain a topology with the same name as the topology in the `.dmp` file.
3. Import the tables from the `.dmp` file that you created when you exported the topology data. Specify the `indexes=N` option.
4. If you have imported the topology tables into a different schema than the one used for the topology in the source database, update the OWNER column value in all rows of the <topology-name>_EXP$ table to reflect the schema name in the current (target) database.
5. Execute the SDO_TOPO.INITIALIZE_AFTER_IMPORT procedure, which creates the topology and performs other operations, as necessary, to make the topology ready for use.

1.10 Cross-Schema Topology Usage and Editing

This section contains requirements and guidelines for using and editing topologies when multiple database users (schemas) are involved.

1.10.1 Cross-Schema Topology Usage

The following considerations apply when one user owns a topology and another user owns a topology geometry layer table. In the following, assume that user A owns the `CITY_DATA` topology and that user B owns the CITY_STREETS topology geometry layer table.

- The owner of the topology must create the topology and initialize the metadata. In this example, user A must perform these actions.
- Only the owner of a topology can add layers to or delete layers from the topology. Therefore, if you add a table owned by another user to a topology, or when you remove such a table from the topology, you must qualify the table name with the schema name. For example, user A could add the CITY_STREETS table owned by user B to the `CITY_DATA` topology with the following statement:

  ```
  EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'B.CITY_STREETS',
  'FEATURE', 'LINE');
  ```

  User A could delete the CITY_STREETS table owned by user B from the `CITY_DATA` topology with the following statement:

  ```
  EXECUTE SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER('CITY_DATA', 'B.CITY_STREETS',
  'FEATURE');
  ```

- The owner of the topology should grant the SELECT privilege on the node, edge, and face information tables to the owner of the topology geometry layer table. For example, user A should grant the SELECT privilege on the CITY_DATA_NODE$, CITY_DATA_EDGE$, and CITY_DATA_FACE$ tables to user B.
- The owner of the topology geometry layer table should grant the SELECT and INDEX privileges on that table to the owner of the topology. For example, user B should grant the SELECT and INDEX privileges on the CITY_STREETS table to user A.

  The owner of the topology geometry layer table should also grant appropriate privileges to other users that need to access the table. For read-only access, grant

the SELECT privilege on the table to a user; for read/write access, grant the INSERT, SELECT, and UPDATE privileges.

1.10.2 Cross-Schema Topology Editing

The following considerations apply when one user owns a topology and another user wants to edit the topology. In the following, assume that user A owns the CITY_DATA topology and that user B wants to edit that topology.

- The owner of the topology should grant the following privileges to users who can edit the topology: INSERT, SELECT, and UPDATE on the node, edge, face, and relationship information tables, and SELECT on the node, edge, and face sequences used to generate ID numbers for the topology primitives. For example, user A could grant the following privileges to user B, where the table names end with $ and the sequence names end with _S:

```
GRANT insert,select,update ON city_data_node$ TO b;
GRANT insert,select,update ON city_data_edge$ TO b;
GRANT insert,select,update ON city_data_face$ TO b;
GRANT insert,select,update ON city_data_relation$ TO b;
GRANT select ON city_data_node_s TO b;
GRANT select ON city_data_edge_s TO b;
GRANT select ON city_data_face_s TO b;
```

- When a user who does not own the topology edits that topology, the owner's schema name should be specified with the topology name in functions and procedures that accept the topology name as an input parameter. For example, user B should specify the topology as A.CITY_DATA, not just CITY_DATA.

For information about editing topologies, see Chapter 2.

1.11 Function-Based Indexes Not Supported

You cannot create a function-based index on a column of type SDO_TOPO_GEOMETRY. (Function-based indexes are explained in *Oracle Database Advanced Application Developer's Guide* and *Oracle Database Administrator's Guide*.)

1.12 Topology Examples (PL/SQL)

This section presents simplified PL/SQL examples that perform topology data model operations. The examples refer to concepts that are explained in this chapter. They use SDO_TOPO and SDO_TOPO_MAP functions and procedures, which are documented in Chapter 3 and Chapter 4, and the SDO_ANYINTERACT topology operator (see Section 1.8.1).

Both examples are based on the "city data" topology shown in Figure 1–1 in Section 1.2, and the features shown in Figure 1–3 in Section 1.3.1. However, the topologies created are not identical, because the topology built from Spatial geometries (Example 1–13) does not contain all the edges, nodes, and faces that are defined for the topology build from topology data (Example 1–12).

1.12.1 Topology Built from Topology Data

Example 1–12 uses a topology built from edge, node, and face data.

Example 1–12 Topology Built from Topology Data

```
------------------------------
```

```
-- Main steps for using the topology data model with a topology
-- built from edge, node, and face data
------------------------------
-- 1. Create a topology.
-- 2. Load (normally bulk-load) topology data (node, edge, and face tables).
-- 3. Create feature tables.
-- 4. Associate feature tables with the topology.
-- 5. Initialize topology metadata.
-- 6. Load feature tables using the SDO_TOPO_GEOMETRY constructor.
-- 7. Query the data.
-- 8. Optionally, edit data using the PL/SQL or Java API.

-- 1. Create the topology. (Null SRID in this example.)
EXECUTE SDO_TOPO.CREATE_TOPOLOGY('CITY_DATA', 0.00005);

-- 2. Load topology data (node, edge, and face tables).
--  Use INSERT statements here instead of a bulk-load utility.

-- 2A. Insert data into <topology_name>_EDGE$ table.

-- E1
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(1, 1, 1, 1, 1, -1, -1, 1, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(8,30, 16,30, 16,38, 3,38, 3,30, 8,30)));
-- E2
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(2, 2, 2, 3, -3, -2, -2, 2, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(25,30, 31,30, 31,40, 17,40, 17,30, 25,30)));
-- E3
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(3, 2, 3, -3, 2, 2, 3, 2, 2,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(25,30, 25,35)));
-- E4
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(4, 5, 6, -5, -4, 4, 5, -1, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(36,38, 38,35, 41,34, 42,33, 45,32, 47,28, 50,28, 52,32,
57,33)));
-- E5
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(5, 7, 6, -4, -5, 5, 4, -1, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(41,40, 45,40, 47,42, 62,41, 61,38, 59,39, 57,36,
57,33)));
-- E6
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
```

```
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(6, 16, 17, 7, 21, -21, 19, -1, 3,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,22, 21,22)));
-- E7
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(7, 17, 18, 8, 6, -19, 17, -1, 4,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(21,22, 35,22)));
-- E8
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(8, 18, 19, -15, 7, -17, 15, -1, 5,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,22, 47,22)));
-- E9
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(9, 15, 14, 19, -21, -22, 20, 3, 6,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,14, 21,14)));
-- E10
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(10, 13, 14, -20, 18, 17, -19, 7, 4,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,14, 21,14)));
-- E11
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(11, 13, 12, 15, -17, -18, 16, 5, 8,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,14, 47,14)));
-- E12
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(12, 8, 9, 20, -22, 22, -13, 6, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,6, 21,6)));
-- E13
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(13, 9, 10, 18, -20, -12, -14, 7, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(21,6, 35,6)));
-- E14
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(14, 10, 11, 16, -18, -13, -16, 8, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,6, 47,6)));
```

```
-- E15
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(15, 12, 19, -8, 11, -16, 8, 5, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(47,14, 47,22)));
-- E16
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(16, 11, 12, -11, 14, -14, -15, 8, -1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(47,6, 47,14)));
-- E17
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(17, 13, 18, -7, -10, 11, -8, 4, 5,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,14, 35,22)));
-- E18
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(18, 10, 13, 10, 13, 14, -11, 7, 8,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(35,6, 35,14)));
-- E19
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(19, 14, 17, -6, 9, -10, -7, 3, 4,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(21,14, 21,22)));
-- E20
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(20, 9, 14, -9, 12, 13, 10, 6, 7,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(21,6, 21,14)));
-- E21
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(21, 15, 16, 6, 22, 9, -6, -1, 3,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,14, 9,22)));
-- E22
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(22, 8, 15, 21, -12, 12, -9, -1, 6,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,6, 9,14)));
-- E25
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
```

```
 VALUES(25, 21, 22, -25, -25, 25, 25, 1, 1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,35, 13,35)));
-- E26
INSERT INTO city_data_edge$ (edge_id, start_node_id, end_node_id,
    next_left_edge_id, prev_left_edge_id, next_right_edge_id,
    prev_right_edge_id, left_face_id, right_face_id, geometry)
 VALUES(26, 20, 20, 26, 26, -26, -26, 9, 1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(4,31, 7,31, 7,34, 4,34, 4,31)));

-- 2B. Insert data into <topology_name>_NODE$ table.

-- N1
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(1, 1, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8,30,NULL), NULL, NULL));
-- N2
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(2, 2, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(25,30,NULL), NULL, NULL));
-- N3
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(3, -3, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(25,35,NULL), NULL, NULL));
-- N4
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(4, NULL, 2,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(20,37,NULL), NULL, NULL));
-- N5
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(5, 4, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(36,38,NULL), NULL, NULL));
-- N6
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(6, -4, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(57,33,NULL), NULL, NULL));
-- N7
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(7, 5, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(41,40,NULL), NULL, NULL));
-- N8
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(8, 12, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,6,NULL), NULL, NULL));
-- N9
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(9, 20, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(21,6,NULL), NULL, NULL));
-- N10
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(10, 18, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(35,6,NULL), NULL, NULL));
-- N11
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(11, -14, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(47,6,NULL), NULL, NULL));
-- N12
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(12, 15, NULL,
```

```
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(47,14,NULL), NULL, NULL));
-- N13
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(13, 17, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(35,14,NULL), NULL, NULL));
-- N14
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(14, 19, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(21,14,NULL), NULL, NULL));
-- N15
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(15, 21, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,14,NULL), NULL, NULL));
-- N16
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(16, 6, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,22,NULL), NULL, NULL));
-- N17
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(17, 7, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(21,22,NULL), NULL, NULL));
-- N18
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(18, 8, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(35,22,NULL), NULL, NULL));
-- N19
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(19, -15, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(47,22,NULL), NULL, NULL));
-- N20
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(20, 26, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(4,31,NULL), NULL, NULL));
-- N21
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(21, 25, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,35,NULL), NULL, NULL));
-- N22
INSERT INTO city_data_node$ (node_id, edge_id, face_id, geometry)
 VALUES(22, -25, NULL,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(13,35,NULL), NULL, NULL));

-- 2C. Insert data into <topology_name>_FACE$ table.

-- F0 (id = -1, not 0)
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(-1, NULL, SDO_LIST_TYPE(-1, -2, 4, 6),
   SDO_LIST_TYPE(), NULL);
-- F1
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(1, 1, SDO_LIST_TYPE(25, -26), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(3,30, 15,38)));
-- F2
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(2, 2, SDO_LIST_TYPE(), SDO_LIST_TYPE(4),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
```

```
    SDO_ORDINATE_ARRAY(17,30, 31,40)));
-- F3
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(3, 19, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(9,14, 21,22)));
-- F4
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(4, 17, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(21,14, 35,22)));
-- F5
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(5, 15, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(35,14, 47,22)));
-- F6
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(6, 20, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(9,6, 21,14)));
-- F7
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(7, 10, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(21,6, 35,14)));
-- F8
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(8, 16, SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(35,6, 47,14)));
-- F9
INSERT INTO city_data_face$ (face_id, boundary_edge_id,
    island_edge_id_list, island_node_id_list, mbr_geometry)
 VALUES(9,26,SDO_LIST_TYPE(), SDO_LIST_TYPE(),
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(4,31, 7,34)));

-- 3. Create feature tables.

CREATE TABLE land_parcels ( -- Land parcels (selected faces)
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE city_streets ( -- City streets (selected edges)
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE traffic_signs ( -- Traffic signs (selected nodes)
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

-- 4. Associate feature tables with the topology.
--    Add the three topology geometry layers to the CITY_DATA topology.
```

```
--    Any order is OK.

EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'LAND_PARCELS','FEATURE',
'POLYGON');
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'TRAFFIC_SIGNS','FEATURE',
'POINT');
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'CITY_STREETS',
'FEATURE','LINE');

--  As a result, Spatial generates a unique TG_LAYER_ID for each layer in
--  the topology metadata (USER/ALL_SDO_TOPO_METADATA).

-- 5. Initialize topology metadata.
EXECUTE SDO_TOPO.INITIALIZE_METADATA('CITY_DATA');

-- 6. Load feature tables using the SDO_TOPO_GEOMETRY constructor.

-- Each topology feature can consist of one or more objects (face, edge, node)
-- of an appropriate type. For example, a land parcel can consist of one face,
-- or two or more faces, as specified in the SDO_TOPO_OBJECT_ARRAY.

-- There are typically fewer features than there are faces, nodes, and edges.
-- In this example, the only features are these:
-- Area features (land parcels): P1, P2, P3, P4, P5
-- Point features (traffic signs): S1, S2, S3, S4
-- Linear features (roads/streets): R1, R2, R3, R4

-- 6A. Load LAND_PARCELS table.

-- P1
INSERT INTO land_parcels VALUES ('P1', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 3), -- face_id = 3
      SDO_TOPO_OBJECT (6, 3))) -- face_id = 6
);
-- P2
INSERT INTO land_parcels VALUES ('P2', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (4, 3), -- face_id = 4
      SDO_TOPO_OBJECT (7, 3))) -- face_id = 7
);
-- P3
INSERT INTO land_parcels VALUES ('P3', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (5, 3), -- face_id = 5
      SDO_TOPO_OBJECT (8, 3))) -- face_id = 8
);
-- P4
```

```
INSERT INTO land_parcels VALUES ('P4', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (2, 3))) -- face_id = 2
);
-- P5 (Includes F1, but not F9.)
INSERT INTO land_parcels VALUES ('P5', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    3, -- Topology geometry type (polygon/multipolygon)
    1, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (1, 3))) -- face_id = 1
);

-- 6B. Load TRAFFIC_SIGNS table.

-- S1
INSERT INTO traffic_signs VALUES ('S1', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    1, -- Topology geometry type (point)
    2, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (14, 1))) -- node_id = 14
);
-- S2
INSERT INTO traffic_signs VALUES ('S2', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    1, -- Topology geometry type (point)
    2, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (13, 1))) -- node_id = 13
);
-- S3
INSERT INTO traffic_signs VALUES ('S3', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    1, -- Topology geometry type (point)
    2, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (6, 1))) -- node_id = 6
);
-- S4
INSERT INTO traffic_signs VALUES ('S4', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    1, -- Topology geometry type (point)
    2, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (4, 1))) -- node_id = 4
);

-- 6C. Load CITY_STREETS table.
-- (Note: "R" in feature names is for "Road", because "S" is used for signs.)
```

```
-- R1
INSERT INTO city_streets VALUES ('R1', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    2, -- Topology geometry type (line string)
    3, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (9, 2),
      SDO_TOPO_OBJECT (-10, 2),
      SDO_TOPO_OBJECT (11, 2))) -- edge_ids = 9, -10, 11
);
-- R2
INSERT INTO city_streets VALUES ('R2', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    2, -- Topology geometry type (line string)
    3, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (4, 2),
      SDO_TOPO_OBJECT (-5, 2))) -- edge_ids = 4, -5
);
-- R3
INSERT INTO city_streets VALUES ('R3', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    2, -- Topology geometry type (line string)
    3, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (25, 2))) -- edge_id = 25
);
-- R4
INSERT INTO city_streets VALUES ('R4', -- Feature name
  SDO_TOPO_GEOMETRY(
    'CITY_DATA', -- Topology name
    2, -- Topology geometry type (line string)
    3, -- TG_LAYER_ID for this topology (from ALL_SDO_TOPO_METADATA)
    SDO_TOPO_OBJECT_ARRAY (
      SDO_TOPO_OBJECT (3, 2))) -- edge_id = 3
);

-- 7. Query the data.

SELECT a.feature_name, a.feature.tg_id, a.feature.get_geometry()
FROM land_parcels a;

/* Window is city_streets */
SELECT  a.feature_name, b.feature_name
  FROM city_streets b,
     land_parcels a
  WHERE  b.feature_name like 'R%' AND
     sdo_anyinteract(a.feature, b.feature) = 'TRUE'
  ORDER BY b.feature_name, a.feature_name;

-- Find all streets that have any interaction with land parcel P3.
-- (Should return only R1.)
SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
   SDO_ANYINTERACT (c.feature, l.feature) = 'TRUE';

-- Find all land parcels that have any interaction with traffic sign S1.
```

```
-- (Should return P1 and P2.)
SELECT l.feature_name FROM land_parcels l, traffic_signs t
  WHERE t.feature_name = 'S1' AND
   SDO_ANYINTERACT (l.feature, t.feature) = 'TRUE';

-- Get the geometry for land parcel P1.
SELECT l.feature_name, l.feature.get_geometry()
  FROM land_parcels l WHERE l.feature_name = 'P1';

-- Get the boundary of face with face_id 3.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 3) FROM DUAL;

-- Get the topological elements for land parcel P2.
-- CITY_DATA layer, land parcels (tg_ layer_id = 1), parcel P2 (tg_id = 2)
SELECT SDO_TOPO.GET_TOPO_OBJECTS('CITY_DATA', 1, 2) FROM DUAL;
```

1.12.2 Topology Built from Spatial Geometries

Example 1–13 uses a topology built from Oracle Spatial geometry data.

Example 1–13 Topology Built from Spatial Geometries

```
------------------------------
-- Main steps for using the topology data model with a topology
-- built from Spatial geometry data
------------------------------
-- 1. Create the topology.
-- 2. Insert the universe face (F0). (id = -1, not 0)
-- 3. Create feature tables.
-- 4. Associate feature tables with the topology.
-- 5. Create a TopoMap object and load the whole topology into
--     cache for updating.
-- 6. Load feature tables, inserting data from the spatial tables and
--     using SDO_TOPO_MAP.CREATE_FEATURE.
-- 7. Initialize topology metadata.
-- 8. Query the data.
-- 9. Optionally, edit the data using the PL/SQL or Java API.

-- Preliminary work for this example (things normally done to use
-- data with Oracle Spatial):
-- * Create the spatial tables.
-- * Update the Spatial metadata (USER_SDO_GEOM_METADATA).
-- * Load data into the spatial tables.
-- * Validate the spatial data (validate the layers).
-- * Create the spatial indexes.

-- Create spatial tables of geometry features: names and geometries.

CREATE TABLE city_streets_geom ( -- City streets/roads
  name VARCHAR2(30) PRIMARY KEY,
  geometry SDO_GEOMETRY);

CREATE TABLE traffic_signs_geom ( -- Traffic signs
  name VARCHAR2(30) PRIMARY KEY,
  geometry SDO_GEOMETRY);

CREATE TABLE land_parcels_geom ( -- Land parcels
  name VARCHAR2(30) PRIMARY KEY,
  geometry SDO_GEOMETRY);
```

```
INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
     DIMINFO,
     SRID)
  VALUES (
    'CITY_STREETS_GEOM',
    'GEOMETRY',
    SDO_DIM_ARRAY(
      SDO_DIM_ELEMENT('X', 0, 65, 0.005),
      SDO_DIM_ELEMENT('Y', 0, 45, 0.005)
      ),
    NULL -- SRID
  );

INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
     DIMINFO,
     SRID)
  VALUES (
    'TRAFFIC_SIGNS_GEOM',
    'GEOMETRY',
    SDO_DIM_ARRAY(
      SDO_DIM_ELEMENT('X', 0, 65, 0.005),
      SDO_DIM_ELEMENT('Y', 0, 45, 0.005)
      ),
    NULL -- SRID
  );

INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
     DIMINFO,
     SRID)
  VALUES (
    'LAND_PARCELS_GEOM',
    'GEOMETRY',
    SDO_DIM_ARRAY(
      SDO_DIM_ELEMENT('X', 0, 65, 0.005),
      SDO_DIM_ELEMENT('Y', 0, 45, 0.005)
      ),
    NULL -- SRID
  );

-- Load these tables (names and geometries for city streets/roads,
-- traffic signs, and land parcels).

-- Insert data into city street line geometries.

-- R1
INSERT INTO city_streets_geom VALUES('R1',
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,14, 21,14, 35,14, 47,14)));

-- R2
INSERT INTO city_streets_geom VALUES('R2',
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(36,38, 38,35, 41,34, 42,33, 45,32, 47,28, 50,28, 52,32,
57,33, 57,36, 59,39, 61,38, 62,41, 47,42, 45,40, 41,40)));
```

```
-- R3
INSERT INTO city_streets_geom VALUES('R3',
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(9,35, 13,35)));

-- R4
INSERT INTO city_streets_geom VALUES('R4',
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(25,30, 25,35)));

-- Insert data into traffic sign point geometries.

-- S1
INSERT INTO traffic_signs_geom VALUES('S1',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(21,14,NULL), NULL, NULL));

-- S2
INSERT INTO traffic_signs_geom VALUES('S2',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(35,14,NULL), NULL, NULL));

-- S3
INSERT INTO traffic_signs_geom VALUES('S3',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(57,33,NULL), NULL, NULL));

-- S4
INSERT INTO traffic_signs_geom VALUES('S4',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(20,37,NULL), NULL, NULL));

-- Insert data into land parcel polygon geometries.

-- P1
INSERT INTO land_parcels_geom VALUES('P1',
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
    SDO_ORDINATE_ARRAY(9,6, 21,6, 21,14, 21,22, 9,22, 9,14, 9,6)));

-- P2
INSERT INTO land_parcels_geom VALUES('P2',
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1),
    SDO_ORDINATE_ARRAY(21,6, 35,6, 35,14, 35,22, 21,22, 21,14, 21,6)));

-- P3
INSERT INTO land_parcels_geom VALUES('P3',
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1),
    SDO_ORDINATE_ARRAY(35,6, 47,6, 47,14, 47,22, 35,22, 35,14, 35,6)));

-- P4
INSERT INTO land_parcels_geom VALUES('P4',
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1),
    SDO_ORDINATE_ARRAY(17,30, 31,30, 31,40, 17,40, 17,30)));

-- P5 (polygon with a hole; exterior ring and one interior ring)
INSERT INTO land_parcels_geom VALUES('P5',
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1, 11,2003,1),
      SDO_ORDINATE_ARRAY(3,30, 16,30, 16,38, 3,38, 3,30, 4,31, 4,34, 7,34, 7,31,
4,31)));

-- Validate the layers.
create table val_results (sdo_rowid ROWID, result VARCHAR2(2000));
call SDO_GEOM.VALIDATE_LAYER_WITH_CONTEXT('CITY_STREETS_GEOM','GEOMETRY','VAL_
```

```
RESULTS');
SELECT * from val_results;
truncate table val_results;
call SDO_GEOM.VALIDATE_LAYER_WITH_CONTEXT('TRAFFIC_SIGNS_GEOM','GEOMETRY','VAL_
RESULTS');
SELECT * from val_results;
truncate table val_results;
call SDO_GEOM.VALIDATE_LAYER_WITH_CONTEXT('LAND_PARCELS_GEOM','GEOMETRY','VAL_
RESULTS');
SELECT * from val_results;
drop table val_results;

-- Create the spatial indexes.
CREATE INDEX city_streets_geom_idx ON city_streets_geom(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
CREATE INDEX traffic_signs_geom_idx ON traffic_signs_geom(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
CREATE INDEX land_parcels_geom_idx ON land_parcels_geom(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;

-- Start the main steps for using the topology data model with a
-- topology built from Spatial geometry data.

-- 1. Create the topology. (Null SRID in this example.)
EXECUTE SDO_TOPO.CREATE_TOPOLOGY('CITY_DATA', 0.00005);

-- 2. Insert the universe face (F0). (id = -1, not 0)
INSERT INTO CITY_DATA_FACE$ values (
  -1, NULL, SDO_LIST_TYPE(), SDO_LIST_TYPE(), NULL);

COMMIT;

-- 3. Create feature tables.

CREATE TABLE city_streets ( -- City streets/roads
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE traffic_signs ( -- Traffic signs
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

CREATE TABLE land_parcels ( -- Land parcels
  feature_name VARCHAR2(30) PRIMARY KEY,
  feature SDO_TOPO_GEOMETRY);

-- 4. Associate feature tables with the topology.
--    Add the three topology geometry layers to the CITY_DATA topology.
--    Any order is OK.

EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'CITY_STREETS',
'FEATURE','LINE');
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'TRAFFIC_SIGNS','FEATURE',
'POINT');
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'LAND_PARCELS','FEATURE',
'POLYGON');

--  As a result, Spatial generates a unique TG_LAYER_ID for each layer in
--  the topology metadata (USER/ALL_SDO_TOPO_METADATA).
```

```
-- 5. Create a TopoMap object and load the whole topology into cache for updating.

EXECUTE SDO_TOPO_MAP.CREATE_TOPO_MAP('CITY_DATA', 'CITY_DATA_TOPOMAP');
EXECUTE SDO_TOPO_MAP.LOAD_TOPO_MAP('CITY_DATA_TOPOMAP', 'true');

-- 6. Load feature tables, inserting data from the spatial tables and
--     using SDO_TOPO_MAP.CREATE_FEATURE.

BEGIN
  FOR street_rec IN (SELECT name, geometry FROM city_streets_geom) LOOP
   INSERT INTO city_streets VALUES(street_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'CITY_STREETS', 'FEATURE',
         street_rec.geometry));
  END LOOP;

  FOR sign_rec IN (SELECT name, geometry FROM traffic_signs_geom) LOOP
   INSERT INTO traffic_signs VALUES(sign_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'TRAFFIC_SIGNS', 'FEATURE',
         sign_rec.geometry));
  END LOOP;

  FOR parcel_rec IN (SELECT name, geometry FROM land_parcels_geom) LOOP
   INSERT INTO land_parcels VALUES(parcel_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'LAND_PARCELS', 'FEATURE',
         parcel_rec.geometry));
  END LOOP;
END;
/

CALL SDO_TOPO_MAP.COMMIT_TOPO_MAP();
CALL SDO_TOPO_MAP.DROP_TOPO_MAP('CITY_DATA_TOPOMAP');

-- 7. Initialize topology metadata.
EXECUTE SDO_TOPO.INITIALIZE_METADATA('CITY_DATA');

-- 8. Query the data.

SELECT a.feature_name, a.feature.tg_id, a.feature.get_geometry()
FROM land_parcels a;

SELECT a.feature_name, a.feature.tg_id, a.feature.get_geometry()
FROM city_streets a;

SELECT a.feature_name, a.feature.tg_id, a.feature.get_geometry()
FROM traffic_signs a;

SELECT sdo_topo.get_face_boundary('CITY_DATA', face_id), face_id
FROM city_data_face$;

SELECT sdo_topo.get_face_boundary('CITY_DATA', face_id), face_id
FROM city_data_face$;

SELECT sdo_topo.get_face_boundary('CITY_DATA', face_id, 'TRUE'), face_id
FROM city_data_face$;

-- Get topological elements.
SELECT a.FEATURE_NAME,
 sdo_topo.get_topo_objects('CITY_DATA', a.feature.TG_LAYER_ID, a.feature.TG_ID)
FROM land_parcels a;
```

```
SELECT a.FEATURE_NAME,
 sdo_topo.get_topo_objects('CITY_DATA', a.feature.TG_LAYER_ID, a.feature.TG_ID)
FROM city_streets a;

SELECT a.FEATURE_NAME,
 sdo_topo.get_topo_objects('CITY_DATA', a.feature.TG_LAYER_ID, a.feature.TG_ID)
FROM traffic_signs a;

SELECT sdo_topo.get_topo_objects('CITY_DATA', sdo_geometry(2003,null, null,
       sdo_elem_info_array(1,1003,3),
        sdo_ordinate_array(1,1, 20,20)))
        FROM DUAL;

SELECT sdo_topo.get_topo_objects('CITY_DATA', sdo_geometry(2003,null, null,
       sdo_elem_info_array(1,1003,3),
        sdo_ordinate_array(17,30, 31,40)))
        FROM DUAL;

-- Find all city streets interacting with a query window.
SELECT c.feature_name FROM city_streets c WHERE
  SDO_ANYINTERACT(
    c.feature,
    SDO_GEOMETRY(2003, NULL, NULL,
      SDO_ELEM_INFO_ARRAY(1, 1003, 3),
      SDO_ORDINATE_ARRAY(5,5, 30,40)))
= 'TRUE';

-- Find all streets that have any interaction with land parcel P3.
-- (Should return only R1.)
SELECT c.feature_name FROM city_streets c, land_parcels l
  WHERE l.feature_name = 'P3' AND
   SDO_ANYINTERACT (c.feature, l.feature) = 'TRUE';

-- Find all land parcels that have any interaction with traffic sign S1.
-- (Should return P1 and P2.)
SELECT l.feature_name FROM land_parcels l, traffic_signs t
  WHERE t.feature_name = 'S1' AND
   SDO_ANYINTERACT (l.feature, t.feature) = 'TRUE';

-- Get the geometry for land parcel P1.
SELECT l.feature_name, l.feature.get_geometry()
  FROM land_parcels l WHERE l.feature_name = 'P1';

-- Query SDO_TOPO_GEOMETRY attributes,
SELECT s.feature.tg_type FROM city_streets s;
SELECT s.feature.tg_id FROM city_streets s;
SELECT s.feature.tg_layer_id FROM city_streets s;
SELECT s.feature.topology_id FROM city_streets s;

-- Topology-specific functions

-- Get the boundary of face with face_id 3.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 3) FROM DUAL;
-- Try 'TRUE' as third parameter.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 3, 'TRUE') FROM DUAL;
-- Get the boundary of face with face_id 2.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 2) FROM DUAL;
-- Try 'TRUE' as third parameter.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 2, 'TRUE') FROM DUAL;
-- Get the boundary of face with face_id 1.
```

```
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 1) FROM DUAL;
-- Specify 'TRUE' for the all_edges parameter.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 1, 'TRUE') FROM DUAL;

-- CITY_DATA layer, land parcels (tg_ layer_id = 1), parcel P2 (tg_id = 2)
SELECT SDO_TOPO.GET_TOPO_OBJECTS('CITY_DATA', 1, 2) FROM DUAL;

-- 10. Optionally, edit the data using the PL/SQL or Java API.
```

1.13 README File for Spatial and Related Features

A `README.txt` file supplements the information in the following manuals: *Oracle Spatial Developer's Guide*, *Oracle Spatial GeoRaster Developer's Guide*, and *Oracle Spatial Topology and Network Data Models Developer's Guide* (this manual). This file is located at:

$ORACLE_HOME`/md/doc/README.txt`

2
Editing Topologies

This chapter explains how to edit node and edge data in a topology. The operations include adding, moving, and removing nodes and edges, and updating the coordinates of an edge.

This chapter explains two approaches to editing topology data, and it explains why one approach (creating and using a special cache) is better in most cases. It also describes the behavior and implications of some major types of editing operations. It contains the following major sections:

- Section 2.1, "Approaches for Editing Topology Data"
- Section 2.2, "Performing Operations on Nodes"
- Section 2.3, "Performing Operations on Edges"

The explanations in this chapter refer mainly to the PL/SQL application programming interface (API) provided in the MDSYS.SDO_TOPO_MAP package, which is documented in Chapter 4. However, you can also perform topology editing operations using the client-side Java API, which is introduced in Section 1.8.2 and is explained in the Javadoc-generated documentation.

To edit topology data, always use the PL/SQL or Java API. Do not try to perform editing operations by directly modifying the node, edge, or face information tables.

For cross-schema topology editing considerations, see Section 1.10.2.

2.1 Approaches for Editing Topology Data

Whenever you need to edit a topology, you can use PL/SQL or Java API. In both cases, Oracle Spatial uses an in-memory topology cache, specifically, a TopoMap object (described in Section 2.1.1):

- If you use the PL/SQL API, you can either explicitly create and use the cache or allow Spatial to create and use the cache automatically.
- If you use the Java API, you must explicitly create and use the cache.

Allowing Spatial to create and manage the cache automatically is simpler, because it involves fewer steps than creating and using a cache. However, because allowing Spatial to create and manage the cache involves more database activity and disk accesses, it is less efficient when you need to edit more than a few topological elements.

2.1.1 TopoMap Objects

A **TopoMap object** is associated with an in-memory cache that is associated with a topology. If you explicitly create and use a cache for editing a topology, you must create a TopoMap object to be associated with a topology, load all or some of the topology into the cache, edit objects, periodically update the topology to write changes to the database, commit the changes made in the cache, and clear the cache.

Although this approach involves more steps than allowing Spatial to create and use the cache automatically, it is much faster and more efficient for most topology editing sessions, which typically affect hundreds or thousands of topological elements. It is the approach shown in most explanations and illustrations.

A TopoMap object can be updatable or read-only, depending on the value of the `allow_updates` parameter when you call the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure:

- With a read-only TopoMap object, topological elements (primitives) are loaded but not locked.
- With an updatable TopoMap object, topological elements (primitives) are loaded and locked. If you specified a rectangular window for an updatable TopoMap object, you can edit only those topological elements inside the specified window. (The TopoMap object may also contain locked topological elements that you cannot edit directly, but that Oracle Spatial can modify indirectly as needed.)

  For more information about what occurs when you use an updatable TopoMap object, see the Usage Notes for the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure.

The following procedures set an updatable TopoMap object to be read-only:

- SDO_TOPO_MAP.COMMIT_TOPO_MAP
- SDO_TOPO_MAP.ROLLBACK_TOPO_MAP
- SDO_TOPO_MAP.CLEAR_TOPO_MAP

Within a user session at any given time, there can be no more than one updatable TopoMap object. However, multiple different user sessions can work with updatable TopoMap objects based on the same topology, as long as their editing windows do not contain any topological elements that are in any other updatable TopoMap objects. There can be multiple read-only TopoMap objects within and across user sessions.

Two or more users can edit a topology at the same time as long as their editing windows (specified in the call to the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure) do not overlap.

2.1.2 Specifying the Editing Approach with the Topology Parameter

For many SDO_TOPO_MAP package functions and procedures that edit topologies, such as SDO_TOPO_MAP.ADD_NODE or SDO_TOPO_MAP.MOVE_EDGE, you indicate the approach you are using for editing by specifying either a topology name or a null value for the first parameter, which is named `topology`:

- If you specify a topology name, Spatial checks to see if an updatable TopoMap object already exists in the user session; and if one does not exist, Spatial creates an internal TopoMap object, uses that cache to perform the editing operation, commits the change (or rolls back changes to the savepoint at the beginning of the process if an exception occurred), and deletes the TopoMap object. (If an updatable TopoMap object already exists, an exception is raised.) For example, the

following statement removes the node with node ID value 99 from the MY_TOPO topology:

```
CALL SDO_TOPO_MAP.REMOVE_NODE('MY_TOPO', 99);
```

- If you specify a null value, Spatial checks to see if an updatable TopoMap object already exists in the user session; and if one does exist, Spatial performs the operation in the TopoMap object's cache. (If no updatable TopoMap object exists, an exception is raised.) For example, the following statement removes the node with node ID value 99 from the current updatable TopoMap object:

```
CALL SDO_TOPO_MAP.REMOVE_NODE(null, 99);
```

2.1.3 Using GET_xxx Topology Functions

Some SDO_TOPO_MAP package functions that get information about topologies have topology and topo_map as their first two parameters. Examples of such functions are SDO_TOPO_MAP.GET_EDGE_COORDS and SDO_TOPO_MAP.GET_NODE_STAR. To use these functions, specify a valid value for one parameter and a null value for the other parameter, as follows:

- If you specify a valid topology parameter value, Spatial retrieves the information for the specified topology. It creates an internal TopoMap object, uses that cache to perform the operation, and deletes the TopoMap object. For example, the following statement returns the edge coordinates of the edge with an ID value of 1 from the CITY_DATA topology:

```
SELECT SDO_TOPO_MAP.GET_EDGE_COORDS('CITY_DATA', null, 1) FROM DUAL;
```

- If you specify a null topology parameter value and a valid topo_map parameter value, Spatial uses the specified TopoMap object (which can be updatable or read-only) to retrieve the information for the specified topology. For example, the following statement returns the edge coordinates of the edge with an ID value of 1 from the CITY_DATA_TOPOMAP TopoMap object:

```
SELECT SDO_TOPO_MAP.GET_EDGE_COORDS(null, 'CITY_DATA_TOPOMAP', 1) FROM DUAL;
```

- If you specify a null or invalid value for both the topology and topo_map parameters, an exception is raised.

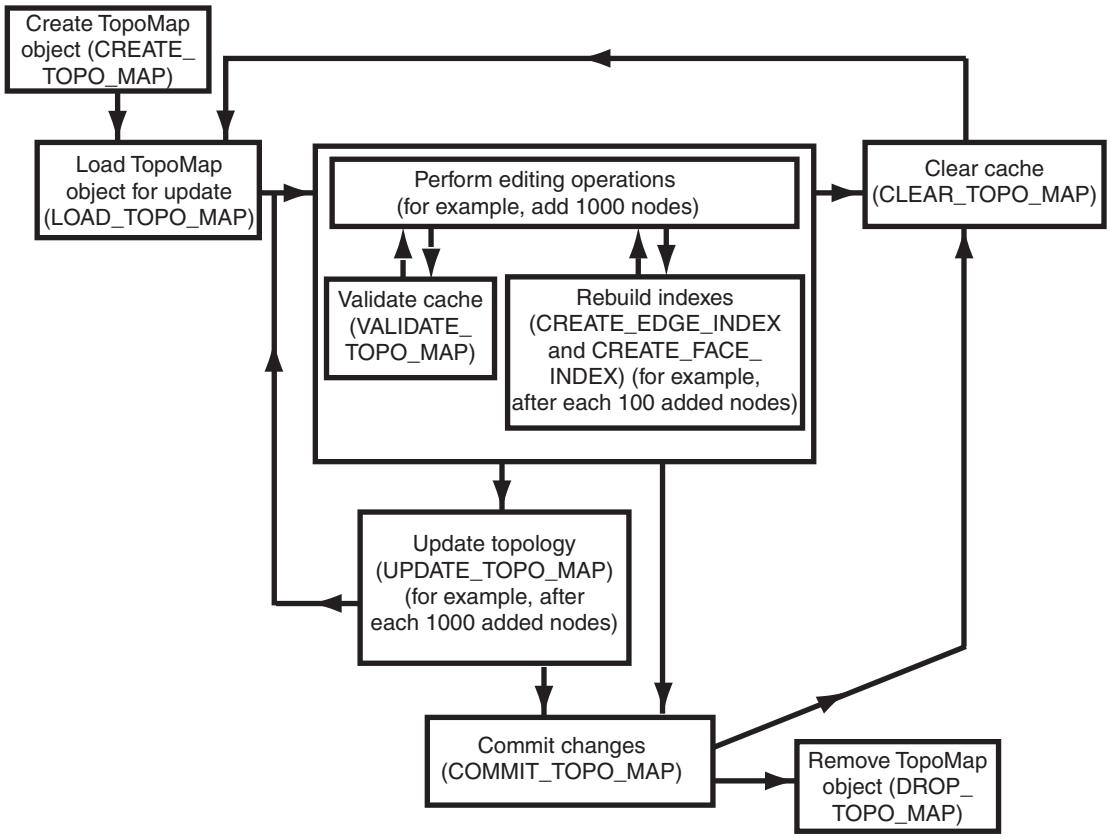
Some SDO_TOPO_MAP package functions that get information about topology editing operations have no parameters. Examples of such functions are SDO_TOPO_MAP.GET_FACE_ADDITIONS and SDO_TOPO_MAP.GET_NODE_CHANGES. These functions use the current updatable TopoMap object. If no updatable TopoMap object exists, an exception is raised. For example, the following statement returns an SDO_NUMBER_ARRAY object (described in Section 1.6.7) with the node ID values of nodes that have been added to the current updatable TopoMap object:

```
SELECT SDO_TOPO_MAP.GET_NODE_ADDITIONS FROM DUAL;
```

2.1.4 Process for Editing Using Cache Explicitly (PL/SQL API)

Figure 2–1 shows the recommended process for editing topological elements using the PL/SQL API and explicitly using a TopoMap object and its associated cache.

Figure 2–1 Editing Topologies Using the TopoMap Object Cache (PL/SQL API)



As Figure 2–1 shows, the basic sequence is as follows:

1. Create the TopoMap object, using the SDO_TOPO_MAP.CREATE_TOPO_MAP procedure.

   This creates an in-memory cache for editing objects associated with the specified topology.

2. Load the entire topology or a rectangular window from the topology into the TopoMap object cache for update, using the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure.

   You can specify that in-memory R-tree indexes be built on the edges and faces that are being loaded. These indexes use some memory resources and take some time to create and periodically rebuild; however, they significantly improve performance if you edit a large number of topological elements in the session. (They can also improve performance for queries that use a read-only TopoMap object.)

3. Perform a number of topology editing operations (for example, add 1000 nodes).

   Periodically, validate the cache by calling the SDO_TOPO_MAP.VALIDATE_TOPO_MAP function.

   You can rebuild existing in-memory R-tree indexes on edges and faces in the TopoMap object, or create new indexes if none exist, by using the SDO_TOPO_MAP.CREATE_EDGE_INDEX and SDO_TOPO_MAP.CREATE_FACE_INDEX procedures. For best index performance, these indexes should be rebuilt periodically when you are editing a large number of topological elements.

If you want to discard edits made in the cache, call the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure. This procedure fails if there are any uncommitted updates; otherwise, it clears the data in the cache and sets the cache to be read-only.

4. Update the topology by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.
5. Repeat Steps 3 and 4 (editing objects, validating the cache, rebuilding the R-tree indexes, and updating the topology) as often as needed, until you have finished the topology editing operations.
6. Commit the topology changes by calling the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure. (The SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure automatically performs the actions of the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure before it commits the changes.) After the commit operation, the cache is made read-only (that is, it is no longer updatable). However, if you want to perform further editing operations using the same TopoMap object, you can load it again and use it (that is, repeat Steps 2 through 5, clearing the cache first if necessary).

   To perform further editing operations, clear the TopoMap object cache by calling the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure, and then go to Step 2.

   If you want to discard all uncommitted topology changes, you can call the SDO_TOPO_MAP.ROLLBACK_TOPO_MAP procedure at any time. After the rollback operation, the cache is cleared.
7. Remove the TopoMap object by calling the SDO_TOPO_MAP.DROP_TOPO_MAP procedure.

   This procedure removes the TopoMap object and frees any resources that it had used. (If you forget to drop the TopoMap object, it will automatically be dropped when the user session ends.) This procedure also rolls back any topology changes in the cache that have not been committed.

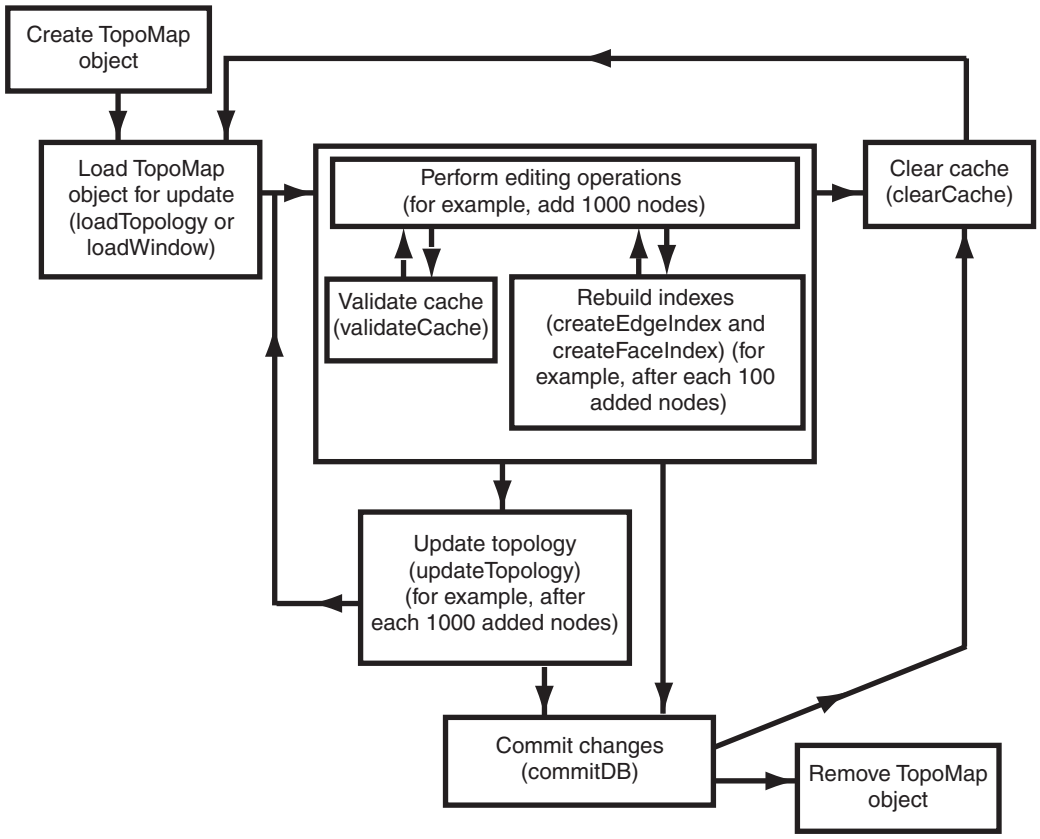
   If the application terminates abnormally, all uncommitted changes to the database will be discarded.

If you plan to perform a very large number of topology editing operations, you can divide the operations among several editing sessions, each of which performs Steps 1 through 7 in the preceding list.

2.1.5 Process for Editing Using the Java API

Figure 2–2 shows the recommended process for editing topological elements using the client-side Java API, which is introduced in Section 1.8.2 and is explained in the Javadoc-generated documentation. The Java API requires that you create and manage a TopoMap object and its associated cache.

Figure 2–2 Editing Topologies Using the TopoMap Object Cache (Java API)



As Figure 2–2 shows, the basic sequence is as follows:

1. Create the TopoMap object, using a constructor of the `TopoMap` class, specifying a topology and a database connection.

   This creates an in-memory cache for editing objects associated with the specified topology.

2. Load the entire topology or a rectangular window from the topology into the TopoMap object cache for update, using the `loadTopology` or `loadWindow` method of the `TopoMap` class.

   You can specify that in-memory R-tree indexes be built on the edge and edge face that are being affected. These indexes use some memory resources and take some time to create and periodically rebuild; however, they significantly improve performance if you edit a large number of topological elements during the database connection.

3. Perform a number of topology editing operations (for example, add 1000 nodes), and update the topology by calling the `updateTopology` method of the `TopoMap` class.

   Periodically, validate the cache by calling the `validateCache` method of the `TopoMap` class.

   If you caused in-memory R-tree indexes to be created when you loaded the TopoMap object (in Step 2), you can periodically (for example, after each addition of 100 nodes) rebuild the indexes by calling the `createEdgeIndex` and `createFaceIndex` methods of the `TopoMap` class. For best index performance,

these indexes should be rebuilt periodically when you are editing a large number of topological elements.

If you do not want to update the topology but instead want to discard edits made in the cache since the last update, call the `clearCache` method of the `TopoMap` class. The `clearCache` method fails if there are any uncommitted updates; otherwise, it clears the data in the cache and sets the cache to be read-only.

4. Update the topology by calling the `updateTopology` method of the `TopoMap` class.
5. Repeat Steps 3 and 4 (editing objects, validating the cache, rebuilding the R-tree indexes, and updating the topology) as often as needed, until you have finished the topology editing operations.
6. Commit the topology changes by calling the `commitDB` method of the `TopoMap` class. (The `commitDB` method automatically calls the `updateTopology` method before it commits the changes.) After the commit operation, the cache is made read-only (that is, it is no longer updatable). However, if you want to perform further editing operations using the same TopoMap object, you can load it again and use it (that is, repeat Steps 2 through 5, clearing the cache first if necessary).

   To perform further editing operations, clear the TopoMap object cache by calling the `clearCache` method of the `TopoMap` class, and then go to Step 2.

   If you want to discard all uncommitted topology changes, you can call the `rollbackDB` method of the `TopoMap` class at any time. After the rollback operation, the cache is cleared.
7. Remove the TopoMap object by setting the TopoMap object to null, which makes the object available for garbage collection and frees any resources that it had used. (If you forget to remove the TopoMap object, it will automatically be garbage collected when the application ends.)

   If the application terminates abnormally, all uncommitted changes to the database will be discarded.

If you plan to perform a very large number of topology editing operations, you can divide the operations among several editing sessions, each of which performs Steps 1 through 7 in the preceding list.

2.1.6 Error Handling for Topology Editing

This section discusses the following conditions:

- Input parameter errors
- All exceptions

2.1.6.1 Input Parameter Errors

When an SDO_TOPO_MAP PL/SQL subprogram or a public method in the `TopoMap` Java class is called, it validates the values of the input parameters, and it uses or creates a TopoMap object to perform the editing or read-only operation. Whenever there is an input error, an `oracle.spatial.topo.TopoDataException` exception is thrown. Other errors may occur when the underlying TopoMap object performs an operation.

If the method is called from SQL or PL/SQL, the caller gets the following error message:

```
ORA-29532: Java call terminated by uncaught Java exception:
```

```
<specific error message text>
```

The following PL/SQL example shows how you can handle a `TopoDataException` exception:

```
DECLARE
  topo_data_error EXCEPTION;
  PRAGMA EXCEPTION_INIT(topo_data_error, -29532);
BEGIN
  sdo_topo_map.create_topo_map(null, null, 100, 100, 100);
EXCEPTION
  WHEN topo_data_error THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

The preceding example generates the following output:

```
ORA-29532: Java call terminated by uncaught Java exception:
oracle.spatial.topo.TopoDataException: invalid TopoMap name
```

2.1.6.2 All Exceptions

The following actions are performed automatically when any exception occurs in a call to any of the following SDO_TOPO_MAP PL/SQL subprograms or their associated methods in the `TopoMap` Java class: SDO_TOPO_MAP.ADD_EDGE (`addEdge`), SDO_TOPO_MAP.ADD_ISOLATED_NODE (`addIsolatedNode`), SDO_TOPO_MAP.ADD_LOOP (`addLoop`), SDO_TOPO_MAP.ADD_NODE (`addNode`), SDO_TOPO_MAP.ADD_POINT_GEOMETRY (`addPointGeometry`), SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY (`addPolygonGeometry`), SDO_TOPO_MAP.CHANGE_EDGE_COORDS (`changeEdgeCoords`), SDO_TOPO_MAP.MOVE_ISOLATED_NODE (`moveIsolatedNode`), SDO_TOPO_MAP.MOVE_NODE (`moveNode`), SDO_TOPO_MAP.MOVE_EDGE (`moveEdge`), SDO_TOPO_MAP.REMOVE_EDGE (`removeEdge`), SDO_TOPO_MAP.REMOVE_NODE (`removeNode`), and SDO_TOPO_MAP.UPDATE_TOPO_MAP (`updateTopology`).

- The transaction is rolled back.
- The TopoMap object cache is cleared.
- The TopoMap object is made read-only.

2.2 Performing Operations on Nodes

This section contains sections that describe the effects of adding, moving, and removing nodes, and that explain how to perform these operations using the PL/SQL API.

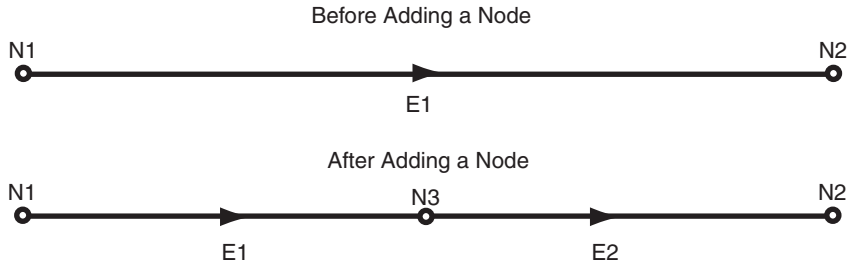
2.2.1 Adding a Node

Adding a non-isolated node adds the node to an edge at a point that is currently on the edge. This operation also splits the edge, causing the original edge to be divided into two edges. Spatial automatically adjusts the definition of the original edge and creates a new edge (assigning it an ID value that is unique among edges in the topology).

To add a non-isolated node, use the SDO_TOPO_MAP.ADD_NODE function. To add an isolated node, use the SDO_TOPO_MAP.ADD_ISOLATED_NODE function.

Figure 2–3 shows the addition of a node (N3) on edge E1.

Figure 2–3 Adding a Non-Isolated Node

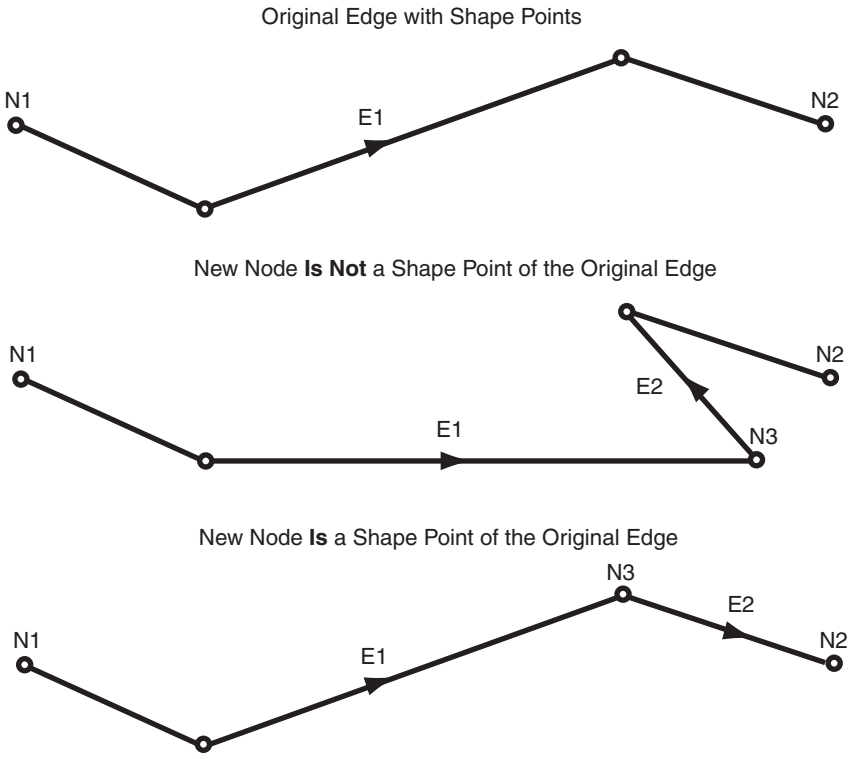


As a result of the operation shown in Figure 2–3:

- Edge E1 is redefined to be between the original edge's start point and the point at the added node (N3).
- Edge E2 is created. Its start point is the point at node N3, and its end point is the end point of the original edge.
- If any linear features were defined on the original edge, they are automatically redefined to be on both resulting edges, the edge is split, and a record is added to the history information table (explained in Section 1.5.5) for the topology. For example, if a street named Main Street had been defined on the original edge E1 in Figure 2–3, then after the addition of node N3, Main Street would be defined on both edges E1 and E2.

Figure 2–4 shows a more complicated example of adding a node, where the result depends on whether or not the added node is a new shape point of the original edge (that is, on the value of the `is_new_shape_point` parameter to the SDO_TOPO_MAP.ADD_NODE function).

Figure 2–4 Effect of is_new_shape_point Value on Adding a Node



In Figure 2–4:

- In the top part of the figure, the original edge (E1) starts at node N1, ends at node N2, and has two intermediate shape points.
- In the middle part of the figure, a new node (N3) is added that is not a shape point of the original edge, but instead is a new shape point (that is, `is_new_shape_point=>'TRUE'`). The new node is added at the location specified with the `point` parameter, and is added after the vertex specified in the `coord_index` parameter (in this case, `coord_index=>1` to indicate after the first vertex). The new node becomes the end node for edge E1 and the start node for the new edge E2, which ends at node N2.
- In the bottom part of the figure, a new node (N3) is added that is a shape point of the original edge, and is thus not a new shape point (that is, `is_new_shape_point=>'FALSE'`). Because it is not a new shape point, the node is added at the vertex specified with the `coord_index` parameter (in this case, `coord_index=>2`). As in the middle part of the figure, the new node becomes the end node for edge E1 and the start node for the new edge E2, which ends at node N2.

2.2.2 Moving a Node

Moving a non-isolated node to a new position causes the ends of all edges that are attached to the node to move with the node. You must specify the vertices for all edges affected by the moving of the node; each point (start or end) that is attached to the node must have the same coordinates as the new location of the node, and the other end points (not the moved node) of each affected edge must remain the same.

To move a non-isolated node, use the SDO_TOPO_MAP.MOVE_NODE procedure. To move an isolated node, use the SDO_TOPO_MAP.MOVE_ISOLATED_NODE procedure.

Figure 2–5 shows the original topology before node N1 is moved.

Figure 2–5 Topology Before Moving a Non-Isolated Node

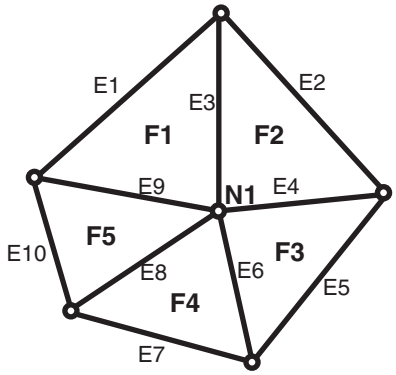
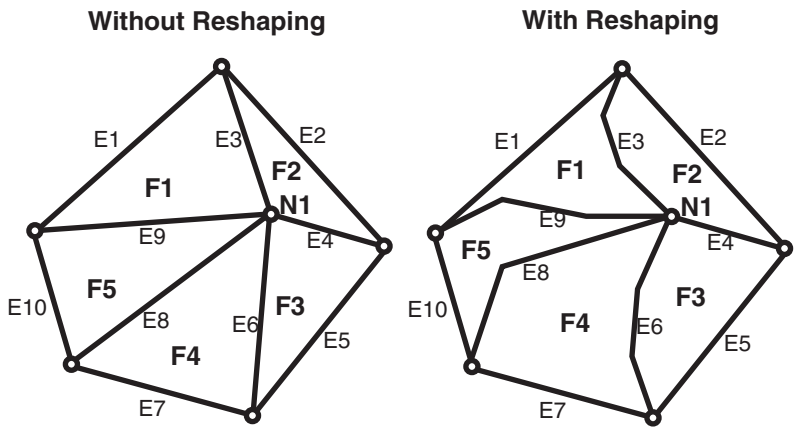


Figure 2–6 shows two cases of the original topology after node N1 is moved. In one case, no reshaping occurs; that is, all edges affected by the movement are specified as straight lines. In the other case, reshaping occurs; that is, one or more affected edges are specified as line segments with multiple vertices.

Figure 2–6 Topology After Moving a Non-Isolated Node



In both cases shown in Figure 2–6:

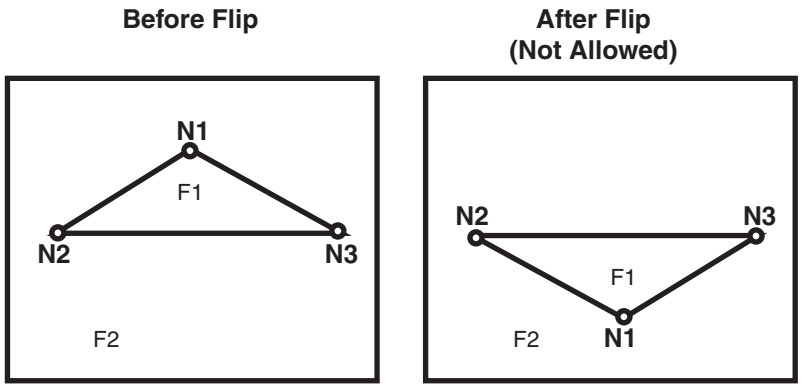
- The topology does not change. That is, the number of nodes, edges, and faces does not change, and the relationships (such as adjacency and connectivity) among the nodes, edges, and faces do not change.
- All features defined on the nodes, edges, and faces retain their definitions.

Any isolated nodes and edges might remain in the same face or be moved to a different face as a result of a move operation on a non-isolated node. The SDO_TOPO_MAP.MOVE_NODE procedure has two output parameters, `moved_iso_nodes` and `moved_iso_edges`, that store the ID numbers of any isolated nodes and edges that were moved to a different face as a result of the operation.

A node cannot be moved if, as a result of the move, any of the following would happen:

- Any edges attached to the node would intersect any other edge. For example, assume that the original topology shown in Figure 2–6 had included another edge E20 that passed just above and to the right of node N1. If the movement of node N1 would cause edge E3, E4, E6, E8, or E9 to intersect edge E20, the move operation is not performed.
- The node would be moved to a face that does not currently bound the node. For example, if the movement of node N1 would place it outside the original topology shown in Figure 2–6, the move operation is not performed.
- The node would be moved to the opposite side of an isolated face. This is not allowed because the move would change the topology by changing one or more of the following: the relationship or ordering of edges around the face, and the left and right face for each edge. Figure 2–7 shows a node movement (flipping node N1 from one side of isolated face F1 to the other side) that would not be allowed.

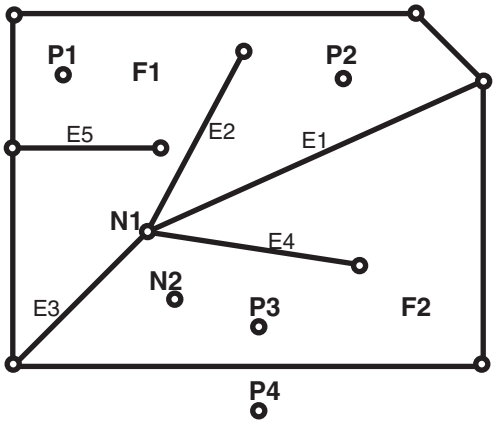
Figure 2–7 Node Move Is Not Allowed



2.2.2.1 Additional Examples of Allowed and Disallowed Node Moves

This section provides additional examples of node movement operations that are either allowed or not allowed. All refer to the topology shown in Figure 2–8.

Figure 2–8 Topology for Node Movement Examples



In the topology shown in Figure 2–8:

- Attempts will be made to move node N1 to points P1, P2, P3, and P4. (These points are locations but are not existing nodes.)
- The edges have no shape points, either before or after the move operation.
- New vertices are specified for the edges E1, E2, E3, and E4, but the ID values of the start and end points for the edges remain the same.

When the following node move operations are attempted using the topology shown in Figure 2–8, the following results occur:

- Moving node N1 to point P1: Not allowed, because one or more of the four attached edges would intersect edge E5. (Edge E3 would definitely intersect edge E5 if the move were allowed.)
- Moving node N1 to point P2: Allowed.
- Moving node N1 to point P3: Allowed. However, this operation causes the isolated node N2 to change from face F2 to face F1, and this might cause the application to want to roll back or disallow the movement of node N1. Similarly, if the movement of a node would cause any isolated edges or faces to change from one face to another, the application might want to roll back or disallow the node move operation.
- Moving node N1 to point P4: Not allowed, because the node must be moved to a point in a face that bounds the original (current) position of the node.

2.2.3 Removing a Node

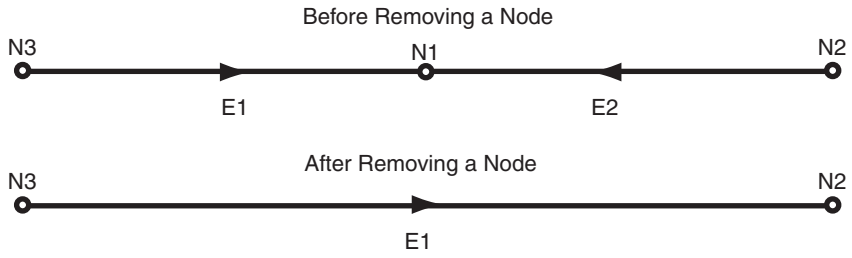
You can remove individual nodes (isolated or non-isolated), as explained in this section, and you can remove all obsolete nodes in a topology, as explained in Section 2.2.4.

Removing a non-isolated node deletes the node and merges the edges that were attached to the node into a single edge. (Spatial applies complex rules, which are not documented, to determine the ID value and direction of the resulting edge.)

To remove a non-isolated or isolated node, use the SDO_TOPO_MAP.REMOVE_NODE procedure.

Figure 2–9 shows the removal of a node (N1) that is attached to edges E1 and E2.

Figure 2–9 Removing a Non-Isolated Node



As a result of the operation shown in Figure 2–9:

- Edge E1 is redefined to consist of the line segments that had represented the original edges E1 and E2.
- Edge E2 is deleted.

- If any linear features were defined on both original edges, they are automatically redefined to be on the resulting edge, and a record is added to the history information table (explained in Section 1.5.5) for the topology. For example, if a street named Main Street had been defined on the original edges E1 and E2 in Figure 2–9, then after the removal of node N1, Main Street would be defined on edge E1.

A node cannot be removed if one or more of the following are true:

- A point feature is defined on the node. For example, if a point feature named Metropolitan Art Museum had been defined on node N1 in Figure 2–9, node N1 cannot be removed. Before you can remove the node in this case, you must remove the definition of any point features on the node.
- A linear feature defined on either original edge is not also defined on both edges. For example, if a linear feature named Main Street had been defined on edge E1 but not edge E2 in Figure 2–9, node N1 cannot be removed.

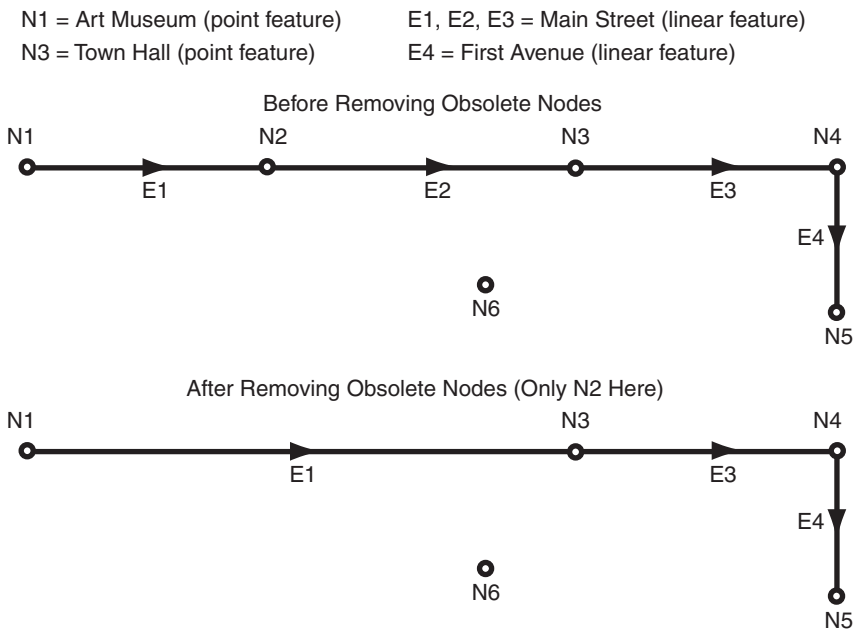
2.2.4 Removing Obsolete Nodes

An **obsolete node** is a node that is connected to only two distinct edges, is not assigned to any point feature, and does not serve as the demarcation between different linear features. Obsolete nodes can result when the SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY function is used repeatedly to build a topology, or when edges have been removed during editing operations, leaving some unnecessary nodes. Therefore, it is recommended that you use the SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES procedure to remove obsolete nodes in such cases.

Spatial automatically updates the appropriate entries in the <topology-name>_NODE$ and <topology-name>_EDGE$ tables, and in the <topology-name>_FACE$ table if necessary.

Figure 2–10 shows the removal of obsolete nodes in a simple topology. In this topology, node N1 has a point feature named Art Museum defined on it, and node N3 has a point feature named Town Hall defined on it. Edges E1, E2, and E3 have a linear feature named Main Street defined on them, and edge E4 has a linear feature named First Avenue defined on it.

Figure 2–10 Removing Obsolete Nodes



In Figure 2–10, the only node removed is N2, because only that node satisfies all the criteria for an obsolete node. As for the other nodes:

- N1 is connected to only one edge (E1), and it has a point feature defined on it (Art Museum).
- N3 has a point feature defined on it (Town Hall).
- N4 is the demarcation between two different linear features (Main Street and First Avenue).
- N5 is connected to only one edge (E4).
- Node N6 is an isolated node (not connected to any edges).

Also as a result of the operation shown in Figure 2–10, edge E2 was removed as a result of the removal of node N2.

2.3 Performing Operations on Edges

This section describes the effects of adding, moving, removing, and updating edges, and explains how to perform these operations using the PL/SQL API.

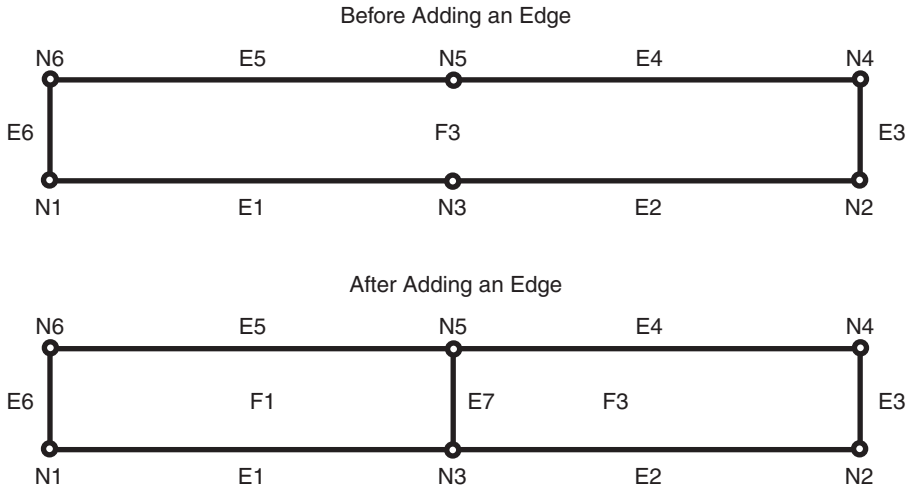
2.3.1 Adding an Edge

Adding a non-isolated edge adds the edge to a face. It also splits the face, causing the original face to be divided into two faces. Spatial automatically adjusts the definition of the original face and creates a new face (assigning it an ID value that is unique among faces in the topology).

To add an edge, use the SDO_TOPO_MAP.ADD_EDGE procedure. You must specify existing nodes as the start and end nodes of the added edge.

Figure 2–11 shows the addition of an edge (E7) between nodes N3 and N5 on face F3.

Figure 2–11 Adding a Non-Isolated Edge



As a result of the operation shown in Figure 2–11, face F3 is redefined to be two faces, F1 and F3. (Spatial applies complex rules, which are not documented, to determine the ID values of the resulting faces.)

Any polygon features that were defined on the original face are automatically redefined to be on both resulting faces. For example, if a park named Walden State Park had been defined on the original face F3 in Figure 2–11, then after the addition of edge E7, Walden State Park would be defined on both faces F1 and F3.

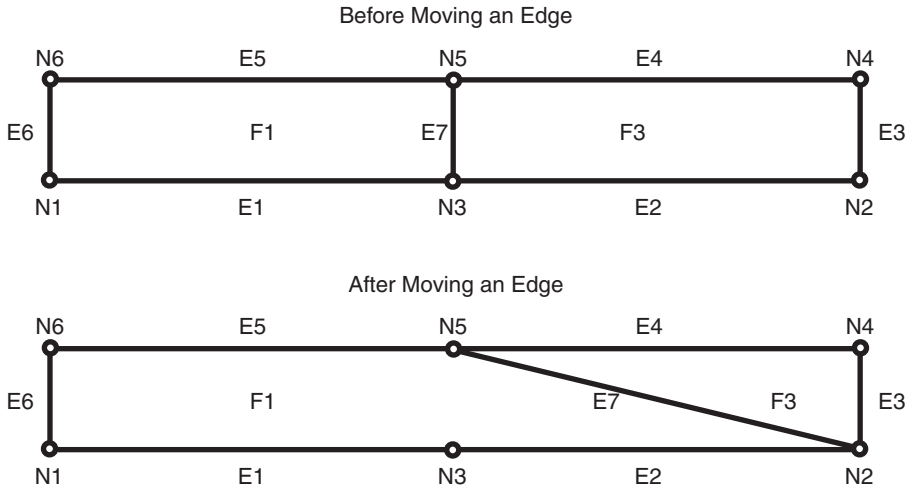
2.3.2 Moving an Edge

Moving a non-isolated edge keeps the start or end point of the edge in the same position and moves the other of those two points to another existing node position. You must specify the source node (location before the move of the node to be moved), the target node (location after the move of the node being moved), and the vertices for the moved edge.

To move an edge, use the SDO_TOPO_MAP.MOVE_EDGE procedure.

Figure 2–12 shows the movement of edge E7, which was originally between nodes N3 and N5, to be between nodes N2 and N5.

Figure 2–12 Moving a Non-Isolated Edge



As a result of the operation shown in Figure 2–12, faces F1 and F3 are automatically redefined to reflect the coordinates of their edges, including the new coordinates for edge E7.

Any isolated nodes and edges might remain in the same face or be moved to a different face as a result of a move operation on a non-isolated edge. The SDO_TOPO_MAP.MOVE_EDGE procedure has two output parameters, `moved_iso_nodes` and `moved_iso_edges`, that store the ID numbers of any isolated nodes and edges that were moved to a different face as a result of the operation.

An edge cannot be moved if, as a result of the move, any of the following would happen:

- The moved edge would intersect any other edge. For example, assume that the topology before the move, as shown in Figure 2–12, had included another edge (E10) that was between nodes N3 and N4. In this case, the movement of edge E7 would cause it to intersect edge E10, and therefore the move operation is not performed.
- The node would be moved to a face that does not currently bound the edge. For example, if the movement of edge E7 would place its terminating point at a node outside the faces shown in Figure 2–12 (F1 and F3), the move operation is not performed.

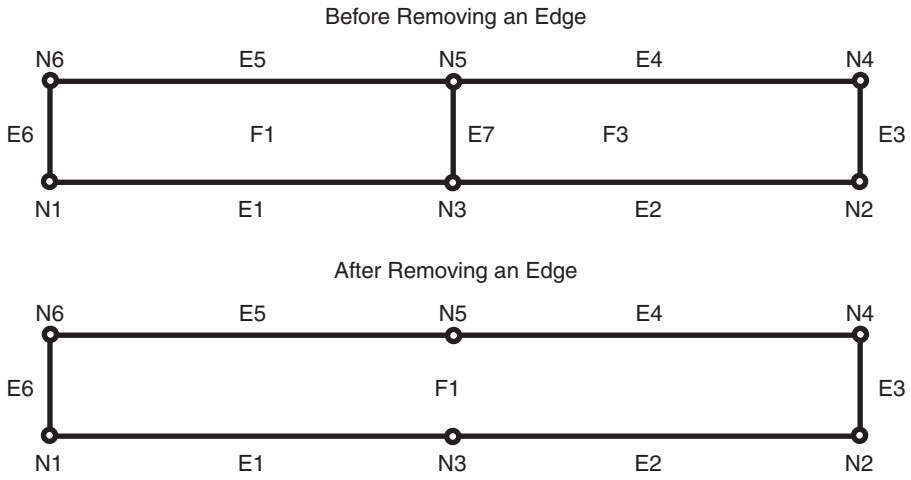
2.3.3 Removing an Edge

Removing a non-isolated edge deletes the edge and merges the faces that bounded the edge. (Spatial applies complex rules, which are not documented, to determine the ID value of the resulting face.)

To remove an edge, use the SDO_TOPO_MAP.REMOVE_EDGE procedure.

Figure 2–13 shows the removal of an edge (E7) that is bounded by faces F1 and F3.

Figure 2–13 Removing a Non-Isolated Edge



As a result of the operation shown in Figure 2–13:

- Face F1 is redefined to consist of the area of the original faces F1 and F3.
- Face F3 is deleted.
- The start and end nodes of the deleted edge (nodes N3 and N5) are not removed.

Any polygon features that were defined on both original faces are automatically redefined to be on the resulting face. For example, if a park named Adams Park had been defined on the original faces F1 and F3 in Figure 2–13, then after the removal of edge E7, Adams Park would be defined on face F1.

A non-isolated edge cannot be removed if one or more of the following are true:

- A linear feature is defined on the edge. For example, if a linear feature named Main Street had been defined on edge E7 in Figure 2–13, edge E7 cannot be removed. Before you can remove the edge in this case, you must remove the definition of any linear features on the edge.
- A polygon feature defined on either original face is not also defined on both faces. For example, if a linear feature named Adams Park had been defined on face F1 but not face F3 in Figure 2–13, edge E7 cannot be removed.

2.3.4 Updating an Edge

Updating an isolated edge means changing one or more coordinates of the edge, but without changing the start point and end point.

To update an edge, use the SDO_TOPO_MAP.CHANGE_EDGE_COORDS procedure.

Any isolated nodes and edges might remain in the same face or be moved to a different face as a result of an update operation on a non-isolated edge. The SDO_TOPO_MAP.CHANGE_EDGE_COORDS procedure has two output parameters, `moved_iso_nodes` and `moved_iso_edges`, that store the ID numbers of any isolated nodes and edges that were moved to a different face as a result of the operation.

An edge cannot be updated if, as a result of the operation, it would intersect any other edge. See the Usage Notes for the SDO_TOPO_MAP.CHANGE_EDGE_COORDS procedure for more information about updating an edge.

3

SDO_TOPO Package Subprograms

The MDSYS.SDO_TOPO package contains subprograms (functions and procedures) that constitute part of the PL/SQL application programming interface (API) for the Spatial topology data model. This package mainly contains subprograms for creating and managing topologies.

To use the subprograms in this chapter, you must understand the conceptual information about topology in Chapter 1.

Another package, SDO_TOPO_MAP, mainly contains subprograms related to editing topologies. Reference information for the SDO_TOPO_MAP package is in Chapter 4.

The rest of this chapter provides reference information about the SDO_TOPO subprograms, listed in alphabetical order.

SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER

Format

```
SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER(
   topology                 IN VARCHAR2,
   table_name               IN VARCHAR2,
   column_name              IN VARCHAR2,
   topo_geometry_layer_type IN VARCHAR2,
   relation_table_storage   IN VARCHAR2 DEFAULT NULL,
   child_layer_id           IN NUMBER DEFAULT NULL);
```

Description

Adds a topology geometry layer to a topology.

Parameters

topology
Topology to which to add the topology geometry layer containing the topology geometries in the specified column. The topology must have been created using the SDO_TOPO.CREATE_TOPOLOGY procedure.

table_name
Name of the topology geometry layer table containing the column specified in `column_name`.

column_name
Name of the column (of type SDO_TOPO_GEOMETRY) containing the topology geometries in the topology geometry layer to be added to the topology.

topo_geometry_layer_type
Type of topology geometry layer: POINT, LINE, CURVE, POLYGON, or COLLECTION.

relation_table_storage
Physical storage parameters used internally to create the <topology-name>_RELATION$ table (described in Section 1.5.4). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K)`. If you do not specify this parameter, the default physical storage values are used.

child_layer_id
Layer ID number of the child topology geometry layer for this layer, if the topology has a topology geometry layer hierarchy. (Topology geometry layer hierarchy is explained in Section 1.4.) If you do not specify this parameter and if the topology has a topology geometry layer hierarchy, the topology geometry layer is added to the lowest level (level 0) of the hierarchy.

If the topology does not have a topology geometry layer hierarchy, do not specify this parameter when adding any of the topology geometry layers.

Usage Notes

The first call to this procedure for a given topology creates the <topology-name>_RELATION$ table, which is described in Section 1.5.4.

This procedure automatically performs a commit operation, and therefore it cannot be rolled back. To delete the topology that you just created, call the SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER procedure.

The procedure creates a spatial index on the spatial features in the topology geometries, and a B-tree index on the combination of `tg_type` and `tg_id` in the topology geometries.

Users granted CONNECT and RESOURCE roles must also be granted the CREATE VIEW privilege to call the procedure. This is necessary because effective with Oracle Database 10*g* Release 2, the CONNECT role privilege reduction feature removed the following privileges from the CONNECT role: CREATE CLUSTER, CREATE DATABASE LINK, CREATE SEQUENCE, ALTER SESSION, CREATE SYNONYM, CREATE TABLE, and CREATE VIEW.

The topology geometry layer table (`table_name` parameter) cannot be an object table.

An exception is raised if `topology`, `table_name`, or `column_name` does not exist, or if `topo_geometry_layer_type` is not one of the supported values.

Examples

The following example adds a topology geometry layer to the `CITY_DATA` topology. The topology geometry layer consists of polygon geometries in the FEATURE column of the LAND_PARCELS table. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER('CITY_DATA', 'LAND_PARCELS', 'FEATURE',
'POLYGON');
```

SDO_TOPO.CREATE_TOPOLOGY

Format

```
SDO_TOPO.CREATE_TOPOLOGY(
   topology                IN VARCHAR2,
   tolerance               IN NUMBER,
   srid                    IN NUMBER DEFAULT NULL,
   node_table_storage      IN VARCHAR2 DEFAULT NULL,
   edge_table_storage      IN VARCHAR2 DEFAULT NULL,
   face_table_storage      IN VARCHAR2 DEFAULT NULL,
   history_table_storage   IN VARCHAR2 DEFAULT NULL.
   digits_right_of_decimal IN VARCHAR2 DEFAULT 16);
```

Description

Creates a topology.

Parameters

topology
Name of the topology to be created. Must not exceed 20 characters.

tolerance
Tolerance value associated with topology geometries in the topology. (Tolerance is explained in Section 1.2.1.)

srid
Coordinate system (spatial reference system) associated with all topology geometry layers in the topology. The default is null: no coordinate system is associated; otherwise, it must be a value from the SRID column of the SDO_COORD_REF_SYS table (described in *Oracle Spatial Developer's Guide*).

node_table_storage
Physical storage parameters used internally to create the <topology-name>_NODE$ table (described in Section 1.5.2). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K)`. If you do not specify this parameter, the default physical storage values are used.

edge_table_storage
Physical storage parameters used internally to create the <topology-name>_EDGE$ table (described in Section 1.5.1). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K)`. If you do not specify this parameter, the default physical storage values are used.

face_table_storage
Physical storage parameters used internally to create the <topology-name>_FACE$ table (described in Section 1.5.3). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K`

NEXT 200K). If you do not specify this parameter, the default physical storage values are used.

history_table_storage
Physical storage parameters used internally to create the <topology-name>_HISTORY$ table (described in Section 1.5.5. Must be a valid string for use with the CREATE TABLE statement. For example: TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K). If you do not specify this parameter, the default physical storage values are used.

digits_right_of_decimal
The number of digits permitted to the right of the decimal point in the expression of any coordinate position when features are added to an existing topology. All incoming features (those passed as arguments to the addLinearGeometry, addPolygonGeometry, or addPointGeometry method in the Java API or the equivalent PL/SQL subprograms) will be automatically snapped (truncated) to the number of digits right of the decimal that is specified in this parameter. The default is 16.

This value should be set to match the last digit right of the decimal point that is considered valid based on the accuracy of the incoming data. This mechanism is provided to improve the stability of the computational geometry during the feature insertion process, and to minimize the creation of sliver polygons and other undesired results.

Usage Notes

This procedure creates the <topology-name>_EDGE$, <topology-name>_NODE$, <topology-name>_FACE$, and <topology-name>_HISTORY$ tables, which are described in Section 1.5, and it creates B-tree indexes on the primary keys of these tables. This procedure also creates the metadata for the topology.

In the srid parameter, you can specify a geodetic coordinate system; however, all Spatial internal operations on the topology will use Cartesian (not geodetic) arithmetic operations. (Geodetic and non-geodetic coordinate systems are discussed in *Oracle Spatial Developer's Guide*.)

Node, edge, face, and history tables are created without partitions; however, you can alter any of these tables to make it partitioned. You can also create a partitioned spatial index on a partitioned table, as explained in *Oracle Spatial Developer's Guide*.

This procedure automatically performs a commit operation, and therefore it cannot be rolled back. To delete the topology that you just created, call the SDO_TOPO.DROP_TOPOLOGY procedure.

An exception is raised if the topology already exists.

Examples

The following example creates a topology named CITY_DATA. The spatial geometries in this topology have a tolerance value of 0.5 and use the WGS 84 coordinate system (longitude and latitude, SRID value 8307). (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.CREATE_TOPOLOGY('CITY_DATA', 0.5, 8307);
```

SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER

Format

```
SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER(
   topology        IN VARCHAR2,
   table_name      IN VARCHAR2,
   column_name     IN VARCHAR2);
```

Description

Deletes a topology geometry layer from a topology.

Parameters

topology
Topology from which to delete the topology geometry layer containing the topology geometries in the specified column. The topology must have been created using the SDO_TOPO.CREATE_TOPOLOGY procedure.

table_name
Name of the table containing the column specified in `column_name`.

column_name
Name of the column containing the topology geometries in the topology geometry layer to be deleted from the topology.

Usage Notes

This procedure deletes data associated with the specified topology geometry layer from the <topology-name>_RELATION$ table (described in Section 1.5.4). If this procedure is deleting the only remaining topology geometry layer from the topology, it also deletes the <topology-name>_RELATION$ table.

This procedure automatically performs a commit operation, and therefore it cannot be rolled back.

Examples

The following example deletes the topology geometry layer that is based on the geometries in the FEATURE column of the LAND_PARCELS table from the topology named `CITY_DATA`. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER('CITY_DATA', 'LAND_PARCELS',
'FEATURE');
```

SDO_TOPO.DROP_TOPOLOGY

Format

```
SDO_TOPO.DROP_TOPOLOGY(
   topology  IN VARCHAR2);
```

Description

Deletes a topology.

Parameters

topology
Name of the topology to be deleted. The topology must have been created using the SDO_TOPO.CREATE_TOPOLOGY procedure.

Usage Notes

This procedure deletes the <topology-name>_EDGE$, <topology-name>_NODE$, <topology-name>_FACE$, <topology-name>_NODE$, <topology-name>_RELATION$, and <topology-name>_HISTORY$ tables (described in Section 1.5).

If there are no topology layers associated with the topology, the topology is removed from the Spatial metadata.

This procedure automatically performs a commit operation, and therefore it cannot be rolled back.

A database user that owns a topology cannot be deleted. Therefore, before you can use the DROP USER ... CASCADE statement on a database user that owns a topology, you must connect as that user and execute the SDO_TOPO.DROP_TOPOLOGY procedure.

An exception is raised if the topology contains any topology geometries from any topology geometry layers. If you encounter this exception, delete all topology geometry layers in the topology using the SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER procedure for each topology geometry layer, and then drop the topology.

Examples

The following example drops the topology named CITY_DATA. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.DROP_TOPOLOGY('CITY_DATA');
```

SDO_TOPO.GET_FACE_BOUNDARY

Format

```
SDO_TOPO.GET_FACE_BOUNDARY(
   topology    IN VARCHAR2,
   face_id     IN NUMBER,
   all_edges   IN VARCHAR2 DEFAULT 'FALSE'
   ) RETURN SDO_LIST_TYPE;
```

Description

Returns a list of the signed ID numbers of the edges for the specified face.

Parameters

topology
Name of the topology that contains the face. Must not exceed 20 characters.

face_id
Face ID value of the face.

all_edges
`TRUE` includes all edges that bound the face (that is, that have the face on one or both sides); `FALSE` (the default) includes only edges that constitute the external boundary of the face. (See the examples for this function.)

Usage Notes

None.

Examples

The following examples return the ID numbers of the edges for the face whose face ID value is 1. The first example accepts the default value of `'FALSE'` for the `all_edges` parameter. The second example specifies `'TRUE'` for `all_edges`, and the list includes the ID numbers of the boundary edge and the two isolated edges on the face. (The examples refer to definitions and data from Section 1.12.)

```
-- Get the boundary of face with face_id 1.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 1) FROM DUAL;

SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA',1)
--------------------------------------------------------------------------------
SDO_LIST_TYPE(1)

-- Specify 'TRUE' for the all_edges parameter.
SELECT SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA', 1, 'TRUE') FROM DUAL;

SDO_TOPO.GET_FACE_BOUNDARY('CITY_DATA',1,'TRUE')
--------------------------------------------------------------------------------
SDO_LIST_TYPE(1, -26, 25)
```

SDO_TOPO.GET_TOPO_OBJECTS

Format

```
SDO_TOPO.GET_TOPO_OBJECTS(
   topology    IN VARCHAR2,
   geometry  IN SDO_GEOMETRY
   ) RETURN SDO_TOPO_OBJECT_ARRAY;
```

or

```
SDO_TOPO.GET_TOPO_OBJECTS(
   topology                  IN VARCHAR2,
   topo_geometry_layer_id  IN NUMBER,
   topo_geometry_id          IN NUMBER
   ) RETURN SDO_TOPO_OBJECT_ARRAY;
```

Description

Returns an array of SDO_TOPO_OBJECT objects that interact with a specified geometry object or topology geometry object.

Parameters

topology
Name of the topology. Must not exceed 20 characters.

geometry
Geometry object to be checked.

topo_geometry_layer_id
ID number of the topology geometry layer that contains the topology geometry object to be checked.

topo_geometry_id
ID number of the topology geometry object to be checked.

Usage Notes

The SDO_TOPO_OBJECT_ARRAY data type is described in Section 1.6.2.1.

For a topology that has a topology geometry layer hierarchy, this function works for all levels of the hierarchy, and it always returns the leaf-level (lowest-level) objects. (Topology geometry layer hierarchy is explained in Section 1.4.)

Examples

The following example returns the topology geometry objects that interact with land parcel P2 in the CITY_DATA topology. (The example refers to definitions and data from Section 1.12.1.)

```
-- CITY_DATA layer, land parcels (topo_geometry_ layer_id = 1),
-- parcel P2 (topo_geometry_id = 2)
SELECT SDO_TOPO.GET_TOPO_OBJECTS('CITY_DATA', 1, 2) FROM DUAL;
```

```
SDO_TOPO.GET_TOPO_OBJECTS('CITY_DATA',1,2)(TOPO_ID, TOPO_TYPE)
--------------------------------------------------------------------------------
SDO_TOPO_OBJECT_ARRAY(SDO_TOPO_OBJECT(9, 1), SDO_TOPO_OBJECT(10, 1), SDO_TOPO_OB
JECT(13, 1), SDO_TOPO_OBJECT(14, 1), SDO_TOPO_OBJECT(17, 1), SDO_TOPO_OBJECT(18,
 1), SDO_TOPO_OBJECT(6, 2), SDO_TOPO_OBJECT(7, 2), SDO_TOPO_OBJECT(8, 2), SDO_TO
PO_OBJECT(9, 2), SDO_TOPO_OBJECT(10, 2), SDO_TOPO_OBJECT(11, 2), SDO_TOPO_OBJECT
(12, 2), SDO_TOPO_OBJECT(13, 2), SDO_TOPO_OBJECT(14, 2), SDO_TOPO_OBJECT(17, 2),
 SDO_TOPO_OBJECT(18, 2), SDO_TOPO_OBJECT(19, 2), SDO_TOPO_OBJECT(20, 2), SDO_TOP
O_OBJECT(-6, 2), SDO_TOPO_OBJECT(-7, 2), SDO_TOPO_OBJECT(-8, 2), SDO_TOPO_OBJECT
(-9, 2), SDO_TOPO_OBJECT(-10, 2), SDO_TOPO_OBJECT(-11, 2), SDO_TOPO_OBJECT(-12,
2), SDO_TOPO_OBJECT(-13, 2), SDO_TOPO_OBJECT(-14, 2), SDO_TOPO_OBJECT(-17, 2), S
DO_TOPO_OBJECT(-18, 2), SDO_TOPO_OBJECT(-19, 2), SDO_TOPO_OBJECT(-20, 2), SDO_TO
PO_OBJECT(-1, 3), SDO_TOPO_OBJECT(3, 3), SDO_TOPO_OBJECT(4, 3), SDO_TOPO_OBJECT(
5, 3), SDO_TOPO_OBJECT(6, 3), SDO_TOPO_OBJECT(7, 3), SDO_TOPO_OBJECT(8, 3))
```

SDO_TOPO.INITIALIZE_AFTER_IMPORT

Format

```
SDO_TOPO.INITIALIZE_AFTER_IMPORT(
   topology  IN VARCHAR2);
```

Description

Creates (initializes) a topology that was imported from another database.

Parameters

topology
Name of the topology to be created. The topology must have been exported from a source database.

Usage Notes

This procedure creates the specified topology and all related database structures, adjusts (if necessary) the topology ID values in all feature tables, and creates the feature layers in the correct order.

Before calling this procedure, connect to the database as the user for the schema that is to own the topology to be created.

You must use this procedure after following all other required steps for exporting and importing the topology, as explained in Section 1.9.

Examples

The following example creates the topology named `CITY_DATA`, using information from the imported tables, including CITY_DATA_EXP$. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.INITIALIZE_AFTER_IMPORT('CITY_DATA');
```

SDO_TOPO.INITIALIZE_METADATA

Format

```
SDO_TOPO.INITIALIZE_METADATA(
   topology  IN VARCHAR2);
```

Description

Initializes the topology metadata: sets sequence information for the node, edge, and face tables, and creates (or re-creates) required indexes on these tables.

Parameters

topology
Name of the topology for which to initialize the sequences. The topology must have been created using the SDO_TOPO.CREATE_TOPOLOGY procedure.

Usage Notes

You should run this procedure after loading data into the node, edge, or face tables, to initialize the sequences for these tables with numeric values 2 higher than the highest ID values stored in those tables. This ensures that no attempt is made to reuse the unique ID values in these tables. (The node, edge, and face tables are described in Section 1.5.)

This procedure creates spatial indexes on the geometry or MBR geometry columns in the node, edge, and face tables. If the indexes were dropped before a bulk load operation, running this procedure after the bulk load will re-create these indexes.

Examples

The following example initializes the metadata for the topology named `CITY_DATA`. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.INITIALIZE_METADATA('CITY_DATA');
```

SDO_TOPO.PREPARE_FOR_EXPORT

Format

```
SDO_TOPO.PREPARE_FOR_EXPORT(
   topology  IN VARCHAR2);
```

Description

Prepares a topology to be exported to another database.

Parameters

topology
Name of the topology to be prepared for export. The topology must have been created using the SDO_TOPO.CREATE_TOPOLOGY procedure.

Usage Notes

This procedure prepares the specified topology in the current database (the source database) to be exported to another database (the target database).

This procedure creates a table in the current schema with a table name in the format <topology-name>_EXP$. This table contains the same columns as the USER_SDO_TOPO_INFO and ALL_SDO_TOPO_INFO views. These columns are described in Table 1–8 in Section 1.7.1.

Before calling this procedure, connect to the database as the owner of the topology.

For information about exporting and importing topologies, including the steps to be followed, see Section 1.9.

Examples

The following example prepares the topology named `CITY_DATA` for export to a target database. (The example refers to definitions and data from Section 1.12.1.)

```
EXECUTE SDO_TOPO.PREPARE_FOR_EXPORT('CITY_DATA');
```

SDO_TOPO.RELATE

Format

```
SDO_TOPO.RELATE(
   geom1  IN SDO_TOPO_GEOMETRY,
   geom2  IN SDO_TOPO_GEOMETRY,
   mask     IN VARCHAR2
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO.RELATE(
   feature1  IN SDO_TOPO_GEOMETRY,
   feature2  IN SDO_GEOMETRY,
   mask      IN VARCHAR2
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO.RELATE(
   geom               IN SDO_TOPO_GEOMETRY,
   topo_elem_array IN SDO_TOPO_OBJECT_ARRAY,
   mask               IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Examines two topology geometry objects, or a topology geometry and a spatial geometry, or a topology geometry and a topology object array (SDO_TOPO_OBJECT_ARRAY object), to determine their spatial relationship.

Parameters

geom1
Topology geometry object.

geom2
Topology geometry object.

feature1
Topology geometry object.

feature2
Spatial geometry object.

geom
Topology geometry object.

topo_elem_array
Topology object array (of type SDO_TOPO_OBJECT_ARRAY, which is described in Section 1.6.2.1).

mask
Specifies one or more relationships to check. See the list of keywords in the Usage Notes.

Usage Notes

The topology operators (described in Section 1.8.1) provide better performance than the SDO_TOPO.RELATE function if you are checking a large number of objects; however, if you are checking just two objects or a small number, the SDO_TOPO.RELATE function provides better performance. In addition, sometimes you may need to use the SDO_TOPO.RELATE function instead of a topology operator. For example, you cannot specify the DETERMINE mask keyword with the topology operators.

The following keywords can be specified in the `mask` parameter to determine the spatial relationship between two objects:

- ANYINTERACT: Returns TRUE if the objects are not disjoint.
- CONTAINS: Returns TRUE if the second object is entirely within the first object and the object boundaries do not touch; otherwise, returns FALSE.
- COVEREDBY: Returns TRUE if the first object is entirely within the second object and the object boundaries touch at one or more points; otherwise, returns FALSE.
- COVERS: Returns TRUE if the second object is entirely within the first object and the boundaries touch in one or more places; otherwise, returns FALSE.
- DETERMINE: Returns the one relationship keyword that best matches the geometries.
- DISJOINT: Returns TRUE if the objects have no common boundary or interior points; otherwise, returns FALSE.
- EQUAL: Returns TRUE if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns FALSE.
- INSIDE: Returns TRUE if the first object is entirely within the second object and the object boundaries do not touch; otherwise, returns FALSE.
- ON: Returns TRUE if the boundary and interior of a line (the first object) is completely on the boundary of a polygon (the second object); otherwise, returns FALSE.
- OVERLAPBDYDISJOINT: Returns TRUE if the objects overlap, but their boundaries do not interact; otherwise, returns FALSE.
- OVERLAPBDYINTERSECT: Returns TRUE if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns FALSE.
- TOUCH: Returns TRUE if the two objects share a common boundary point, but no interior points; otherwise, returns FALSE.

Values for `mask` (except for DETERMINE) can be combined using the logical Boolean operator OR. For example, 'INSIDE + TOUCH' returns the string TRUE or FALSE depending on the outcome of the test.

Examples

The following example finds whether or not the ANYINTERACT relationship exists between each topology geometry object in the CITY_STREETS table and the P3 land parcel (that is, which streets interact with that land parcel). (The example refers to definitions and data from Section 1.12. The output is reformatted for readability.)

```
SELECT c.feature_name,
  SDO_TOPO.RELATE(c.feature, l.feature, 'anyinteract') Any_Interaction
  FROM city_streets c, land_parcels l WHERE l.feature_name = 'P3';

FEATURE_NAME  ANY_INTERACTION
------------  ---------------
R1            TRUE
R2            FALSE
R3            FALSE
R4            FALSE
```

The following example finds whether or not the ANYINTERACT relationship exists between each topology geometry object in the CITY_STREETS table and an SDO_TOPO_OBJECT_ARRAY object that happens to be identical to the land parcel feature named P3. (This example uses definitions and data from Section 1.12.) The output is identical to that in the preceding example, and is reformatted for readability.

```
SELECT c.feature_name,
  SDO_TOPO.RELATE(c.feature,
    SDO_TOPO_OBJECT_ARRAY (SDO_TOPO_OBJECT (5, 3), SDO_TOPO_OBJECT (8, 3)),
    'anyinteract') Any_Interaction
  FROM city_streets c, land_parcels l WHERE l.feature_name = 'P3';

FEATURE_NAME  ANY_INTERACTION
------------  ---------------
R1            TRUE
R2            FALSE
R3            FALSE
R4            FALSE
```

4

SDO_TOPO_MAP Package Subprograms

The MDSYS.SDO_TOPO_MAP package contains subprograms (functions and procedures) that constitute part of the PL/SQL application programming interface (API) for the Spatial topology data model. This package contains subprograms related to editing topologies. These subprograms use a TopoMap object, either one that you previously created or that Spatial creates implicitly.

To use the subprograms in this chapter, you must understand the conceptual information about topology in Chapter 1, as well as the information about editing topologies in Chapter 2.

The rest of this chapter provides reference information about the SDO_TOPO_MAP subprograms, listed in alphabetical order.

SDO_TOPO_MAP.ADD_EDGE

Format

```
SDO_TOPO_MAP.ADD_EDGE(
   topology    IN VARCHAR2,
   node_id1    IN NUMBER,
   node_id2    IN NUMBER,
   geom        IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

Description

Adds an edge to a topology, and returns the edge ID of the added edge.

Parameters

topology
Name of the topology to which to add the edge, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

node_id1
Node ID of the start node for the edge to be added.

node_id2
Node ID of the end node for the edge to be added.

geom
SDO_GEOMETRY object (line or contiguous line string geometry) representing the edge to be added.

Usage Notes

Spatial automatically assigns an edge ID to the added edge. If `topology` is not null, the appropriate entry is inserted in the <topology-name>_EDGE$ table; and if the addition of the edge affects the face information table, the appropriate entries in the <topology-name>_FACE$ table are updated. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

If `node_id1` and `node_id2` are the same value, a loop edge is created.

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addEdge` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds an edge connecting node N3 to node N4 in the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.ADD_EDGE(null, 3, 4,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
```

```
    SDO_ORDINATE_ARRAY(25,35, 20,37)))
  INTO :res_number;

Call completed.

SQL> PRINT res_number;

RES_NUMBER
----------
        29
```

SDO_TOPO_MAP.ADD_ISOLATED_NODE

Format

```
SDO_TOPO_MAP.ADD_ISOLATED_NODE(
   topology    IN VARCHAR2,
   face_id     IN NUMBER,
   point       IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.ADD_ISOLATED_NODE(
   topology    IN VARCHAR2,
   point       IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.ADD_ISOLATED_NODE(
   topology    IN VARCHAR2,
   face_id     IN NUMBER,
   x           IN NUMBER,
   y           IN NUMBER
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.ADD_ISOLATED_NODE(
   topology    IN VARCHAR2,
   x           IN NUMBER,
   y           IN NUMBER
   ) RETURN NUMBER;
```

Description

Adds an isolated node (that is, an island node) to a topology, and returns the node ID of the added isolated node.

Parameters

topology
Name of the topology to which to add the isolated node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

face_id
Face ID of the face on which the isolated node is to be added. (An exception is raised if the specified point is not on the specified face.)

point
SDO_GEOMETRY object (point geometry) representing the isolated node to be added.

x
X-axis value of the point representing the isolated node to be added.

y
Y-axis value of the point representing the isolated node to be added.

Usage Notes

Spatial automatically assigns a node ID to the added node. If `topology` is not null, the appropriate entry is inserted in the <topology-name>_NODE$ table, and the <topology-name>_FACE$ table is updated to include an entry for the added isolated node. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

If you know the ID of the face on which the isolated node is to be added, you can specify the `face_id` parameter. If you specify this parameter, there are two benefits:

- Validation: The function checks to see if the point is on the specified face, and raises an exception if it is not. Otherwise, the function checks to see if the point is on any face in the topology, and raises an exception if it is not.
- Performance: The function checks only if the point is on the specified face. Otherwise, it checks potentially all faces in the topology to see if the point is on any face.

To add a non-isolated node, use the SDO_TOPO_MAP.ADD_NODE function.

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addIsolatedNode` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds an isolated node to the right of isolated node N4 on face F2, and it returns the node ID of the added node. It uses the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
DECLARE
  result_num NUMBER;
BEGIN
result_num := SDO_TOPO_MAP.ADD_ISOLATED_NODE(null, 2,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(22,37,NULL), NULL, NULL));
DBMS_OUTPUT.PUT_LINE('Result = ' || result_num);
END;
/
Result = 24

PL/SQL procedure successfully completed.
```

SDO_TOPO_MAP.ADD_LINEAR_GEOMETRY

Format

```
SDO_TOPO_MAP.ADD_LINEAR_GEOMETRY(
   topology  IN VARCHAR2,
   curve     IN SDO_GEOMETRY
   ) RETURN SDO_NUMBER_ARRAY;
```

or

```
SDO_TOPO_MAP.ADD_LINEAR_GEOMETRY(
   topology  IN VARCHAR2,
   coords    IN SDO_NUMBER_ARRAY
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Adds a linear (line string or multiline string) geometry to the topology, inserting edges and nodes as necessary based on the full intersection of the geometry with the edges and nodes in the topology graph, and an array of the edge IDs of the inserted and shared edges in sequence from the start to the end of the geometry.

Parameters

topology
Name of the topology to which to add the edge or edges, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

curve
SDO_GEOMETRY object (curve or line string geometry) representing the edge or edges to be added.

coords
SDO_NUMBER_ARRAY object specifying the coordinates of the edge or edges to be added.

Usage Notes

This function creates at least one new edge, and more edges if necessary. For example, if the line string geometry intersects an existing edge, two edges are created for the added line string, and the existing edge (the one being intersected) is divided into two edges. If `topology` is not null, Spatial automatically updates the <topology-name>_EDGE$ table as needed. (If `topology` is null, you can update this table at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

This function returns an array of the edge IDs of the inserted and shared edges in sequence from the start to the end of the geometry. If a segment in the added geometry overlaps an existing edge in the topology, the sign of the returned edge depends on the directions of the added segment and the existing edge: if the direction of the existing edge is the same as the linear geometry, the returned edge element is positive; if the direction of the existing edge is the opposite of the linear geometry, the returned edge element is negative.

An exception is raised if the object in the `curve` or `coords` parameter contains any line segments that run together (overlap) in any manner; however, the line segments can cross at one or more points.

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addLinearGeometry` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds an edge representing a specified line string geometry, and it returns the edge ID of the added edge. It uses the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.ADD_LINEAR_GEOMETRY(null,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,2,1),
    SDO_ORDINATE_ARRAY(50,10, 55,10, 57,11)))
FROM DUAL;

SDO_TOPO_MAP.ADD_LINEAR_GEOMETRY(NULL,SDO_GEOMETRY(2002,NULL,NULL,SDO_ELEM_INFO_
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(31)
```

SDO_TOPO_MAP.ADD_LOOP

Format

```
SDO_TOPO_MAP.ADD_LOOP(
   topology    IN VARCHAR2,
   node_id     IN NUMBER,
   geom        IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

Description

Adds an edge that loops and connects to the same node, and returns the edge ID of the added edge.

Parameters

topology
Name of the topology to which to add the edge, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

node_id
Node ID of the node to which to add the edge that will start and end at this node.

geom
SDO_GEOMETRY object (line string geometry) representing the edge to be added. The start and end points of the line string must be the same point representing `node_id`.

Usage Notes

This function creates a new edge, as well as a new face consisting of the interior of the loop. If the edge is added at an isolated node, the edge is an isolated edge. If `topology` is not null, Spatial automatically updates the <topology-name>_EDGE$ and <topology-name>_FACE$ tables as needed. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addLoop` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds an edge loop starting and ending at node N4, and it returns the edge ID of the added edge. It uses the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.ADD_LOOP(null, 4,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(20,37, 20,39, 25,39, 20,37)))
  INTO :res_number;

Call completed.
```

```
SQL> PRINT res_number;

RES_NUMBER
----------
        30
```

SDO_TOPO_MAP.ADD_NODE

Format

```
SDO_TOPO_MAP.ADD_NODE(
   topology            IN VARCHAR2,
   edge_id             IN NUMBER,
   point               IN SDO_GEOMETRY,
   coord_index         IN NUMBER,
   is_new_shape_point  IN VARCHAR2
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.ADD_NODE(
   topology            IN VARCHAR2,
   edge_id             IN NUMBER,
   x                   IN NUMBER,
   y                   IN NUMBER,
   coord_index         IN NUMBER,
   is_new_shape_point  IN VARCHAR2
   ) RETURN NUMBER;
```

Description

Adds a non-isolated node to a topology to split an existing edge, and returns the node ID of the added node.

Parameters

topology
Name of the topology to which to add the node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

edge_id
Edge ID of the edge on which the node is to be added.

point
SDO_GEOMETRY object (point geometry) representing the node to be added. The point must be an existing shape point or a new point that breaks a line segment connecting two consecutive shape points.

x
X-axis value of the point representing the node to be added. The point must be an existing shape point or a new point that breaks a line segment connecting two consecutive shape points.

y
Y-axis value of the point representing the node to be added. The point must be an existing shape point or a new point that breaks a line segment connecting two consecutive shape points.

coord_index
The index (position) of the array position in the edge coordinate array on or after which the node is to be added. Each vertex (node or shape point) has a position in the edge coordinate array. The start point (node) is index (position) 0, the first point after the start point is 1, and so on. (However, the `coord_index` value cannot be the index of the last vertex.) For example, if the edge coordinates are (2,2, 5,2, 8,3) the index of the second vertex (5,2) is 1.

is_new_shape_point
`TRUE` if the added node is to be a new shape point following the indexed vertex (`coord_index` value) of the edge; `FALSE` if the added node is exactly on the indexed vertex.

A value of `TRUE` lets you add a node at a new point, breaking an edge segment at the coordinates specified in the `point` parameter or the `x` and `y` parameter pair. A value of `FALSE` causes the coordinates in the `point` parameter or the `x` and `y` parameter pair to be ignored, and causes the node to be added at the existing shape point associated with the `coord_index` value.

Usage Notes

Spatial automatically assigns a node ID to the added node and creates a new edge. The split piece at the beginning of the old edge is given the edge ID of the old edge. If `topology` is not null, appropriate entries are inserted in the <topology-name>_NODE$ and <topology-name>_EDGE$ tables. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

To add an isolated node (that is, an island node), use the SDO_TOPO_MAP.ADD_ISOLATED_NODE function.

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addNode` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds a non-isolated node to the right of node N2 on edge E2, and it returns the node ID of the added node. It uses the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
DECLARE
  result_num NUMBER;
BEGIN
result_num := SDO_TOPO_MAP.ADD_NODE(null, 2,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(27,30,NULL), NULL, NULL),
  0, 'TRUE');
DBMS_OUTPUT.PUT_LINE('Result = ' || result_num);
END;
/
Result = 26

PL/SQL procedure successfully completed.
```

SDO_TOPO_MAP.ADD_POINT_GEOMETRY

Format

```
SDO_TOPO_MAP.ADD_POINT_GEOMETRY(
   topology  IN VARCHAR2,
   point       IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.ADD_POINT_GEOMETRY(
   topology  IN VARCHAR2,
   coord      IN SDO_NUMBER_ARRAY
   ) RETURN NUMBER;
```

Description

Adds a node representing a specified point geometry or coordinate pair, and returns the node ID of the added node.

Parameters

topology
Name of the topology to which to add the node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

point
SDO_GEOMETRY object (point geometry) representing the node to be added.

coord
SDO_NUMBER_ARRAY object specifying the coordinates of the node to be added.

Usage Notes

If the point coincides with an existing node, no changes are made to the topology. Otherwise, an isolated node or a node splitting an edge is added.

For information about adding and deleting nodes and edges, see Chapter 2.

This function is equivalent to using the `addPointGeometry` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds a node representing a specified point geometry, and it returns the node ID of the added node. It uses the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.ADD_POINT_GEOMETRY(null,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(57,12,NULL), NULL, NULL))
FROM DUAL;

SDO_TOPO_MAP.ADD_POINT_GEOMETRY(NULL,SDO_GEOMETRY(2001,NULL,SDO_POINT_TYPE(57,12
--------------------------------------------------------------------------------
                                                                              29
```

The following example adds a node at the specified coordinates (58, 12), and it returns the node ID of the added node. It uses the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.ADD_POINT_GEOMETRY(null, SDO_NUMBER_ARRAY(58,12))
FROM DUAL;

SDO_TOPO_MAP.ADD_POINT_GEOMETRY(NULL,SDO_NUMBER_ARRAY(58,12))
------------------------------------------------------------
                                                          30
```

SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY

Format

```
SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY(
   topology  IN VARCHAR2,
   polygon   IN SDO_GEOMETRY
   ) RETURN SDO_NUMBER_ARRAY;
```

or

```
SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY(
   topology  IN VARCHAR2,
   coords    IN SDO_NUMBER_ARRAY
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Adds one or more faces representing a specified polygon geometry, and returns the face ID of each added face.

Parameters

topology
Name of the topology to which to add the face or faces, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

polygon
SDO_GEOMETRY object (polygon or multipolygon geometry) representing the face or faces to be added. Each polygon in the object must have a single exterior ring that can contain any number of interior rings.

coords
SDO_NUMBER_ARRAY object specifying the coordinates of a single polygon geometry representing the face or faces to be added. The vertices of the polygon must be in counterclockwise order, with the last vertex the same as the first vertex.

Usage Notes

This function creates at least one new face, and more faces if necessary. For example, if the polygon geometry intersects an existing face, faces are created for the added polygon, and the existing face (the one being intersected) definition is adjusted. If `topology` is not null, Spatial automatically updates the <topology-name>_FACE$ table as needed. (If `topology` is null, you can update this table at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

If the polygon coincides with an existing face, no changes are made to the topology.

For a multipolygon geometry, no exterior ring may overlap any other exterior ring. For example, you cannot add a face representing the following single multipolygon geometry: a park (exterior ring) containing a lake (interior ring) with an island in the lake (exterior ring inside the preceding interior ring).

This function is equivalent to using the `addPolygonGeometry` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds a face representing a specified polygon geometry, and it returns and prints the face ID of the added edge. It uses the current updatable TopoMap object.

```
DECLARE
  res_number_array SDO_NUMBER_ARRAY;
  face_count NUMBER;
  face_ctr NUMBER;
  this_face NUMBER;
BEGIN
res_number_array := SDO_TOPO_MAP.ADD_POLYGON_GEOMETRY(null,
  SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1,1003,1),
    SDO_ORDINATE_ARRAY(61,10, 70,10, 70,15, 65,15, 61,10)));
-- DBMS_OUTPUT.PUT_LINE('Result = ' || res_number_array);
-- Print each face associated with the geometry.
face_count := res_number_array.count;
for face_ctr in 1..face_count loop
  this_face := res_number_array(face_ctr);
  dbms_output.put_line ('this face = '|| this_face);
  end loop;  -- printed each face
END;
/
this face = 12
```

SDO_TOPO_MAP.CHANGE_EDGE_COORDS

Format

```
SDO_TOPO_MAP.CHANGE_EDGE_COORDS(
   topology    IN VARCHAR2,
   edge_id     IN NUMBER,
   geom        IN SDO_GEOMETRY);
```

or

```
SDO_TOPO_MAP.CHANGE_EDGE_COORDS(
   topology            IN VARCHAR2,
   edge_id             IN NUMBER,
   geom                IN SDO_GEOMETRY,
   moved_iso_nodes  OUT SDO_NUMBER_ARRAY,
   moved_iso_edges  OUT SDO_NUMBER_ARRAY,
   allow_iso_moves    IN VARCHAR2);
```

Description

Changes the coordinates and related information about an edge.

Parameters

topology
Name of the topology containing the edge, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

edge_id
Edge ID of the edge whose coordinates are to be changed.

geom
SDO_GEOMETRY object (line or contiguous line string geometry) representing the modified edge. The start and end points of the modified edge must be the same as for the original edge.

moved_iso_nodes
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the node ID values of any isolated nodes that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the node ID values of any isolated nodes that did not move but that would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

moved_iso_edges
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the edge ID values of any isolated edges that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the edge ID values of any isolated edges that did not move but that

would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

allow_iso_moves
`TRUE` causes Spatial to allow an edge coordinates change operation that would cause any isolated nodes or edges to be in a different face, and to adjust the containing face information for such isolated nodes and edges; `FALSE` causes Spatial not to allow an edge coordinates change operation that would cause any isolated nodes or edges to be in a different face.

If you use the format that does not include the `allow_iso_moves` parameter, Spatial allows edge move operations that would cause any isolated nodes or edges to be in a different face, and it adjusts the containing face information for such isolated nodes and edges.

Usage Notes

If this procedure modifies a boundary between faces, Spatial automatically performs the following operations and updates the topology data model tables as needed: reassigning island nodes and faces, and adjusting the MBRs of the faces on both sides.

If `topology` is not null, this procedure modifies the information about the specified edge in the <topology-name>_EDGE$ table (described in Section 1.5.1). (If `topology` is null, you can update this table at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

You cannot use this procedure to change the start point or the end point, or both, of the specified edge. To do any of these operations, you must delete the edge, delete the node or nodes for the start or end point (or both) to be changed, add the necessary new node or nodes, and add the edge.

For information about editing topological elements, see Chapter 2.

This procedure is equivalent to using the `changeEdgeCoords` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example changes the coordinates of edge E1. (It changes only the third point, from 16,38 to 16,39.) It uses the current updatable TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.CHANGE_EDGE_COORDS(null, 1,
  SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1),
    SDO_ORDINATE_ARRAY(8,30, 16,30, 16,39, 3,38, 3,30, 8,30)));
```

SDO_TOPO_MAP.CLEAR_TOPO_MAP

Format

```
SDO_TOPO_MAP.CLEAR_TOPO_MAP(
   topo_map  IN VARCHAR2);
```

Description

Clears all objects and changes in the cache associated with a TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

Usage Notes

If the TopoMap object is updatable, this procedure changes it to be read-only.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

Contrast this procedure with the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure, which applies the changes in the cache associated with the TopoMap object to the topology. You cannot call the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure if you previously used the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure on the specified TopoMap object.

This procedure is equivalent to using the `clearCache` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example clears the cache associated with the TopoMap object named `CITY_DATA_TOPOMAP`, which is associated with the topology named `CITY_DATA`. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.CLEAR_TOPO_MAP('CITY_DATA_TOPOMAP');
```

SDO_TOPO_MAP.COMMIT_TOPO_MAP

Format

SDO_TOPO_MAP.COMMIT_TOPO_MAP;

Description

Updates the topology to reflect changes made to the current updatable TopoMap object, commits all changes to the database, and makes the TopoMap object read-only.

Parameters

None.

Usage Notes

Use this procedure when you are finished with a batch of edits to a topology and you want to commit all changes to the database. After the commit operation completes, you cannot edit the TopoMap object. To make further edits to the topology, you must either clear the cache (using the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure) or create a new TopoMap object (using the SDO_TOPO_MAP.CREATE_TOPO_MAP procedure), and then load the topology into the TopoMap object for update (using the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure).

Contrast this procedure with the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure, which leaves the TopoMap object available for editing operations and which does not perform a commit operation (and thus does not end the database transaction).

To roll back all TopoMap object changes, use the SDO_TOPO_MAP.ROLLBACK_TOPO_MAP procedure.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

This procedure is equivalent to using the `commitDB` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example commits to the database all changes to the current updatable TopoMap object, and prevents further editing of the TopoMap object.

```
EXECUTE SDO_TOPO_MAP.COMMIT_TOPO_MAP;
```

SDO_TOPO_MAP.CREATE_EDGE_INDEX

Format

```
SDO_TOPO_MAP.CREATE_EDGE_INDEX(
   topo_map  IN VARCHAR2);
```

Description

Creates an internal R-tree index (or rebuilds the index if one already exists) on the edges in the cache associated with a TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

Usage Notes

You can cause Spatial to create in-memory R-tree indexes to be built on the edges and faces in the specified TopoMap object. These indexes use some memory resources and take some time to create; however, they significantly improve performance if you edit a large number of topological elements in the session. They can also improve performance for queries that use a read-only TopoMap object. If the TopoMap object is updatable and if you are performing many editing operations, you should probably rebuild the indexes periodically; however, if the TopoMap object will not be updated, create the indexes when or after loading the read-only TopoMap object or after calling the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure.

Compare this procedure with the SDO_TOPO_MAP.CREATE_FACE_INDEX procedure, which creates an internal R-tree index (or rebuilds the index if one already exists) on the faces in the cache associated with a TopoMap object.

This procedure is equivalent to using the `createEdgeIndex` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example creates an internal R-tree index (or rebuilds the index if one already exists) on the edges in the cache associated with the TopoMap object named `CITY_DATA_TOPOMAP`, which is associated with the topology named `CITY_DATA`. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.CREATE_EDGE_INDEX('CITY_DATA_TOPOMAP');
```

SDO_TOPO_MAP.CREATE_FACE_INDEX

Format

```
SDO_TOPO_MAP.CREATE_FACE_INDEX(
   topo_map  IN VARCHAR2);
```

Description

Creates an internal R-tree index (or rebuilds the index if one already exists) on the faces in the cache associated with a TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

Usage Notes

You can cause Spatial to create in-memory R-tree indexes to be built on the edges and faces in the specified TopoMap object. These indexes use some memory resources and take some time to create; however, they significantly improve performance if you edit a large number of topological elements in the session. They can also improve performance for queries that use a read-only TopoMap object. If the TopoMap object is updatable and if you are performing many editing operations, you should probably rebuild the indexes periodically; however, if the TopoMap object will not be updated, create the indexes when or after loading the read-only TopoMap object or after calling the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure.

Compare this procedure with the SDO_TOPO_MAP.CREATE_EDGE_INDEX procedure, which creates an internal R-tree index (or rebuilds the index if one already exists) on the edges in the cache associated with a TopoMap object.

This procedure is equivalent to using the `createFaceIndex` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example creates an internal R-tree index (or rebuilds the index if one already exists) on the faces in the cache associated with the TopoMap object named `CITY_DATA_TOPOMAP`, which is associated with the topology named `CITY_DATA`. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.CREATE_FACE_INDEX('CITY_DATA_TOPOMAP');
```

SDO_TOPO_MAP.CREATE_FEATURE

Format (no topology geometry layer hierarchy or lowest level in a hierarchy)

```
SDO_TOPO_MAP.CREATE_FEATURE(
   topology          IN VARCHAR2,
   table_name        IN VARCHAR2,
   column_name       IN VARCHAR2,
   geom              IN SDO_GEOMETRY
   ) RETURN SDO_TOPO_GEOMETRY;
```

Format (parent level in a hierarchy)

```
SDO_TOPO_MAP.CREATE_FEATURE(
   topology          IN VARCHAR2,
   table_name        IN VARCHAR2,
   column_name       IN VARCHAR2,
   dml_condition     IN VARCHAR2
   ) RETURN SDO_TOPO_GEOMETRY;
```

Description

Creates a feature from Oracle Spatial geometries. (This function is intended to be used for inserting rows into a feature table.)

- The first format (with the `geom` parameter and without the `dml_condition` parameter) is for creating a feature in a topology without a topology geometry layer hierarchy or in the lowest level of a topology with a topology geometry layer hierarchy.
- The second format (with the `dml_condition` parameter and without the `geom` parameter) is for creating a feature in a parent level of a topology with a topology geometry layer hierarchy.

Parameters

topology
Topology having the associated specified feature table and feature column.

table_name
Name of the feature table containing the feature column specified in `column_name`.

column_name
Name of the feature column (of type SDO_TOPO_GEOMETRY) containing the topology geometries.

geom
Geometry objects.

dml_condition
For topologies with a topology geometry layer hierarchy (described in Section 1.4): DML condition for selecting rows from a child layer to be inserted into a parent layer. Specify the condition in a quoted string, but without the word WHERE. For example, to select only rows where the STATE_ABBR column value is `MA`, specify the following: `'state_abbr=''MA'''`

Usage Notes

This function is used to create features from existing geometries stored in a spatial table. Creating features from existing geometries is one approach to creating topology features; the other approach is to load the topology data into the node, edge, and face information tables. Both approaches are described in Section 1.1, which contains the following subsections:

- Section 1.1.1, "Using a Topology Built from Topology Data"
- Section 1.1.2, "Using a Topology Built from Spatial Geometries" (that is, the approach using the CREATE_FEATURE function)

When you use the first format of this function, you must first create and load an updatable TopoMap object. To create a topology feature or an associated topological element, the function internally calls the `addPointGeometry`, `addLinearGeometry`, or `addPolygonGeometry` method of the updatable TopoMap object, depending on the SDO_GTYPE value of the geometry object, and it calls the `updateTopology` method of the updatable TopoMap object to write topological elements to the database. If this function is called in an INSERT or UPDATE statement, a feature is created or updated in the feature table. When the function completes, it has the effect of overlaying the geometry onto the topology.

When you use the second format of this function, you do not need to create an updatable TopoMap object. The function internally collects TG_ID values of features in the child level based on the `dml_condition` parameter value, and it assembles an SDO_TGL_OBJECT_ARRAY object to create the SDO_GEOMETRY object.

To ensure that this function works correctly with all geometries, use a loop to call the function for each geometry. Do not use this function in a subquery in an INSERT or UPDATE statement, because doing so may cause inconsistencies in the topology, and you may not receive any error or warning messages about the inconsistencies.

An exception is raised if one or more of the following conditions exist:

- `topology`, `table_name`, or `column_name` does not exist.
- `geom` specifies geometry objects of a type inconsistent with the topology geometry layer type. For example, you cannot use line string geometries to create land parcel features.
- `dml_condition` is used with a topology that does not have a topology geometry layer hierarchy.
- The input geometries include any optimized shapes, such as optimized rectangles or circles.
- A line string or multiline string geometry contains any overlapping line segments.
- In a multipolygon geometry, an exterior ring overlaps any other exterior ring.

Examples

The following example populates the FEATURE column in the CITY_STREETS, TRAFFIC_SIGNS, and LAND_PARCELS feature tables with all geometries in the

GEOMETRY column in the CITY_STREETS_GEOM, TRAFFIC_SIGNS_GEOM, and LAND_PARCELS_GEOM spatial tables, respectively. This example assumes that an updatable TopoMap object has been created and loaded for the `CITY_DATA` topology. (The example refers to definitions and data from Section 1.12.2.)

```
BEGIN
  FOR street_rec IN (SELECT name, geometry FROM city_streets_geom) LOOP
   INSERT INTO city_streets VALUES(street_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'CITY_STREETS', 'FEATURE',
         street_rec.geometry));
  END LOOP;

  FOR sign_rec IN (SELECT name, geometry FROM traffic_signs_geom) LOOP
   INSERT INTO traffic_signs VALUES(sign_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'TRAFFIC_SIGNS', 'FEATURE',
         sign_rec.geometry));
  END LOOP;

  FOR parcel_rec IN (SELECT name, geometry FROM land_parcels_geom) LOOP
   INSERT INTO land_parcels VALUES(parcel_rec.name,
     SDO_TOPO_MAP.CREATE_FEATURE('CITY_DATA', 'LAND_PARCELS', 'FEATURE',
         parcel_rec.geometry));
  END LOOP;
END;
/
```

The following example creates a topology that has a topology geometry layer hierarchy with two layers: counties and states. The calls to the CREATE_FEATURE function that create parent layer (state) features include the `dml_condition` parameter (for example, `'p_name=''NH'''`).

```
declare
 name varchar2(64);
 cursor c1 is select state_abrv, county from
   counties order by 1, 2;
 stateabrv varchar2(2);
begin

 -- Initialize.
 sdo_topo_map.create_topo_map('cnty', 'm2', 10000, 10000, 10000);
 sdo_topo_map.load_topo_map('m2', -180, -90, 180, 90, 'true');

 -- Insert one county at a time.
 for cnty_rec in c1 loop
   stateabrv := cnty_rec.state_abrv;
   name := cnty_rec.county;
   insert into cnty_areas select state_abrv || '-' ||county,
     sdo_topo_map.create_feature('CNTY', 'CNTY_AREAS', 'FEATURE', geom) from
     counties where state_abrv=stateabrv and county=name;
 end loop;

 -- Roll back topology.
 sdo_topo_map.rollback_topo_map();
 sdo_topo_map.drop_topo_map('m2');

 -- Roll back inserts.
 rollback;

exception
 when others then
```

```
   dbms_output.put_line(SQLERRM);
   sdo_topo_map.rollback_topo_map();
   sdo_topo_map.drop_topo_map('m2');
   rollback;
end;
/

-- Add parent feature layer.
--
--   The following commented out statement can be used to populate the
--     child_layer_id parameter in sdo_topo.add_topo_geometry_layer.
--
--   select tg_layer_id
--     from user_sdo_topo_info
--     where TOPOLOGY = 'SC'
--       and table_name = 'SC_AREAS';
--
execute sdo_topo.add_topo_geometry_layer('SC','SC_P_AREAS', 'FEATURE', -
                                         'POLYGON', NULL, child_layer_id => 1);

-- Create and insert state features (logically) from county features.
insert into sc_p_areas (f_name, p_name, feature) values ('NH', 'US',
  sdo_topo_map.create_feature('SC','SC_P_AREAS','FEATURE','p_name=''NH'''));
insert into sc_p_areas (f_name, p_name, feature) values ('CT', 'US',
  sdo_topo_map.create_feature('SC','SC_P_AREAS','FEATURE','p_name=''CT'''));
insert into sc_p_areas (f_name, p_name, feature) values ('ME', 'US',
  sdo_topo_map.create_feature('SC','SC_P_AREAS','FEATURE','p_name=''ME'''));
insert into sc_p_areas (f_name, p_name, feature) values ('MA', 'US',
  sdo_topo_map.create_feature('SC','SC_P_AREAS','FEATURE','p_name=''MA'''));
commit;
```

SDO_TOPO_MAP.CREATE_TOPO_MAP

Format

```
SDO_TOPO_MAP.CREATE_TOPO_MAP(
   topology          IN VARCHAR2,
   topo_map          IN VARCHAR2,
   number_of_edges   IN NUMBER DEFAULT 100,
   number_of_nodes   IN NUMBER DEFAULT 80,
   number_of_faces   IN NUMBER DEFAULT 30);
```

Description

Creates a TopoMap object cache associated with an existing topology.

Parameters

topology
Name of the topology. Must not exceed 20 characters.

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

number_of_edges
An estimate of the maximum number of edges that will be in the TopoMap object at any given time. If you do not specify this parameter, a default value of 100 is used.

number_of_nodes
An estimate of the maximum number of nodes that will be in the TopoMap object at any given time. If you do not specify this parameter, a default value of 80 is used.

number_of_faces
An estimate of the maximum number of faces that will be in the TopoMap object at any given time. If you do not specify this parameter, a default value of 30 is used.

Usage Notes

The `number_of_edges`, `number_of_nodes`, and `number_of_faces` parameters let you improve the performance and memory usage of the procedure when you have a good idea of the approximate number of edges, nodes, or faces (or any combination) that will be placed in the cache associated with the specified TopoMap object. Spatial initially allocates memory cache for the specified or default number of objects of each type, and incrementally increases the allocation later if more objects need to be accommodated.

You can create more than one TopoMap object in a user session; however, there can be no more than one updatable TopoMap object at any given time in a user session.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

Using this procedure is equivalent to calling the constructor of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example creates a TopoMap object named CITY_DATA_TOPOMAP and its associated cache, and it associates the TopoMap object with the topology named CITY_DATA. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.CREATE_TOPO_MAP('CITY_DATA', 'CITY_DATA_TOPOMAP');
```

SDO_TOPO_MAP.DROP_TOPO_MAP

Format

```
SDO_TOPO_MAP.DROP_TOPO_MAP(
   topo_map  IN VARCHAR2);
```

Description

Deletes a TopoMap object from the current user session.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

Usage Notes

This procedure rolls back any uncommitted changes if the TopoMap object is updatable (that is, performs the equivalent of an SDO_TOPO_MAP.ROLLBACK_TOPO_MAP operation). It clears the cache associated with the TopoMap object, and removes the TopoMap object from the session.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

Using this procedure is equivalent to setting the variable of the TopoMap object to a null value in a client-side Java application. (The client-side Java API is described in Section 1.8.2.)

Examples

The following example drops the TopoMap object named `CITY_DATA_TOPOMAP`. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.DROP_TOPO_MAP('CITY_DATA_TOPOMAP');
```

SDO_TOPO_MAP.GET_CONTAINING_FACE

Format

```
SDO_TOPO_MAP.GET_CONTAINING_FACE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   point         IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.GET_CONTAINING_FACE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   x             IN NUMBER,
   y             IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the face ID number of the face that contains the specified point.

Parameters

topology
Name of the topology that contains the face and the point, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

point
Geometry object specifying the point.

x
X-axis value of the point.

y
Y-axis value of the point.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function determines, from the faces in the specified TopoMap object (including any island faces), which one face (if any) contains the specified point in its open set, excluding islands. (The open set, excluding islands, of a face consists of all points inside, but not on the boundary of, the face.) If the point is exactly on the boundary of a face, the function returns a value of 0 (zero).

If the entire topology has been loaded into the TopoMap object and if the point is not in any finite face in the cache, this function returns a value of -1 (for the universe face). If a window from the topology has been loaded into the TopoMap object and if the point is not in any finite face in the cache, this function returns a value of -1 (for the universe face) if the point is inside the window and a value of 0 (zero) if the point is outside the window. If neither the entire topology nor a window has been loaded, this function returns 0 (zero).

This function is equivalent to using the `getContainingFace` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the face ID number of the face that contains the point at (22, 37) in the `CITY_DATA_TOPOMAP` TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_CONTAINING_FACE(null, 'CITY_DATA_TOPOMAP',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(22,37,NULL), NULL, NULL))
  FROM DUAL;

SDO_TOPO_MAP.GET_CONTAINING_FACE(NULL,'CITY_DATA_TOPOMAP',SDO_GEOMETRY(2001,NULL
--------------------------------------------------------------------------------
                                                                               2
```

SDO_TOPO_MAP.GET_EDGE_ADDITIONS

Format

SDO_TOPO_MAP.GET_EDGE_ADDITIONS() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of edge ID numbers of edges that have been added to the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the edge ID numbers of edges in the current updatable TopoMap object that have been added since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no additions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getEdgeAdditions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID numbers of edges that have been added to the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_EDGE_ADDITIONS FROM DUAL;

GET_EDGE_ADDITIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(28, 29, 30, 32)
```

SDO_TOPO_MAP.GET_EDGE_CHANGES

Format

SDO_TOPO_MAP.GET_EDGE_CHANGES() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of edge ID numbers of edges that have been changed (modified) in the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the edge ID numbers of edges in the current updatable TopoMap object that have been changed since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no changes during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getEdgeChanges` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID numbers of edges that have been changed in the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_EDGE_CHANGES FROM DUAL;

GET_EDGE_CHANGES
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(3, 2, 1)
```

SDO_TOPO_MAP.GET_EDGE_COORDS

Format

```
SDO_TOPO_MAP.GET_EDGE_COORDS(
   topology     IN VARCHAR2,
   topo_map  IN VARCHAR2,
   edge_id     IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array with the coordinates of the start node, shape points, and end node for the specified edge.

Parameters

topology
Name of the topology that contains the edge, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

edge_id
Edge ID value of the edge.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function is equivalent to using the `getEdgeCoords` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the coordinates of the start node, shape points, and end node for the edge whose edge ID value is 1. The returned array contains coordinates for six points. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_EDGE_COORDS(null, 'CITY_DATA_TOPOMAP', 1) FROM DUAL;

SDO_TOPO_MAP.GET_EDGE_COORDS(NULL,'CITY_DATA_TOPOMAP',1)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(8, 30, 16, 30, 16, 38, 3, 38, 3, 30, 8, 30)
```

SDO_TOPO_MAP.GET_EDGE_DELETIONS

Format

SDO_TOPO_MAP.GET_EDGE_DELETIONS() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of edge ID numbers of edges that have been deleted from the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the edge ID numbers of edges in the current updatable TopoMap object that have been deleted since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no deletions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getEdgeDeletions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID numbers of edges that have been deleted from the current updatable TopoMap object. In this case, the return of an empty SDO_NUMBER_ARRAY object indicates that no edges have been deleted.

```
SELECT SDO_TOPO_MAP.GET_EDGE_DELETIONS FROM DUAL;

GET_EDGE_DELETIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY()
```

SDO_TOPO_MAP.GET_EDGE_NODES

Format

```
SDO_TOPO_MAP.GET_EDGE_NODES(
   topology     IN VARCHAR2,
   topo_map  IN VARCHAR2,
   edge_id     IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array with the ID numbers of the start and end nodes on the specified edge.

Parameters

topology
Name of the topology that contains the edge, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

edge_id
Edge ID value of the edge.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

If the edge starts and ends at a node, the ID number of the node is the first and last number in the array.

This function has no exact equivalent method in the `TopoMap` class of the client-side Java API (described in Section 1.8.2). The `getEdge` method returns a Java edge object of the `oracle.spatial.topo.Edge` class.

Examples

The following example returns the ID numbers of the nodes on the edge whose edge ID value is 1. The returned array contains two nodes ID numbers, both of them 1 (for the same node), because the specified edge starts and ends at the node with node ID 1 and has a loop edge. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_EDGE_NODES(null, 'CITY_DATA_TOPOMAP', 1) FROM DUAL;

SDO_TOPO_MAP.GET_EDGE_NODES(NULL,'CITY_DATA_TOPOMAP',1)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(1, 1)
```

SDO_TOPO_MAP.GET_FACE_ADDITIONS

Format

SDO_TOPO_MAP.GET_FACE_ADDITIONS() RETURN SDO_NUMBER_ARRAY

Description

Returns an array of face ID numbers of faces that have been added to the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the face ID numbers of faces in the current updatable TopoMap object that have been added since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no additions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getFaceAdditions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the face ID numbers of faces that have been added to the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_FACE_ADDITIONS FROM DUAL;

GET_FACE_ADDITIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(11)
```

SDO_TOPO_MAP.GET_FACE_CHANGES

Format

SDO_TOPO_MAP.GET_FACE_CHANGES() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of face ID numbers of faces that have been changed (modified) in the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the face ID numbers of faces in the current updatable TopoMap object that have been changed since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no changes during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getFaceChanges` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the face ID numbers of faces that have been changed in the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_FACE_CHANGES FROM DUAL;

GET_FACE_CHANGES
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(2, 1, -1)
```

SDO_TOPO_MAP.GET_FACE_BOUNDARY

Format

```
SDO_TOPO_MAP.GET_FACE_BOUNDARY(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   face_id      IN NUMBER
   option       IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array with the edge ID numbers of the edges that make up the boundary for the specified face.

Parameters

topology
Name of the topology that contains the face, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

face_id
Face ID value of the face.

option
One of the following numbers to indicate an option for computing the boundary: 0 for an external boundary ring without spurs (that is, without doubly traced edges), 1 for external and internal rings without spurs, or 2 for external and internal rings with spurs. A value of 2 returns the full, though possibly degenerate, boundary.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function is equivalent to using the `getFaceBoundary` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edges in the external boundary ring without spurs for the face whose face ID value is 3. The returned array contains four edge ID values. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_FACE_BOUNDARY(null, 'CITY_DATA_TOPOMAP', 3, 0) FROM DUAL;

SDO_TOPO_MAP.GET_FACE_BOUNDARY(NULL,'CITY_DATA_TOPOMAP',3,0)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(19, 6, 21, 9)
```

SDO_TOPO_MAP.GET_FACE_DELETIONS

Format

SDO_TOPO_MAP.GET_FACE_DELETIONS() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of face ID numbers of faces that have been deleted from the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the face ID numbers of faces in the current updatable TopoMap object that have been deleted since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no deletions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getFaceDeletions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the face ID numbers of faces that have been deleted from the current updatable TopoMap object. In this case, the return of an empty SDO_NUMBER_ARRAY object indicates that no faces have been deleted.

```
SELECT SDO_TOPO_MAP.GET_FACE_DELETIONS FROM DUAL;

GET_FACE_DELETIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY()
```

SDO_TOPO_MAP.GET_NEAREST_EDGE

Format

```
SDO_TOPO_MAP.GET_NEAREST_EDGE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   point         IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.GET_NEAREST_EDGE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   x               IN NUMBER,
   y               IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the edge ID number of the edge that is nearest (closest to) the specified point.

Parameters

topology
Name of the topology that contains the edge and the point, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

point
Geometry object specifying the point.

x
X-axis value of the point.

y
Y-axis value of the point.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

The nearest edge is determined from the representation of the topology in the database, using the spatial index. If there are changed, added, or deleted edges in the instance and the database has not been updated to reflect those changes, the result may not reflect the true situation in the TopoMap object cache.

If multiple edges are equally close to the point, any one of the edge ID values is returned. If no edges exist in the topology, this function returns 0 (zero).

This function is equivalent to using the `getNearestEdge` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID number of the edge that is closest to the point at (8, 8) in the `CITY_DATA_TOPOMAP` TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NEAREST_EDGE(null, 'CITY_DATA_TOPOMAP',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8,8,NULL), NULL, NULL))
  FROM DUAL;

SDO_TOPO_MAP.GET_NEAREST_EDGE(NULL,'CITY_DATA_TOPOMAP',SDO_GEOMETRY(2001,NULL,SD
--------------------------------------------------------------------------------
                                                                              22
```

SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE

Format

```
SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE(
   topo_map  IN VARCHAR2,
   point     IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE(
   topo_map  IN VARCHAR2,
   x         IN NUMBER,
   y         IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the edge ID number of the edge that, of the edges loaded in the specified TopoMap object, is nearest (closest to) the specified point.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

point
Geometry object specifying the point.

x
X-axis value of the point.

y
Y-axis value of the point.

Usage Notes

If multiple edges are equally close to the point, any one of the edge ID values is returned. If no topology data is loaded or if no edges exist in the cache, this function returns 0 (zero).

This function is equivalent to using the `getNearestEdgeInCache` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID number of the edge that is closest to the point at (8, 8) in the `CITY_DATA_TOPOMAP` TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE('CITY_DATA_TOPOMAP',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8,8,NULL), NULL, NULL))
  FROM DUAL;
```

```
SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE('CITY_DATA_TOPOMAP',SDO_GEOMETRY(2001,NUL
--------------------------------------------------------------------------------
                                                                              22
```

SDO_TOPO_MAP.GET_NEAREST_NODE

Format

```
SDO_TOPO_MAP.GET_NEAREST_NODE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   point         IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.GET_NEAREST_NODE(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   x               IN NUMBER,
   y               IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID number of the node that is nearest (closest to) the specified point.

Parameters

topology
Name of the topology that contains the node and the point, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

point
Geometry object specifying the point.

x
X-axis value of the point.

y
Y-axis value of the point.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

The nearest node is determined from the representation of the topology in the database, using the spatial index. If there are changed, added, or deleted nodes in the instance and the database has not been updated to reflect those changes, the result may not reflect the true situation in the TopoMap object cache.

If multiple nodes are equally close to the point, any one of the node ID values is returned.

This function is equivalent to using the `getNearestNode` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the node ID number of the node that is closest to the point at (8, 8) in the `CITY_DATA_TOPOMAP` TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NEAREST_NODE(null, 'CITY_DATA_TOPOMAP',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8,8,NULL), NULL, NULL))
  FROM DUAL;

SDO_TOPO_MAP.GET_NEAREST_NODE(NULL,'CITY_DATA_TOPOMAP',SDO_GEOMETRY(2001,NULL,SD
--------------------------------------------------------------------------------
                                                                               8
```

SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE

Format

```
SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE(
   topo_map  IN VARCHAR2,
   point     IN SDO_GEOMETRY
   ) RETURN NUMBER;
```

or

```
SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE(
   topo_map  IN VARCHAR2,
   x         IN NUMBER,
   y         IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID number of the node that, of the nodes loaded in the specified TopoMap object, is nearest (closest to) the specified point.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

point
Geometry object specifying the point.

x
X-axis value of the point.

y
Y-axis value of the point.

Usage Notes

If multiple nodes are equally close to the point, any one of the node ID values is returned. If no topology data is loaded or if no nodes exist in the cache, this function returns 0 (zero).

This function is equivalent to using the `getNearestNodeInCache` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the node ID number of the node that is closest to the point at (8, 8) in the `CITY_DATA_TOPOMAP` TopoMap object. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE('CITY_DATA_TOPOMAP',
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8,8,NULL), NULL, NULL))
  FROM DUAL;
```

```
SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE('CITY_DATA_TOPOMAP',SDO_GEOMETRY(2001,NUL
--------------------------------------------------------------------------------
                                                                               8
```

SDO_TOPO_MAP.GET_NODE_ADDITIONS

Format

SDO_TOPO_MAP.GET_NODE_ADDITIONS() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of node ID numbers of nodes that have been added to the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the node ID numbers of nodes in the current updatable TopoMap object that have been added since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no additions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getNodeAdditions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the node ID numbers of nodes that have been added to the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_NODE_ADDITIONS FROM DUAL;

GET_NODE_ADDITIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(24, 25, 26, 27, 28)
```

SDO_TOPO_MAP.GET_NODE_CHANGES

Format

SDO_TOPO_MAP.GET_NODE_CHANGES() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of node ID numbers of nodes that have been changed (modified) in the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the node ID numbers of nodes in the current updatable TopoMap object that have been changed since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no changes during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getNodeChanges` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the node ID numbers of nodes that have been changed in the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_NODE_CHANGES FROM DUAL;

GET_NODE_CHANGES
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(2, 4)
```

SDO_TOPO_MAP.GET_NODE_COORD

Format

```
SDO_TOPO_MAP.GET_NODE_COORD(
   topology    IN VARCHAR2,
   topo_map  IN VARCHAR2,
   node_id     IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns an SDO_GEOMETRY object with the coordinates of the specified node.

Parameters

topology
Name of the topology that contains the node, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

node_id
Node ID value of the node.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function is equivalent to using the `getNodeCoord` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns a geometry object with the coordinates of the node whose node ID value is 14. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NODE_COORD(null, 'CITY_DATA_TOPOMAP', 14) FROM DUAL;

SDO_TOPO_MAP.GET_NODE_COORD(NULL,'CITY_DATA_TOPOMAP',14)(SDO_GTYPE, SDO_SRID, SD
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, 0, SDO_POINT_TYPE(21, 14, NULL), NULL, NULL)
```

SDO_TOPO_MAP.GET_NODE_DELETIONS

Format

SDO_TOPO_MAP.GET_NODE_DELETIONS() RETURN SDO_NUMBER_ARRAY;

Description

Returns an array of node ID numbers of nodes that have been deleted from the current updatable TopoMap object.

Parameters

None.

Usage Notes

This function returns the node ID numbers of nodes in the current updatable TopoMap object that have been deleted since the object was most recently loaded (using SDO_TOPO_MAP.LOAD_TOPO_MAP), updated (using SDO_TOPO_MAP.UPDATE_TOPO_MAP), cleared (using SDO_TOPO_MAP.CLEAR_TOPO_MAP), committed (using SDO_TOPO_MAP.COMMIT_TOPO_MAP), or rolled back (using SDO_TOPO_MAP.ROLLBACK_TOPO_MAP). If there have been no deletions during that time, the function returns an empty SDO_NUMBER_ARRAY object.

This function is equivalent to using the `getNodeDeletions` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the node ID numbers of nodes that have been deleted from the current updatable TopoMap object. In this case, the return of an empty SDO_NUMBER_ARRAY object indicates that no nodes have been deleted.

```
SELECT SDO_TOPO_MAP.GET_NODE_DELETIONS FROM DUAL;

GET_NODE_DELETIONS
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY()
```

SDO_TOPO_MAP.GET_NODE_FACE_STAR

Format

```
SDO_TOPO_MAP.GET_NODE_FACE_STAR(
   topology     IN VARCHAR2,
   topo_map  IN VARCHAR2,
   node_id     IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an SDO_NUMBER_ARRAY object with the face ID numbers, in clockwise order, of the faces that are connected to the specified node.

Parameters

topology
Name of the topology that contains the node, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

node_id
Node ID value of the node.

Usage Notes

The **node face star** of a node is the faces that are connected to the node. One face is returned for each edge connected to the node. For an isolated node, the containing face is returned. A face may appear more than once in the list.

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function is equivalent to using the `getNodeFaceStar` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

To return the node star of a node, use the SDO_TOPO_MAP.GET_NODE_STAR function.

Examples

The following example returns the node face star of the node whose node ID value is 14. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NODE_FACE_STAR(null, 'CITY_DATA_TOPOMAP', 14) FROM DUAL;

SDO_TOPO_MAP.GET_NODE_FACE_STAR(NULL,'CITY_DATA_TOPOMAP',14)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(4, 7, 6, 3)
```

SDO_TOPO_MAP.GET_NODE_STAR

Format

```
SDO_TOPO_MAP.GET_NODE_STAR(
   topology     IN VARCHAR2,
   topo_map  IN VARCHAR2,
   node_id     IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an SDO_NUMBER_ARRAY object with the edge ID numbers, in clockwise order, of the edges that are connected to the specified node.

Parameters

topology
Name of the topology that contains the node, or a null value, as explained in Section 2.1.3. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Section 2.1.3. (TopoMap objects are explained in Section 2.1.1.)

node_id
Node ID value of the node.

Usage Notes

The **node star** of a node is the edges that are connected to the node. A positive edge ID represents an edge for which the node is its start node. A negative edge ID represents an edge for which the node is its end node. If any loops are connected to the node, edges may appear in the list twice with opposite signs.

The `topology` or `topo_map` parameter should specify a valid name, as explained in Section 2.1.3.

This function is equivalent to using the `getNodeStar` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

To return the node face star of a node, use the SDO_TOPO_MAP.GET_NODE_FACE_STAR function.

Examples

The following example returns the node star of the node whose node ID value is 14. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_NODE_STAR(null, 'CITY_DATA_TOPOMAP', 14) FROM DUAL;

SDO_TOPO_MAP.GET_NODE_STAR(NULL,'CITY_DATA_TOPOMAP',14)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(19, -10, -20, -9)
```

SDO_TOPO_MAP.GET_TOPO_NAME

Format

```
SDO_TOPO_MAP.GET_TOPO_NAME(
   topo_map  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the topology associated with the specified TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

Usage Notes

This function is equivalent to using the `getTopoName` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the name of the topology associated with the TopoMap object named `CITY_DATA_TOPOMAP`. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.GET_TOPO_NAME('CITY_DATA_TOPOMAP') FROM DUAL;

SDO_TOPO_MAP.GET_TOPO_NAME('CITY_DATA_TOPOMAP')
--------------------------------------------------------------------------------
CITY_DATA
```

SDO_TOPO_MAP.GET_TOPO_TRANSACTION_ID

Format

SDO_TOPO_MAP.GET_TOPO_TRANSACTION_ID() RETURN NUMBER;

Description

Returns the topology transaction ID number, if data has been loaded into the current updatable TopoMap object.

Parameters

None.

Usage Notes

For each row in the history information table for a topology, the TOPO_TX_ID column contains the topology transaction ID number. The history information table is described in Section 1.5.5.

This function is equivalent to using the `getTopoTransactionId` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the topology transaction ID number for the current updatable TopoMap object.

```
SELECT SDO_TOPO_MAP.GET_TOPO_TRANSACTION_ID FROM DUAL;

GET_TOPO_TRANSACTION_ID
-----------------------
                      1
```

SDO_TOPO_MAP.LIST_TOPO_MAPS

Format

SDO_TOPO_MAP.LIST_TOPO_MAPS() RETURN VARCHAR2;

Description

Returns a comma-delimited list of entries for each TopoMap object currently active in the session, or an empty string if there are no currently active TopoMap objects.

Parameters

None.

Usage Notes

Each entry in the comma-delimited list contains the following information: the name of the TopoMap object, the name of the topology associated with the TopoMap object, and either `updatable` if the TopoMap object can be updated (that is, edited) or `read-only` if the TopoMap object cannot be updated.

For more information about TopoMap objects, including updatable and read-only status, see Section 2.1.1.

To remove a TopoMap object from the session, use the SDO_TOPO_MAP.DROP_TOPO_MAP procedure.

Examples

The following example lists the Topomap object name, topology name, and whether the object is updatable or read-only for each TopoMap object currently active in the session. (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.LIST_TOPO_MAPS FROM DUAL;

LIST_TOPO_MAPS
--------------------------------------------------------------------------------
(CITY_DATA_TOPOMAP, CITY_DATA, updatable)
```

SDO_TOPO_MAP.LOAD_TOPO_MAP

Format (Function)

```
SDO_TOPO_MAP.LOAD_TOPO_MAP(
   topo_map        IN VARCHAR2,
   allow_updates   IN VARCHAR2,
   build_indexes    IN VARCHAR2 DEFAULT 'TRUE'
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO_MAP.LOAD_TOPO_MAP(
   topo_map        IN VARCHAR2,
   xmin             IN NUMBER,
   ymin             IN NUMBER,
   xmax             IN NUMBER,
   ymax             IN NUMBER,
   allow_updates   IN VARCHAR2,
   build_indexes   IN VARCHAR2 DEFAULT 'TRUE'
   ) RETURN VARCHAR2;
```

Format (Procedure)

```
SDO_TOPO_MAP.LOAD_TOPO_MAP(
   topo_map        IN VARCHAR2,
   allow_updates   IN VARCHAR2,
   build_indexes    IN VARCHAR2 DEFAULT 'TRUE');'
```

or

```
SDO_TOPO_MAP.LOAD_TOPO_MAP(
   topo_map        IN VARCHAR2,
   xmin             IN NUMBER,
   ymin             IN NUMBER,
   xmax             IN NUMBER,
   ymax             IN NUMBER,
   allow_updates   IN VARCHAR2);
   build_indexes    IN VARCHAR2 DEFAULT 'TRUE');'
```

Description

Loads the topological elements (primitives) for an entire topology or a window (rectangular portion) of a topology into a TopoMap object. If you use a function format, returns the string `TRUE` if topological elements were loaded into the cache, and `FALSE` if no topological elements were loaded into the cache.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

xmin
Lower-left X coordinate value for the window (rectangular portion of the topology) to be loaded.

See the Usage Notes and Figure 4–1 for information about which topological elements are loaded when you specify a window.

ymin
Lower-left Y coordinate value for the window (rectangular portion of the topology) to be loaded.

xmax
Upper-right X coordinate value for the window (rectangular portion of the topology) to be loaded.

ymax
Upper-right Y coordinate value for the window (rectangular portion of the topology) to be loaded.

allow_updates
`TRUE` makes the TopoMap object updatable; that is, it allows topology editing operations to be performed on the TopoMap object and changes to be written back to the database. `FALSE` makes the TopoMap object read-only with respect to the database; that is, it allows topology editing operations to be performed on the TopoMap object but does not allow changes to be written back to the database.

Making a TopoMap object updatable causes the topological elements in the TopoMap object to be locked, which means that they cannot be included in an updatable TopoMap object in the session of another database user. (Within any given user session, there can be no more than one updatable TopoMap object active.)

build_indexes
`TRUE` (the default) builds in-memory R-tree indexes for edge and face data; `FALSE` does not build in-memory R-tree indexes for edge and face data. The indexes improve the performance of editing operations, especially with large topologies.

Usage Notes

Using a procedure format for loading the TopoMap object is more efficient than using the function format, if you do not need to know if any topological elements were loaded (for example, if the specified topology or rectangular area is empty). Using a function format lets you know if any topological elements were loaded.

You must create the TopoMap object (using the SDO_TOPO_MAP.CREATE_TOPO_MAP procedure) before you load data into it.

You cannot use this function or procedure if the TopoMap object already contains data. If the TopoMap object contains any data, you must do one of the following before calling this function or procedure: commit the changes (using the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure) and clear the cache (using the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure), or roll back the changes (using the SDO_TOPO_MAP.ROLLBACK_TOPO_MAP procedure).

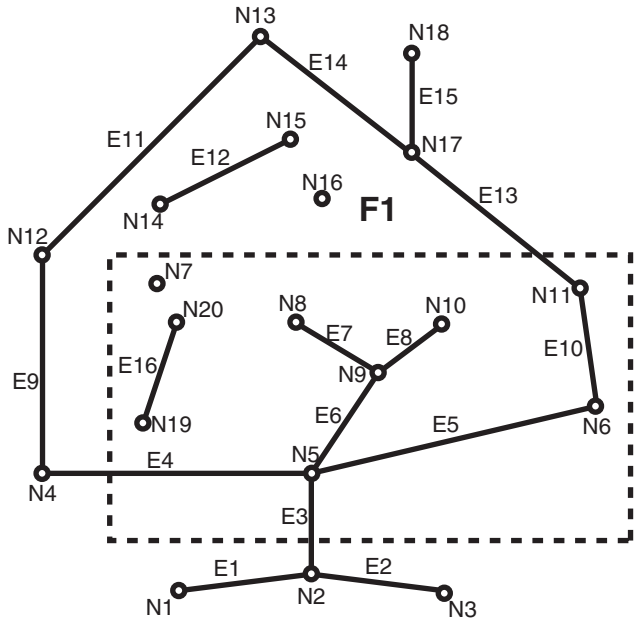
For information about using an in-memory cache to edit topological elements, see Section 2.1.

This function or procedure is equivalent to using the `loadTopology` or `loadWindow` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Every TopoMap object, whether for an entire topology or for a window specified using the `xmin`, `ymin`, `xmax`, and `ymax` parameters, has a region associated with it. For an updatable TopoMap object, updates are allowed only within this region. (The region might also contain topological elements that you cannot update directly, but that might be modified by Oracle Spatial as needed as a result of your editing operations.)

When a TopoMap object is loaded, all nodes, faces, and edges that intersect the region for the TopoMap object are loaded. When a face is loaded, all edges and nodes that are on the boundary of the face are loaded. When an edge is loaded, the start node and end node of the edge are loaded. Consider the topology and the window (shown by a dashed line) in Figure 4–1.

Figure 4–1 Loading Topological Elements into a Window



With the window shown in Figure 4–1:

- Face F1 is loaded because it partially overlaps the window.
- The following edges are loaded: E3, E4, E5, E6, E7, E8, E9, E10, E11, E12, E13, E14, E16.

  Edge E3 is loaded because it partially overlaps the window.

  Edge E9 is loaded because it bounds a face (F1) that partially overlaps a window.

  Edge E12 is loaded because it is an island edge in a face (F1) that partially overlaps the window.

  Edge E1 is not loaded because it is not associated with any face that interacts with the window.
- The following nodes are loaded: N2, N5, N6, N7, N8, N9, N10, N11, N12, N16, N19, N20.

Non-isolated node N2 is loaded because edge E3 is loaded.

Non-isolated node N12 is loaded because edges E9 and E11 are loaded.

Isolated node N16 is loaded because it is an isolated (island) node inside a locked face.

Examples

The following example loads all CITY_DATA topology elements into its associated TopoMap object for editing and builds the in-memory R-tree indexes by default. It returns a result indicating that the operation was successful and that some topological elements were loaded into the cache. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.LOAD_TOPO_MAP('CITY_DATA_TOPOMAP', 'TRUE') INTO :res_varchar;

Call completed.

PRINT res_varchar;

RES_VARCHAR
--------------------------------------------------------------------------------
TRUE
```

SDO_TOPO_MAP.MOVE_EDGE

Format

```
SDO_TOPO_MAP.MOVE_EDGE(
   topology          IN VARCHAR2,
   edge_id           IN NUMBER,
   s_node_id         IN NUMBER,
   t_node_id         IN NUMBER,
   edge_coords       IN SDO_NUMBER_ARRAY);
```

or

```
SDO_TOPO_MAP.MOVE_EDGE(
   topology          IN VARCHAR2,
   edge_id           IN NUMBER,
   s_node_id         IN NUMBER,
   t_node_id         IN NUMBER,
   edge_coords       IN SDO_NUMBER_ARRAY,
   moved_iso_nodes   OUT SDO_NUMBER_ARRAY,
   moved_iso_edges   OUT SDO_NUMBER_ARRAY,
   allow_iso_moves   IN VARCHAR2);
```

Description

Moves a non-isolated edge.

Parameters

topology
Name of the topology in which to move the edge, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

edge_id
Edge ID of the edge to be moved.

edge_coords
An array of coordinates of the resulting moved edge, from start point to end point.

s_node_id
Node ID of the source node, which identifies the point (start node or end node of the edge) affected by the move, before the move occurs. For example, if the end point of edge E19 is to be moved from node N17 to node N16, the `s_node_id` value is the node ID number for node N17.

t_node_id
Node ID of the target node, which identifies the point affected by the move, after the move occurs. For example, if the end point of edge E19 is to be moved from node N17 to node N16, the `t_node_id` value is the node ID number for node N16.

moved_iso_nodes
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the node ID values of any isolated nodes that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the node ID values of any isolated nodes that did not move but that would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

moved_iso_edges
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the edge ID values of any isolated edges that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the edge ID values of any isolated edges that did not move but that would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

allow_iso_moves
`TRUE` causes Spatial to allow an edge move operation that would cause any isolated nodes or edges to be in a different face, and to adjust the containing face information for such isolated nodes and edges; `FALSE` causes Spatial not to allow an edge move operation that would cause any isolated nodes or edges to be in a different face.

If you use the format that does not include the `allow_iso_moves` parameter, Spatial allows an edge move operation that would cause any isolated nodes or edges to be in a different face, and it adjusts the containing face information for such isolated nodes and edges.

Usage Notes

For information about moving edges, see Section 2.3.2.

This procedure is equivalent to using the `moveEdge` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example moves the edge with edge ID value 19, and it displays the edge coordinates before and after the move. The edge move operation moves the end point of the edge from the node with node ID value 17 to the node with node ID value 16. (The edge being moved is E19 in Figure 1–2 in Section 1.2; and the edge is being changed from going vertically up to node N17, to going diagonally up and left to node N16. The example refers to definitions and data from Section 1.12.1.)

```
-- Get coordinates of edge E19.
SELECT SDO_TOPO_MAP.GET_EDGE_COORDS(null, 'CITY_DATA_TOPOMAP', 19) FROM DUAL;

SDO_TOPO_MAP.GET_EDGE_COORDS(NULL,'CITY_DATA_TOPOMAP',19)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(21, 14, 21, 22)

-- Move edge E19: from N14 -> N17 to N14 -> N16. The 3rd and 4th parameters
-- identify N17 and N16.
CALL SDO_TOPO_MAP.MOVE_EDGE(null, 19, 17, 16,
  SDO_NUMBER_ARRAY(21,14, 9,22));

Call completed.

-- Get coordinates of edge E19 after the move.
SELECT SDO_TOPO_MAP.GET_EDGE_COORDS(null, 'CITY_DATA_TOPOMAP', 19) FROM DUAL;
```

```
SDO_TOPO_MAP.GET_EDGE_COORDS(NULL,'CITY_DATA_TOPOMAP',19)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(21, 14, 9, 22)
```

SDO_TOPO_MAP.MOVE_ISOLATED_NODE

Format

```
SDO_TOPO_MAP.MOVE_ISOLATED_NODE(
   topology    IN VARCHAR2,
   node_id     IN NUMBER,
   point       IN SDO_GEOMETRY);
```

or

```
SDO_TOPO_MAP.MOVE_ISOLATED_NODE(
   topology    IN VARCHAR2,
   node_id     IN NUMBER,
   x           IN NUMBER,
   y           IN NUMBER);
```

Description

Moves an isolated (island) node.

Parameters

topology
Name of the topology in which to move the node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

node_id
Node ID of the node to be moved.

point
SDO_GEOMETRY object (point geometry) representing the location to which the isolated node is to be moved.

x
X-axis value of the point representing the location to which the isolated node is to be moved.

y
Y-axis value of the point representing the location to which the isolated node is to be moved.

Usage Notes

For information about moving nodes, see Section 2.2.2.

The node must be moved to a location inside the face in which it is currently located. Otherwise, you must delete the node and re-create it.

This procedure is equivalent to using the `moveIsolatedNode` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example adds an isolated node and then moves it. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.ADD_ISOLATED_NODE(null, 2,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(22,38,NULL), NULL, NULL))
  INTO :res_number;

-- Move the just-added isolated node (from 20,38 to 22,39).
CALL SDO_TOPO_MAP.MOVE_ISOLATED_NODE( null, :res_number,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(22,39,NULL), NULL, NULL));
```

SDO_TOPO_MAP.MOVE_NODE

Format

```
SDO_TOPO_MAP.MOVE_NODE(
   topology          IN VARCHAR2,
   node_id           IN NUMBER,
   edges_coords      IN SDO_EDGE_ARRAY);
```

or

```
SDO_TOPO_MAP.MOVE_NODE(
   topology          IN VARCHAR2,
   node_id           IN NUMBER,
   edges_coords      IN SDO_EDGE_ARRAY,
   moved_iso_nodes   OUT SDO_NUMBER_ARRAY,
   moved_iso_edges   OUT SDO_NUMBER_ARRAY,
   allow_iso_moves   IN VARCHAR2);
```

Description

Moves a non-isolated node and its attached edges.

Parameters

topology
Name of the topology in which to move the node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

node_id
Node ID of the node to be moved.

edges_coords
An array of arrays, of type SDO_EDGE_ARRAY (described in Section 1.6.7). Each inner array consists of coordinates of each resulting attached edge, from start point to end point. The outer array consists of the attached edge arrays, starting with the start edge of the node to be moved and proceeding in clockwise order (with the sequence of the edges as would be obtained in a call to the SDO_TOPO_MAP.GET_NODE_STAR function).

The array for each edge must include the start and end points. Any loops that connect twice at the moved node must be specified twice in the array.

moved_iso_nodes
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the node ID values of any isolated nodes that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the node ID values of any isolated nodes that did not move but that would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

moved_iso_edges
Output parameter in which, if the `allow_iso_moves` parameter value is `TRUE`, Spatial stores the edge ID values of any isolated edges that have moved to a different face as a result of this procedure. If the `allow_iso_moves` parameter value is `FALSE`, Spatial stores the edge ID values of any isolated edges that did not move but that would have moved to a different face if the `allow_iso_moves` parameter value had been `TRUE`.

allow_iso_moves
`TRUE` causes Spatial to allow a node move operation that would cause any isolated nodes or edges to be in a different face, and to adjust the containing face information for such isolated nodes and edges; `FALSE` causes Spatial not to allow a node move operation that would cause any isolated nodes or edges to be in a different face.

If you use the format that does not include the `allow_iso_moves` parameter, Spatial allows a node move operation that would cause any isolated nodes or edges to be in a different face, and it adjusts the containing face information for such isolated nodes and edges.

Usage Notes

For information about moving nodes, see Section 2.2.2.

This procedure is equivalent to using the `moveNode` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example moves node N3 and adjusts the coordinates of the only attached edge. (The example refers to definitions and data from Section 1.12.1.)

```
-- Move node N3 to right: from 25,35 to 26,35.
-- E3 is changed from 25,30 -> 25,35 to 25,30 -> 26,35.
CALL SDO_TOPO_MAP.MOVE_NODE(null, 3,
  SDO_EDGE_ARRAY(SDO_NUMBER_ARRAY(25,30, 26,35)));
```

SDO_TOPO_MAP.REMOVE_EDGE

Format

```
SDO_TOPO_MAP.REMOVE_EDGE(
   topology     IN VARCHAR2,
   edge_id      IN NUMBER);
```

Description

Removes an edge from a topology.

Parameters

topology
Name of the topology from which to remove the edge, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

edge_id
Edge ID of the edge to be removed.

Usage Notes

If `topology` is not null, Spatial automatically updates the appropriate entries in the <topology-name>_EDGE$ and <topology-name>_FACE$ tables. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

For information about removing an edge from a topology, see Section 2.3.3.

Examples

The following example removes the edge with edge ID value 99 from the current updatable TopoMap object.

```
CALL SDO_TOPO_MAP.REMOVE_EDGE(null, 99);
```

SDO_TOPO_MAP.REMOVE_NODE

Format

```
SDO_TOPO_MAP.REMOVE_NODE(
   topology     IN VARCHAR2,
   node_id      IN NUMBER);
```

Description

Removes a node from a topology.

Parameters

topology
Name of the topology from which to remove the node, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

node_id
Node ID of the node to be removed.

Usage Notes

If `topology` is not null, Spatial automatically updates the appropriate entries in the <topology-name>_NODE$ and <topology-name>_EDGE$ tables, and in the <topology-name>_FACE$ table if necessary. (If `topology` is null, you can update these tables at any time by calling the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure.)

For information about removing a node from a topology, see Section 2.2.3.

Examples

The following example removes the node with node ID value 500 from the current updatable TopoMap object.

```
CALL SDO_TOPO_MAP.REMOVE_NODE(null, 500);
```

SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES

Format

```
SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES(
   topology     IN VARCHAR2);
```

Description

Removes obsolete nodes from a topology. (Obsolete nodes are explained in Section 2.2.4.)

Parameters

topology
Name of the topology from which to remove obsolete nodes, or null if you are using an updatable TopoMap object (see Section 2.1.2). Must not exceed 20 characters.

Usage Notes

For information about removing obsolete nodes from a topology, see Section 2.2.4.

Examples

The following example removes all obsolete nodes from the current updatable TopoMap object.

```
CALL SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES(null);
```

SDO_TOPO_MAP.ROLLBACK_TOPO_MAP

Format

SDO_TOPO_MAP.ROLLBACK_TOPO_MAP;

Description

Rolls back all changes to the database that were made using the current updatable TopoMap object, discards any changes in the object, clears the object's cache structure, and makes the object read-only.

Parameters

None.

Usage Notes

Use this procedure when you are finished with a batch of edits to a topology and you want to discard (that is, not commit) all changes to the database and in the cache. After the rollback operation completes, you cannot edit the TopoMap object. To make further edits to the topology, you can load the topology into the same TopoMap object for update (using the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure), or you can create a new TopoMap object (using the SDO_TOPO_MAP.CREATE_TOPO_MAP procedure) and load the topology into that TopoMap object for update.

To commit all TopoMap object changes, use the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

This procedure is equivalent to using the `rollbackDB` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example rolls back from the database all changes associated with the current updatable TopoMap object.

```
EXECUTE SDO_TOPO_MAP.ROLLBACK_TOPO_MAP;
```

SDO_TOPO_MAP.SEARCH_EDGE_RTREE_TOPO_MAP

Format

```
SDO_TOPO_MAP.SEARCH_EDGE_RTREE_TOPO_MAP(
   topo_map  IN VARCHAR2,
   xmin      IN NUMBER,
   ymin      IN NUMBER,
   xmax      IN NUMBER,
   ymax      IN NUMBER,
   capacity   IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array with the edge ID numbers of the edges that interact with a specified query window. The query uses the edge R-tree built on the specified TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

xmin
Lower-left X coordinate value for the query window.

ymin
Lower-left Y coordinate value for the query window.

xmax
Upper-right X coordinate value for the query window.

ymax
Upper-right Y coordinate value for the query window.

capacity
Maximum number of edge ID values to be returned. If you specify 0 or a negative number, 100 is used.

Usage Notes

This procedure is equivalent to using the `searchEdgeRTree` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the edge ID numbers (up to 200) of edge that interact with a query window whose lower-left corner is at (5,5) and upper-right corner is at (30,40). (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.SEARCH_EDGE_RTREE_TOPO_MAP('CITY_DATA_TOPOMAP', -
  5,5, 30,40, 200) FROM DUAL;
```

```
SDO_TOPO_MAP.SEARCH_EDGE_RTREE_TOPO_MAP('CITY_DATA_TOPOMAP',5,5,30,40,200)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(12, 13, 22, 20, 9, 21, 19, 6, 10, 7, 26, 3, 1, 25, 2)
```

SDO_TOPO_MAP.SEARCH_FACE_RTREE_TOPO_MAP

Format

```
SDO_TOPO_MAP.SEARCH_FACE_RTREE_TOPO_MAP(
   topo_map  IN VARCHAR2,
   xmin      IN NUMBER,
   ymin      IN NUMBER,
   xmax      IN NUMBER,
   ymax      IN NUMBER,
   capacity   IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array with the face ID numbers of the faces that interact with a specified query window. The query uses the face R-tree built on the specified TopoMap object.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

xmin
Lower-left X coordinate value for the query window.

ymin
Lower-left Y coordinate value for the query window.

xmax
Upper-right X coordinate value for the query window.

ymax
Upper-right Y coordinate value for the query window.

capacity
Maximum number of face ID values to be returned. If you specify 0 or a negative number, 100 is used.

Usage Notes

This procedure is equivalent to using the `searchFaceRTree` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example returns the face ID numbers (up to 200) of faces that interact with a query window whose lower-left corner is at (5,5) and upper-right corner is at (30,40). (The example refers to definitions and data from Section 1.12.1.)

```
SELECT SDO_TOPO_MAP.SEARCH_FACE_RTREE_TOPO_MAP('CITY_DATA_TOPOMAP', -
  5,5, 30,40, 200) FROM DUAL;
```

```
SDO_TOPO_MAP.SEARCH_FACE_RTREE_TOPO_MAP('CITY_DATA_TOPOMAP',5,5,30,40,200)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(6, 7, 3, 4, 9, 1, 2)
```

SDO_TOPO_MAP.SET_MAX_MEMORY_SIZE

Format

```
SDO_TOPO_MAP.SET_MAX_MEMORY_SIZE(
   maxsize     IN NUMBER DEFAULT 268435456);
```

Description

Sets the Java maximum heap size for an application to run in an Oracle Java virtual machine.

Parameters

maxsize
Number of bytes for the Java maximum heap size. The default value is 268435456 (256 MB).

Usage Notes

If you encounter the `java.lang.OutOfMemoryError` exception, you can use this procedure to increase the maximum heap size.

If you specify a value greater than the system limit, the system limit is used.

Examples

The following example sets the Java maximum heap size to 536870912 (512 MB).

```
EXECUTE SDO_TOPO_MAP.SET_MAX_MEMORY_SIZE(536870912);
```

SDO_TOPO_MAP.UPDATE_TOPO_MAP

Format

SDO_TOPO_MAP.UPDATE_TOPO_MAP;

Description

Updates the topology to reflect edits made to the current updatable TopoMap object.

Parameters

None.

Usage Notes

Use this procedure to update the topology periodically during an editing session, as explained in Section 2.1.4. Updates are made, as needed, to the edge, face, and node information tables (described in Section 1.5). The TopoMap object remains open for further editing operations. The updates are not actually committed to the database until you call the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure.

This procedure performs a level-0 validation of the TopoMap object before it updates the topology. (See the explanation of the `level` parameter for the SDO_TOPO_MAP.VALIDATE_TOPO_MAP function.)

If you caused in-memory R-tree indexes to be created when you loaded the TopoMap object (by specifying or accepting the default value of `TRUE` for the `build_indexes` parameter with the SDO_TOPO_MAP.LOAD_TOPO_MAP function or procedure), you can rebuild these indexes by using the SDO_TOPO_MAP.CREATE_EDGE_INDEX and SDO_TOPO_MAP.CREATE_FACE_INDEX procedures. For best index performance, these indexes should be rebuilt periodically when you are editing a large number of topological elements.

Contrast this procedure with the SDO_TOPO_MAP.CLEAR_TOPO_MAP procedure, which clears the cache associated with a specified TopoMap object and makes the object read-only.

To commit all TopoMap object changes, use the SDO_TOPO_MAP.COMMIT_TOPO_MAP procedure.

For information about using an in-memory cache to edit topological elements, see Section 2.1.

This procedure is equivalent to using the `updateTopology` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example updates the topology associated with the current updatable TopoMap object to reflect changes made to that object.

```
EXECUTE SDO_TOPO_MAP.UPDATE_TOPO_MAP;
```

SDO_TOPO_MAP.VALIDATE_TOPO_MAP

Format

```
SDO_TOPO_MAP.VALIDATE_TOPO_MAP(
   topo_map  IN VARCHAR2,
   level     IN NUMBER DEFAULT 1
   ) RETURN VARCHAR2;
```

Description

Performs a first-order validation of a TopoMap object, and optionally (by default) checks the computational geometry also; returns the string `TRUE` if the structure of the topological elements in TopoMap object is consistent, and raises an exception if the structure of the topological elements in TopoMap object is not consistent.

Parameters

topo_map
Name of the TopoMap object. (TopoMap objects are explained in Section 2.1.1.)

level
A value of 0 checks for the following conditions as part of a first-order validation:

- All faces are closed, and none have infinite loops.
- All previous and next edge pointers are consistent.
- All edges meet at nodes.
- Each island node is associated with a face.
- All edges on a face boundary are associated with the face.

A value of 1 (the default) checks for all conditions associated with a value of 0, plus the following conditions related to computational geometry:

- Each island is inside the boundary of its associated face.
- No edge intersects itself or another edge.
- Start and end coordinates of edges match coordinates of nodes.
- Node stars are properly ordered geometrically.

Usage Notes

This function checks the consistency of all pointer relationships among edges, nodes, and faces. You can use this function to validate an updatable TopoMap object before you update the topology (using the SDO_TOPO_MAP.UPDATE_TOPO_MAP procedure) or to validate a read-only TopoMap object before issuing queries.

This function uses a tolerance value of 10E-15 for its internal computations, as explained in Section 1.2.1.

This function is equivalent to using the `validateCache` method of the `TopoMap` class of the client-side Java API (described in Section 1.8.2).

Examples

The following example validates the topology in the TopoMap object named CITY_DATA_TOPOMAP, and it returns a result indicating that the topology is valid. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.VALIDATE_TOPO_MAP('CITY_DATA_TOPOMAP') INTO :res_varchar;

Call completed.

PRINT res_varchar;

RES_VARCHAR
--------------------------------------------------------------------------------
TRUE
```

SDO_TOPO_MAP.VALIDATE_TOPOLOGY

Format

```
SDO_TOPO_MAP.VALIDATE_TOPOLOGY(
   topology  IN VARCHAR2,
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO_MAP.VALIDATE_TOPOLOGY(
   topology          IN VARCHAR2,
   prevent_updates   IN VARCHAR2,
   level             IN NUMBER DEFAULT 1
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO_MAP.VALIDATE_TOPOLOGY(
   topology          IN VARCHAR2,
   xmin              IN NUMBER,
   ymin              IN NUMBER,
   xmax              IN NUMBER,
   ymax              IN NUMBER
   ) RETURN VARCHAR2;
```

or

```
SDO_TOPO_MAP.VALIDATE_TOPOLOGY(
   topology          IN VARCHAR2,
   xmin              IN NUMBER,
   ymin              IN NUMBER,
   xmax              IN NUMBER,
   ymax              IN NUMBER,
   prevent_updates   IN VARCHAR2,
   level             IN NUMBER DEFAULT 1
   ) RETURN VARCHAR2;
```

Description

Loads an entire topology or a window (rectangular portion) of a topology into a TopoMap object; returns the string `TRUE` if the structure of the topology is consistent, and raises an exception if the structure of the topology is not consistent.

Parameters

topology
Name of the topology to be validated. Must not exceed 20 characters.

xmin
Lower-left X coordinate value for the window (rectangular portion of the topology) to be validated.

ymin
Lower-left Y coordinate value for the window (rectangular portion of the topology) to be validated.

xmax
Upper-right X coordinate value for the window (rectangular portion of the topology) to be validated.

ymax
Upper-right Y coordinate value for the window (rectangular portion of the topology) to be validated.

prevent_updates
`TRUE` prevents other users from updating the topology while the validation is being performed; `FALSE` allows other users to update the topology while the validation is being performed. If you specify `FALSE`, any topology changes made by other users while the validation is being performed will not be considered by this function and will not affect the result.

level
A value of 0 checks for the following conditions:

- All faces are closed, and none have infinite loops.
- All previous and next edge pointers are consistent.
- All edges meet at nodes.
- Each island node is associated with a face.
- All edges on a face boundary are associated with the face.

A value of 1 (the default) checks for all conditions associated with a value of 0, plus the following conditions related to computational geometry:

- Each island is inside the boundary of its associated face.
- No edge intersects itself or another edge.
- Start and end coordinates of edges match coordinates of nodes.
- Node stars are properly ordered geometrically.

Usage Notes

This function implicitly creates a TopoMap object, and removes the object after the validation is complete. (TopoMap objects are described in Section 2.1.1.)

This function uses a tolerance value of 10E-15 for its internal computations, as explained in Section 1.2.1.

Examples

The following example validates the topology named CITY_DATA, and it returns a result indicating that the topology is valid. (The example refers to definitions and data from Section 1.12.1.)

```
CALL SDO_TOPO_MAP.VALIDATE_TOPOLOGY('CITY_DATA') INTO :res_varchar;

Call completed.

PRINT res_varchar;

RES_VARCHAR
--------------------------------------------------------------------------------
TRUE
```

Part II

Network Data Model

This document has two main parts:

- Part I provides conceptual, usage, and reference information about the topology data model of Oracle Spatial.
- Part II provides conceptual, usage, and reference information about the network data model of Oracle Spatial.

Part II contains the following chapters:

5 Network Data Model Overview

This chapter explains the concepts and operations related to the Oracle Spatial network data model. It assumes that you are familiar with the main Oracle Spatial concepts, data types, and operations, as documented in *Oracle Spatial Developer's Guide*:

Although this chapter discusses some network-related terms as they relate to Oracle Spatial, it assumes that you are familiar with basic network data modeling concepts.

This chapter contains the following major sections:

5.1 Introduction to Network Modeling

In many applications, capabilities or objects are modeled as nodes and links in a network. The network model contains logical information such as connectivity relationships among nodes and links, directions of links, and costs of nodes and links. With logical network information, you can analyze a network and answer questions, many of them related to path computing and tracing. For example, for a biochemical pathway, you can find all possible reaction paths between two chemical compounds; or for a road network, you can find the following information:

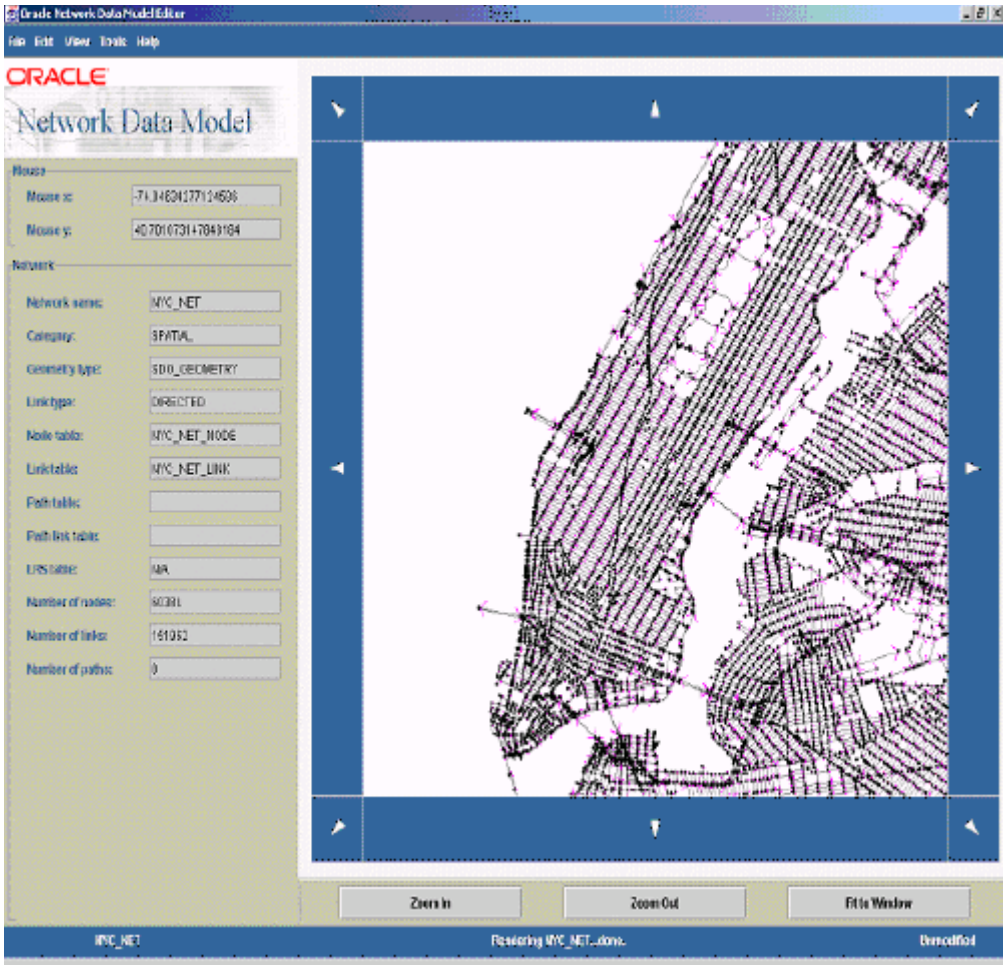
- What is the shortest (distance) or fastest (travel time) path between two cities?

- What is the closest hotel to a specific airport, and how can I get there?

In additional to logical network information, spatial information such as node locations and link geometries can be associated with the network. This information can help you to model the logical information (such as the cost of a route, because its physical length can be directly computed from its spatial representation).

The Spatial network data model can be used for large, complex networks. For example, Figure 5–1 shows New York City nodes and links, which have been defined using the network data model, displayed using the Network Editor demo tool (described in Section 5.14).

Figure 5–1 New York City Nodes and Links



The generic data model and network analysis capability can model and analyze many kinds of network applications in addition to traditional geographical information systems (GIS). For example, in biochemistry, applications may need to model reaction pathway networks for living organisms; and in the pharmaceutical industry, applications that model the drug discovery process may need to model protein-protein interaction.

The network modeling capabilities of Spatial include schema objects and an application programming interface (API). The schema objects include metadata and network tables. The API includes a server-side PL/SQL API (the SDO_NET and SDO_

NET_MEM packages) for creating, managing, editing, and analyzing networks in the database, and a middle-tier (or client-side) Java API for network editing and analysis.

5.2 Main Steps in Using the Network Data Model

This section summarizes the main steps for working with the network data model in Oracle Spatial. It refers to important concepts, structures, and operations that are described in detail in other sections.

There are two basic approaches to creating a network:

- Let Spatial perform most operations, using procedures with names in the form CREATE_<*network-type*>_NETWORK. (See Section 5.2.1.)
- Perform the operations yourself: create the necessary network tables and update the network metadata. (See Section 5.2.2.)

With each approach, you must insert the network data into the network tables. You can then use the network data model PL/SQL and Java application programming interfaces (APIs) to update the network and perform other operations. (The PL/SQL and Java APIs are described in Section 5.11.)

5.2.1 Letting Spatial Perform Most Operations

To create a network by letting Spatial perform most of the necessary operations, follow these steps:

1. Create the network using a procedure with a name in the form CREATE_<*network-type*>_NETWORK, where <*network-type*> reflects the type of network that you want to create:
   - SDO_NET.CREATE_SDO_NETWORK for a spatial network with non-LRS SDO_GEOMETRY objects
   - SDO_NET.CREATE_LRS_NETWORK for a spatial network with LRS SDO_GEOMETRY objects
   - SDO_NET.CREATE_TOPO_NETWORK for a spatial network with topology geometry (SDO_TOPO_GEOMETRY) objects
   - SDO_NET.CREATE_LOGICAL_NETWORK for a logical network that does not contain spatial information

   Each of these procedures creates the necessary network data model tables (described in Section 5.9) and inserts a row with the appropriate network metadata information into the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

   Each procedure has two formats: one format creates all network data model tables using default names for the tables and certain columns, and other format lets you specify names for the tables and certain columns. The default names for the network data model tables are <*network-name*>_NODE$, <*network-name*>_LINK$, <*network-name*>_PATH$, and <*network-name*>_PLINK$. The default name for cost columns in the network data model tables is COST, and the default name for geometry columns is GEOMETRY.
2. Insert data into the node and link tables, and if necessary into the path and path-link tables. (The node, link, path, and path-link tables are described in Section 5.9.)
3. Validate the network, using the SDO_NET.VALIDATE_NETWORK function.

4. For a spatial (SDO or LRS) network, insert the appropriate information into the USER_SDO_GEOM_METADATA view, and create spatial indexes on the geometry columns.

   If you plan to use a view as a node, link, or path table, you must specify the view name for the TABLE_NAME column value when you insert information about the node, link, or path table in the USER_SDO_GEOM_METADATA view.

5.2.2 Performing the Operations Yourself

To create a network by performing the necessary operations yourself, follow these steps:

1. Create the node table, using the SDO_NET.CREATE_NODE_TABLE procedure. (The node table is described in Section 5.9.1.)
2. Insert data into the node table.
3. Create the link table, using the SDO_NET.CREATE_LINK_TABLE procedure. (The link table is described in Section 5.9.2).
4. Insert data into the link table.
5. Optionally, create the path table, using the SDO_NET.CREATE_PATH_TABLE procedure. (The path table is described in Section 5.9.3).
6. If you created the path table, create the path-link table, using the SDO_NET.CREATE_PATH_LINK_TABLE procedure. (The path-link table is described in Section 5.9.4).
7. If you created the path table and if you want to create paths, insert data into the table.
8. If you inserted data into the path table, insert the appropriate rows into the path-link table.
9. Insert a row into the USER_SDO_NETWORK_METADATA view with information about the network. (The USER_SDO_NETWORK_METADATA view is described in Section 5.10.1.)

   If you plan to use a view as a node, link, path, or path-link table, you must specify the view name for the relevant columns when you insert information about the network in the USER_SDO_NETWORK_METADATA view.
10. For a spatial (SDO or LRS) network, insert the appropriate information into the USER_SDO_GEOM_METADATA view, and create spatial indexes on the geometry columns.

    If you plan to use a view as a node, link, or path table, you must specify the view name for the TABLE_NAME column value when you insert information about the node, link, or path table in the USER_SDO_GEOM_METADATA view.
11. Validate the network, using the SDO_NET.VALIDATE_NETWORK function.

You can change the sequence of some of these steps. For example, you can create both the node and link tables first, and then insert data into each one; and you can insert the row into the USER_SDO_NETWORK_METADATA view before you create the node and link tables.

5.3 Network Data Model Concepts

A network is a type of mathematical graph that captures relationships between objects using connectivity. The connectivity may or may not be based on spatial proximity. For example, if two towns are on opposite sides of a lake, the shortest path based on spatial proximity (a straight line across the middle of the lake) is not relevant if you want to drive from one town to the other. Instead, to find the shortest driving distance, you need connectivity information about roads and intersections and about the "cost" of individual links.

A network consists of a set of nodes and links. Each link (sometimes also called an edge or a segment) specifies two nodes.

A network can be **directed** (that is, by default, the start and end nodes determine link direction) or **undirected** (that is, links can be traversed in either direction).

The following are some key terms related to the network data model:

- A **node** represents an object of interest. An **isolated node** is a node that is not included in any links. (A non-isolated node will become isolated if all links that include that node are deleted.)
- A **link** represents a relationship between two nodes. Within a directed network, any link can be **undirected** (that is, able to be traversed either from the start node to the end node or from the end node to the start node) or **directed** (that is, able to be traversed only from the start node to the end node). Within an undirected network, all links are undirected.
- A **path** is an alternating sequence of nodes and links, beginning and ending with nodes, and usually with no nodes and links appearing more than once. (Repeating nodes and links within a path are permitted, but are rare in most network applications.)
- A **subpath** is partial path along a path, created either as a result of a network analysis operation or explicitly by a user. Subpaths are explained and illustrated in Section 5.3.1.
- A **logical network** contains connectivity information but no geometric information. This is the model used for network analysis. A logical network can be treated as a directed graph or undirected graph, depending on the application.
- A **spatial network** contains both connectivity information and geometric information. In a spatial network, the nodes and links are SDO_GEOMETRY geometry objects without LRS information (an **SDO network**) or with LRS information (an **LRS network**), or SDO_TOPO_GEOMETRY objects (a **topology geometry network**).

  In an LRS network, each node includes a geometry ID value and a measure value, and each link includes a geometry ID value and start and end measure values; and the geometry ID value in each case refers to an SDO_GEOMETRY object with LRS information. A spatial network can be directed or undirected, depending on the application.
- A **feature** is an object of interest in a network application that is associated with a node or link. For example, in a transportation network, features include exits and intersections (mapped to nodes), and highways and streets (mapped to links).
- **Cost** is a non-negative numeric attribute that can be associated with links or nodes for computing the **minimum cost path**, which is the path that has the minimum total cost from a start node to an end node. You can specify a single cost factor,

such as driving time or driving distance for links, in the network metadata, and network analytical functions that examine cost will use this specified cost factor.

- **Duration** is a non-negative numeric attribute that can be associated with links or nodes to specify a duration value for the link or node. The duration value can indicate a number of minutes or any other user-determined significance. You can specify a single duration factor, such as driving time for links, in the network metadata; however, if you use duration instead of cost to indicate elapsed time, network analytical functions that examine cost will not consider the specified duration factor.
- **State** is a string attribute, either `ACTIVE` or `INACTIVE`, that is associated with links or nodes to specify whether or not a link or node will be considered by network analysis functions. For example, if the state of a node is `INACTIVE`, any links from or to that node are ignored in the computation of the shortest path between two nodes. The state is `ACTIVE` by default when a link or node is created, but you can set the state `INACTIVE`.
- **Type** is a string attribute that can be associated with links or nodes to specify a user-defined value for the type of a link or a node.
- **Temporary** links, nodes, and paths exist only in a network memory object, and are not written to the database when the network memory object is written. For example, during a network analysis and editing session you might create temporary nodes to represent street addresses for shortest-path computations, but not save these temporary nodes when you save the results of editing operations.
- **Reachable nodes** are all nodes that can be reached from a given node. **Reaching nodes** are all nodes that can reach a given node.
- The **degree** of a node is the number of links to (that is, incident upon) the node. The **in-degree** is the number of inbound links, and the **out-degree** is the number of outbound links.
- A **connected component** is a group of network nodes that are directly or indirectly connected. If node A can reach node B, they must belong to the same connected component. If two nodes are not connected, it is concluded that there is no possible path between them. This information can be used as a filter to avoid unnecessary path computations.
- A **spanning tree** of a connected graph is a tree (that is, a graph with no cycles) that connects all nodes of the graph. (The directions of links are ignored in a spanning tree.) The **minimum cost spanning tree** is the spanning tree that connects all nodes and has the minimum total cost.
- A **partitioned network** is a network that contains multiple partitions. Partitioning a large network enables only the necessary partitions to be loaded on demand into memory, thus providing better overall performance.

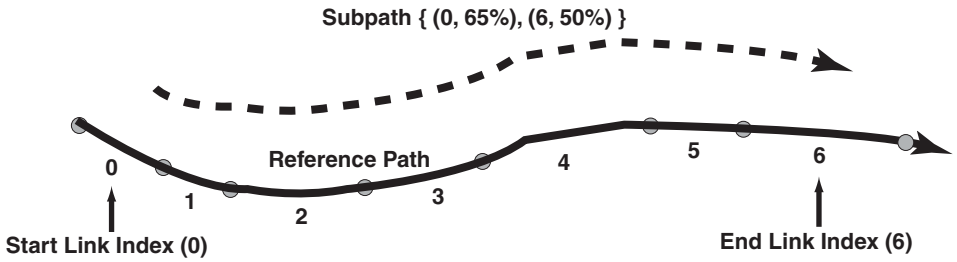
  Network partitions are sub-networks, each covering a subset of nodes and links of the entire network. Network partitions are the basic processing units for load on demand analysis. They are created by assigning every node in the network to only one partition ID. Network partition information is stored in a partition table.
- **Load on demand** (load on demand analysis) is an approach that divides large networks into manageable partitions and only loads needed partitions during analysis, thus removing memory limitation as a consideration and providing better overall performance.
- **Partition BLOBs** are binary representations for network partitions. They provide faster partition loading time. They are stored in a partition BLOB table.

- The load on demand **partition cache** is an in-memory placeholder for network partitions loaded into memory during network analysis. You can configure the partition cache.
- **User-defined data** is the information (not related to connectivity) that users want to associate with a network representation. User-defined data can be defined at the node, link, path, and subpath level, and is stored in columns in the node, link, path, and subpath tables.

5.3.1 Subpaths

A **subpath** is partial path along a path, created either as a result of a network analysis operation or explicitly by a user. The start and end points of a subpath are defined as link indexes and the percentage of the distance from the previous node in the path, as shown in Figure 5–2.

Figure 5–2 Path and Subpaths



A subpath refers to an existing path (the **reference path**) using the following parameters:

- Reference path ID: the path ID of the reference path.
- Start link index: the start link index on the reference path. (Link index 0 refers to the link between the first and second nodes on the path.) In Figure 5–2, link index 0 is the start link index.
- Start percentage: the percentage of the distance along the start link for the start node of the subpath. In Figure 5–2, the subpath starts at 65 percent of the distance between the start and end of link index 0.
- End link index: the end link index on the reference path. In Figure 5–2, link index 6 is the end link index.
- End percentage: the percentage of the distance along the end link for the end node of the subpath. In Figure 5–2, the subpath ends at 50 percent of the distance between the start and end of link index 6.

5.4 Network Applications

Networks are used in applications to find how different objects are connected to each other. The connectivity is often expressed in terms of adjacency and path relationships. Two nodes are adjacent if they are connected by a link. There are often several paths between any two given nodes, and you may want to find the path with the minimum cost.

This section describes some typical examples of different kinds of network applications.

5.4.1 Road Network Example

In a typical road network, the intersections of roads are nodes and the road segments between two intersections are links. The spatial representation of a road is not inherently related to the nodes and links in the network. For example, a shape point in the spatial representation of a road (reflecting a sharp turn in the road) is not a node in the network if that shape point is not associated with an intersection; and a single spatial object may make up several links in a network (such as a straight segment intersected by three crossing roads). An important operation with a road network is to find the path from a start point to an end point, minimizing either the travel time or distance. There may be additional constraints on the path computation, such as having the path go through a particular landmark or avoid a particular intersection.

5.4.2 Train (Subway) Network Example

The subway network of any major city is probably best modeled as a logical network, assuming that precise spatial representation of the stops and track lines is unimportant. In such a network, all stops on the system constitute the nodes of the network, and a link is the connection between two stops if a train travels directly between these two stops. Important operations with a train network include finding all stations that can be reached from a specified station, finding the number of stops between two specified stations, and finding the travel time between two stations.

5.4.3 Utility Network Example

Utility networks, such as power line or cable networks, must often be configured to minimize the cost. An important operation with a utility network is to determine the connections among nodes, using minimum cost spanning tree algorithms, to provide the required quality of service at the minimum cost. Another important operation is reachability analysis, so that, for example, if a station in a water network is shut down, you know which areas will be affected.

5.4.4 Biochemical Network Example

Biochemical processes can be modeled as biochemical networks to represent reactions and regulations in living organisms. For example, metabolic pathways are networks involved in enzymatic reactions, while regulatory pathways represent protein-protein interactions. In this example, a pathway is a network; genes, proteins, and chemical compounds are nodes; and reactions among nodes are links. Important operations for a biochemical network include computing paths and the degrees of nodes.

5.5 Network Hierarchy

Some network applications require representations at different levels of abstraction. For example, two major processes might be represented as nodes with a link between them at the highest level of abstraction, and each major process might have several subordinate processes that are represented as nodes and links at the next level down.

A **network hierarchy** enables you to represent a network with multiple levels of abstraction by assigning a hierarchy level to each node. (Links are not assigned a hierarchy level, and links can be between nodes in the same hierarchy level or in different levels.) The lowest (most detailed) level in the hierarchy is level 1, and successive higher levels are numbered 2, 3, and so on.

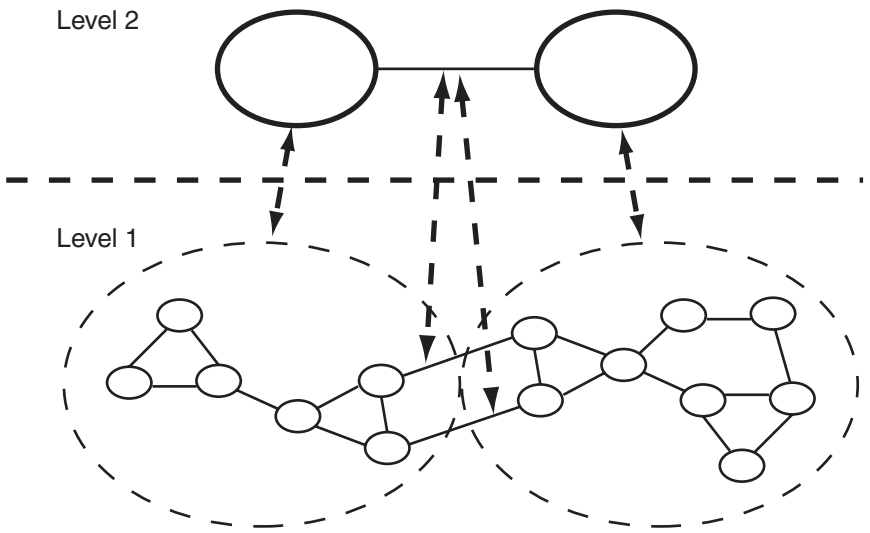
Nodes at adjacent levels of a network hierarchy have parent-child relationships. Each node at the higher level can be the **parent node** for one or more nodes at the lower

level. Each node at the lower level can be a **child node** of one node at the higher level. **Sibling nodes** are nodes that have the same parent node.

Links can also have parent-child relationships. However, because links are not assigned to a hierarchy level, there is not necessarily a relationship between link parent-child relationships and network hierarchy levels. **Sibling links** are links that have the same parent link.

Figure 5–3 shows a simple hierarchical network, in which there are two levels.

Figure 5–3 Network Hierarchy



As shown in Figure 5–3:

- The top level (level 2) contains two nodes. Each node is the parent node of several nodes in the bottom level. The link between the nodes in the top level is the parent link of two links between nodes in the bottom level.
- The bottom level (level 1) shows the nodes that make up each node in the top level. It also shows the links between nodes that are child nodes of each parent node in the top level, and two links between nodes that have different parent nodes.
- The links between nodes in the bottom level that have different parent nodes are shown with dark connecting lines. These links are child links of the single link between the nodes in the top level in the hierarchy. (However, these two links in the bottom level could also be defined as not being child links of any parent link between nodes in a higher level.)
- The parent-child relationships between each parent node and link and its child nodes and links are shown with dashed lines with arrowheads at both ends.

Although it is not shown in Figure 5–3, links can cross hierarchy levels. For example, a link could be defined between a node in the top level and any node in the bottom level. In this case, there would not be a parent-child relationship between the links.

5.6 Network Constraints

Network constraints are restrictions defined on network analysis computations. For example, a network constraint might list a series of prohibited turns in a roads

network due to one-way streets and "No Left Turn" signs, with each prohibited turn represented as a pair of links (a start link and an end link onto which a turn cannot be made from the start link). As another example, a network constraint might require that driving routes must not include toll roads or must not include expressways.

To create a network constraint, you must create a Java class that implements the constraint, and you must register the constraint by using the SDO_NET.REGISTER_CONSTRAINT procedure. To apply a network constraint to a network analysis operation, specify the constraint using the `constraint` parameter with the appropriate SDO_NET_MEM subprogram.

Examples of Java classes to implement network constraints are provided in the network data model demo files, which are described in Section 5.14. For example, the `ProhibitedTurns.java` file creates a network constraint that defines a series of prohibited turns, and it then returns the shortest path between two nodes, first without applying the constraint and then applying the constraint.

5.7 Network Analysis Using Load on Demand

Load on demand means that during network analysis, a network partition is not loaded into memory until the analysis has reached this partition while exploring the network. With load on demand, Oracle Spatial performs most partitioning and loading operations automatically, and this usually results in more efficient memory utilization with very large networks.

Load on demand analysis involves the following major steps: network creation, network partition, partition cache configuration, and network analysis.

1. Create the network, using one of the approaches described in Section 5.2.
2. Partition the network using the SDO_NET.SPATIAL_PARTITION procedure, as explained in Section 5.7.1.
3. Optionally, generate partition BLOBs, as explained in Section 5.7.2.
4. Configure the load on demand environment, including the partition cache, as explained in Section 5.7.3.
5. Analyze the network, as explained in Section 5.7.4.

> **Note:** Load on demand analysis also works with nonpartitioned networks by treating the entire network as one partition. For a small network, there may be no performance benefit in partitioning it, in which case you can skip the partitioning but still use load on demand APIs.

For examples of performing load on demand network analysis and configuring the partition cache, see Section 5.13.5.

Additional examples of partitioning and load on demand analysis are included on the Oracle Database Examples media (see *Oracle Database Examples Installation Guide*). For more information about network data model example and demo files, see Section 5.14.

5.7.1 Partitioning a Network

You can partition a network using the SDO_NET.SPATIAL_PARTITION procedure, specifying the maximum number of nodes in each partition. The partition result is stored in a partition table, which is automatically generated, and partition metadata

information is inserted into the network metadata. (As an alternative to using the procedure, you can partition a network by creating and populating a partition table.) You can use other SDO_NET subprograms to query the partitioning metadata.

A good partition strategy is to minimize the number of links between partitions, which reduces the number of partitions that need to be loaded and the probable number of times that the same partitions need to be reloaded. Moreover, partitions that are too small require excessive loading and unloading of partitions during analysis.

The recommended maximum number of nodes per partition, assuming 1 GB of memory, is between 5,000 and 10,000. You can tune the number and see what is best for your applications, considering the available memory, type of analysis, and network size. You should also consider configuring the partition caching size.

5.7.2 Generating Partition BLOBs

To enhance the performance of network loading, you can optionally store partitions as BLOBs in a network partition BLOB table. This information needs to be stored in the network metadata view in order to take advantage of faster partition loading time. Note that if a network or partition information is updated, the partition BLOBs need to be regenerated as well.

A **partition BLOB** is a binary stream of data containing the network partition information, such as number of nodes, number of links, properties of each node, properties of each link, and so on. If a partition BLOB exists, Spatial uses it to read information during the load operation, rather than performing time-consuming database queries.

To generate partition BLOBs, use the SDO_NET.GENERATE_PARTITION_BLOBS procedure. The partition BLOBs and their metadata are stored in the partition BLOB table, which is described in Section 5.9.7.

5.7.3 Configuring the Partition Cache

Before you perform network analysis, you can configure the network partition cache to optimize performance, by modifying an XML configuration file to override the default configuration. You can specify the following:

- Cache size: the maximum number of nodes in partition cache
- Partitions source: from network tables or partition BLOBs
- Resident partitions: IDs of partitions that will not be flushed out of the cache, but will stay in memory once they are loaded
- Cache flushing policy: class name of the `CachingHandler` implementation

  The default caching policy is `LeastRecentlyUsed`, which flushes out the oldest partition out of memory when the cache is full. You can specify other caching policies by implementing the `CachingHandler` interface.

A copy of the default load on demand configuration file is included in the supplementary documentation, described in Section 5.14.

5.7.4 Analyzing the Network

After you have created and partitioned the network, and optionally configured the partition cache, you can issue analysis queries. Analysis results are returned in Java representation or XML responses, depending on whether you used the Java or XML

API. For details, see the load on demand (LOD) Javadoc and XML schemas (the latter described in Section 5.14).

You can write the analysis results to the database using the load on demand Java API.

5.7.5 Using Link Levels for Priority Modeling

Although the load on demand approach reduces the effect of memory limitations in analyzing large networks, analysis operations still can sometimes be very slow. For example, shortest path analysis of two nodes diagonally across the entire network is likely to require traversing almost every link in the network, and this will take a significant amount of time for a network with more than, for example, two million nodes.

To further reduce network analysis time, you can perform analysis on different link levels. **Link level** is a positive integer assigned to a link indicating the level of preference of this link. The higher the link level, the higher the preference. For example, a road network may consist of two link levels, level 1 for local roads and level 2 for highways. During network analysis, highways are preferred to local roads, and the minimum link level is 1. (If no link level is assigned to a link, the default link level of 1 is used for the link.)

Link levels have an implicit inheritance property, which means that a network at higher link levels must be a subnetwork of a network at a lower link level. That is, link level 2 is a subnetwork of link level 1, link level 3 is a subnetwork of link level 2, and so on.

You can specify a link level when you load a network or a partition, which causes links at that level and higher levels to be loaded. Using the road network example, with link level 1 for local roads and link level 2 for highways, specifying link level 1 on a load operation loads links at link levels 1 and 2 (that is, local roads and highways), but specifying link level 2 on a load operation loads only the highways links. If you wanted to perform analysis using only highways links, you could optimize the performance by specifying link level 2 for the load operation.

5.7.6 Precomputed Analysis Results

Some analysis operations, such as connected component analysis, can be time consuming. To improve run-time performance, you can call the SDO_NET.FIND_CONNECTED_COMPONENTS procedure, which computes the connected components in the network and stores the results in the connected component table, which is described Section 5.9.8.

At run time, before calling shortest path analysis or reachability analysis, you can check whether the nodes of interest belong to the same connected component by querying the connected component table. If precomputed component information does not exist, it may take a long time for shortest path and reachability analysis to discover that two nodes are, in fact, not connected.

5.8 Network Editing and Analysis Using the In-Memory Approach

Note: The in-memory approach will be deprecated in the next release of Spatial, and future development will enhance the load on demand approach, which is described in Section 5.7. You are encouraged use the load on demand approach whenever possible.

This section describes how to perform network editing and analysis operations using a **network memory object**, which is a cache in virtual memory. You can load a network or a hierarchy level in a network into a network memory object, perform operations on network objects in the memory object, and then either discard any changes or write the changes to the network in the database.

Multiple network memory objects can exist at a time for a specified network, but only one can be updatable; any others must be read-only. For better performance, if you plan to use the network memory object only to retrieve information or to perform network analysis operations, make the network memory object read-only (that is, specify `allow_updates=>'FALSE'` with the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure).

To work with a network memory object, you can use either the PL/SQL API (specifically, the SDO_NET_MEM package) or the Java API. Both APIs are introduced in Section 5.11.

In the network data model PL/SQL API, the subprograms in the SDO_NET package operate on the network in the database, and the subprograms in the SDO_NET_MEM package operate on the network memory object in the cache. For some network editing operations (such as adding a node, link, or path), you can use either an SDO_NET or SDO_NET_MEM procedure; however, if you are performing a large number of editing operations, using the cache (SDO_NET_MEM procedures) offers better performance. Most network operations, though, can be performed only by a subprogram in the SDO_NET or SDO_NET_MEM package, and in these cases your decision about whether to use a network memory object depends on your specific needs.

Example 5–1 uses a network memory object to add a new node and a new link to an existing network, perform a shortest path analysis, print the analysis results, and save the changes and analysis results in the database. These steps assume that a logical network named XYZ_NETWORK has already been created and populated using the statements shown in Example 5–5 in Section 5.13.4.

Example 5–1 Using a Network Memory Object for Editing and Analysis (PL/SQL)

```
DECLARE
  path_id     NUMBER;
  res_numeric NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN
-- Create a network memory object in the user session for the
-- logical network named XYZ_NETWORK. This creates a network
-- object and reads all metadata, nodes, links, and paths in
-- the network, and it allows for updates to be performed.
sdo_net_mem.network_manager.read_network(net_mem=>'XYZ_NETWORK',
  allow_updates=>'TRUE');

-- Add a node with ID=901, and set its name to N901 and cost to 5.
sdo_net_mem.network.add_node(net_mem=>'XYZ_NETWORK', node_id=>901,
  node_name=>'N901', external_network_id=>0, external_node_id=>0);
sdo_net_mem.node.set_cost(net_mem=>'XYZ_NETWORK', node_id=>901, cost=>5);

-- Add a link with ID=9901, name=N901N1, cost=20 from node N901 to node N1.
sdo_net_mem.network.add_link(net_mem=>'XYZ_NETWORK', link_id=>9901,
  link_name=>'N901N1', start_node_id=>901, end_node_id=>101, cost=>20);

-- Perform a shortest path analysis from node N1 to node N5.
path_id := sdo_net_mem.network_manager.shortest_path('XYZ_NETWORK', 101, 105);
```

```
DBMS_OUTPUT.PUT_LINE('The ID of the shortest path from N1 to N5 is: ' || path_id);

-- List the properties of the path: cost, nodes, and links.
res_numeric := sdo_net_mem.path.get_cost('XYZ_NETWORK', path_id);
DBMS_OUTPUT.PUT_LINE('The cost of this path is: ' || res_numeric);
res_array:= sdo_net_mem.path.get_node_ids('XYZ_NETWORK', path_id);
DBMS_OUTPUT.PUT('This path has the following nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
res_array:= sdo_net_mem.path.get_link_ids('XYZ_NETWORK', path_id);
DBMS_OUTPUT.PUT('This path has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- Add the path to the network memory object.
sdo_net_mem.network.add_path(net_mem=>'XYZ_NETWORK', path_id=>path_id);

-- Write changes to the database and commit changes.
sdo_net_mem.network_manager.write_network(net_mem=>'XYZ_NETWORK');

-- Drop the network memory object.
sdo_net_mem.network_manager.drop_network(net_mem=>'XYZ_NETWORK');

END;
/
The ID of the shortest path from N1 to N5 is: 1
The cost of this path is: 50
This path has the following nodes: 101 103 104 105
This path has the following links: 1102 1104 1105
```

5.9 Network Data Model Tables

The connectivity information for a spatial network is stored in two tables: a node table and a link table. In addition, path information can be stored in a path table and a path-link table. You can have Spatial create these tables automatically when you create the network using a CREATE_<*network-type*>_NETWORK procedure; or you can create these tables using the SDO_NET.CREATE_NODE_TABLE, SDO_NET.CREATE_LINK_TABLE, SDO_NET.CREATE_PATH_TABLE, and SDO_NET.CREATE_PATH_LINK_TABLE procedures.

These tables contain columns with predefined names, and you must not change any of the predefined column names; however, you can add columns to the tables by using the ALTER TABLE statement with the ADD COLUMN clause. For example, although each link and path table is created with a single COST column, you can create additional columns and associate them with other comparable attributes. Thus, to assign a driving time, scenic appeal rating, and a danger rating to each link, you could use the COST column for driving time, add columns for SCENIC_APPEAL and DANGER to the link table, and populate all three columns with values to be interpreted by applications.

The following considerations apply to schema, table, and column names that are stored in any Oracle Spatial metadata views. For example, these considerations apply

to the names of node, link, path, and path-link tables, and to the names of any columns in these tables that are stored in the network metadata views described in Section 5.10.

- The name must contain only letters, numbers, and underscores. For example, the name cannot contain a space ( ), an apostrophe ('), a quotation mark ("), or a comma (,).
- All letters in the names are converted to uppercase before the names are stored in metadata views or before the tables are accessed. This conversion also applies to any schema name specified with the table name.

5.9.1 Node Table

Each network has a node table that can contain the columns described in Table 5–1. (The specific columns depend on the network type and whether the network is hierarchical or not.)

Table 5–1 Node Table Columns

Column Name	Data Type	Description
NODE_ID	NUMBER	ID number that uniquely identifies this node within the network
NODE_NAME	VARCHAR2(32)	Name of the node
NODE_TYPE	VARCHAR2(24)	User-defined string to identify the node type
ACTIVE	VARCHAR2(1)	Contains `Y` if the node is active (visible in the network), or `N` if the node is not active.
PARTITION_ID	NUMBER	(Not used. Instead, node and partition relationships are stored in the partition table, which is described in Section 5.9.6.)
<*node_geometry_column*>, or GEOM_ID and MEASURE	SDO_GEOMETRY, or SDO_TOPO_GEOMETRY, or NUMBER	For a spatial (SDO, non-LRS) network, the SDO_GEOMETRY object associated with the node For a spatial topology network, the SDO_TOPO_GEOMETRY object associated with the node For a spatial LRS network, GEOM_ID and MEASURE column values (both of type NUMBER) for the geometry objects associated with the node For a logical network, this column is not used. For a spatial SDO or topology network, the actual column name is either a default name or what you specified as the `geom_column` parameter value in the call to the SDO_NET.CREATE_NODE_TABLE procedure.
<*node_cost_column*>	NUMBER	Cost value to be associated with the node, for use by applications that use the network. The actual column name is either a default name or what you specified as the `cost_column` parameter value in the call to the SDO_NET.CREATE_NODE_TABLE procedure. The cost value can represent anything you want, for example, the toll to be paid at a toll booth.
HIERARCHY_LEVEL	NUMBER	For hierarchical networks only: number indicating the level in the network hierarchy for this node. (Section 5.5 explains network hierarchy.)
PARENT_NODE_ID	NUMBER	For hierarchical networks only: node ID of the parent node of this node. (Section 5.5 explains network hierarchy.)

5.9.2 Link Table

Each network has a link table that contains the columns described in Table 5–2.

Table 5–2 Link Table Columns

Column Name	Data Type	Description
LINK_ID	NUMBER	ID number that uniquely identifies this link within the network
LINK_NAME	VARCHAR2(32)	Name of the link
START_NODE_ID	NUMBER	Node ID of the node that starts the link
END_NODE_ID	NUMBER	Node ID of the node that ends the link
LINK_TYPE	VARCHAR2(24)	User-defined string to identify the link type
ACTIVE	VARCHAR2(1)	Contains `Y` if the link is active (visible in the network), or `N` if the link is not active.
LINK_LEVEL	NUMBER	Priority level for the link; used for network analysis, so that links with higher priority levels can be considered first in computing a path
<link_geometry_column>; or GEOM_ID, START_MEASURE, and END_MEASURE	SDO_GEOMETRY, or SDO_TOPO_GEOMETRY, or NUMBER	For a spatial (SDO, non-LRS) network, the SDO_GEOMETRY object associated with the link For a spatial topology network, the SDO_TOPO_GEOMETRY object associated with the link For a spatial LRS network, GEOM_ID, START_MEASURE, and END_MEASURE column values (all of type NUMBER) for the geometry objects associated with the link For a logical network, this column is not used. For a spatial SDO or topology network, the actual column name is either a default name or what you specified as the `geom_column` parameter value in the call to the SDO_NET.CREATE_LINK_TABLE procedure.
<link_cost_column>	NUMBER	Cost value to be associated with the link, for use by applications that use the network. The actual column name is either a default name or what you specified as the `cost_column` parameter value in the call to the SDO_NET.CREATE_LINK_TABLE procedure. The cost value can represent anything you want, for example, the estimated driving time for the link.
PARENT_LINK_ID	NUMBER	For hierarchical networks only: link ID of the parent link of this link. (Section 5.5 explains parent-child relationships in a network hierarchy.)
BIDIRECTED	VARCHAR2(1)	For directed networks only: contains `Y` if the link is undirected (that is, can be traversed either from the start node to the end node or from the end node to the start node), or `N` if the link is directed (in one direction only, from the start node to the end node).

5.9.3 Path Table

Each network can have a path table. A path is an ordered sequence of links, and is usually created as a result of network analysis. A path table provides a way to store the result of this analysis. For each path table, you must create an associated path-link

table (described in Section 5.9.4). Each path table contains the columns described in Table 5–3.

Table 5–3 Path Table Columns

Column Name	Data Type	Description
PATH_ID	NUMBER	ID number that uniquely identifies this path within the network
PATH_NAME	VARCHAR2(32)	Name of the path
PATH_TYPE	VARCHAR2(24)	User-defined string to identify the path type
START_NODE_ID	NUMBER	Node ID of the node that starts the first link in the path
END_NODE_ID	NUMBER	Node ID of the node that ends the last link in the path
COST	NUMBER	Cost value to be associated with the path, for use by applications that use the network. The cost value can represent anything you want, for example, the estimated driving time for the path.
SIMPLE	VARCHAR2(1)	Contains `Y` if the path is a simple path, or `N` if the path is a complex path. In a simple path, the links form an ordered list that can be traversed from the start node to the end node with each link visited once. In a complex path, there are multiple options for going from the start node to the end node.
<path_geometry_column>	SDO_GEOMETRY	For all network types except logical, the geometry object associated with the path. The actual column name is either a default name or what you specified as the `geom_column` parameter value in the call to the SDO_NET.CREATE_PATH_TABLE procedure. For a logical network, this column is not used.

5.9.4 Path-Link Table

For each path table (described in Section 5.9.3), you must create a path-link table. Each row in the path-link table uniquely identifies a link within a path in a network; that is, each combination of PATH_ID, LINK_ID, and SEQ_NO values must be unique within the network. The order of rows in the path-link table is not significant. Each path-link table contains the columns described in Table 5–4.

Table 5–4 Path-Link Table Columns

Column Name	Data Type	Description
PATH_ID	NUMBER	ID number of the path in the network
LINK_ID	NUMBER	ID number of the link in the network
SEQ_NO	NUMBER	Unique sequence number of the link in the path. (The sequence numbers start at 1.) Sequence numbers allow paths to contain repeating nodes and links.

5.9.5 Subpath Table

Each path can have one or more associated subpaths, and information about all subpaths in a network is stored in the subpath table. A subpath is a partial path along a path, as explained in Section 5.3. The subpath table contains the columns described in Table 5–5.

Table 5–5 Subpath Table Columns

Column Name	Data Type	Description
SUBPATH_ID	NUMBER	ID number that uniquely identifies this subpath within the reference path
SUBPATH_ NAME	VARCHAR2(32)	Name of the subpath
SUBPATH_ TYPE	VARCHAR2(24)	User-defined string to identify the subpath type
REFERENCE_ PATH_ID	NUMBER	Path ID number of the path that contains this subpath
START_LINK_ INDEX	NUMBER	Link ID of the link used to define the start of the subpath. For example, in Figure 5–2 in Section 5.3, the START_LINK_INDEX is 0, and the START_ PERCENTAGE is 65.
END_LINK_ INDEX	NUMBER	Link ID of the link used to define the end of the subpath. For example, in Figure 5–2 in Section 5.3, the END_LINK_INDEX is 6, and the END_PERCENTAGE is 50.
START_ PERCENTAGE	NUMBER	Percentage of the distance between START_LINK_ INDEX and the next link in the path, representing the start point of the subpath. Can be a positive or negative number. For example, in Figure 5–2 in Section 5.3, the START_LINK_INDEX is 0, and the START_PERCENTAGE is 65.
END_ PERCENTAGE	NUMBER	Percentage of the distance between END_LINK_ INDEX and the next link in the path, representing the end point of the subpath. Can be a positive or negative number. For example, in Figure 5–2 in Section 5.3, the END_LINK_INDEX is 6, and the END_PERCENTAGE is 50.
COST	NUMBER	Cost value to be associated with the subpath, for use by applications that use the network. The cost value can represent anything you want, for example, the estimated driving time for the path.
GEOM	SDO_ GEOMETRY	For all network types except logical, the geometry object associated with the subpath. The actual column name is either a default name or what you specified as the `geom_column` parameter value in the call to the SDO_NET.CREATE_SUBPATH_TABLE procedure. For a logical network, this column is not used.

5.9.6 Partition Table

Each partitioned network has a partition table. For information about partitioned networks, see Section 5.7. Each partition table contains the columns described in Table 5–6.

Table 5–6 Partition Table Columns

Column Name	Data Type	Description
NODE_ID	NUMBER	ID number of the node
PARTITION_ID	NUMBER	ID number of the partition. Must be unique within the network.

Table 5–6 (Cont.) Partition Table Columns

Column Name	Data Type	Description
LINK_LEVEL	NUMBER	Link level (Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in computing a path.)

5.9.7 Partition BLOB Table

Each partitioned network can have a partition BLOB table, which stores binary large object (BLOB) representations for each combination of link level and partition ID in the network. Having BLOB representations of partitions enables better performance for network load on demand analysis operations. To create the partition BLOB table, use the SDO_NET.GENERATE_PARTITION_BLOBS procedure, where you specify the partition BLOB table name as one of the parameters. For information about partitioned networks, see Section 5.7.

Note: You should never directly modify the partition BLOB table. This table is automatically updated as a result of calls to the SDO_NET.GENERATE_PARTITION_BLOBS and SDO_NET.GENERATE_PARTITION_BLOB procedures.

Each partition table contains the columns described in Table 5–7.

Table 5–7 Partition BLOB Table Columns

Column Name	Data Type	Description
LINK_LEVEL	VARCHAR2(32)	Link level (Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in computing a path.)
PARTITION_ID	NUMBER	ID number of the partition
BLOB	BLOB	Binary large object (BLOB) representing the specified link level within the specified partition
NUM_INODES	NUMBER	Number of internal nodes in the BLOB (that is, total number of nodes in the BLOB)
NUM_ENODES	NUMBER	Number of external nodes. An external node is a node that is outside the BLOB, but is one end of a link in which the other node is inside the BLOB.
NUM_ILINKS	NUMBER	Number of internal links in the BLOB (that is, links completely inside the BLOB)
NUM_ELINKS	NUMBER	Number of external links. An external link is a link in which one node is internal (inside the BLOB) and one node is external (outside the BLOB).
NUM_INLINKS	NUMBER	Number of incoming links. An incoming link is an external link in which the start node is outside the BLOB and the end node is inside the BLOB.
NUM_ OUTLINKS	NUMBER	Number of outgoing links. An outgoing link is an external link in which the start node is inside the BLOB and the end node is outside the BLOB.
USER_DATA_ INCLUDED	VARCHAR2(1)	Contains `Y` if the BLOB can include user data, or `N` if the BLOB cannot include user data.

5.9.8 Connected Component Table

Each network can have a connected component table, which stores the component ID for each node. Nodes of the same connected component have the same component ID. Having this information in the table enables better performance for many network analysis operations. To create the connected component table, and to update the contents of the table at any time afterwards, use the SDO_NET.FIND_CONNECTED_COMPONENTS procedure, where you specify the connected component table name as one of the parameters. For more information about using the precomputed information about connected components, see Section 5.7.6.

Each connected component table contains the columns described in Table 5–8.

Table 5–8 Connected Component Table Columns

Column Name	Data Type	Description
LINK_LEVEL	NUMBER	Link level of the component assignment. (Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in computing a path.)
NODE_ID	NUMBER	ID number of the node from which to compute all other components that are reachable.
COMPONENT_ID	NUMBER	ID number of a component that is reachable from the specified node.

5.10 Network Data Model Metadata Views

There is a set of network metadata views for each schema (user): xxx_SDO_NETWORK_METADATA, where *xxx* can be USER or ALL. These views are created by Spatial.

5.10.1 xxx_SDO_NETWORK_METADATA Views

The following views contain information about networks:

- USER_SDO_NETWORK_METADATA contains information about all networks owned by the user.
- ALL_SDO_NETWORK_METADATA contains information about all networks on which the user has SELECT permission.

If you create a network using one of the CREATE_<*network-type*>_NETWORK procedures, the information in these views is automatically updated to reflect the new network; otherwise, you must insert information about the network into the USER_SDO_NETWORK_METADATA view.

The USER_SDO_NETWORK_METADATA and ALL_SDO_NETWORK_METADATA views contain the same columns, as shown Table 5–9, except that the USER_SDO_NETWORK_METADATA view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5–9 Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the network (ALL_SDO_NETWORK_METADATA view only)
NETWORK	VARCHAR2(24)	Name of the network

Table 5–9 (Cont.) Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
NETWORK_ID	NUMBER	ID number of the network; assigned by Spatial
NETWORK_ CATEGORY	VARCHAR2(12)	Contains `SPATIAL` if the network nodes and links are associated with spatial geometries; contains `LOGICAL` if the network nodes and links are not associated with spatial geometries. A value of `LOGICAL` causes the network data model PL/SQL and Java APIs to ignore any spatial attributes of nodes, links, and paths.
GEOMETRY_TYPE	VARCHAR2(24)	If `NETWORK_CATEGORY` is `SPATIAL`, contains a value indicating the geometry type of nodes and links: `SDO_GEOMETRY` for non-LRS SDO_ GEOMETRY objects, `LRS_GEOMETRY` for LRS SDO_GEOMETRY objects, `TOPO_GEOMETRY` for SDO_TOPO_GEOMETRY objects.
NETWORK_TYPE	VARCHAR2(24)	User-defined string to identify the network type.
NO_OF_ HIERARCHY_ LEVELS	NUMBER	Number of levels in the network hierarchy. Contains 1 if there is no hierarchy. (See Section 5.5 for information about network hierarchy.)
NO_OF_ PARTITIONS	NUMBER	(Not currently used)
LRS_TABLE_NAME	VARCHAR2(32)	If `GEOMETRY_TYPE` is `SDO_GEOMETRY`, contains the name of the table containing geometries associated with nodes.
LRS_GEOM_ COLUMN	VARCHAR2(32)	If `LRS_TABLE_NAME` contains a table name, identifies the geometry column in that table.
NODE_TABLE_ NAME	VARCHAR2(32)	If `GEOMETRY_TYPE` is `SDO_GEOMETRY`, contains the name of the table containing geometries associated with nodes. (The node table is described in Section 5.9.1.)
NODE_GEOM_ COLUMN	VARCHAR2(32)	If `NODE_TABLE_NAME` contains a table name, identifies the geometry column in that table.
NODE_COST_ COLUMN	VARCHAR2(1024)	If `NODE_TABLE_NAME` contains a table name, identifies the cost column in that table, or a PL/SQL function to compute the cost value.
NODE_PARTITION_ COLUMN	VARCHAR2(32)	(Not currently used).
NODE_DURATION_ COLUMN	VARCHAR2(32)	If `NODE_TABLE_NAME` contains a table name, identifies the optional duration column in that table. This column can contain a numeric value that has any user-defined significance, such as a number of minutes associated with the node.
LINK_TABLE_NAME	VARCHAR2(32)	If `GEOMETRY_TYPE` is `SDO_GEOMETRY`, contains the name of the table containing geometries associated with links. (The link table is described in Section 5.9.2.)
LINK_GEOM_ COLUMN	VARCHAR2(32)	If `LINK_TABLE_NAME` contains a table name, identifies the geometry column in that table.
LINK_DIRECTION	VARCHAR2(12)	Contains a value indicating the type for all links in the network: `UNDIRECTED` or `DIRECTED`.

Table 5–9 (Cont.) Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
LINK_COST_ COLUMN	VARCHAR2(1024)	If LINK_TABLE_NAME contains a table name, identifies the optional numeric column containing a cost value for each link, or a PL/SQL function to compute the cost value.
LINK_PARTITION_ COLUMN	VARCHAR2(32)	(Not currently used)
LINK_DURATION_ COLUMN	VARCHAR2(32)	If LINK_TABLE_NAME contains a table name, identifies the optional duration column in that table. This column can contain a numeric value that has any user-defined significance, such as a number of minutes associated with the link.
PATH_TABLE_ NAME	VARCHAR2(32)	Contains the name of an optional table containing information about paths. (The path table is described in Section 5.9.3.)
PATH_GEOM_ COLUMN	VARCHAR2(32)	If PATH_TABLE_NAME is associated with a spatial network, identifies the geometry column in that table.
PATH_LINK_ TABLE_NAME	VARCHAR2(32)	Contains the name of an optional table containing information about links for each path. (The path-link table is described in Section 5.9.4.)
SUBPATH_TABLE_ NAME	VARCHAR2(32)	Contains the name of an optional table containing information about subpaths. (The subpath table is described in Section 5.9.5.)
SUBPATH_GEOM_ COLUMN	VARCHAR2(32)	If SUBPATH_TABLE_NAME is associated with a spatial network, identifies the geometry column in that table.
PARTITION_TABLE_ NAME	VARCHAR2(32)	For a partitioned network: the name of the partition table. (The partition table is described in Section 5.9.6.
PARTITION_BLOB_ TABLE_NAME	VARCHAR2(32)	For a partitioned network for which any partition BLOBs have been generated: the name of the partition BLOB table. (The partition BLOB table is described in Section 5.9.7.
COMPONENT_ TABLE_NAME	VARCHAR2(32)	The name of the table containing information about precomputed connected components, as explained in Section 5.7.6. (The connected component table is described in Section 5.9.8.)
NODE_LEVEL_ TABLE_NAME	VARCHAR2(32)	The name of the table containing information about node levels in a hierarchical network. Specify this table as the node_level_table_ name parameter with the SDO_NET.GENERATE_ NODE_LEVELS procedure.
TOPOLOGY	VARCHAR2(32)	For a spatial network containing SDO_TOPO_ GEOMETRY objects (creating using the SDO_ NET.CREATE_TOPO_NETWORK procedure), contains the name of the topology.
USER_DEFINED_ DATA	VARCHAR2(1)	Contains Y if the network contains user-defined data; contains N if the network does not contain user-defined data.
EXTERNAL_ REFERENCES	VARCHAR2(1)	(Not currently used)

5.10.2 xxx_SDO_NETWORK_CONSTRAINTS Views

The following views contain information about network constraints (described in Section 5.6):

- USER_SDO_NETWORK_CONSTRAINTS contains information about all network constraints owned by the user.
- ALL_SDO_NETWORK_CONSTRAINTS contains information about all network constraints on which the user has SELECT permission.

These views are automatically maintained by the SDO_NET.REGISTER_CONSTRAINT and SDO_NET.DEREGISTER_CONSTRAINT procedures. You should not directly modify the contents of these views.

The USER_SDO_NETWORK_CONSTRAINTS and ALL_SDO_NETWORK_CONSTRAINTS views contain the same columns, as shown Table 5–10, except that the USER_SDO_NETWORK_CONSTRAINTS view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5–10 Columns in the xxx_SDO_NETWORK_CONSTRAINTS Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the network constraint (ALL_SDO_NETWORK_CONSTRAINTS view only)
CONSTRAINT	VARCHAR2(32)	Name of the network constraint
DESCRIPTION	VARCHAR2(200)	Descriptive information about the network constraint, such as its purpose and any usage notes
CLASS_NAME	VARCHAR2(4000)	Name of the Java class that implements the network constraint
CLASS	BINARY FILE LOB	The Java class that implements the network constraint

5.10.3 xxx_SDO_NETWORK_USER_DATA Views

The following views contain information about network user-defined data, which is the information (not related to connectivity) that users want to associate with a network representation:

- USER_SDO_NETWORK_USER_DATA contains information about all network user-defined data owned by the user.
- ALL_SDO_NETWORK_USER_DATA contains information about all network user-defined data on which the user has SELECT permission.

The USER_SDO_NETWORK_USER_DATA and ALL_SDO_NETWORK_USER_DATA views contain the same columns, as shown Table 5–10, except that the USER_SDO_NETWORK_USER_DATA view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5–11 Columns in the xxx_SDO_NETWORK_USER_DATA Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the network constraint (ALL_SDO_NETWORK_CONSTRAINTS view only)
NETWORK	VARCHAR2(32)	Name of the network

Table 5–11 (Cont.) Columns in the xxx_SDO_NETWORK_USER_DATA Views

Column Name	Data Type	Purpose
TABLE_TYPE	VARCHAR2(12)	Type of the table containing the user-defined data: NODE, LINK, PATH, or SUBPATH
DATA_NAME	VARCHAR2(32)	Name of column containing the user-defined data
DATA_TYPE	VARCHAR2(12)	Data type of the user-defined data: VARCHAR2, INTEGER, NUMBER, DATE, TIMESTAMP, or SDO_GEOMETRY
DATA_LENGTH	NUMBER(38)	If DATA_TYPE is VARCHAR2, the length of the user-defined data

For an example of using user-defined data, see Section 5.13.6.

5.11 Network Data Model Application Programming Interface

The Oracle Spatial network data model includes two client application programming interfaces (APIs): a PL/SQL interface provided by the SDO_NET and SDO_NET_MEM packages and a Java interface. Both interfaces let you create and update network data, and perform network analysis. It is recommended that you use only PL/SQL or SQL to populate network tables and to create indexes, and that you use either PL/SQL or Java for application development.

The following performance considerations apply to the PL/SQL and Java APIs:

- If you plan to analyze or edit only nonspatial aspects of a spatial network, you can get better performance by setting the NETWORK_CATEGORY column value to LOGICAL in the USER_SDO_NETWORK_METADATA view (described in Section 5.10.1) before performing the analysis or editing, and then changing the value back to SPATIAL afterward.

  For example, you could use this technique when finding the shortest path between two nodes, because the shortest-path computation considers cost values. However, you could not use this technique when setting the spatial geometry object or the end measure value for a link.

- If you do not plan to modify any network objects (that is, if you plan to perform only network analysis operations or to retrieve network information), you can get better performance by creating the network memory object as read-only (that is, by specifying that updates are not allowed).

5.11.1 Network Data Model PL/SQL Interface

The SDO_NET package provides subprograms for creating, accessing, and managing networks on a database server. The SDO_NET_MEM package, which implements capabilities available through the Java API, provides subprograms for editing network objects and performing network analysis using a cache object called a network memory object. Example 5–4 in Section 5.13 shows the use of SDO_NET functions and procedures. Section 5.8 explains how to use a network memory object, and it contains Example 5–1, which uses SDO_NET_MEM functions and procedures.

The SDO_NET subprograms can be grouped into the following logical categories:

- Creating networks:

  SDO_NET.CREATE_SDO_NETWORK

  SDO_NET.CREATE_LRS_NETWORK

SDO_NET.CREATE_TOPO_NETWORK

SDO_NET.CREATE_LOGICAL_NETWORK

- Copying and deleting networks:

  SDO_NET.COPY_NETWORK

  SDO_NET.DROP_NETWORK

- Creating network tables:

  SDO_NET.CREATE_NODE_TABLE

  SDO_NET.CREATE_LINK_TABLE

  SDO_NET.CREATE_PATH_TABLE

  SDO_NET.CREATE_PATH_LINK_TABLE

  SDO_NET.CREATE_LRS_TABLE

- Validating network objects:

  SDO_NET.VALIDATE_NETWORK

  SDO_NET.VALIDATE_NODE_SCHEMA

  SDO_NET.VALIDATE_LINK_SCHEMA

  SDO_NET.VALIDATE_PATH_SCHEMA

  SDO_NET.VALIDATE_LRS_SCHEMA

- Retrieving information (getting information about the network, checking for a characteristic):

  SDO_NET.GET_CHILD_LINKS

  SDO_NET.GET_CHILD_NODES

  SDO_NET.GET_GEOMETRY_TYPE

  SDO_NET.GET_IN_LINKS

  SDO_NET.GET_LINK_COST_COLUMN

  SDO_NET.GET_LINK_DIRECTION

  SDO_NET.GET_LINK_GEOM_COLUMN

  SDO_NET.GET_LINK_GEOMETRY

  SDO_NET.GET_LINK_TABLE_NAME

  SDO_NET.GET_LRS_GEOM_COLUMN

  SDO_NET.GET_LRS_LINK_GEOMETRY

  SDO_NET.GET_LRS_NODE_GEOMETRY

  SDO_NET.GET_LRS_TABLE_NAME

  SDO_NET.GET_NETWORK_TYPE

  SDO_NET.GET_NO_OF_HIERARCHY_LEVELS

  SDO_NET.GET_NO_OF_LINKS

  SDO_NET.GET_NO_OF_NODES

  SDO_NET.GET_NODE_DEGREE

SDO_NET.GET_NODE_GEOM_COLUMN

SDO_NET.GET_NODE_GEOMETRY

SDO_NET.GET_NODE_IN_DEGREE

SDO_NET.GET_NODE_OUT_DEGREE

SDO_NET.GET_NODE_TABLE_NAME

SDO_NET.GET_OUT_LINKS

SDO_NET.GET_PATH_GEOM_COLUMN

SDO_NET.GET_PATH_TABLE_NAME

SDO_NET.IS_HIERARCHICAL

SDO_NET.IS_LOGICAL

SDO_NET.IS_SPATIAL

SDO_NET.LRS_GEOMETRY_NETWORK

SDO_NET.NETWORK_EXISTS

SDO_NET.SDO_GEOMETRY_NETWORK

SDO_NET.TOPO_GEOMETRY_NETWORK

For reference information about each SDO_NET function and procedure, see Chapter 6.

The SDO_NET_MEM subprograms are grouped according to their associated object-related class in the oracle.spatial.network interface or class. You must specify a prefix after SDO_NET_MEM for each program, depending on its associated class (for example, SDO_NET_MEM.NETWORK_MANAGER.CREATE_LOGICAL_NETWORK, SDO_NET_MEM.NETWORK.ADD_NODE, and SDO_NET_MEM.NODE.GET_COST).

Note: Although this manual refers to "the SDO_NET_MEM package," all subprograms except one are actually implemented as methods of several object types. Thus, they are not listed by the statement `DESCRIBE SDO_NET_MEM`. Instead, you can use the DESCRIBE statements listed in Table 7–1 in Chapter 7 to see the subprograms in each grouping; however, because they are member functions and procedures in an object type, the subprograms in each grouping will not be listed in alphabetical order in the DESCRIBE statement output.

The SDO_NET_MEM subprogram groupings are as follows:

- SDO_NET_MEM.NETWORK_MANAGER subprograms are related to the `oracle.spatial.network.NetworkManager` Java class. They enable you to create and drop network memory objects and to perform network analysis.
- SDO_NET_MEM.NETWORK subprograms are related to the `oracle.spatial.network.Network` Java interface. They enable you to add and delete nodes, links, and paths.
- SDO_NET_MEM.NODE subprograms are related to the `oracle.spatial.network.Node` Java interface. They enable you to get and set attributes for nodes.

- SDO_NET_MEM.LINK subprograms are related to the `oracle.spatial.network.Link` Java interface. They enable you to get and set attributes for links.
- SDO_NET_MEM.PATH subprograms are related to the `oracle.spatial.network.Path` Java interface. They enable you to get and set attributes for paths.

The associations between SDO_NET_MEM subprograms and methods of the Java API are not necessarily exact. In some cases, a PL/SQL subprogram may combine operations and options from several methods. In addition, some Java methods do not have PL/SQL counterparts. Thus, the Usage Notes for subprograms state only that the function or procedure is analogous to a specific Java method, to indicate a logical relationship between the two. For detailed information about a specific Java method and others that may be related, see the Javadoc-generated API documentation (briefly explained in Section 5.11.2).

For reference information about each SDO_NET_MEM function and procedure, see Chapter 7.

5.11.2 Network Data Model Java Interface

The network data model Java interface includes the in-memory interface and the load on demand interface. Complete reference information about these interfaces is provided in *Oracle Spatial Java API Reference*. The classes of the in-memory Java interface are in the `oracle.spatial.network package`. The classes of the load on demand Java interface are in the `oracle.spatial.network.lod` package and its subpackages.

5.11.2.1 Network Metadata and Data Management

You can use the Java API to perform network metadata and data management operations such as the following:

- Insert, delete, and modify node and link data
- Load a network from a database
- Store a network in a database
- Store network metadata in a database
- Modify network metadata attributes

5.11.2.2 Network Analysis Using the In-Memory Approach

You can use the `oracle.spatial.network.NetworkManager` class to perform network analysis operations, such as the following, using the in-memory approach:

- Shortest path: typical transitive closure problems in graph theory. Given a start and an end node, find the shortest path.
- Minimum cost spanning tree: Given an undirected graph, find the minimum cost tree that connects all nodes.
- Reachability: Given a node, find all nodes that can reach that node, or find all nodes that can be reached by that node.
- Within-cost analysis: Given a target node and a cost, find all nodes that can be reached by the target node within the given cost.
- Nearest-neighbors analysis: Given a target node and number of neighbors, find the neighbor nodes and their costs to go to the given target node.

- All paths between two nodes: Given two nodes, find all possible paths between them.
- "Traveling salesman problem" (TSP) analysis: Given a set of nodes, find the most efficient (lowest-cost or shortest distance) path that visits all nodes, and optionally require that the start and end nodes be the same.

5.11.2.3 Network Analysis Using the Load on Demand Approach

You can use the `oracle.spatial.network.lod.NetworkAnalyst` class to perform network analysis operations, such as the following, using the load on demand approach:

- Shortest path: typical transitive closure problems in graph theory. Given a start and an end node, find the shortest path.
- Reachability: Given a node, find all nodes that can reach that node, or find all nodes that can be reached by that node.
- Within-cost analysis: Given a target node and a cost, find all nodes that can be reached by the target node within the given cost.
- Nearest-neighbors analysis: Given a target node and number of neighbors, find the neighbor nodes and their costs to go to the given target node.
- Dynamic data input: Create and use a `NetworkUpdate` object with network update information.
- User-defined link and node cost calculators: Define the method for computing the cost of a link or a node.

5.12 Cross-Schema Network Access

If database users other than the network owner need to read a network into memory, you need to do one of the following:

- For each non-owner user, qualify the network tables with the schema of the network owner in the USER_SDO_NETWORK_METADATA view, as explained in Section 5.12.1.
- For each non-owner user, create views on the network data model tables and update the USER_SDO_NETWORK_METADATA view, as explained in Section 5.12.2.

The second approach requires the extra step of creating views, but the views provide you with flexibility in controlling the parts of the network that are accessible. Each view can provide access to all of the network, or it can use a WHERE clause to provide access to just one or more parts (for example, `WHERE STATE_CODE='NY'` to restrict the view users to rows for New York) .

Consider the following example scenario:

- User1 creates (and thus owns) Network1.
- User2 attempts to call the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure to read Network1, but receives an error. The error occurs even though User2 has the appropriate privileges on the network data model tables for Network1.

To work around this problem, you must use the approach in either Section 5.12.1, "Cross-Schema Access by Specifying Owner in Network Metadata" or Section 5.12.2, "Cross-Schema Access by Using Views".

5.12.1 Cross-Schema Access by Specifying Owner in Network Metadata

To enable a non-owner user (with suitable privileges) to access a network, you can specify the network owner in the network metadata. For each non-owner user that will be permitted to access the network, follow these steps:

1. Ensure that the user has SELECT access to the necessary network data model tables. If the user does not have this access, connect as the network owner and grant it. For example, connect as User1 and execute the following statements:

```
GRANT select ON network1_node$ TO user2;
GRANT select ON network1_link$ TO user2;
GRANT select ON network1_path$ TO user2;
GRANT select ON network1_plink$ TO user2;
```

2. Connect as the non-owner user. For example, connect as User2.

3. Use the schema name of the network owner to qualify the network data model tables for the network in the USER_SDO_NETWORK_METADATA view (explained in Section 5.10.1). For example, if the network is not already defined in this view, enter the following while connected as User2:

```
INSERT INTO user_sdo_network_metadata
  (network, network_category, geometry_type,
   node_table_name,node_geom_column,
   link_table_name, link_geom_column, link_direction,
   path_table_name, path_geom_column,
   path_link_table_name)
VALUES
  ('NETWORK1','SPATIAL', 'SDO_GEOMETRY',
   'USER1.NETWORK1_NODE$', 'GEOMETRY',
   'USER1.NETWORK1_LINK$', 'GEOMETRY', 'DIRECTED',
   'USER1.NETWORK1_PATH$', 'GEOMETRY',
   'USER1.NETWORK1_PLINK$');
```

   If the network is already defined in this view, update the definition to qualify each table name with the schema name. For example:

```
UPDATE USER_SDO_NETWORK_METADATA
  SET node_table_name = 'USER1.NETWORK1_NODE$',
      link_table_name = 'USER1.NETWORK1_LINK$',
      path_table_name = 'USER1.NETWORK1_PATH$',
      path_link_table_name = 'USER1.NETWORK1_PLINK$'
  WHERE network = 'NETWORK1';
```

In this scenario, User2 can now read NETWORK1 into memory.

5.12.2 Cross-Schema Access by Using Views

To enable a non-owner user (with suitable privileges) to access a network, or specific parts of a network, you can create views. For each non-owner user that will be permitted to access the network, follow these steps:

1. Ensure that the user has SELECT access to the necessary network data model tables. If the user does not have this access, connect as the network owner and grant it. For example, connect as User1 and execute the following statements:

```
GRANT select ON network1_node$ TO user2;
GRANT select ON network1_link$ TO user2;
GRANT select ON network1_path$ TO user2;
GRANT select ON network1_plink$ TO user2;
```

2. Connect as the non-owner user. For example, connect as User2.
3. Create a view on each of the necessary network data model nodes, with each view selecting all columns in the associated table. Qualify the table name with the schema name of the network owner. For example, while connected as User2:

```
CREATE VIEW network1_node$ AS select * from user1.network1_node$;
CREATE VIEW network1_link$ AS select * from user1.network1_link$;
CREATE VIEW network1_path$ AS select * from user1.network1_path$;
CREATE VIEW network1_plink$ AS select * from user1.network1_plink$;
```

> **Note:** Although this example shows views that include all data in the underlying tables, you can restrict the parts of the network that are available by using a WHERE clause in each view definition (for example, `WHERE STATE_CODE='NY'`).

4. Add a row specifying the newly created views to the USER_SDO_NETWORK_METADATA view (explained in Section 5.10.1). For example, while connected as User2:

```
INSERT INTO user_sdo_network_metadata
  (network, network_category, geometry_type,
   node_table_name,node_geom_column,
   link_table_name, link_geom_column, link_direction,
   path_table_name, path_geom_column,
   path_link_table_name)
VALUES
  ('NETWORK1','SPATIAL', 'SDO_GEOMETRY',
   'NETWORK1_NODE$', 'GEOMETRY',
   'NETWORK1_LINK$', 'GEOMETRY', 'DIRECTED',
   'NETWORK1_PATH$', 'GEOMETRY',
   'NETWORK1_PLINK$');
```

In this scenario, User2 can now read into memory those parts of NETWORK1 that are available through the views that were created.

5.13 Network Examples

This section presents several network data model examples. Most are simplified examples. All examples use the PL/SQL API, and some also use other APIs. This section includes the following subsections:

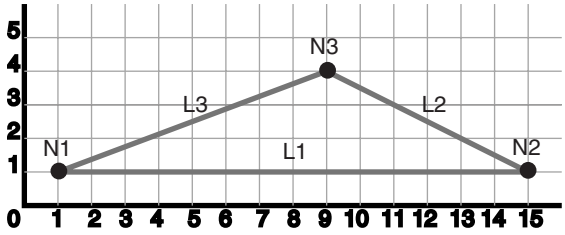
- Section 5.13.1, "Simple Spatial (SDO) Network Example (PL/SQL)"
- Section 5.13.2, "Simple Logical Network Example (PL/SQL)"
- Section 5.13.3, "Spatial (LRS) Network Example (PL/SQL)"
- Section 5.13.4, "Logical Hierarchical Network Example (PL/SQL)"
- Section 5.13.5, "Partitioning and Load on Demand Analysis Examples (PL/SQL, XML, and Java)"
- Section 5.13.6, "User-Defined Data Example (PL/SQL and Java)"

The examples refer to concepts that are explained in this chapter, and they use PL/SQL functions and procedures documented in Chapter 6.

5.13.1 Simple Spatial (SDO) Network Example (PL/SQL)

This section presents an example of a very simple spatial (SDO, not LRS) network that contains three nodes and a link between each node. The network is illustrated in Figure 5–4.

Figure 5–4 Simple Spatial (SDO) Network



As shown in Figure 5–4, node N1 is at point 1,1, node N2 is at point 15,1, and node N3 is at point 9,4. Link L1 is a straight line connecting nodes N1 and N2, link L2 is a straight line connecting nodes N2 and N3, and link L3 is a straight line connecting nodes N3 and N1. There are no other nodes or shape points on any of the links.

Example 5–2 does the following:

- In a call to the SDO_NET.CREATE_SDO_NETWORK procedure, creates the SDO_NET1 directed network; creates the SDO_NET1_NODE$, SDO_NET1_LINK$, SDO_NET1_PATH$, and SDO_NET1_PLINK$ tables; and updates the xxx_SDO_NETWORK_METADATA views. All geometry columns are named GEOMETRY. Both the node and link tables contain a cost column named COST.
- Populates the node, link, path, and path-link tables. It inserts three rows into the node table, three rows into the link table, two rows into the path table, and four rows into the path-link table.
- Updates the Oracle Spatial metadata, and creates spatial indexes on the GEOMETRY columns of the node and link tables. (These actions are not specifically related to network management, but that are necessary if applications are to benefit from spatial indexing on these geometry columns.)

Example 5–2 does not show the use of many SDO_NET functions and procedures; these are included in Example 5–4 in Section 5.13.3.

Example 5–2 Simple Spatial (SDO) Network Example (PL/SQL)

```
-- Create the SDO_NET1 directed network. Also creates the SDO_NET1_NODE$,
-- SDO_NET1_LINK$, SDO_NET1_PATH$, SDO_NET1_PLINK$ tables, and updates
-- USER_SDO_NETWORK_METADATA. All geometry columns are named GEOMETRY.
-- Both the node and link tables contain a cost column named COST.
EXECUTE SDO_NET.CREATE_SDO_NETWORK('SDO_NET1', 1, TRUE, TRUE);

-- Populate the SDO_NET1_NODE$ table.
-- N1
INSERT INTO sdo_net1_node$ (node_id, node_name, active, geometry, cost)
  VALUES(1, 'N1', 'Y',
    SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(1,1,NULL), NULL, NULL),
    5);
-- N2
INSERT INTO sdo_net1_node$ (node_id, node_name, active, geometry, cost)
  VALUES(2, 'N2', 'Y',
    SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(15,1,NULL), NULL, NULL),
```

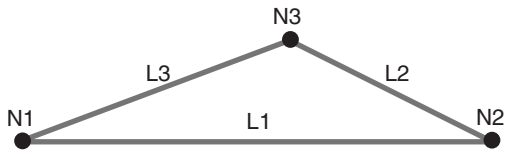
```
    8);
-- N3
INSERT INTO sdo_net1_node$ (node_id, node_name, active, geometry, cost)
  VALUES(3, 'N3', 'Y',
    SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,4,NULL), NULL, NULL),
    4);

-- Populate the SDO_NET1_LINK$ table.
-- L1
INSERT INTO sdo_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, geometry, cost, bidirected)
  VALUES(1, 'L1', 1, 2, 'Y',
    SDO_GEOMETRY(2002, NULL, NULL,
      SDO_ELEM_INFO_ARRAY(1,2,1),
        SDO_ORDINATE_ARRAY(1,1, 15,1)),
    14, 'Y');
-- L2
INSERT INTO sdo_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, geometry, cost, bidirected)
   VALUES(2, 'L2', 2, 3, 'Y',
    SDO_GEOMETRY(2002, NULL, NULL,
      SDO_ELEM_INFO_ARRAY(1,2,1),
        SDO_ORDINATE_ARRAY(15,1, 9,4)),
    10, 'Y');
-- L3
INSERT INTO sdo_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, geometry, cost, bidirected)
  VALUES(3, 'L3', 3, 1, 'Y',
    SDO_GEOMETRY(2002, NULL, NULL,
      SDO_ELEM_INFO_ARRAY(1,2,1),
        SDO_ORDINATE_ARRAY(9,4, 1,1)),
    10, 'Y');

-- Do not populate the SDO_NET1_PATH$ and SDO_NET1_PLINK$ tables now.
-- Do this only when you need to create any paths.

-------------------------------------------------------------------------------
-- REMAINING STEPS NEEDED TO USE SPATIAL INDEXES --
-------------------------------------------------------------------------------
-- Update the USER_SDO_GEOM_METADATA view. This is required before the
-- spatial index can be created. Do this only once for each layer
-- (that is, table-column combination).

INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
     DIMINFO,
     SRID)
  VALUES (
    'SDO_NET1_NODE$',
    'GEOMETRY',
    SDO_DIM_ARRAY(   -- 20X20 grid
      SDO_DIM_ELEMENT('X', 0, 20, 0.005),
      SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
       ),
    NULL   -- SRID (spatial reference system, also called coordinate system)
  );
INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
```

```
    DIMINFO,
    SRID)
  VALUES (
    'SDO_NET1_LINK$',
    'GEOMETRY',
    SDO_DIM_ARRAY(   -- 20X20 grid
      SDO_DIM_ELEMENT('X', 0, 20, 0.005),
      SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
       ),
    NULL   -- SRID (spatial reference system, also called coordinate system)
  );

-- Create the spatial indexes
CREATE INDEX sdo_net1_nodes_idx ON sdo_net1_node$(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
CREATE INDEX sdo_net1_links_idx ON sdo_net1_link$(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

5.13.2 Simple Logical Network Example (PL/SQL)

This section presents an example of a very simple logical network that contains three nodes and a link between the nodes. The network is illustrated in Figure 5–5.

Figure 5–5 Simple Logical Network



As shown in Figure 5–5, link L1 is a straight line connecting nodes N1 and N2, link L2 is a straight line connecting nodes N2 and N3, and link L3 is a straight line connecting nodes N3 and N1. There are no other nodes on any of the links.

Example 5–3 calls the SDO_NET.CREATE_LOGICAL_NETWORK procedure, which does the following: creates the LOG_NET1 directed network; creates the LOG_NET1_NODE$, LOG_NET1_LINK$, LOG_NET1_PATH$, and LOG_NET1_PLINK$ tables; and updates the xxx_SDO_NETWORK_METADATA views. Both the node and link tables contain a cost column named COST. (Because this is a logical network, there are no geometry columns.) The example also populates the node and link tables.

Example 5–3 does not show the use of many SDO_NET functions and procedures; these are included in the logical hierarchical network example (Example 5–5) in Section 5.13.4.

Example 5–3 Simple Logical Network Example (PL/SQL)

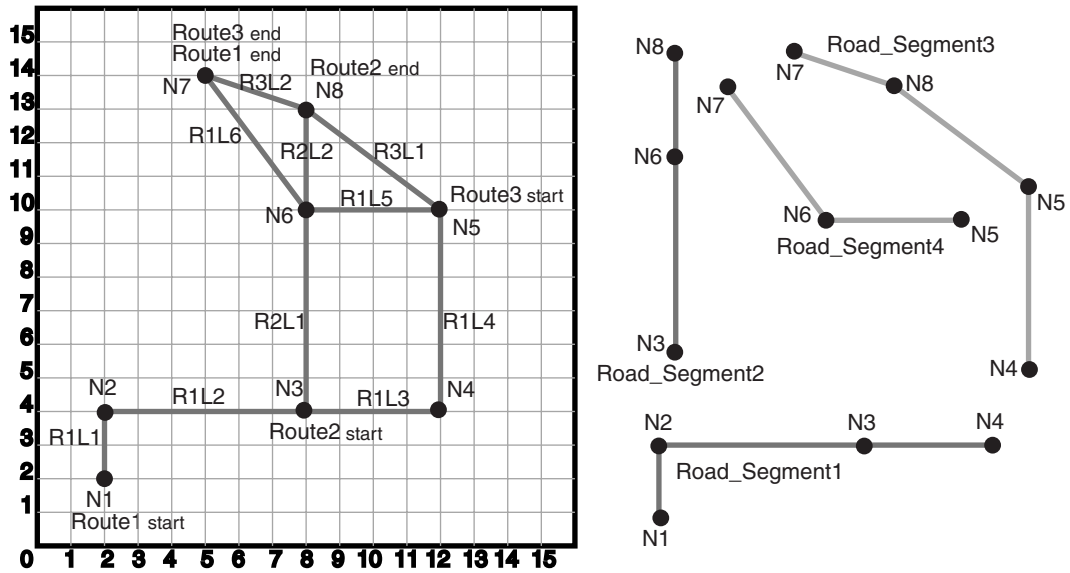
```
-- Creates the LOG_NET1 directed logical network. Also creates the
-- LOG_NET1_NODE$, LOG_NET1_LINK$, LOG_NET1_PATH$,
-- and LOG_NET1_PLINK$ tables, and updates USER_SDO_NETWORK_METADATA.
-- Both the node and link tables contain a cost column named COST.
EXECUTE SDO_NET.CREATE_LOGICAL_NETWORK('LOG_NET1', 1, TRUE, TRUE);

-- Populate the LOG_NET1_NODE$ table.
-- N1
INSERT INTO log_net1_node$ (node_id, node_name, active, cost)
  VALUES (1, 'N1', 'Y', 2);
```

```
-- N2
INSERT INTO log_net1_node$ (node_id, node_name, active, cost)
  VALUES (2, 'N2', 'Y', 3);
-- N3
INSERT INTO log_net1_node$ (node_id, node_name, active, cost)
  VALUES (3, 'N3', 'Y', 2);

-- Populate the LOG_NET1_LINK$ table.
-- L1
INSERT INTO log_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, link_level, cost)
  VALUES (1, 'L1', 1, 2, 'Y', 1, 10);
-- L2
INSERT INTO log_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, link_level, cost)
  VALUES (2, 'L2', 2, 3, 'Y', 1, 7);
-- L3
INSERT INTO log_net1_link$ (link_id, link_name, start_node_id, end_node_id,
     active, link_level, cost)
  VALUES (3, 'L3', 3, 1, 'Y', 1, 8);

-- Do not populate the LOG_NET1_PATH$ and LOG_NET1_PLINK$ tables now.
-- Do this only when you need to create any paths.
```

5.13.3 Spatial (LRS) Network Example (PL/SQL)

This section presents an example of a spatial (LRS) network that uses the roads (routes) illustrated in Figure 5–6. Each road is built from individual line segments (associated with links) taken from one or more road segment geometries, which are also shown in the figure.

Figure 5–6 Roads and Road Segments for Spatial (LRS) Network Example



As shown in Figure 5–6:

- `Route1` starts at point 2,2 and ends at point 5,14. It has the following nodes: `N1`, `N2`, `N3`, `N4`, `N5`, `N6`, and `N7`. It has the following links: `R1L1`, `R1L2`, `R1L3`, `R1L4`, `R1L5`, and `R1L6`.

- Route2 starts at point 8,4 and ends at point 8,13. It has the following nodes: N3, N6, and N8. It has the following links: R2L1 and R2L2.
- Route3 starts at point 12,10 and ends at point 5,14. It has the following nodes: N5, N8, and N7. It has the following links: R3L1 and R3L2.
- The four road segment geometries are shown individually on the right side of the figure. (The points on each segment are labeled with their associated node names, to clarify how each segment geometry fits into the illustration on the left side.)

Example 5–4 does the following:

- Creates a table to hold the road segment geometries.
- Inserts four road segment geometries into the table.
- Inserts the spatial metadata into the USER_SDO_GEOM_METADATA view.
- Creates a spatial index on the geometry column in the ROAD_SEGMENTS table.
- Creates and populates the node table.
- Creates and populates the link table.
- Creates and populates the path table and path-link table, for possible future use. (Before an application can use paths, you must populate these two tables.)
- Inserts network metadata into the USER_SDO_NETWORK_METADATA view.
- Uses various SDO_NET and SDO_NET_MEM functions and procedures.

Example 5–4 Spatial (LRS) Network Example (PL/SQL)

```
--------------------------------------------------------------------------
-- CREATE AND POPULATE TABLE --
--------------------------------------------------------------------------
-- Create a table for road segments. Use LRS.
CREATE TABLE road_segments (
  segment_id  NUMBER PRIMARY KEY,
  segment_name  VARCHAR2(32),
  segment_geom  SDO_GEOMETRY,
  geom_id NUMBER);

-- Populate the table with road segments.
INSERT INTO road_segments VALUES(
  1,
  'Segment1',
  SDO_GEOMETRY(
    3302,  -- line string, 3 dimensions (X,Y,M), 3rd is measure dimension
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    SDO_ORDINATE_ARRAY(
      2,2,0,   -- Starting point - Node1; 0 is measure from start.
      2,4,2,   -- Node2; 2 is measure from start.
      8,4,8,   -- Node3; 8 is measure from start.
      12,4,12) -- Node4; 12 is measure from start.
  ), 1001
);

INSERT INTO road_segments VALUES(
  2,
  'Segment2',
  SDO_GEOMETRY(
```

```
    3302,  -- line string, 3 dimensions (X,Y,M), 3rd is measure dimension
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    SDO_ORDINATE_ARRAY(
      8,4,0,   -- Node3; 0 is measure from start.
      8,10,6,  -- Node6; 6 is measure from start.
      8,13,9)  -- Ending point - Node8; 9 is measure from start.
  ), 1002
);

INSERT INTO road_segments VALUES(
  3,
  'Segment3',
  SDO_GEOMETRY(
    3302,  -- line string, 3 dimensions (X,Y,M), 3rd is measure dimension
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    SDO_ORDINATE_ARRAY(
      12,4,0,     -- Node4; 0 is measure from start.
      12,10,6,    -- Node5; 6 is measure from start.
      8,13,11,    -- Node8; 11 is measure from start.
      5,14,14.16) -- Ending point - Node7; 14.16 is measure from start.
  ), 1003
);

INSERT INTO road_segments VALUES(
  4,
  'Segment4',
  SDO_GEOMETRY(
    3302,  -- line string, 3 dimensions (X,Y,M), 3rd is measure dimension
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    SDO_ORDINATE_ARRAY(
      12,10,0, -- Node5; 0 is measure from start.
      8,10,4,  -- Node6; 4 is measure from start.
      5,14,9)  -- Ending point - Node7; 9 is measure from start.
  ), 1004
);

-----------------------------------------------------------------------------
-- UPDATE THE SPATIAL METADATA --
-----------------------------------------------------------------------------
-- Update the USER_SDO_GEOM_METADATA view. This is required before the
-- spatial index can be created. Do this only once for each layer
-- (that is, table-column combination; here: road_segment and segment_geom).
INSERT INTO user_sdo_geom_metadata
    (TABLE_NAME,
     COLUMN_NAME,
     DIMINFO,
     SRID)
  VALUES (
  'ROAD_SEGMENTS',
  'SEGMENT_GEOM',
  SDO_DIM_ARRAY(   -- 20X20 grid
    SDO_DIM_ELEMENT('X', 0, 20, 0.005),
    SDO_DIM_ELEMENT('Y', 0, 20, 0.005),
    SDO_DIM_ELEMENT('M', 0, 20, 0.005) -- Measure dimension
```

```
    ),
  NULL   -- SRID (spatial reference system, also called coordinate system)
);

--------------------------------------------------------------------
-- CREATE THE SPATIAL INDEX --
--------------------------------------------------------------------
CREATE INDEX road_segments_idx ON road_segments(segment_geom)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX;

--------------------------------
-- USE SDO_NET SUBPROGRAMS
--------------------------------

-- This procedure does not use the CREATE_LRS_NETWORK procedure. Instead,
-- the user creates the network tables and populates the network metadata view.
-- Basic steps:
-- 1. Create and populate the node table.
-- 2. Create and populate the link table.
-- 3. Create the path table and paths and links table (for possible
--    future use, before which they will need to be populated).
-- 4. Populate the network metadata (USER_SDO_NETWORK_METADATA).
--    Note: Can be done before or after Steps 1-3.
-- 5. Use various SDO_NET functions and procedures.
-- 6. Use SDO_NET_MEM functions and procedures for analysis and editing.

-- 1. Create and populate the node table.
EXECUTE SDO_NET.CREATE_NODE_TABLE('ROADS_NODES', 'LRS_GEOMETRY', 'NODE_GEOMETRY',
'COST', 1);

-- Populate the node table.

-- N1
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (1, 'N1', 'Y', 1001, 0);

-- N2
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (2, 'N2', 'Y', 1001, 2);

-- N3
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (3, 'N3', 'Y', 1001, 8);

-- N4
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (4, 'N4', 'Y', 1001, 12);

-- N5
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (5, 'N5', 'Y', 1004, 0);

-- N6
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (6, 'N6', 'Y', 1002, 6);

-- N7
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (7, 'N7', 'Y', 1004, 9);
```

```
-- N8
INSERT INTO roads_nodes (node_id, node_name, active, geom_id, measure)
  VALUES (8, 'N8', 'Y', 1002, 9);

-- 2. Create and populate the link table.
EXECUTE SDO_NET.CREATE_LINK_TABLE('ROADS_LINKS', 'LRS_GEOMETRY', 'LINK_GEOMETRY',
'COST', 1);

-- Populate the link table.

-- Route1, Link1
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (101, 'R1L1', 1, 2, 'Y', 3, 1001, 0, 2);

-- Route1, Link2
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (102, 'R1L2', 2, 3, 'Y', 15, 1001, 2, 8);

 -- Route1, Link3
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (103, 'R1L3', 3, 4, 'Y', 10, 1001, 8, 12);

-- Route1, Link4
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (104, 'R1L4', 4, 5, 'Y', 15, 1003, 0, 6);

-- Route1, Link5
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (105, 'R1L5', 5, 6, 'Y', 10, 1004, 0, 4);

-- Route1, Link6
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (106, 'R1L6', 6, 7, 'Y', 7, 1004, 4, 9);

-- Route2, Link1 (cost = 30, a slow drive)
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (201, 'R2L1', 3, 6, 'Y', 30, 1002, 0, 6);

-- Route2, Link2
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (202, 'R2L2', 6, 8, 'Y', 5, 1002, 6, 9);

-- Route3, Link1
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (301, 'R3L1', 5, 8, 'Y', 5, 1003, 6, 11);

-- Route3, Link2
INSERT INTO roads_links (link_id, link_name, start_node_id, end_node_id, active,
   cost, geom_id, start_measure, end_measure)
VALUES (302, 'R3L2', 8, 7, 'Y', 5, 1003, 11, 14.16);
```

```
-- 3. Create the path table (to store created paths) and the path-link
--    table (to store links for each path) for possible future use,
--    before which they will need to be populated.
EXECUTE SDO_NET.CREATE_PATH_TABLE('ROADS_PATHS', 'PATH_GEOMETRY');
EXECUTE SDO_NET.CREATE_PATH_LINK_TABLE('ROADS_PATHS_LINKS');

-- 4. Populate the network metadata (USER_SDO_NETWORK_METADATA).

INSERT INTO user_sdo_network_metadata
    (NETWORK,
     NETWORK_CATEGORY,
     GEOMETRY_TYPE,
     NETWORK_TYPE,
     NO_OF_HIERARCHY_LEVELS,
     NO_OF_PARTITIONS,
     LRS_TABLE_NAME,
     LRS_GEOM_COLUMN,
     NODE_TABLE_NAME,
     NODE_GEOM_COLUMN,
     NODE_COST_COLUMN,
     LINK_TABLE_NAME,
     LINK_GEOM_COLUMN,
     LINK_DIRECTION,
     LINK_COST_COLUMN,
     PATH_TABLE_NAME,
     PATH_GEOM_COLUMN,
     PATH_LINK_TABLE_NAME)
  VALUES (
    'ROADS_NETWORK',  -- Network name
    'SPATIAL',  -- Network category
    'LRS_GEOMETRY',  -- Geometry type
    'Roadways',  -- Network type (user-defined)
    1,  -- No. of levels in hierarchy
    1,  -- No. of partitions
    'ROAD_SEGMENTS',   -- LRS table name
    'SEGMENT_GEOM' ,  -- LRS geometry column
    'ROADS_NODES',  -- Node table name
    'NODE_GEOMETRY',  -- Node geometry column
    'COST',  -- Node cost column
    'ROADS_LINKS',  -- Link table name
    'LINK_GEOMETRY',  -- Link geometry column
    'DIRECTED',  -- Link direction
    'COST',  -- Link cost column
    'ROADS_PATHS',  -- Path table name
    'PATH_GEOMETRY',  -- Path geometry column
    'ROADS_PATHS_LINKS'  -- Paths and links table
    );

-- 5. Use various SDO_NET functions and procedures.

-- Validate the network.
SELECT SDO_NET.VALIDATE_NETWORK('ROADS_NETWORK') FROM DUAL;

-- Validate parts or aspects of the network.
SELECT SDO_NET.VALIDATE_LINK_SCHEMA('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_LRS_SCHEMA('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_NODE_SCHEMA('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_PATH_SCHEMA('ROADS_NETWORK') FROM DUAL;

-- Retrieve various information (GET_xxx and some other functions).
```

```
SELECT SDO_NET.GET_CHILD_LINKS('ROADS_NETWORK', 101) FROM DUAL;
SELECT SDO_NET.GET_CHILD_NODES('ROADS_NETWORK', 1) FROM DUAL;
SELECT SDO_NET.GET_GEOMETRY_TYPE('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_IN_LINKS('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_INVALID_LINKS('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_INVALID_NODES('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_INVALID_PATHS('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_ISOLATED_NODES('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_COST_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_DIRECTION('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_GEOMETRY('ROADS_NETWORK', 103) FROM DUAL;
SELECT SDO_NET.GET_LINK_TABLE_NAME('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LRS_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LRS_LINK_GEOMETRY('ROADS_NETWORK', 103) FROM DUAL;
SELECT SDO_NET.GET_LRS_NODE_GEOMETRY('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_LRS_TABLE_NAME('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NETWORK_CATEGORY('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NETWORK_ID('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NETWORK_NAME(3) FROM DUAL;
SELECT SDO_NET.GET_NETWORK_TYPE('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_HIERARCHY_LEVELS('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_LINKS('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_NODES('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NODE_DEGREE('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_NODE_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NODE_GEOMETRY('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_NODE_IN_DEGREE('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_NODE_OUT_DEGREE('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_NODE_TABLE_NAME('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NODE_COST_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NODE_HIERARCHY_LEVEL('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_OUT_LINKS('ROADS_NETWORK', 3) FROM DUAL;
SELECT SDO_NET.GET_PATH_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_PATH_TABLE_NAME('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_COMPLEX('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_HIERARCHICAL('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_LOGICAL('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_SIMPLE('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_SPATIAL('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.LRS_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.NETWORK_EXISTS('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.SDO_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;
SELECT SDO_NET.TOPO_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;

-- Copy a network.
EXECUTE SDO_NET.COPY_NETWORK('ROADS_NETWORK', 'ROADS_NETWORK2');

-- Create a trigger.
EXECUTE SDO_NET.CREATE_DELETE_TRIGGER('ROADS_NETWORK');

-- 6. Use SDO_NET_MEM functions and procedures for analysis and editing.

-- Network analysis and other operations (SDO_NET_MEM.NETWORK_MANAGER)

DECLARE
  net_mem    VARCHAR2(100);
  res_string VARCHAR2(1000);

  cost        NUMBER;
```

```
  res_numeric NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

  indx1        NUMBER;
  var1_numeric NUMBER;
  var1_array   SDO_NUMBER_ARRAY;

BEGIN

net_mem := 'ROADS_NETWORK';

-- Read in the network.
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK(net_mem, 'TRUE');

-- Validate the network.
res_string := SDO_NET_MEM.NETWORK_MANAGER.VALIDATE_NETWORK_SCHEMA(net_mem);
DBMS_OUTPUT.PUT_LINE('Is network ' || net_mem || ' valid? ' || res_string);

res_string := SDO_NET_MEM.NETWORK_MANAGER.LIST_NETWORKS;
DBMS_OUTPUT.PUT_LINE('The current in-memory network(s) is/are: ' || res_string);

res_numeric := SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS(net_mem);
DBMS_OUTPUT.PUT_LINE('The number of connected components is: ' || res_numeric);

res_array := SDO_NET_MEM.NETWORK_MANAGER.MCST_LINK(net_mem);
DBMS_OUTPUT.PUT('Network ' || net_mem || ' has the following MCST links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHABLE_NODES(net_mem,1);
DBMS_OUTPUT.PUT_LINE('Reachable nodes from 1: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(net_mem,6,3);
DBMS_OUTPUT.PUT_LINE('Path IDs to the nearest 3 neighbors of node 6 are: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', which contains links: ');
  var1_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
    FOR indx1 IN var1_array.FIRST..var1_array.LAST
    LOOP
      var1_numeric := var1_array(indx1);
      DBMS_OUTPUT.PUT(var1_numeric || ' ');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(net_mem,6,3);
DBMS_OUTPUT.PUT_LINE('Path IDs to the nearest 3 neighbors of node 6 are: ');
```

```
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', whose end node is: ');
  var1_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, res_numeric);
  DBMS_OUTPUT.PUT(var1_numeric);
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_string := SDO_NET_MEM.NETWORK_MANAGER.IS_REACHABLE(net_mem,1,5);
DBMS_OUTPUT.PUT_LINE('Can node 1 reach node 5? ' || res_string);

res_array := SDO_NET_MEM.NETWORK_MANAGER.ALL_PATHS(net_mem,1,5,10,200,5);
DBMS_OUTPUT.PUT_LINE('For each path from node 1 to node 5: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric ||
                       ' has the following properties: ');
  cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
  res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_array(indx));
 DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);
END LOOP;

DBMS_OUTPUT.PUT_LINE(' ');
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(net_mem,1,5);
DBMS_OUTPUT.PUT_LINE('The shortest path from node 1 to node 5 is path ID: ' ||
res_numeric);

DBMS_OUTPUT.PUT_LINE('The following are characteristics of this shortest path: ');
cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' has cost: ' || cost);
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);

DBMS_OUTPUT.PUT_LINE(' ');
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH_DIJKSTRA(net_mem,1,5);
DBMS_OUTPUT.PUT_LINE('The shortest Dijkstra path from node 1 to node 5 is ' ||
res_numeric);

DBMS_OUTPUT.PUT_LINE('The following are characteristics of this shortest path: ');
cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);

res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
```

```
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.WITHIN_COST(net_mem,2,20);
DBMS_OUTPUT.PUT('Path IDs to nodes within cost of 40 from node 2: ');
DBMS_OUTPUT.PUT_LINE(' ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', whose end node is: ');
  var1_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, res_numeric);
  DBMS_OUTPUT.PUT(var1_numeric);
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

END;
/

-- Link editing (SDO_NET_MEM.LINK)

DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  res_geom    SDO_GEOMETRY;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN

net_mem := 'ROADS_NETWORK';

-- GET_COST
res_numeric := SDO_NET_MEM.LINK.GET_COST(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The cost of link 104 is: ' || res_numeric);

-- GET_END_MEASURE
res_numeric := SDO_NET_MEM.LINK.GET_END_MEASURE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The end measure of link 104 is: ' || res_numeric);

-- GET_END_NODE_ID
res_numeric := SDO_NET_MEM.LINK.GET_END_NODE_ID(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The end node of link 104 is: ' || res_numeric);

-- GET_GEOM_ID
res_numeric := SDO_NET_MEM.LINK.GET_GEOM_ID(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The geometry ID of link 104 is: ' || res_numeric);

-- GET_GEOMETRY
res_geom := SDO_NET_MEM.LINK.GET_GEOMETRY(net_mem, 104);

-- GET_NAME
res_string := SDO_NET_MEM.LINK.GET_NAME(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The name of link 104 is: ' || res_string);

-- GET_START_MEASURE
res_numeric := SDO_NET_MEM.LINK.GET_START_MEASURE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The start measure of link 104 is: ' || res_numeric);
```

```
-- GET_START_NODE_ID
res_numeric := SDO_NET_MEM.LINK.GET_START_NODE_ID(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The start node of link 104 is: ' || res_numeric);

-- GET_STATE
res_string := SDO_NET_MEM.LINK.GET_STATE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The state of link 104 is: ' || res_string);

-- IS_ACTIVE
res_string := SDO_NET_MEM.LINK.IS_ACTIVE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('Is link 104 active?: ' || res_string);

-- IS_LOGICAL
res_string := SDO_NET_MEM.LINK.IS_LOGICAL(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('Is link 104 a logical link?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.LINK.IS_TEMPORARY(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('Is link 104 temporary?: ' || res_string);

-- SET_COST
-- Set the cost of link 302 to 6.
SDO_NET_MEM.LINK.SET_COST(net_mem, 302, 6);

-- SET_MEASURE
-- Set the measure value of link 302 as from 111 to 114.16.
SDO_NET_MEM.LINK.SET_MEASURE(net_mem, 302, 111, 114.16);

-- SET_NAME
-- Set the name of link 302 to 'My favorite link'.
SDO_NET_MEM.LINK.SET_NAME(net_mem, 302, 'My favorite link');

-- SET_STATE
-- Set the state of link 302 to 'INACTIVE'.
SDO_NET_MEM.LINK.SET_STATE(net_mem, 302, 'INACTIVE');
-- GET_STATE
res_string := SDO_NET_MEM.LINK.GET_STATE(net_mem, 302);
DBMS_OUTPUT.PUT_LINE('The state of link 302 is: ' || res_string);

-- SET_TYPE
-- Set the type of link 302 to 'Normal street'.
SDO_NET_MEM.LINK.SET_TYPE(net_mem, 302, 'Normal street');
-- GET_TYPE
res_string := SDO_NET_MEM.LINK.GET_TYPE(net_mem, 302);
DBMS_OUTPUT.PUT_LINE('The type of link 302 is: ' || res_string);

END;
/

-- Node editing (SDO_NET_MEM.NODE)

DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  res_geom    SDO_GEOMETRY;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;
```

```
BEGIN

net_mem := 'ROADS_NETWORK';

-- GET_COMPONENT_NO
res_numeric := SDO_NET_MEM.NODE.GET_COMPONENT_NO(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The component number of node 3 is: ' || res_numeric);

-- GET_COST
res_numeric := SDO_NET_MEM.NODE.GET_COST(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The cost of node 3 is: ' || res_numeric);

-- GET_GEOM_ID
res_numeric := SDO_NET_MEM.NODE.GET_GEOM_ID(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The geometry ID of node 3 is: ' || res_numeric);

-- GET_GEOMETRY
res_geom := SDO_NET_MEM.NODE.GET_GEOMETRY(net_mem, 3);

-- GET_IN_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_IN_LINK_IDS(net_mem, 3);
DBMS_OUTPUT.PUT('Node 3 has the following inbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_INCIDENT_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_INCIDENT_LINK_IDS(net_mem, 3);
DBMS_OUTPUT.PUT('Node 3 has the following incident links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_MEASURE
res_numeric := SDO_NET_MEM.NODE.GET_MEASURE(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The measure value of node 3 is: ' || res_numeric);

-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The name of node 3 is: ' || res_string);

-- GET_OUT_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_OUT_LINK_IDS(net_mem, 3);
DBMS_OUTPUT.PUT('Node 3 has the following outbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_STATE
res_string := SDO_NET_MEM.NODE.GET_STATE(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The state of node 3 is: ' || res_string);

-- IS_ACTIVE
res_string := SDO_NET_MEM.NODE.IS_ACTIVE(net_mem, 3);
```

```
DBMS_OUTPUT.PUT_LINE('Is node 3 active?: ' || res_string);

-- IS_LOGICAL
res_string := SDO_NET_MEM.NODE.IS_LOGICAL(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('Is node 3 a logical node?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.NODE.IS_TEMPORARY(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('Is node 3 temporary?: ' || res_string);

-- LINK_EXISTS
res_string := SDO_NET_MEM.NODE.LINK_EXISTS(net_mem, 3, 4);
DBMS_OUTPUT.PUT_LINE('Does a link exist between nodes 3 and 4?: ' || res_string);

-- MAKE_TEMPORARY
-- Make node 7 temporary.
SDO_NET_MEM.NODE.MAKE_TEMPORARY(net_mem, 7);

-- SET_COMPONENT_NO
-- Set the component number of node 7 to 987.
SDO_NET_MEM.NODE.SET_COMPONENT_NO(net_mem, 7, 987);

-- SET_COST
-- Set the cost of node 7 to 40.
SDO_NET_MEM.NODE.SET_COST(net_mem, 7, 40);

-- SET_GEOM_ID
-- Set the geometry ID of node 7 to 99.
SDO_NET_MEM.NODE.SET_GEOM_ID(net_mem, 7, 99);

-- SET_MEASURE
-- Set the measure value of node 7 to 30.
SDO_NET_MEM.NODE.SET_MEASURE(net_mem, 7, 30);

-- SET_NAME
-- Set the name of node 7 to 'My favorite node'.
SDO_NET_MEM.NODE.SET_NAME(net_mem, 7, 'My favorite node');
-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME(net_mem, 7);
DBMS_OUTPUT.PUT_LINE('The name of node 7 is: ' || res_string);

-- SET_STATE
-- Set the state of node 7 to 'INACTIVE'.
SDO_NET_MEM.NODE.SET_STATE(net_mem, 7, 'INACTIVE');
-- GET_STATE
res_string := SDO_NET_MEM.NODE.GET_STATE(net_mem, 7);
DBMS_OUTPUT.PUT_LINE('The state of node 7 is: ' || res_string);

-- SET_TYPE
-- Set the type of node 7 to 'Historic site'.
SDO_NET_MEM.NODE.SET_TYPE(net_mem, 7, 'Historic site');
-- GET_TYPE
res_string := SDO_NET_MEM.NODE.GET_TYPE(net_mem, 7);
DBMS_OUTPUT.PUT_LINE('The type of node 7 is: ' || res_string);

END;
/

-- Path editing (SDO_NET_MEM.PATH)
```

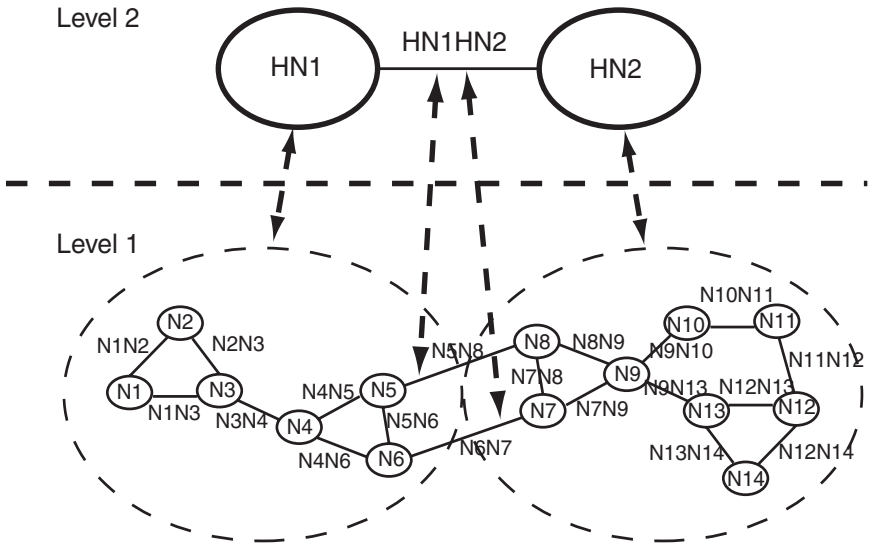
```
DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  res_geom    SDO_GEOMETRY;
  path_id     NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN

net_mem := 'ROADS_NETWORK';

-- Create a path for use with subsequent statements. Here, it is
-- the shortest path between nodes 1 (N1) and 5 (N5).
path_id := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(net_mem,1,5);
DBMS_OUTPUT.PUT_LINE('The shortest path between nodes 1 and 5 is: ' || path_id);

-- GET_LINK_IDS
res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Path ' || path_id || ' has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_COST
res_numeric := SDO_NET_MEM.PATH.GET_COST(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The cost of path ' || path_id || ' is: ' || res_numeric);

-- GET_END_NODE_ID
res_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The end node ID of path ' || path_id || ' is: ' || res_
numeric);

-- GET_GEOMETRY
res_geom := SDO_NET_MEM.PATH.GET_GEOMETRY(net_mem, path_id);
-- doesn't work DBMS_OUTPUT.PUT_LINE('The geometry of path ' || path_id || ' is: '
|| res_geom);

-- GET_LINK_IDS
res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Path ' || path_id || ' has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_NAME
res_string := SDO_NET_MEM.PATH.GET_NAME(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The name of path ' || path_id || ' is: ' || res_string);

-- GET_NO_OF_LINKS
res_numeric := SDO_NET_MEM.PATH.GET_NO_OF_LINKS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The number of links in path ' || path_id || ' is: ' || res_
numeric);

-- GET_NODE_IDS
```

```
res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT('Path ' || path_id || ' has the following nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_START_NODE_ID
res_numeric := SDO_NET_MEM.PATH.GET_START_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The start node ID of path ' || path_id || ' is: ' || res_
numeric);

-- IS_ACTIVE
res_string := SDO_NET_MEM.PATH.IS_ACTIVE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' active?: ' || res_string);

-- IS_CLOSED
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' closed?: ' || res_string);

-- IS_CONNECTED
res_string := SDO_NET_MEM.PATH.IS_CONNECTED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' connected?: ' || res_string);

-- IS_LOGICAL
res_string := SDO_NET_MEM.PATH.IS_LOGICAL(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a logical path?: ' || res_string);

-- IS_SIMPLE
res_string := SDO_NET_MEM.PATH.IS_SIMPLE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a simple path?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.PATH.IS_TEMPORARY(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' temporary?: ' || res_string);

-- SET_NAME
-- Set the name of path to 'My favorite path'.
SDO_NET_MEM.PATH.SET_NAME(net_mem, path_id, 'My favorite path');
-- GET_NAME
res_string := SDO_NET_MEM.PATH.GET_NAME(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The name of path ' || path_id || ' is: ' || res_string);

-- SET_TYPE
-- Set the type of the path to 'Scenic'.
SDO_NET_MEM.PATH.SET_TYPE(net_mem, path_id, 'Scenic');
-- GET_TYPE
res_string := SDO_NET_MEM.PATH.GET_TYPE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The type of path ' || path_id || ' is: ' || res_string);

-- SET_PATH_ID
-- Set (change) the path ID of the path to 6789.
SDO_NET_MEM.PATH.SET_PATH_ID(net_mem, path_id, 6789);

END;
/
```

5.13.4 Logical Hierarchical Network Example (PL/SQL)

This section presents an example of a logical network that contains the nodes and links illustrated in Figure 5–7. Because it is a logical network, there are no spatial geometries associated with it. (Figure 5–7 is essentially the same as Figure 5–3 in Section 5.5, but with the nodes and links labeled.)

Figure 5–7 Nodes and Links for Logical Network Example



As shown in Figure 5–7:

- The network is hierarchical, with two levels. The top level (level 2) consists of two nodes (`HN1` and `HN2`), and the remaining nodes and links are in the bottom level (level 1) of the hierarchy.
- Each node in level 1 is a child node of one of the nodes in level 2. Node `HN1` has the following child nodes: `N1`, `N2`, `N3`, `N4`, `N5`, and `N6`. Node `HN2` has the following child nodes: `N7`, `N8`, `N9`, `N10`, `N11`, `N12`, `N13`, and `N14`.
- One link (`HN1HN2`) links nodes `HN1` and `HN2`, and two links (`N5N8` and `N6N7`) are child links of parent link `HN1HN2`. Note, however, that links are not associated with a specific network hierarchy level.

Example 5–5 does the following:

- Creates and populates the node table.
- Creates and populates the link table.
- Creates and populates the path table and path-link table, for possible future use. (Before an application can use paths, you must populate these two tables.)
- Inserts network metadata into the USER_SDO_NETWORK_METADATA view.
- Uses various SDO_NET functions and procedures.
- Uses SDO_NET_MEM functions and procedures for analysis and editing.

Example 5–5 Logical Network Example (PL/SQL)

```
-- Basic steps:
-- 1. Create and populate the node table.
-- 2. Create and populate the link table.
```

```
-- 3. Create the path table and paths and links table (for possible
--    future use, before which they will need to be populated).
-- 4. Populate the network metadata (USER_SDO_NETWORK_METADATA).
--    Note: Can be done before or after Steps 1-3.
-- 5. Use various SDO_NET functions and procedures.
-- 6. Use SDO_NET_MEM functions and procedures for analysis and editing.

-- 1. Create and populate the node table.
EXECUTE SDO_NET.CREATE_NODE_TABLE('XYZ_NODES', NULL, NULL, NULL, 2);

-- Populate the node table, starting with the highest level in the hierarchy.

-- HN1 (Hierarchy level=2, highest in this network)
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level)
  VALUES (1, 'HN1', 'Y', 2);

-- HN2 (Hierarchy level=2, highest in this network)
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level)
  VALUES (2, 'HN2', 'Y', 2);

-- N1 (Hierarchy level 1, parent node ID = 1 for N1 through N6)
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (101, 'N1', 'Y', 1, 1);

-- N2
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (102, 'N2', 'Y', 1, 1);

-- N3
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (103, 'N3', 'Y', 1, 1);

-- N4
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (104, 'N4', 'Y', 1, 1);

-- N5
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (105, 'N5', 'Y', 1, 1);

-- N6
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (106, 'N6', 'Y', 1, 1);

-- N7 (Hierarchy level 1, parent node ID = 2 for N7 through N14)
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (107, 'N7', 'Y', 1, 2);

-- N8
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (108, 'N8', 'Y', 1, 2);
```

```
-- N9
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (109, 'N9', 'Y', 1, 2);

-- N10
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (110, 'N10', 'Y', 1, 2);

-- N11
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (111, 'N11', 'Y', 1, 2);

-- N12
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (112, 'N12', 'Y', 1, 2);

-- N13
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (113, 'N13', 'Y', 1, 2);

-- N14
INSERT INTO xyz_nodes (node_id, node_name, active, hierarchy_level,
     parent_node_id)
  VALUES (114, 'N14', 'Y', 1, 2);

-- 2. Create and populate the link table.
EXECUTE SDO_NET.CREATE_LINK_TABLE('XYZ_LINKS', NULL, NULL, 'COST', 2);

-- Populate the link table.

-- HN1HN2 (single link in highest hierarchy level: link level = 2)
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level)
  VALUES (1001, 'HN1HN2', 1, 2, 'Y', 2);

-- For remaining links, link level = 1 and cost (10, 20, or 30) varies among
links.
-- N1N2
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1101, 'N1N2', 101, 102, 'Y', 1, 10);

-- N1N3
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1102, 'N1N3', 101, 103, 'Y', 1, 20);

-- N2N3
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1103, 'N2N3', 102, 103, 'Y', 1, 30);

-- N3N4
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
```

```
  VALUES (1104, 'N3N4', 103, 104, 'Y', 1, 10);

-- N4N5
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1105, 'N4N5', 104, 105, 'Y', 1, 20);

-- N4N6
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1106, 'N4N6', 104, 106, 'Y', 1, 30);

-- N5N6
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1107, 'N5N6', 105, 106, 'Y', 1, 10);

-- N5N8 (child of the higher-level link: parent ID = 1001)
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost, parent_link_id)
  VALUES (1108, 'N5N8', 105, 108, 'Y', 1, 20, 1001);

-- N6N7 (child of the higher-level link: parent ID = 1001)
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost, parent_link_id)
  VALUES (1109, 'N6N7', 106, 107, 'Y', 1, 30, 1001);

-- N7N8
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1110, 'N7N8', 107, 108, 'Y', 1, 10);

-- N7N9
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1111, 'N7N9', 107, 109, 'Y', 1, 20);

-- N8N9
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1112, 'N8N9', 108, 109, 'Y', 1, 30);

-- N9N10
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1113, 'N9N10', 109, 110, 'Y', 1, 30);

-- N9N13
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1114, 'N9N13', 109, 113, 'Y', 1, 10);

-- N10N11
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1115, 'N10N11', 110, 111, 'Y', 1, 20);

-- N11N12
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
```

```
  VALUES (1116, 'N11N12', 111, 112, 'Y', 1, 30);

-- N12N13
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1117, 'N12N13', 112, 113, 'Y', 1, 10);

-- N12N14
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1118, 'N12N14', 112, 114, 'Y', 1, 20);

-- N13N14
INSERT INTO xyz_links (link_id, link_name, start_node_id, end_node_id, active,
     link_level, cost)
  VALUES (1119, 'N13N14', 113, 114, 'Y', 1, 30);

-- 3. Create the path table (to store created paths) and the path-link
--    table (to store links for each path) for possible future use,
--    before which they will need to be populated.
EXECUTE SDO_NET.CREATE_PATH_TABLE('XYZ_PATHS', NULL);
EXECUTE SDO_NET.CREATE_PATH_LINK_TABLE('XYZ_PATHS_LINKS');

-- 4. Populate the network metadata (USER_SDO_NETWORK_METADATA).

INSERT INTO user_sdo_network_metadata
    (NETWORK,
     NETWORK_CATEGORY,
     NO_OF_HIERARCHY_LEVELS,
     NO_OF_PARTITIONS,
     NODE_TABLE_NAME,
     LINK_TABLE_NAME,
     LINK_DIRECTION,
     LINK_COST_COLUMN,
     PATH_TABLE_NAME,
     PATH_LINK_TABLE_NAME)
  VALUES (
    'XYZ_NETWORK',  -- Network name
    'LOGICAL',   -- Network category
    2,  -- No. of levels in hierarchy
    1,  -- No. of partitions
    'XYZ_NODES',  -- Node table name
    'XYZ_LINKS',  -- Link table name
    'BIDIRECTED',  -- Link direction
    'COST',  -- Link cost column
    'XYZ_PATHS',  -- Path table name
    'XYZ_PATHS_LINKS'  -- Path-link table name
  );

-- 5. Use various SDO_NET functions and procedures.

-- Validate the network.
SELECT SDO_NET.VALIDATE_NETWORK('XYZ_NETWORK') FROM DUAL;

-- Validate parts or aspects of the network.
SELECT SDO_NET.VALIDATE_LINK_SCHEMA('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_LRS_SCHEMA('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_NODE_SCHEMA('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.VALIDATE_PATH_SCHEMA('XYZ_NETWORK') FROM DUAL;
```

```
-- Retrieve various information (GET_xxx and some other functions).
SELECT SDO_NET.GET_CHILD_LINKS('XYZ_NETWORK', 1001) FROM DUAL;
SELECT SDO_NET.GET_CHILD_NODES('XYZ_NETWORK', 1) FROM DUAL;
SELECT SDO_NET.GET_CHILD_NODES('XYZ_NETWORK', 2) FROM DUAL;
SELECT SDO_NET.GET_IN_LINKS('XYZ_NETWORK', 104) FROM DUAL;
SELECT SDO_NET.GET_LINK_COST_COLUMN('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_DIRECTION('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_LINK_TABLE_NAME('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NETWORK_TYPE('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_HIERARCHY_LEVELS('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_LINKS('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NO_OF_NODES('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.GET_NODE_DEGREE('XYZ_NETWORK', 104) FROM DUAL;
SELECT SDO_NET.GET_NODE_IN_DEGREE('XYZ_NETWORK', 104) FROM DUAL;
SELECT SDO_NET.GET_NODE_OUT_DEGREE('XYZ_NETWORK', 104) FROM DUAL;
SELECT SDO_NET.GET_OUT_LINKS('XYZ_NETWORK', 104) FROM DUAL;
SELECT SDO_NET.GET_PATH_TABLE_NAME('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_HIERARCHICAL('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_LOGICAL('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.IS_SPATIAL('XYZ_NETWORK') FROM DUAL;
SELECT SDO_NET.NETWORK_EXISTS('XYZ_NETWORK') FROM DUAL;

-- Copy a network.
EXECUTE SDO_NET.COPY_NETWORK('XYZ_NETWORK', 'XYZ_NETWORK2');

-- Create a trigger.
EXECUTE SDO_NET.CREATE_DELETE_TRIGGER('XYZ_NETWORK');

-- 6. Use SDO_NET_MEM functions and procedures for analysis and editing.

-- Network analysis and other operations (SDO_NET_MEM.NETWORK_MANAGER)

DECLARE
  net_mem    VARCHAR2(100);
  res_string VARCHAR2(1000);

  cost        NUMBER;
  res_numeric NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

  indx1        NUMBER;
  var1_numeric NUMBER;
  var1_array   SDO_NUMBER_ARRAY;

BEGIN

net_mem := 'XYZ_NETWORK';

-- Read in the network.
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK(net_mem, 'TRUE');

-- Validate the network.
res_string := SDO_NET_MEM.NETWORK_MANAGER.VALIDATE_NETWORK_SCHEMA(net_mem);
DBMS_OUTPUT.PUT_LINE('Is network ' || net_mem || ' valid? ' || res_string);

res_string := SDO_NET_MEM.NETWORK_MANAGER.LIST_NETWORKS;
DBMS_OUTPUT.PUT_LINE('The current in-memory network(s) is/are: ' || res_string);

res_numeric := SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS(net_mem);
```

```
DBMS_OUTPUT.PUT_LINE('The number of connected components is: ' || res_numeric);

res_array := SDO_NET_MEM.NETWORK_MANAGER.MCST_LINK(net_mem);
DBMS_OUTPUT.PUT('Network ' || net_mem || ' has the following MCST links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHABLE_NODES(net_mem,101);
DBMS_OUTPUT.PUT_LINE('Reachable nodes from 101: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHING_NODES(net_mem,101);
DBMS_OUTPUT.PUT_LINE('Nodes from which 101 can be reached: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(net_mem,101,3);
DBMS_OUTPUT.PUT_LINE('Path IDs to the nearest 3 neighbors of node 101 are: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', which contains links: ');
  var1_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
    FOR indx1 IN var1_array.FIRST..var1_array.LAST
    LOOP
      var1_numeric := var1_array(indx1);
      DBMS_OUTPUT.PUT(var1_numeric || ' ');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(net_mem,101,3);
DBMS_OUTPUT.PUT_LINE('Path IDs to the nearest 3 neighbors of node 101 are: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', whose end node is: ');
  var1_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, res_numeric);
  DBMS_OUTPUT.PUT(var1_numeric);
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_string := SDO_NET_MEM.NETWORK_MANAGER.IS_REACHABLE(net_mem,101,105);
DBMS_OUTPUT.PUT_LINE('Can node 101 reach node 105? ' || res_string);

res_array := SDO_NET_MEM.NETWORK_MANAGER.ALL_PATHS(net_mem,101,105,10,200,5);
```

```
DBMS_OUTPUT.PUT_LINE('For each path from node 101 to node 105: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric ||
                       ' has the following properties: ');
  cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
  res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_array(indx));
 DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);
END LOOP;

DBMS_OUTPUT.PUT_LINE(' ');
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(net_mem,101,105);
DBMS_OUTPUT.PUT_LINE('The shortest path from node 101 to node 105 is path ID: ' ||
res_numeric);

DBMS_OUTPUT.PUT_LINE('The following are characteristics of this shortest path: ');
cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' has cost: ' || cost);
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);

res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

DBMS_OUTPUT.PUT_LINE(' ');
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH_DIJKSTRA(net_
mem,101,105);
DBMS_OUTPUT.PUT_LINE('The shortest Dijkstra path from node 101 to node 105 is ' ||
res_numeric);

DBMS_OUTPUT.PUT_LINE('The following are characteristics of this shortest path: ');
cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);

res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.NETWORK_MANAGER.WITHIN_COST(net_mem,102,100);
```

```
DBMS_OUTPUT.PUT('Shortest path IDs to nodes within cost of 100 from node 102: ');
DBMS_OUTPUT.PUT_LINE(' ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', whose end node is: ');
  var1_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, res_numeric);
  DBMS_OUTPUT.PUT(var1_numeric);
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

END;
/

-- Link editing (SDO_NET_MEM.LINK)

DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN

net_mem := 'XYZ_NETWORK';

-- Read in the network.
-- SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK(net_mem, 'TRUE');

-- GET_CHILD_LINKS
res_array := SDO_NET_MEM.LINK.GET_CHILD_LINKS(net_mem, 1001);
DBMS_OUTPUT.PUT('Link 1001 has the following child links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_COST
res_numeric := SDO_NET_MEM.LINK.GET_COST(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The cost of link 1104 is: ' || res_numeric);

-- GET_END_NODE_ID
res_numeric := SDO_NET_MEM.LINK.GET_END_NODE_ID(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The end node of link 1104 is: ' || res_numeric);

-- GET_LEVEL
res_numeric := SDO_NET_MEM.LINK.GET_LEVEL(net_mem, 1001);
DBMS_OUTPUT.PUT_LINE('The hierarchy level of link 1001 is: ' || res_numeric);

-- GET_NAME
res_string := SDO_NET_MEM.LINK.GET_NAME(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The name of link 1104 is: ' || res_string);

-- GET_PARENT_LINK_ID
res_numeric := SDO_NET_MEM.LINK.GET_PARENT_LINK_ID(net_mem, 1108);
DBMS_OUTPUT.PUT_LINE('The parent link of link 1108 is: ' || res_numeric);
```

```
-- GET_SIBLING_LINK_IDS
res_array := SDO_NET_MEM.LINK.GET_SIBLING_LINK_IDS(net_mem, 1108);
DBMS_OUTPUT.PUT('Link 1108 has the following sibling links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_START_NODE_ID
res_numeric := SDO_NET_MEM.LINK.GET_START_NODE_ID(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The start node of link 1104 is: ' || res_numeric);

-- GET_STATE
res_string := SDO_NET_MEM.LINK.GET_STATE(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The state of link 1104 is: ' || res_string);

-- IS_ACTIVE
res_string := SDO_NET_MEM.LINK.IS_ACTIVE(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 active?: ' || res_string);

-- IS_LOGICAL
res_string := SDO_NET_MEM.LINK.IS_LOGICAL(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 a logical link?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.LINK.IS_TEMPORARY(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 temporary?: ' || res_string);

-- SET_COST
-- Set the cost of link 1119 to 40.
SDO_NET_MEM.LINK.SET_COST(net_mem, 1119, 40);

-- SET_END_NODE
-- Set the end node of link 1119 to 109 (N9).
SDO_NET_MEM.LINK.SET_END_NODE(net_mem, 1119, 109);

-- SET_LEVEL
-- Set the hierarchy level of link 1119 to 2.
SDO_NET_MEM.LINK.SET_LEVEL(net_mem, 1119, 2);

-- SET_NAME
-- Set the name of link 1119 to 'My favorite link'.
SDO_NET_MEM.LINK.SET_NAME(net_mem, 1119, 'My favorite link');

-- SET_PARENT_LINK
-- Make link 1001 the parent of link 1119.
SDO_NET_MEM.LINK.SET_PARENT_LINK(net_mem, 1119, 1001);

-- SET_START_NODE
-- Set the start node of link 1119 to 110 (N10).
SDO_NET_MEM.LINK.SET_START_NODE(net_mem, 1119, 110);

-- SET_STATE
-- Set the state of link 1119 to 'INACTIVE'.
SDO_NET_MEM.LINK.SET_STATE(net_mem, 1119, 'INACTIVE');
-- GET_STATE
res_string := SDO_NET_MEM.LINK.GET_STATE(net_mem, 1119);
DBMS_OUTPUT.PUT_LINE('The state of link 1119 is: ' || res_string);
```

```
-- SET_TYPE
-- Set the type of link 1119 to 'Associative'.
SDO_NET_MEM.LINK.SET_TYPE(net_mem, 1119, 'Associative');
-- GET_TYPE
res_string := SDO_NET_MEM.LINK.GET_TYPE(net_mem, 1119);
DBMS_OUTPUT.PUT_LINE('The type of link 1119 is: ' || res_string);

END;
/

-- Node editing (SDO_NET_MEM.NODE)

DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN

net_mem := 'XYZ_NETWORK';

-- GET_ADJACENT_NODE_IDS
res_array := SDO_NET_MEM.NODE.GET_ADJACENT_NODE_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following adjacent nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_CHILD_NODE_IDS
res_array := SDO_NET_MEM.NODE.GET_CHILD_NODE_IDS(net_mem, 1);
DBMS_OUTPUT.PUT('Node 1 has the following child nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_COMPONENT_NO
res_numeric := SDO_NET_MEM.NODE.GET_COMPONENT_NO(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The component number of node 103 is: ' || res_numeric);

-- GET_COST
res_numeric := SDO_NET_MEM.NODE.GET_COST(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The cost of node 103 is: ' || res_numeric);

-- GET_HIERARCHY_LEVEL
res_numeric := SDO_NET_MEM.NODE.GET_HIERARCHY_LEVEL(net_mem, 1);
DBMS_OUTPUT.PUT_LINE('The hierarchy level of node 1 is: ' || res_numeric);

-- GET_IN_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_IN_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following inbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
```

```
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_INCIDENT_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_INCIDENT_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following incident links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The name of node 103 is: ' || res_string);

-- GET_OUT_LINK_IDS
res_array := SDO_NET_MEM.NODE.GET_OUT_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following outbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_PARENT_NODE_ID
res_numeric := SDO_NET_MEM.NODE.GET_PARENT_NODE_ID(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The parent node of node 103 is: ' || res_numeric);

-- GET_SIBLING_NODE_IDS
res_array := SDO_NET_MEM.NODE.GET_SIBLING_NODE_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following sibling nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_STATE
res_string := SDO_NET_MEM.NODE.GET_STATE(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The state of node 103 is: ' || res_string);

-- IS_ACTIVE
res_string := SDO_NET_MEM.NODE.IS_ACTIVE(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 active?: ' || res_string);

-- IS_LOGICAL
res_string := SDO_NET_MEM.NODE.IS_LOGICAL(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 a logical node?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.NODE.IS_TEMPORARY(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 temporary?: ' || res_string);

-- LINK_EXISTS
res_string := SDO_NET_MEM.NODE.LINK_EXISTS(net_mem, 103, 104);
DBMS_OUTPUT.PUT_LINE('Does a link exist between nodes 103 and 104?: ' || res_
string);

-- MAKE_TEMPORARY
-- Make node 114 temporary.
```

```
SDO_NET_MEM.NODE.MAKE_TEMPORARY(net_mem, 114);

-- SET_COMPONENT_NO
-- Set the component number of node 114 to 987.
SDO_NET_MEM.NODE.SET_COMPONENT_NO(net_mem, 114, 987);

-- SET_COST
-- Set the cost of node 114 to 40.
SDO_NET_MEM.NODE.SET_COST(net_mem, 114, 40);

-- SET_HIERARCHY_LEVEL
-- Set the hierarchy level of node 1 to 2.
SDO_NET_MEM.NODE.SET_HIERARCHY_LEVEL(net_mem, 1, 2);

-- SET_NAME
-- Set the name of node 114 to 'My favorite node'.
SDO_NET_MEM.NODE.SET_NAME(net_mem, 114, 'My favorite node');
-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME(net_mem, 114);
DBMS_OUTPUT.PUT_LINE('The name of node 114 is: ' || res_string);

-- SET_PARENT_NODE
-- Make node 1 the parent of node 114.
SDO_NET_MEM.NODE.SET_PARENT_NODE(net_mem, 114, 1);

-- SET_STATE
-- Set the state of node 111 to 'INACTIVE'.
SDO_NET_MEM.NODE.SET_STATE(net_mem, 111, 'INACTIVE');
-- GET_STATE
res_string := SDO_NET_MEM.NODE.GET_STATE(net_mem, 111);
DBMS_OUTPUT.PUT_LINE('The state of node 111 is: ' || res_string);

-- SET_TYPE
-- Set the type of node 114 to 'Research'.
SDO_NET_MEM.NODE.SET_TYPE(net_mem, 114, 'Research');
-- GET_TYPE
res_string := SDO_NET_MEM.NODE.GET_TYPE(net_mem, 114);
DBMS_OUTPUT.PUT_LINE('The type of node 114 is: ' || res_string);

END;
/

-- Path editing (SDO_NET_MEM.PATH)

DECLARE
  net_mem     VARCHAR2(32);
  res_string  VARCHAR2(100);
  res_numeric NUMBER;
  path_id     NUMBER;
  res_array   SDO_NUMBER_ARRAY;
  indx        NUMBER;

BEGIN

net_mem := 'XYZ_NETWORK';

-- Create a path for use with subsequent statements. Here, it is
-- the shortest path between nodes 101 (N1) and 105 (N5).
path_id := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(net_mem,101,105);
DBMS_OUTPUT.PUT_LINE('The shortest path between nodes 101 and 105 is: ' || path_
```

```
id);

-- GET_LINK_IDS
res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Path ' || path_id || ' has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_COST
res_numeric := SDO_NET_MEM.PATH.GET_COST(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The cost of path ' || path_id || ' is: ' || res_numeric);

-- GET_END_NODE_ID
res_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The end node ID of path ' || path_id || ' is: ' || res_
numeric);

-- GET_LINK_IDS
res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Path ' || path_id || ' has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_NO_OF_LINKS
res_numeric := SDO_NET_MEM.PATH.GET_NO_OF_LINKS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The number of links in path ' || path_id || ' is: ' || res_
numeric);

-- GET_NODE_IDS
res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT('Path ' || path_id || ' has the following nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

-- GET_START_NODE_ID
res_numeric := SDO_NET_MEM.PATH.GET_START_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The start node ID of path ' || path_id || ' is: ' || res_
numeric);

-- IS_ACTIVE
res_string := SDO_NET_MEM.PATH.IS_ACTIVE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' active?: ' || res_string);

-- IS_CLOSED
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' closed?: ' || res_string);

-- IS_CONNECTED
res_string := SDO_NET_MEM.PATH.IS_CONNECTED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' connected?: ' || res_string);
```

```
-- IS_LOGICAL
res_string := SDO_NET_MEM.PATH.IS_LOGICAL(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a logical path?: ' || res_string);

-- IS_SIMPLE
res_string := SDO_NET_MEM.PATH.IS_SIMPLE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a simple path?: ' || res_string);

-- IS_TEMPORARY
res_string := SDO_NET_MEM.PATH.IS_TEMPORARY(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' temporary?: ' || res_string);

-- SET_NAME
-- Set the name of path to 'My favorite path'.
SDO_NET_MEM.PATH.SET_NAME(net_mem, path_id, 'My favorite path');
-- GET_NAME
res_string := SDO_NET_MEM.PATH.GET_NAME(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The name of path ' || path_id || ' is: ' || res_string);

-- SET_TYPE
-- Set the type of the path to 'Logical connections'.
SDO_NET_MEM.PATH.SET_TYPE(net_mem, path_id, 'Logical connections');
-- GET_TYPE
res_string := SDO_NET_MEM.PATH.GET_TYPE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The type of path ' || path_id || ' is: ' || res_string);

-- SET_PATH_ID
-- Set (change) the path ID of the path to 6789.
SDO_NET_MEM.PATH.SET_PATH_ID(net_mem, path_id, 6789);

-- Get maximum link, node, path, subpath IDs.
SELECT SDO_NET_MEM.NETWORK.GET_MAX_LINK_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum link ID = ' || res_numeric);
SELECT SDO_NET_MEM.NETWORK.GET_MAX_NODE_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum node ID = ' || res_numeric);
SELECT SDO_NET_MEM.NETWORK.GET_MAX_PATH_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum path ID = ' || res_numeric);
SELECT SDO_NET_MEM.NETWORK.GET_MAX_SUBPATH_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum subpath ID = ' || res_numeric);

END;
/
```

5.13.5 Partitioning and Load on Demand Analysis Examples (PL/SQL, XML, and Java)

This section presents examples of partitioning a network, including related operations, and performing load on demand network analysis. The examples illustrate concepts and techniques explained in Section 5.7.

Example 5–6 partitions a spatial network named NYC_NET. (Assume that this network already exists and its metadata, node, and link tables are populated.)

Example 5–6 Partitioning a Spatial Network

```
exec sdo_net.spatial_partition(
  network->'NYC_NET', -- network name
```

```
  partition_table_name->'NYC_PART$', -- partition table name
  max_num_nodes->5000, -- max. number of nodes per partition
  log_loc->'MDDIR', -- partition log directory
  log_file->'nyc_part.log', --partition log file name
  open_mode->'w', -- partition log file open mode
  link_level->1); -- link level
```

Example 5–7 generates partition BLOBs for the network.

Example 5–7 Generating Partition BLOBs

```
exec sdo_net.generate_partition_blobs(
  network->'NYC_NET', ,-- network name
  link_level ->1, -- link level
  partition_blob_table_name->'NYC_PBLOB$', -- partition blob table name
  includeUserdata->FALSE, -- include user data in partition blobs?
  log_loc->'MYDIR',  -- partition log directory
  log_file->'nyc_part.log', --partition log file name
  open_mode->'a'); -- partition log file open mode
```

Example 5–6 and Example 5–7 generate the necessary partition tables for the NYC_NET network. After executing these examples, you can check the .log file for the current status or any errors encountered during partitioning or BLOB generation.

Example 5–8 shows the XML for configuring the load on demand environment, including the partition cache.

Example 5–8 Configuring the Load on Demand Environment, Including Partition Cache

```
<?xml version="1.0" encoding="UTF-8" ?>
<LODConfigs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.oracle.com/spatial/network/lodLODConfigs.xsd"
     xmlns="http://www.oracle.com/spatial/network/lod">
  <!-- default configuration for networks not configured -->
  <defaultLODConfig>
    <LODConfig>
      <readPartitionFromBlob>false</readPartitionFromBlob>

<partitionBlobTranslator>oracle.spatial.network.lod.PartitionBlobTranslator11g</pa
rtitionBlobTranslator>
      <userDataIO>oracle.spatial.network.lod.LODUserDataIOSDO</userDataIO>
      <cachingPolicy>
        <linkLevelCachingPolicy>
          <linkLevel>1</linkLevel>
          <maxNodes>500000</maxNodes>
          <residentPartitions>-1</residentPartitions>
          <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
        </linkLevelCachingPolicy>
      </cachingPolicy>
    </LODConfig>
  </defaultLODConfig>
  …
  <networkLODConfig>
    <!-- network to be configured -->
    <networkName> NYC_NET </networkName>
    <LODConfig>
       <!- read partitions from partition table or from partition blob table -->
      <readPartitionFromBlob>true</readPartitionFromBlob>

<partitionBlobTranslator>oracle.spatial.network.lod.PartitionBlobTranslator11g</pa
rtitionBlobTranslator>
```

```
      <userDataIO>oracle.spatial.network.lod.LODUserDataIOSDO</userDataIO>
      <cachingPolicy>
        <linkLevelCachingPolicy>
          <linkLevel>1</linkLevel>
          <!-- Maximum number of nodes allowed in cache -->
          <maxNodes>500000</maxNodes>
          <!-- resident partitions -->
          <residentPartitions>-1</residentPartitions>
          <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
        </linkLevelCachingPolicy>
        <linkLevelCachingPolicy>
          <linkLevel>2</linkLevel>
          <maxNodes>500000</maxNodes>
          <residentPartitions>*</residentPartitions>
          <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
        </linkLevelCachingPolicy>
      </cachingPolicy>
    </LODConfig>
  </networkLODConfig>
</LODConfigs>
```

Example 5–9 and Example 5–10 show the Java and PL/SQL APIs, respectively, for reloading the load on demand configuration.

Example 5–9 Reloading the Load on Demand Configuration (Java API)

```
InputStream config = ClassLoader.getSystemResourceAsStream(
                     "netlodcfg.xml");
LODNetworkManager.getConfigManager().loadConfig(config);
```

Example 5–10 Reloading the Load on Demand Configuration (PL/SQL API)

```
EXECUTE SDO_NET.LOAD_CONFIG('WORK_DIR', 'netlodcfg.xml');
```

Example 5–11 returns the estimated size in bytes for a specified network partition.

Example 5–11 Getting Estimated Partition Size

```
SELECT SDO_NET.GET_PARTITION_SIZE (
  NETWORK->'NYC_NET',
  PARTITION_ID->1,
  LINK_LEVEL ->1,
  INCLUDE_USER_DATA->'FALSE',
  INCLUDE_SPATIAL_DATA->'TRUE') FROM DUAL;
```

Example 5–12 uses the load on demand Java API (oracle.spatial.network.lod) to issue a shortest-path query on a network.

Example 5–12 Network Analysis: Shortest Path (LOD Java API)

```
Connection conn = LODNetworkManager.getConnection(dbUrl, dbUser, dbPassword);
// get LOD network IO Adapter
String networkName = "NYC_NET";
NetworkIO reader = LODNetworkManager.getCachedNetworkIO(conn, networkName,
networkName, null);
// get analysis module
NetworkAnalyst analyst = LODNetworkManager.getNetworkAnalyst(reader);
// compute the shortest path
LogicalSubPath path = analyst.shortestPathDijkstra(new PointOnNet(startNodeId),
      new PointOnNet(endNodeId), null);
// print path result
```

```
PrintUtility.print(System.out, path, false, 0, 0);
. . .
```

Example 5–13 uses the XML API (oracle.spatial.network.xml) to issue a shortest-path query on a network. It includes the request and the response.

Example 5–13 Network Analysis: Shortest Path (XML API)

```
<?xml version="1.0" encoding="UTF-8"?>
<ndm:networkAnalysisRequest
    xmlns:ndm="http://xmlns.oracle.com/spatial/network"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gml="http://www.opengis.net/gml">
  <ndm:networkName>NYC_NET</ndm:networkName>
  <ndm:shortestPath>
    <ndm:startPoint>
      <ndm:nodeID>65</ndm:nodeID>
    </ndm:startPoint>
    <ndm:endPoint>
      <ndm:nodeID>115</ndm:nodeID>
    </ndm:endPoint>
    <ndm:subPathRequestParameter>
      <ndm:isFullPath> true </ndm:isFullPath>
      <ndm:startLinkIndex> true </ndm:startLinkIndex>
      <ndm:startPercentage> true </ndm:startPercentage>
      <ndm:endLinkIndex> true </ndm:endLinkIndex>
      <ndm:endPercentage> true </ndm:endPercentage>
      <ndm:geometry>false</ndm:geometry>
    <ndm:pathRequestParameter>
      <ndm:cost> true </ndm:cost>
      <ndm:isSimple> true </ndm:isSimple>
      <ndm:startNodeID>true</ndm:startNodeID>
      <ndm:endNodeID>true</ndm:endNodeID>
      <ndm:noOfLinks>true</ndm:noOfLinks>
      <ndm:linksRequestParameter>
        <ndm:onlyLinkID>true</ndm:onlyLinkID>
      </ndm:linksRequestParameter>
      <ndm:nodesRequestParameter>
        <ndm:onlyNodeID>true</ndm:onlyNodeID>
      </ndm:nodesRequestParameter>
      <ndm:geometry>true</ndm:geometry>
    </ndm:pathRequestParameter>
    </ndm:subPathRequestParameter>
  </ndm:shortestPath>
</ndm:networkAnalysisRequest>

<?xml version = '1.0' encoding = 'UTF-8'?>
<ndm:networkAnalysisResponse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ndm="http://xmlns.oracle.com/spatial/network"
xmlns:gml="http://www.opengis.net/gml">
  <ndm:networkName>NYC_NET</ndm:networkName>
  <ndm:shortestPath>
    <ndm:subPathResponse>
      <ndm:isFullPath>true</ndm:isFullPath>
      <ndm:startLinkIndex>0</ndm:startLinkIndex>
      <ndm:startPercentage>0.0</ndm:startPercentage>
      <ndm:endLinkIndex>17</ndm:endLinkIndex>
      <ndm:endPercentage>1.0</ndm:endPercentage>
      <ndm:pathResponse>
        <ndm:cost>6173.212694405703</ndm:cost>
```

```
        <ndm:isSimple>true</ndm:isSimple>
        <ndm:startNodeID>65</ndm:startNodeID>
        <ndm:endNodeID>115</ndm:endNodeID>
        <ndm:noOfLinks>18</ndm:noOfLinks>
        <ndm:linkIDs>145477046 145477044 145477042 145477039 145476926 145476930
145480892 145480891 145476873 145476871 145477023 145489019 145489020 145476851
145488986 145488987 145476913 145476905
        </ndm:linkIDs>
        <ndm:nodeIDs>65 64 60 57 58 61 71 70 73 87 97 95 91 101 102 104 117 120
115
        </ndm:nodeIDs>
        <ndm:geometry>
          <gml:LineString>
            <gml:coordinates>-71.707462,43.555262 -71.707521,43.555601…
            </gml:coordinates>
          </gml:LineString>
        </ndm:geometry>
      </ndm:pathResponse>
    </ndm:subPathResponse>
  </ndm:shortestPath>
</ndm:networkAnalysisResponse>
```

Additional examples of using load on demand analysis with partitioned networks are included in the demo files, described in Section 5.14.

5.13.6 User-Defined Data Example (PL/SQL and Java)

This section presents an example of using network user-defined data, which is the information (not related to connectivity) that users want to associate with a network representation. The USER_SDO_NETWORK_USER_DATA and ALL_SDO_NETWORK_USER_DATA metadata views (described in Section 5.10.3) contain information about user-defined data.

Example 5–14 inserts link-related user-defined data into the network metadata.

Example 5–14 Inserting User-Defined Data into Network Metadata

```
-- Insert link user data named 'interaction' of
-- type varchar2 (50) in network 'bi_test'.
--'interaction' is a column of type varchar2(50) in the link table of network 'bi_
test'.

insert into user_sdo_network_user_data (network,table_type,data_name,data_
type,data_length) values                                    ('bi_
test','LINK','interaction','VARCHAR2',50) ;

-- insert link user data named 'PROB' of type Number.
--'PROB' is a column of type NUMBER in the link table of network 'bi_test'.

insert into user_sdo_network_user_data (network,table_type,data_name,data_type)
values ('bi_test','LINK','PROB','NUMBER') ;
```

After a network or network partition is loaded, user-defined data is available in Java representations. You can access user-defined data through the `getUserData` and `setUserData` methods for the `Node`, `Link`, `Path`, and `SubPath` interfaces. For example:

```
String interaction = (String)link.getUserData("interaction");
double prob = ((Double)link.getUserData("Prob")).doubleValue();
```

5.14 Network Data Model Documentation and Demo Files

Oracle Spatial provides several network data model supplementary documentation files, as well as demo files that you can use to reinforce your learning and to create models for coding certain operations. Some of these files are for the Network Editor, an unsupported tool that enables you to visualize and edit network data. (The Network Editor interface is shown in Figure 5–1 in Section 5.1.) If you installed the demo files from the Oracle Database Examples media (see *Oracle Database Examples Installation Guide*), those related to the network data model are under the following directory:

- The supplementary documentation files are installed under the `$ORACLE_HOME/md/doc` directory.
- If you installed the demo files from the Oracle Database Examples media, these files and additional code examples are installed under the `$ORACLE_HOME/md/demo` directory.

Each directory and most of the subdirectories have a `README.txt` file that explains the purpose of the files, and many files contain internal explanatory comments.

The `$ORACLE_HOME/md/doc` directory contains supplementary documentation.The `sdondmxml.zip` file in that directory contains XML Schema (.xsd) files and the default load on demand configuration XML file.

If you installed the demo files from the Oracle Database Examples media, the `$ORACLE_HOME/md/demo/network` directory tree includes additional examples (network data, Java, PL/SQL, and SQL*Loader) , as well as files for setting up and using two graphical tools: a network editor and a load on demand analysis viewer. Both tools are provided as a convenience, and are not officially supported by Oracle.

5.14.1 Network Example Files

The `$ORACLE_HOME/md/demo/network/examples` directory tree contains the following kinds of examples: network data, Java, PL/SQL, and SQL*Loader.

5.14.2 Network Editor Tool

The Network Editor enables you to visualize and edit network data. (The Network Editor interface is shown in Figure 5–1 in Section 5.1.) The files for the Network Editor are in the following directory:

$ORACLE_HOME`/md/demo/network/editor`

Read the `README.txt` file in that directory for information about this tool. Sample data for the network data model is under the following directory:

$ORACLE_HOME`/md/demo/network/examples/data`

For example, Hillsborough (New Hampshire, United States) sample network data is in the following file:

$ORACLE_HOME`/md/demo/network/examples/data/hillsborough_network.dmp`

Read the `README.txt` file in that directory for information about loading the Hillsborough sample network data into the database.

5.14.3 Load on Demand Analysis Viewer Tool

A viewer tool is provided to help you visualize the results of load on demand analysis. This viewer is a standalone Java application built on top of the load on demand Java API and Oracle MapViewer. It can perform most load on demand analysis functions and provides simple viewing capabilities.

The files for the load on demand analysis viewer tool Network Editor are in the following directory:

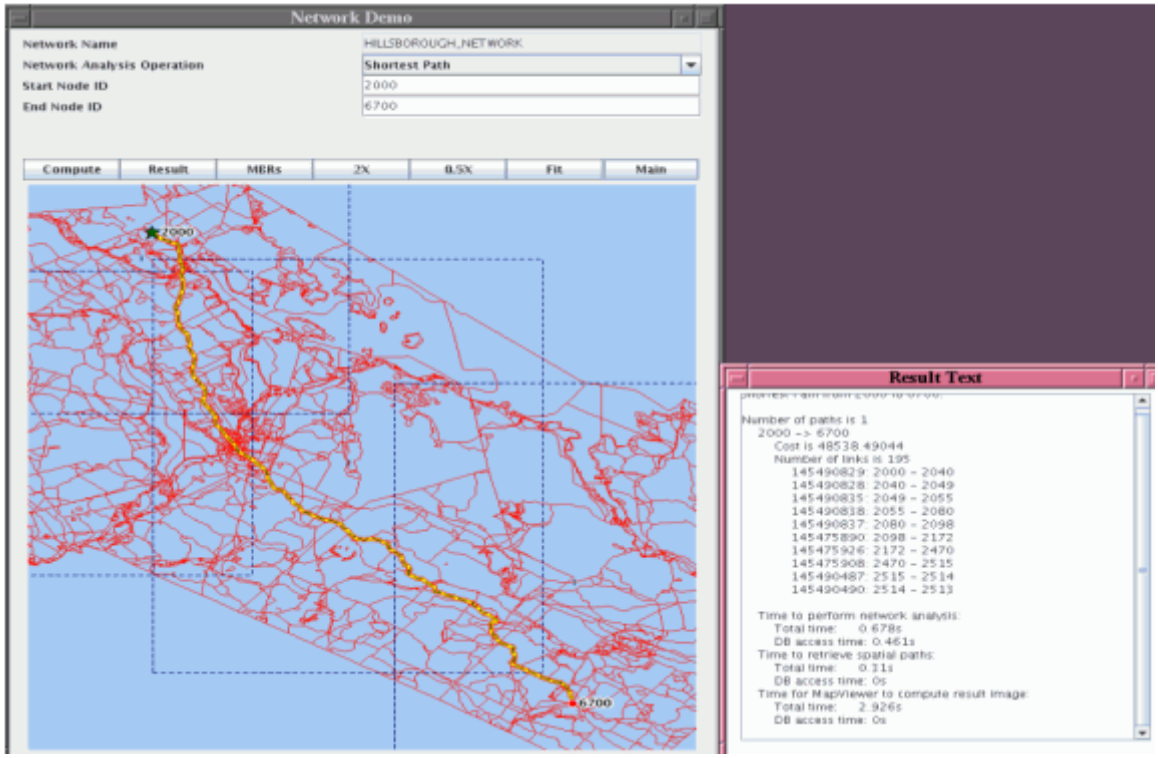
$ORACLE_HOME/md/demo/network/lodviewer

To use the viewer tool, first enter database connection information for the database where the network data model (including partition information) is stored. Next, enter MapViewer connection information. For information on how to install and use MapViewer, see the *MapViewer User's Guide*.

After the connections are validated, you can perform analysis operations on the selected network. Zoom-in, zoom-out, and zoom-window functions are available for viewing analysis results. A text window prints out analysis results and statistics.

Figure 5–8 shows a shortest-path analysis result in the load on demand analysis viewer.

Figure 5–8 Load on Demand Analysis Viewer Tool



5.15 README File for Spatial and Related Features

A `README.txt` file supplements the information in the following manuals: *Oracle Spatial Developer's Guide*, *Oracle Spatial GeoRaster Developer's Guide*, and *Oracle Spatial Topology and Network Data Models Developer's Guide* (this manual). This file is located at:

$ORACLE_HOME/md/doc/README.txt

6

SDO_NET Package Subprograms

The MDSYS.SDO_NET package contains subprograms (functions and procedures) for managing networks. To use the subprograms in this chapter, you must understand the conceptual information in Chapter 5.

For a listing of the subprograms grouped in logical categories, see Section 5.11.1. The rest of this chapter provides reference information about the subprograms, listed in alphabetical order.

SDO_NET.COMPUTE_PATH_GEOMETRY

Format

```
SDO_NET.COMPUTE_PATH_GEOMETRY(
   network    IN VARCHAR2,
   path_id    IN NUMBER,
   tolerance  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the spatial geometry for a path.

Parameters

network
Network name.

path_id
Path ID number.

tolerance
Tolerance value associated with geometries in the network. (Tolerance is explained in Chapter 1 of *Oracle Spatial Developer's Guide*.) This value should be consistent with the tolerance values of the geometries in the link table and node table for the network.

Usage Notes

This function computes and returns the SDO_GEOMETRY object for the specified path.

This function and the SDO_NET_MEM.PATH.COMPUTE_GEOMETRY procedure (documented in Chapter 6) both compute a path geometry, but they have the following differences:

- The SDO_NET.COMPUTE_PATH_GEOMETRY function computes the path from the links in the database, and does not use a network memory object. It returns the path geometry.
- The SDO_NET_MEM.PATH.COMPUTE_GEOMETRY procedure computes the path using a network memory object that has been loaded. It does not return the path geometry; you must use the SDO_NET_MEM.PATH.GET_GEOMETRY function to get the geometry.

Examples

The following example computes and returns the spatial geometry of the path with path ID 1 in the network named `SDO_NET1`, using a tolerance value of 0.005. The returned path geometry is a straight line from (1,1) to (15,1) because this path consists of a single link.

```
SELECT SDO_NET.COMPUTE_PATH_GEOMETRY('SDO_NET1', 1, 0.005) FROM DUAL;

SDO_NET.COMPUTE_PATH_GEOMETRY('SDO_NET1',1,0.005)(SDO_GTYPE, SDO_SRID, SDO_POINT
--------------------------------------------------------------------------------
```

```
SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
1, 1, 15, 1))
```

SDO_NET.COPY_NETWORK

Format

```
SDO_NET.COPY_NETWORK(
   source_network  IN VARCHAR2,
   target_network   IN VARCHAR2);
```

Description

Creates a copy of a network, including its metadata tables.

Parameters

source_network
Name of the network to be copied.

target_network
Name of the network to be created as a copy of `source_network`.

Usage Notes

This procedure creates an entry in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1) for `target_network` that has the same information as for `source_network`, except for the new network name.

This procedure also creates a new node table, link table, and path table (if a path table exists for `source_network`) for `target_network` based on the metadata and data in these tables for `source_network`. These tables have names in the form <*target-network*>_NODE$, <*target-network*>_LINK$, and <*target-network*>_PATH$. For example, if `target_network` has the value `ROADS_NETWORK2` and if `source_network` has a path table, the names of the created metadata tables are ROADS_NETWORK2_NODE$, ROADS_NETWORK2_LINK$, and ROADS_NETWORK2_PATH$.

Examples

The following example creates a new network named `ROADS_NETWORK2` that is a copy of the network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET.COPY_NETWORK('ROADS_NETWORK', 'ROADS_NETWORK2');
```

SDO_NET.CREATE_LINK_TABLE

Format

```
SDO_NET.CREATE_LINK_TABLE(
   table_name              IN VARCHAR2,
   geom_type               IN VARCHAR2,
   geom_column             IN VARCHAR2,
   cost_column             IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER);
```

Description

Creates a link table for a network.

Parameters

table_name
Name of the link table.

geom_type
For a spatial network, specify a value indicating the geometry type of links: `SDO_GEOMETRY` for non-LRS SDO_GEOMETRY objects, `LRS_GEOMETRY` for LRS SDO_GEOMETRY objects, or `TOPO_GEOMETRY` for SDO_TOPO_GEOMETRY objects.

geom_column
For a spatial network, the name of the column containing the geometry objects associated with the links.

cost_column
Name of the column containing the cost values to be associated with the links.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

Usage Notes

The link table is described in Section 5.9.2.

Examples

The following example creates a link table named ROADS_LINKS, with a geometry column named LINK_GEOMETRY that will contain LRS geometries, a cost column named COST, and a single hierarchy level.

```
EXECUTE SDO_NET.CREATE_LINK_TABLE('ROADS_LINKS', 'LRS_GEOMETRY', 'LINK_GEOMETRY',
'COST', 1);
```

SDO_NET.CREATE_LOGICAL_NETWORK

Format

```
SDO_NET.CREATE_LOGICAL_NETWORK(
   network                IN VARCHAR2,
   no_of_hierarchy_levels IN NUMBER,
   is_directed            IN BOOLEAN,
   node_with_cost         IN BOOLEAN DEFAULT FALSE);
```

or

```
SDO_NET.CREATE_LOGICAL_NETWORK(
   network                IN VARCHAR2,
   no_of_hierarchy_levels IN NUMBER,
   is_directed            IN BOOLEAN,
   node_table_name        IN VARCHAR2,
   node_cost_column       IN VARCHAR2,
   link_table_name        IN VARCHAR2,
   link_cost_column       IN VARCHAR2,
   path_table_name        IN VARCHAR2,
   path_link_table_name   IN VARCHAR2);
```

Description

Creates a logical network, creates all necessary tables, and updates the network metadata.

Parameters

network
Network name.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A Boolean value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

node_with_cost
A Boolean value. `TRUE` causes a column named COST to be included in the <*network-name*>_NODE$ table; `FALSE` (the default) causes a column named COST not to be included in the <*network-name*>_NODE$ table.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, a node table named <*network-name*>_NODE$ is created.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, the geometry column is named COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, a link table named <*network-name*>_LINK$ is created.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, the geometry column is named COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, a path table named <*network-name*>_PATH$ is created.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

Usage Notes

This procedure provides a convenient way to create a logical network when the node, link, and optional related tables do not already exist. The procedure creates the network; creates the node, link, path, and path-link tables for the network; and inserts the appropriate information in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

An exception is generated if any of the tables to be created already exists.

The procedure has two formats. The simpler format creates the tables using default values for the table name and the cost column name. The other format lets you specify names for the tables and the cost column.

As an alternative to using this procedure, you can create the network using the SDO_NET.CREATE_LOGICAL_NETWORK procedure; create the tables using the SDO_NET.CREATE_NODE_TABLE, SDO_NET.CREATE_LINK_TABLE, SDO_NET.CREATE_PATH_TABLE, and SDO_NET.CREATE_PATH_LINK_TABLE procedures; and insert the appropriate row in the USER_SDO_NETWORK_METADATA view.

Examples

The following example creates a directed logical network named `LOG_NET1`. The example creates the LOG_NET1_NODE$, LOG_NET1_LINK$,LOG_NET1_PATH$, and LOG_NET1_PLINK$ tables, and updates the xxx_SDO_NETWORK_METADATA views. Both the node and link tables contain a cost column named COST.

```
EXECUTE SDO_NET.CREATE_LOGICAL_NETWORK('LOG_NET1', 1, TRUE, TRUE);
```

SDO_NET.CREATE_LRS_NETWORK

Format

```
SDO_NET.CREATE_LRS_NETWORK(
   network                 IN VARCHAR2,
   lrs_table_name          IN VARCHAR2,
   lrs_geom_column         IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_with_cost          IN BOOLEAN DEFAULT FALSE,
   is_complex              IN BOOLEAN DEFAULT FALSE);
```

or

```
SDO_NET.CREATE_LRS_NETWORK(
   network                 IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_table_name         IN VARCHAR2,
   node_cost_column        IN VARCHAR2,
   link_table_name         IN VARCHAR2,
   link_cost_column        IN VARCHAR2,
   lrs_table_name          IN VARCHAR2,
   lrs_geom_column         IN VARCHAR2,
   path_table_name         IN VARCHAR2,
   path_geom_column        IN VARCHAR2,
   path_link_table_name    IN VARCHAR2,
   is_complex              IN BOOLEAN DEFAULT FALSE);
```

Description

Creates a spatial network containing LRS SDO_GEOMETRY objects, creates all necessary tables, and updates the network metadata.

Parameters

network
Network name.

lrs_table_name
Name of the table containing the LRS geometry column.

lrs_geom_column
Name of the column in `lrs_table_name` that contains LRS geometries (that is, SDO_GEOMETRY objects that include measure information for linear referencing).

is_directed
A Boolean value. TRUE indicates that the links are directed; FALSE indicates that the links are undirected (not directed).

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

node_with_cost
A Boolean value. TRUE causes a column named COST to be included in the <*network-name*>_NODE$ table; FALSE (the default) causes a column named COST not to be included in the <*network-name*>_NODE$ table.

is_complex
Reserved for future use. Ignored for the current release.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, a node table named <*network-name*>_NODE$ is created.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, the geometry column is named COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, a link table named <*network-name*>_LINK$ is created.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, the geometry column is named COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, a path table named <*network-name*>_PATH$ is created.

path_geom_column
Name of the geometry column in the path table. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, the geometry column is named GEOMETRY.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

Usage Notes

This procedure provides a convenient way to create a spatial network of LRS geometries when the node, link, and optional related tables do not already exist. The procedure creates the network; creates the node, link, path, and path-link tables for the

network; and inserts the appropriate information in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

An exception is generated if any of the tables to be created already exists.

The procedure has two formats. The simpler format creates the tables using default values for the table name and the geometry and cost column names. The other format lets you specify names for the tables and the geometry and cost columns.

As an alternative to using this procedure, you can create the network using the SDO_NET.CREATE_LRS_NETWORK procedure; create the tables using the SDO_NET.CREATE_NODE_TABLE, SDO_NET.CREATE_LINK_TABLE, SDO_NET.CREATE_PATH_TABLE, and SDO_NET.CREATE_PATH_LINK_TABLE procedures; and insert the appropriate row in the USER_SDO_NETWORK_METADATA view.

Examples

The following example creates a directed spatial network named `LRS_NET1`. The LRS geometries are in the column named LRS_GEOM in the table named LRS_TAB. The example creates the LRS_NET1_NODE$, LRS_NET1_LINK$, LRS_NET1_PATH$, and LRS_NET1_PLINK$ tables, and updates the xxx_SDO_NETWORK_METADATA views. All geometry columns are named GEOMETRY. Both the node and link tables contain a cost column named COST.

```
EXECUTE SDO_NET.CREATE_LRS_NETWORK('LRS_NET1', 'LRS_TAB', 'LRS_GEOM', 1, TRUE,
TRUE);
```

SDO_NET.CREATE_LRS_TABLE

Format

```
SDO_NET.CREATE_LRS_TABLE(
   table_name  IN VARCHAR2,
   geom_column  IN VARCHAR2);
```

Description

Creates a table for storing Oracle Spatial linear referencing system (LRS) geometries.

Parameters

table_name
Name of the table containing the geometry column specified in `geom_column`.

geom_column
Name of the column (of type SDO_GEOMETRY) to contain geometry objects.

Usage Notes

This procedure creates a table named `table_name` with two columns: GEOM_ID of type NUMBER and `geom_column` of type SDO_GEOMETRY.

Although the created table does not need to be used to store LRS geometries, the procedure is intended as a convenient method for creating a table to store such geometries. You will probably want to modify the table to add other columns before you store data in the table.

Examples

The following example creates a table named HIGHWAYS with a geometry column named GEOM.

```
EXECUTE SDO_NET.CREATE_LRS_TABLE('HIGHWAYS', 'GEOM');

PL/SQL procedure successfully completed.

DESCRIBE highways
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 GEOM_ID                                   NOT NULL NUMBER
 GEOM                                               MDSYS.SDO_GEOMETRY
```

SDO_NET.CREATE_NODE_TABLE

Format

```
SDO_NET.CREATE_NODE_TABLE(
   table_name              IN VARCHAR2,
   geom_type               IN VARCHAR2,
   geom_column             IN VARCHAR2,
   cost_column             IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_complex              IN BOOLEAN DEFAULT FALSE,
   storage_parameters      IN VARCHAR2 DEFAULT NULL);
```

or

```
SDO_NET.CREATE_NODE_TABLE(
   table_name              IN VARCHAR2,
   geom_type               IN VARCHAR2,
   geom_column             IN VARCHAR2,
   cost_column             IN VARCHAR2,
   partition_column        IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_complex              IN BOOLEAN DEFAULT FALSE,
   storage_parameters      IN VARCHAR2 DEFAULT NULL);
```

Description

Creates a node table.

Parameters

table_name
Name of the node table.

geom_type
For a spatial network, specify a value indicating the geometry type of nodes: `SDO_GEOMETRY` for non-LRS SDO_GEOMETRY objects, `LRS_GEOMETRY` for LRS SDO_GEOMETRY objects, or `TOPO_GEOMETRY` for SDO_TOPO_GEOMETRY objects.

geom_column
For a spatial network, the name of the column containing the geometry objects associated with the nodes.

cost_column
Name of the column containing the cost values to be associated with the nodes.

partition_column
Name of the column containing the partition ID values to be associated with the nodes.

no_of_hierarchy_levels
Number of hierarchy levels for nodes in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_complex
Reserved for future use. Ignored for the current release.

storage_parameters
Physical storage parameters used internally to create the <network-name>_NODE$ table (described in Section 5.9.1). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K)`. If you do not specify this parameter, the default physical storage values are used.

Usage Notes

This procedure has two formats, one without the `partition_column` parameter and one with the `partition_column` parameter.

The node table is described in Section 5.9.1.

Examples

The following example creates a node table named ROADS_NODES with a geometry column named NODE_GEOMETRY that will contain LRS geometries, no cost column, and a single hierarchy level.

```
EXECUTE SDO_NET.CREATE_NODE_TABLE('ROADS_NODES', 'LRS_GEOMETRY', 'NODE_GEOMETRY',
NULL, 1);
```

SDO_NET.CREATE_PARTITION_TABLE

Format

```
SDO_NET.CREATE_PARTITION_TABLE(
   table_name  IN VARCHAR2);
```

Description

Creates a partition table.

Parameters

table_name
Name of the partition table.

Usage Notes

The partition table is described in Section 5.9.6.

For information about using partitioned networks to perform analysis using the load on demand approach, see Section 5.7.

Examples

The following example creates a partition table named MY_PART_TAB.

```
EXECUTE SDO_NET.CREATE_PARTITION_TABLE('MY_PART_TAB');
```

SDO_NET.CREATE_PATH_LINK_TABLE

Format

```
SDO_NET.CREATE_PATH_LINK_TABLE(
   table_name  IN VARCHAR2);
```

Description

Creates a path-link table, that is, a table with a row for each link in each path in the path table.

Parameters

table_name
Name of the path-link table.

Usage Notes

The path-link table is described in Section 5.9.4.

To use paths with a network, you must populate the path-link table.

Examples

The following example creates a path-link table named ROADS_PATHS_LINKS.

```
EXECUTE SDO_NET.CREATE_PATH_LINK_TABLE('ROADS_PATHS_LINKS');
```

SDO_NET.CREATE_PATH_TABLE

Format

```
SDO_NET.CREATE_PATH_TABLE(
   table_name  IN VARCHAR2,
   geom_column  IN VARCHAR2);
```

Description

Creates a path table.

Parameters

table_name
Name of the path table.

geom_column
For a spatial network, name of the column containing the geometry objects associated with the paths.

Usage Notes

The path table is described in Section 5.9.3.

To use paths with a network, after you create the path table, you must create the path-link table using the SDO_NET.CREATE_PATH_LINK_TABLE procedure, and populate the path-link table.

Examples

The following example creates a path table named ROADS_PATHS that contains a geometry column named PATH_GEOMETRY.

```
EXECUTE SDO_NET.CREATE_PATH_TABLE('ROADS_PATHS', 'PATH_GEOMETRY');
```

SDO_NET.CREATE_SDO_NETWORK

Format

```
SDO_NET.CREATE_SDO_NETWORK(
   network                 IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_with_cost          IN BOOLEAN DEFAULT FALSE);
```

or

```
SDO_NET.CREATE_SDO_NETWORK(
   network                 IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_table_name         IN VARCHAR2,
   node_geom_column        IN VARCHAR2,
   node_cost_column        IN VARCHAR2,
   link_table_name         IN VARCHAR2,
   link_geom_column        IN VARCHAR2,
   link_cost_column        IN VARCHAR2,
   path_table_name         IN VARCHAR2,
   path_geom_column        IN VARCHAR2,
   path_link_table_name    IN VARCHAR2);
```

Description

Creates a spatial network containing non-LRS SDO_GEOMETRY objects, creates all necessary tables, and updates the network metadata.

Parameters

network
Network name.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A Boolean value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

node_with_cost
A Boolean value. `TRUE` causes a column named COST to be included in the <*network-name*>_NODE$ table; `FALSE` (the default) causes a column named COST not to be included in the <*network-name*>_NODE$ table.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, a node table named <*network-name*>_NODE$ is created.

node_geom_column
Name of the geometry column in the node table. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, the geometry column is named GEOMETRY.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, the geometry column is named COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, a link table named <*network-name*>_LINK$ is created.

link_geom_column
Name of the geometry column in the link table. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, the geometry column is named GEOMETRY.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, the geometry column is named COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, a path table named <*network-name*>_PATH$ is created.

path_geom_column
Name of the geometry column in the path table. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, the geometry column is named GEOMETRY.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

Usage Notes

This procedure provides a convenient way to create a spatial network when the node, link, and optional related tables do not already exist. The procedure creates the network; creates the node, link, path, and path-link tables for the network; and inserts the appropriate information in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

An exception is generated if any of the tables to be created already exists.

The procedure has two formats. The simpler format creates the tables using default values for the table name and the geometry and cost column names. The other format lets you specify names for the tables and the geometry and cost columns.

As an alternative to using this procedure, you can create the network using the SDO_NET.CREATE_SDO_NETWORK procedure; create the tables using the SDO_NET.CREATE_NODE_TABLE, SDO_NET.CREATE_LINK_TABLE, SDO_NET.CREATE_PATH_TABLE, and SDO_NET.CREATE_PATH_LINK_TABLE procedures; and insert the appropriate row in the USER_SDO_NETWORK_METADATA view.

Examples

The following example creates a directed spatial network named `SDO_NET1`. The example creates the SDO_NET1_NODE$, SDO_NET1_LINK$, SDO_NET1_PATH$, and SDO_NET1_PLINK$ tables, and updates the xxx_SDO_NETWORK_METADATA views. All geometry columns are named GEOMETRY. Both the node and link tables contain a cost column named COST.

```
EXECUTE SDO_NET.CREATE_SDO_NETWORK('SDO_NET1', 1, TRUE, TRUE);
```

SDO_NET.CREATE_SUBPATH_TABLE

Format

```
SDO_NET.CREATE_PATH_TABLE(
   table_name          IN VARCHAR2,
   geom_column         IN VARCHAR2,
   storage_parameters  IN VARCHAR2 DEFAULT NULL);
```

Description

Creates a subpath table.

Parameters

table_name
Name of the subpath table.

geom_column
For a spatial network, name of the column containing the geometry objects associated with the subpaths.

storage_parameters
Physical storage parameters used internally to create the subpath table (described in Section 5.9.1). Must be a valid string for use with the CREATE TABLE statement. For example: `TABLESPACE tbs_3 STORAGE (INITIAL 100K NEXT 200K)`. If you do not specify this parameter, the default physical storage values are used.

Usage Notes

The subpath table is described in Section 5.9.5.

To use subpaths with a network, you must create one or more path tables and their associated path-link tables.

Examples

The following example creates a subpath table named ROADS_SUBPATHS that contains a geometry column named SUBPATH_GEOMETRY.

```
EXECUTE SDO_NET.CREATE_SUBPATH_TABLE('ROADS_SUBPATHS', 'SUBPATH_GEOMETRY');
```

SDO_NET.CREATE_TOPO_NETWORK

Format

```
SDO_NET.CREATE_TOPO_NETWORK(
   network                 IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_with_cost          IN BOOLEAN DEFAULT FALSE);
```

or

```
SDO_NET.CREATE_TOPO_NETWORK(
   network                 IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN BOOLEAN,
   node_table_name         IN VARCHAR2,
   node_cost_column        IN VARCHAR2,
   link_table_name         IN VARCHAR2,
   link_cost_column        IN VARCHAR2,
   path_table_name         IN VARCHAR2,
   path_geom_column        IN VARCHAR2,
   path_link_table_name    IN VARCHAR2);
```

Description

Creates a spatial topology network containing SDO_TOPO_GEOMETRY objects, creates all necessary tables, and updates the network metadata.

Parameters

network
Network name.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A Boolean value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

node_with_cost
A Boolean value. `TRUE` causes a column named COST to be included in the <*network-name*>_NODE$ table; `FALSE` (the default) causes a column named COST not to be included in the <*network-name*>_NODE$ table.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, a node table named <*network-name*>_NODE$ is created.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If you use the format that does not specify this parameter, the geometry column is named COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, a link table named <*network-name*>_LINK$ is created.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If you use the format that does not specify this parameter, the geometry column is named COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, a path table named <*network-name*>_PATH$ is created.

path_geom_column
Name of the geometry column in the path table. (The path table is explained in Section 5.9.3.) If you use the format that does not specify this parameter, the geometry column is named GEOMETRY.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

Usage Notes

This procedure provides a convenient way to create a spatial network when the node, link, and optional related tables do not already exist. The procedure creates the network; creates the node, link, path, and path-link tables for the network; and inserts the appropriate information in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1). The node and link tables contain a topology geometry column named TOPO_GEOMETRY of type SDO_TOPO_GEOMETRY.

An exception is generated if any of the tables to be created already exists.

The procedure has two formats. The simpler format creates the tables using default values for the table name and the geometry and cost column names. The other format lets you specify names for the tables and the geometry and cost columns.

As an alternative to using this procedure, you can create the network using the SDO_NET.CREATE_TOPO_NETWORK procedure; create the tables using the SDO_NET.CREATE_NODE_TABLE, SDO_NET.CREATE_LINK_TABLE, SDO_NET.CREATE_PATH_TABLE, and SDO_NET.CREATE_PATH_LINK_TABLE procedures; and insert the appropriate row in the USER_SDO_NETWORK_METADATA view.

Examples

The following example creates a directed spatial topology geometry network named TOPO_NET1. The example creates the TOPO_NET1_NODE$, TOPO_NET1_LINK$, TOPO_NET1_PATH$, and TOPO_NET1_PLINK$ tables, and updates the xxx_SDO_NETWORK_METADATA views. The topology geometry columns are named TOPO_GEOMETRY. Both the node and link tables contain a cost column named COST.

```
EXECUTE SDO_NET.CREATE_TOPO_NETWORK('TOPO_NET1', 1, TRUE, TRUE);
```

SDO_NET.DELETE_LINK

Format

```
SDO_NET.DELETE_LINK(
   network  IN VARCHAR2,
   link_id  IN NUMBER);
```

Description

Deletes a link and all dependent network elements.

Parameters

network
Network name.

link_id
ID of the link to delete.

Usage Notes

This procedure deletes the specified link from the link table (described in Section 5.9.2), and it deletes any other network elements that depend on this link. For example, if the specified link is included in any paths and subpaths, those paths and subpaths are deleted also.

Examples

The following example deletes the link in the SDO_NET2 network whose link ID is 1.

```
SELECT SDO_NET.DELETE_LINK('SDO_NET2', 1);
```

SDO_NET.DELETE_NODE

Format

```
SDO_NET.DELETE_NODE(
   network  IN VARCHAR2,
   node_id  IN NUMBER);
```

Description

Deletes a node and all dependent network elements.

Parameters

network
Network name.

node_id
ID of the node to delete.

Usage Notes

This procedure deletes the specified node from the node table (described in Section 5.9.1), and it deletes any other network elements that depend on this node. For example, if the specified node is included in any link definitions, those links are deleted; and if any of the deleted links are included in any paths and subpaths, those paths and subpaths are deleted also.

Examples

The following example deletes the node in the SDO_NET2 network whose node ID is 1.

```
SELECT SDO_NET.DELETE_NODE('SDO_NET2', 1);
```

SDO_NET.DELETE_PATH

Format

```
SDO_NET.DELETE_PATH(
   network  IN VARCHAR2,
   path_id  IN NUMBER);
```

Description

Deletes a path and all dependent network elements.

Parameters

network
Network name.

path_id
ID of the path to delete.

Usage Notes

This procedure deletes the specified path from the path table (described in Section 5.9.3), and it deletes any other network elements that depend on this path. For example, if the specified path has any subpaths, those subpaths are deleted also.

Examples

The following example deletes the path in the SDO_NET2 network whose path ID is 1.

```
SELECT SDO_NET.DELETE_PATH('SDO_NET2', 1);
```

SDO_NET.DELETE_SUBPATH

Format

```
SDO_NET.DELETE_SUBPATH(
   network  IN VARCHAR2,
   path_id  IN NUMBER);
```

Description

Deletes a subpath.

Parameters

network
Network name.

path_id
ID of the path to delete.

Usage Notes

This procedure deletes the specified subpath from the path table (described in Section 5.9.3). It does not delete any other network elements, because no other elements depend on a subpath definition.

Examples

The following example deletes the subpath in the SDO_NET2 network whose subpath ID is 17.

```
SELECT SDO_NET.DELETE_SUBPATH('SDO_NET2', 17);
```

SDO_NET.DEREGISTER_CONSTRAINT

Format

```
SDO_NET.DEREGISTER_CONSTRAINT(
   constraint_name  IN VARCHAR2);
```

Description

Unloads (removes) the class for the specified network constraint from the Java repository in the database, and deletes the row for that constraint from the USER_SDO_NETWORK_CONSTRAINTS view (described in Section 5.10.2).

Parameters

constraint_name
Name of the network constraint. Must match a value in the CONSTRAINT column of the USER_SDO_NETWORK_CONSTRAINTS view.

Usage Notes

Use this procedure if you want to disable a network constraint that you had previously enabled, such as by using the SDO_NET.REGISTER_CONSTRAINTprocedure. For more information about network constraints, see Section 5.6.

This procedure is analogous to using the `deregisterConstraint` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example deregisters (disables) a network constraint named `GivenProhibitedTurn`.

```
EXECUTE SDO_NET.DEREGISTER_CONSTRAINT('GivenProhibitedTurn');
```

SDO_NET.DROP_NETWORK

Format

```
SDO_NET.DROP_NETWORK(
   network  IN VARCHAR2);
```

Description

Drops (deletes) a network.

Parameters

network
Name of the network to be dropped.

Usage Notes

This procedure also deletes the node, link, and path tables associated with the network, and the network metadata for the network.

Examples

The following example drops the network named ROADS_NETWORK.

```
EXECUTE SDO_NET.DROP_NETWORK('ROADS_NETWORK');
```

SDO_NET.FIND_CONNECTED_COMPONENTS

Format

```
SDO_NET.FIND_CONNECTED_COMPONENTS(
   network  IN VARCHAR2,
   link_level  IN NUMBER,
   component_table_name  IN VARCHAR2,
   log_loc  IN VARCHAR2,
   log_file  IN VARCHAR2,
   open_mode  IN VARCHAR2);
```

Description

Finds all connected components for a specified link level in a network, and stores the information in the connected component table.

Parameters

network
Network name.

link_level
Link level for which to find connected components. Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in computing a path.

component_table_name
Name of the connected component table, which is created by this procedure. (If an existing table with the specified name already exists, it is updated with information for the specified link level.) The connected component table is described in Section 5.9.8.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about Spatial network operations, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

This procedure finds, for each node in the specified network, information about all other nodes that are reachable from that node, and it stores the information in the specified connected component table. Having this information in the table enables better performance for many network analysis operations.

Examples

The following example finds the connected components for link level 1 in the SDO_PARTITIONED network, and creates or updates the SDO_PARTITIONED_CONN_COMP_TAB table. Information about the operation is added (`open_mode => 'a'`) to the `sdo_partitioned.log` file, located in the location associated with the directory object named `LOG_DIR`.

```
EXECUTE SDO_NET.FIND_CONNECTED_COMPONENTS(-
  network => 'SDO_PARTITIONED', -
  link_level => 1,-
  component_table_name => 'sdo_partitioned_conn_comp_tab',-
  log_loc => 'LOG_DIR', log_file=> 'sdo_partitioned.log',-
  open_mode => 'a');
```

SDO_NET.GENERATE_NODE_LEVELS

Format

```
SDO_NET.GENERATE_NODE_LEVELS(
   network                IN VARCHAR2,
   node_level_table_name IN VARCHAR2,
   overwrite              IN BOOLEAN DEFAULT FALSE,
   log_loc                IN VARCHAR2,
   log_file               IN VARCHAR2,
   open_mode              IN VARCHAR2 DEFAULT 'a');
```

Description

Generates node levels for a specified hierarchical (multilevel) network, and stores the information in a table.

Parameters

network
Network name.

node_level_table_name
Table in which to store node level information. This table must have the following definition: `(node_id NUMBER PRIMARY KEY, link_level NUMBER)`

overwrite
Controls the behavior if the table specified in `node_level_table_name` already exists: `TRUE` replaces the contents of that table with new data; `FALSE` (the default) generates an error. (This parameter has no effect if the table specified in `node_level_table_name` does not exist.)

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about Spatial network operations, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

If `network` is not a hierarchical network (one with multiple link levels), this procedure does not perform any operation. For information about hierarchical networks, see Section 5.5.

The node level table name is stored in the NODE_LEVEL_TABLE_NAME column of the USER_SDO_NETWORK_METADATA view, shich is described in Section 5.10.1.

Examples

The following example generates the node level information for the MY_MULTILEVEL_NET network, and stores the information in the MY_NET_NODE_LEVELS table. Information about the operation is added (`open_mode => 'a'`) to the `my_multilevel_net.log` file, located in the location associated with the directory object named `LOG_DIR`.

```
EXECUTE SDO_NET.GENERATE_NODE_LEVELS(-
  network => 'MY_MULTILEVEL_NET', -
  node_level_table_name => 'MY_NET_NODE_LEVELS',-
  overwrite => FALSE,-
  log_loc => 'LOG_DIR', log_file=> 'my_multilevel_net.log',-
  open_mode => 'a');
```

SDO_NET.GENERATE_PARTITION_BLOB

Format

```
SDO_NET.GENERATE_PARTITION_BLOB(
   network  IN VARCHAR2,
   link_level  IN NUMBER,
   partition_id  IN VARCHAR2,
   include_user_data  IN BOOLEAN,
   log_loc  IN VARCHAR2,
   log_file  IN VARCHAR2,
   open_mode  IN VARCHAR2);
```

Description

Generates a single binary large object (BLOB) representation for a specified partition associated with a specified link level in the network, and stores the information in the existing partition BLOB table.

Parameters

network
Network name.

link_level
Link level for links to be included in the BLOB.

partition_id
Partition ID number. Network elements associated with the specified combination of link level and partition ID are included in the generated BLOB.

include_user_data
`TRUE` if the BLOB should include any user data associated with the network elements represented in each BLOB, or `FALSE` if the BLOB should not include any user data.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about Spatial network operations, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

This procedure adds a single new BLOB or replaces a single existing BLOB in the partition BLOB table, which must have been previously created using the SDO_NET.GENERATE_PARTITION_BLOBS procedure.

One use for this procedure is to perform a relatively quick update of the BLOB for a desired partition in a network that contains multiple large partitions, as opposed to than updating the BLOBs for all partitions using the SDO_NET.GENERATE_PARTITION_BLOBS procedure.

Examples

The following example generates the partition BLOB for the partition associated with partition ID 1 and link level 1 in the SDO_PARTITIONED network, and adds or replaces the appropriate BLOB in the SDO_PARTITIONED_PART_BLOB_TAB table. Any user data associated with the network elements is also included. Information about the operation is added (`open_mode => 'a'`) to the `sdo_partitioned.log` file, located in the location associated with the directory object named `LOG_DIR`.

```
EXECUTE SDO_NET.GENERATE_PARTITION_BLOB(-
  network => 'SDO_PARTITIONED', -
  link_level => 1,-
  partition_id => 1,-
  include_user_data => true,-
  log_loc => 'LOG_DIR', log_file=> 'sdo_partitioned.log',-
  open_mode => 'a');
```

SDO_NET.GENERATE_PARTITION_BLOBS

Format

```
SDO_NET.GENERATE_PARTITION_BLOBS(
   network  IN VARCHAR2,
   link_level  IN NUMBER,
   partition_blob_table_name  IN VARCHAR2,
   include_user_data  IN BOOLEAN,
   log_loc  IN VARCHAR2,
   log_file  IN VARCHAR2,
   open_mode  IN VARCHAR2);
```

Description

Generates a binary large object (BLOB) representation for partitions associated with a specified link level in the network, and stores the information in the partition BLOB table.

Parameters

network
Network name.

link_level
Link level for links to be included in each BLOB. Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in computing a path.

partition_blob_table_name
Name of the partition BLOB table, which is created by this procedure. (If an existing table with the specified name already exists, it is updated with information for the specified link level.) The partition BLOB table is described in Section 5.9.7.

include_user_data
`TRUE` if each BLOB should include any user data associated with the network elements represented in each BLOB, or `FALSE` if each BLOB should not include any user data.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about Spatial network operations, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

Generating partition BLOBs enables better performance for many network analysis operations, especially with large networks.

If the network is not partitioned, this procedure generates a single BLOB representing the entire network.

Do not confuse this procedure with SDO_NET.GENERATE_PARTITION_BLOB, which regenerates a single BLOB for a specified combination of link level and partition ID, and adds that information to the existing partition BLOB table.

Examples

The following example generates partition BLOBs for link level 1 in the SDO_PARTITIONED network, and creates or updates the SDO_PARTITIONED_PART_BLOB_TAB table. Any user data associated with the network elements is also included. Information about the operation is added (`open_mode => 'a'`) to the `sdo_partitioned.log` file, located in the location associated with the directory object named `LOG_DIR`.

```
EXECUTE SDO_NET.GENERATE_PARTITION_BLOBS(-
  network => 'SDO_PARTITIONED', -
  link_level => 1,-
  partition_blob_table_name => 'sdo_partitioned_part_blob_tab',-
  include_user_data => true,-
  log_loc => 'LOG_DIR', log_file=> 'sdo_partitioned.log',-
  open_mode => 'a');
```

SDO_NET.GET_CHILD_LINKS

Format

```
SDO_NET.GET_CHILD_LINKS(
   network  IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the child links of a link.

Parameters

network
Network name.

link_id
ID of the link for which to return the child links.

Usage Notes

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the child links of the link in the XYZ_NETWORK network whose link ID is 1001.

```
SELECT SDO_NET.GET_CHILD_LINKS('XYZ_NETWORK', 1001) FROM DUAL;

SDO_NET.GET_CHILD_LINKS('XYZ_NETWORK',1001)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(1108, 1109)
```

SDO_NET.GET_CHILD_NODES

Format

```
SDO_NET.GET_CHILD_NODES(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the child nodes of a node.

Parameters

network
Network name.

node_id
ID of the node for which to return the child nodes.

Usage Notes

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the child nodes of the node in the XYZ_NETWORK network whose node ID is 1.

```
SELECT SDO_NET.GET_CHILD_NODES('XYZ_NETWORK', 1) FROM DUAL;

SDO_NET.GET_CHILD_NODES('XYZ_NETWORK',1)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(101, 102, 103, 104, 105, 106)
```

SDO_NET.GET_GEOMETRY_TYPE

Format

```
SDO_NET.GET_GEOMETRY_TYPE(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the geometry type for a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the GEOMETRY_TYPE column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the geometry type for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_GEOMETRY_TYPE('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_GEOMETRY_TYPE('ROADS_NETWORK')
--------------------------------------------------------------------------------
LRS_GEOMETRY
```

SDO_NET.GET_IN_LINKS

Format

```
SDO_NET.GET_IN_LINKS(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array of link ID numbers of the inbound links to a node.

Parameters

network
Network name.

node_id
ID of the node for which to return the array of inbound links.

Usage Notes

For information about inbound links and related network data model concepts, see Section 5.3.

Examples

The following example returns an array of link ID numbers of the inbound links into the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_IN_LINKS('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_IN_LINKS('ROADS_NETWORK',3)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(102)
```

SDO_NET.GET_INVALID_LINKS

Format

```
SDO_NET.GET_INVALID_LINKS(
   network  IN VARCHAR2
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the invalid links in a network.

Parameters

network
Network name.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with a comma-delimited list of node ID numbers of invalid links in the specified network. If there are no invalid links, this function returns a null value.

Examples

The following example returns the invalid links in the SDO_PARTITIONED network.

```
SELECT SDO_NET.GET_INVALID_LINKS('SDO_PARTITIONED') FROM DUAL;
```

SDO_NET.GET_INVALID_NODES

Format

```
SDO_NET.GET_INVALID_NODES(
   network  IN VARCHAR2
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the invalid nodes in a network.

Parameters

network
Network name.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with a comma-delimited list of node ID numbers of invalid nodes in the specified network. If there are no invalid nodes, this function returns a null value.

Examples

The following example returns the invalid nodes in the SDO_PARTITIONED network.

```
SELECT SDO_NET.GET_INVALID_NODES('SDO_PARTITIONED') FROM DUAL;
```

SDO_NET.GET_INVALID_PATHS

Format

```
SDO_NET.GET_INVALID_PATHS(
   network  IN VARCHAR2
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the invalid paths in a network.

Parameters

network
Network name.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with a comma-delimited list of node ID numbers of invalid paths in the specified network. If there are no invalid paths, this function returns a null value.

Examples

The following example returns the invalid paths in the SDO_PARTITIONED network.

```
SELECT SDO_NET.GET_INVALID_PATHS('SDO_PARTITIONED') FROM DUAL;
```

SDO_NET.GET_ISOLATED_NODES

Format

```
SDO_NET.GET_ISOLATED_NODES(
   network  IN VARCHAR2
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the isolated nodes in a network.

Parameters

network
Network name.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with a comma-delimited list of node ID numbers of isolated nodes in the specified network. If there are no isolated nodes, this function returns a null value.

For a brief explanation of isolated nodes in a network, see Section 5.3.

Examples

The following example returns the isolated nodes in the SDO_PARTITIONED network.

```
SELECT SDO_NET.GET_ISOLATED_NODES('SDO_PARTITIONED') FROM DUAL;
```

SDO_NET.GET_LINK_COST_COLUMN

Format

```
SDO_NET.GET_LINK_COST_COLUMN(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the link cost column for a network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LINK_COST_COLUMN column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the link cost column for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LINK_COST_COLUMN('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LINK_COST_COLUMN('ROADS_NETWORK')
--------------------------------------------------------------------------------
COST
```

SDO_NET.GET_LINK_DIRECTION

Format

```
SDO_NET.GET_LINK_DIRECTION(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the link direction for a network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LINK_DIRECTION column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the link direction for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LINK_DIRECTION('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LINK_DIRECTION('ROADS_NETWORK')
--------------------------------------------------------------------------------
DIRECTED
```

SDO_NET.GET_LINK_GEOM_COLUMN

Format

```
SDO_NET.GET_LINK_GEOM_COLUMN(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the link geometry column for a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LINK_GEOM_COLUMN column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the link geometry column for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LINK_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LINK_GEOM_COLUMN('ROADS_NETWORK')
--------------------------------------------------------------------------------
LINK_GEOMETRY
```

SDO_NET.GET_LINK_GEOMETRY

Format

```
SDO_NET.GET_LINK_GEOMETRY(
   network  IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the geometry associated with a link in a spatial network.

Parameters

network
Network name.

link_id
ID number of the link for which to return the geometry.

Usage Notes

None.

Examples

The following example returns the geometry associated with the link whose link ID is 103 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LINK_GEOMETRY('ROADS_NETWORK', 103) FROM DUAL;

SDO_NET.GET_LINK_GEOMETRY('ROADS_NETWORK',103)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
8, 4, 12, 4))
```

SDO_NET.GET_LINK_TABLE_NAME

Format

```
SDO_NET.GET_LINK_TABLE_NAME(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the link table for a network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LINK_TABLE_NAME column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the link table for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LINK_TABLE_NAME('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LINK_TABLE_NAME('ROADS_NETWORK')
--------------------------------------------------------------------------------
ROADS_LINKS
```

SDO_NET.GET_LINKS_IN_PATH

Format

```
SDO_NET.GET_LINKS_IN_PATH(
   network  IN VARCHAR2,
   path_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the links in a path.

Parameters

network
Network name.

path_id
ID of the path for which to return the links.

Usage Notes

For an explanation of links and paths, see Section 5.3.

Examples

The following example returns the link ID values of links in the path in the XYZ_NETWORK network whose path ID is 1.

```
SELECT SDO_NET.GET_CHILD_LINKS('XYZ_NETWORK', 1001) FROM DUAL;

SDO_NET.GET_CHILD_LINKS('XYZ_NETWORK',1001)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(1108, 1109)
```

SDO_NET.GET_LRS_GEOM_COLUMN

Format

```
SDO_NET.GET_LRS_GEOM_COLUMN(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the LRS geometry column for a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LRS_GEOM_COLUMN column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the LRS geometry column for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LRS_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LRS_GEOM_COLUMN('ROADS_NETWORK')
--------------------------------------------------------------------------------
ROAD_GEOM
```

SDO_NET.GET_LRS_LINK_GEOMETRY

Format

```
SDO_NET.GET_LRS_LINK_GEOMETRY(
   network  IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the LRS geometry associated with a link in a spatial LRS network.

Parameters

network
Network name.

link_id
ID number of the link for which to return the geometry.

Usage Notes

None.

Examples

The following example returns the LRS geometry associated with the link whose link ID is 103 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LRS_LINK_GEOMETRY('ROADS_NETWORK', 103) FROM DUAL;

SDO_NET.GET_LRS_LINK_GEOMETRY('ROADS_NETWORK',103)(SDO_GTYPE, SDO_SRID, SDO_POIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
8, 4, 12, 4))
```

SDO_NET.GET_LRS_NODE_GEOMETRY

Format

```
SDO_NET.GET_LRS_NODE_GEOMETRY(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the LRS geometry associated with a node in a spatial LRS network.

Parameters

network
Network name.

node_id
ID number of the node for which to return the geometry.

Usage Notes

None.

Examples

The following example returns the LRS geometry associated with the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LRS_NODE_GEOMETRY('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_LRS_NODE_GEOMETRY('ROADS_NETWORK',3)(SDO_GTYPE, SDO_SRID, SDO_POINT(
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8, 4, NULL), NULL, NULL)
```

SDO_NET.GET_LRS_TABLE_NAME

Format

```
SDO_NET.GET_LRS_TABLE_NAME(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the table containing LRS geometries in a spatial LRS network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the LRS_TABLE_NAME column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the table that contains LRS geometries for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_LRS_TABLE_NAME('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_LRS_TABLE_NAME('ROADS_NETWORK')
--------------------------------------------------------------------------------
ROADS
```

SDO_NET.GET_NETWORK_TYPE

Format

```
SDO_NET.GET_NETWORK_TYPE(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the network type.

Parameters

network
Network name.

Usage Notes

This function returns the value of the NETWORK_TYPE column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the network type for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NETWORK_TYPE('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NETWORK_TYPE('ROADS_NETWORK')
--------------------------------------------------------------------------------
Roadways
```

SDO_NET.GET_NO_OF_HIERARCHY_LEVELS

Format

```
SDO_NET.GET_NO_OF_HIERARCHY_LEVELS(
   network  IN VARCHAR2
   ) RETURN NUMBER;
```

Description

Returns the number of hierarchy levels for a network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the NO_OF_HIERARCHY_LEVELS column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

For an explanation of network hierarchy, see Section 5.5.

Examples

The following example returns the number of hierarchy levels for the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NO_OF_HIERARCHY_LEVELS('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NO_OF_HIERARCHY_LEVELS('ROADS_NETWORK')
---------------------------------------------------
                                                  1
```

SDO_NET.GET_NO_OF_LINKS

Format

```
SDO_NET.GET_NO_OF_LINKS(
   network  IN VARCHAR2
   ) RETURN NUMBER;
```

or

```
SDO_NET.GET_NO_OF_LINKS(
   network  IN VARCHAR2,
   hierarchy_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of links for a network or a hierarchy level in a network.

Parameters

network
Network name.

hierarchy_id
Hierarchy level number for which to return the number of links.

Usage Notes

None.

Examples

The following example returns the number of links in the network named `ROADS_NETWORK`.

```
SELECT SDO_NET.GET_NO_OF_LINKS('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NO_OF_LINKS('ROADS_NETWORK')
----------------------------------------
                                      10
```

SDO_NET.GET_NO_OF_NODES

Format

```
SDO_NET.GET_NO_OF_NODES(
   network  IN VARCHAR2
   ) RETURN NUMBER;
```

or

```
SDO_NET.GET_NO_OF_NODES(
   network  IN VARCHAR2,
   hierarchy_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of nodes for a network or a hierarchy level in a network.

Parameters

network
Network name.

hierarchy_id
Hierarchy level number for which to return the number of nodes.

Usage Notes

For information about nodes and related concepts, see Section 5.3.

Examples

The following example returns the number of nodes in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NO_OF_NODES('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NO_OF_NODES('ROADS_NETWORK')
----------------------------------------
                                       8
```

SDO_NET.GET_NODE_DEGREE

Format

```
SDO_NET.GET_NODE_DEGREE(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of links to a node.

Parameters

network
Network name.

node_id
Node ID of the node for which to return the number of links.

Usage Notes

For information about node degree and related network data model concepts, see Section 5.3.

Examples

The following example returns the number of links to the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_DEGREE('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_NODE_DEGREE('ROADS_NETWORK',3)
------------------------------------------
                                         3
```

SDO_NET.GET_NODE_GEOM_COLUMN

Format

```
SDO_NET.GET_NODE_GEOM_COLUMN(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the geometry column for nodes in a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the NODE_GEOM_COLUMN column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the geometry column for nodes in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NODE_GEOM_COLUMN('ROADS_NETWORK')
--------------------------------------------------------------------------------
NODE_GEOMETRY
```

SDO_NET.GET_NODE_GEOMETRY

Format

```
SDO_NET.GET_NODE_GEOMETRY(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the LRS geometry associated with a node in a spatial network.

Parameters

network
Network name.

node_id
ID number of the node for which to return the geometry.

Usage Notes

None.

Examples

The following example returns the geometry associated with the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_GEOMETRY('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_NODE_GEOMETRY('ROADS_NETWORK',3)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(8, 4, NULL), NULL, NULL)
```

SDO_NET.GET_NODE_IN_DEGREE

Format

```
SDO_NET.GET_NODE_IN_DEGREE(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of inbound links to a node.

Parameters

network
Network name.

node_id
Node ID of the node for which to return the number of inbound links.

Usage Notes

For information about node degree and related network data model concepts, see Section 5.3.

Examples

The following example returns the number of inbound links to the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_IN_DEGREE('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_NODE_IN_DEGREE('ROADS_NETWORK',3)
---------------------------------------------
                                            1
```

SDO_NET.GET_NODE_OUT_DEGREE

Format

```
SDO_NET.GET_NODE_OUT_DEGREE(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of outbound links from a node.

Parameters

network
Network name.

node_id
Node ID of the node for which to return the number of outbound links.

Usage Notes

For information about node degree and related network data model concepts, see Section 5.3.

Examples

The following example returns the number of outbound links from the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_OUT_DEGREE('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_NODE_OUT_DEGREE('ROADS_NETWORK',3)
----------------------------------------------
                                             2
```

SDO_NET.GET_NODE_TABLE_NAME

Format

```
SDO_NET.GET_NODE_TABLE_NAME(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the table that contains the nodes in a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the NODE_TABLE_NAME column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the table that contains the nodes in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_NODE_TABLE_NAME('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_NODE_TABLE_NAME('ROADS_NETWORK')
--------------------------------------------------------------------------------
ROADS_NODES
```

SDO_NET.GET_OUT_LINKS

Format

```
SDO_NET.GET_OUT_LINKS(
   network  IN VARCHAR2,
   node_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array of link ID numbers of the outbound links from a node.

Parameters

network
Network name.

node_id
ID of the node for which to return the array of outbound links.

Usage Notes

For information about outbound links and related network data model concepts, see Section 5.3.

Examples

The following example returns an array of link ID numbers of the outbound links from the node whose node ID is 3 in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_OUT_LINKS('ROADS_NETWORK', 3) FROM DUAL;

SDO_NET.GET_OUT_LINKS('ROADS_NETWORK',3)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(103, 201)
```

SDO_NET.GET_PARTITION_SIZE

Format

```
SDO_NET.GET_PARTITION_SIZE(
   network              IN VARCHAR2,
   partition_id         IN VARCHAR2,
   link_level           IN NUMBER,
   include_user_data    IN VARCHAR2,
   include_spatial_data IN VARCHAR2) RETURN NUMBER;
```

Description

Gets the estimated size (in bytes) for a specified combination of partition ID and link level.

Parameters

network
Network name.

partition_id
Partition ID number.

link_level
Link level.

include_user_data
`TRUE` if the size should include any user data associated with the network elements represented in each BLOB, or `FALSE` if the size should not include any user data.

include_spatial_data
`TRUE` if the size should include spatial geometry definitions associated with the network elements represented in each BLOB, or `FALSE` if the size should not include spatial geometry definitions.

Usage Notes

The returned size of a network partition is a rough estimate and might vary depending on the Java Virtual Machine and garbage collection.

For information about using partitioned networks to perform analysis using the load on demand approach, see Section 5.7.

Examples

The following example returns the number of bytes for the partition associated with partition ID 1 and link level 1 in the SDO_PARTITIONED network, not including any user data or spatial data.

```
SELECT SDO_NET.GET_PARTITION_SIZE('SDO_PARTITIONED', 1, 1, 'N', 'N') FROM DUAL;

SDO_NET.GET_PARTITION_SIZE('SDO_PARTITIONED',1,1,'FALSE','FALSE')
-----------------------------------------------------------
                                                       5192
```

SDO_NET.GET_PATH_GEOM_COLUMN

Format

```
SDO_NET.GET_PATH_GEOM_COLUMN(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the geometry column for paths in a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the PATH_GEOM_COLUMN column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the geometry column for paths in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_PATH_GEOM_COLUMN('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_PATH_GEOM_COLUMN('ROADS_NETWORK')
--------------------------------------------------------------------------------
PATH_GEOMETRY
```

SDO_NET.GET_PATH_TABLE_NAME

Format

```
SDO_NET.GET_PATH_TABLE_NAME(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the name of the table that contains the paths in a spatial network.

Parameters

network
Network name.

Usage Notes

This function returns the value of the PATH_TABLE_NAME column for the network in the USER_SDO_NETWORK_METADATA view (see Table 5–9 in Section 5.10.1).

Examples

The following example returns the name of the table that contains the paths in the network named ROADS_NETWORK.

```
SELECT SDO_NET.GET_PATH_TABLE_NAME('ROADS_NETWORK') FROM DUAL;

SDO_NET.GET_PATH_TABLE_NAME('ROADS_NETWORK')
--------------------------------------------------------------------------------
ROADS_PATHS
```

SDO_NET.GET_PERCENTAGE

Format

```
SDO_NET.GET_PERCENTAGE(
   network  IN VARCHAR2,
   link_id  IN NUMBER,
   pt_geom  IN SDO_GEOMETRY
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the percentage of the distance along a link's line string geometry of a point geometry.

Parameters

network
Network name.

link_id
ID number of the link.

pt_geom
Point geometry.

Usage Notes

This function returns a value between 0 and 1. For example, if the point is 25 percent (one-fourth) of the distance between the start node and end node for the link, the function returns .25.

If `pt_geom` is not on the link geometry, the nearest point on the link geometry to `pt_geom` is used.

To find the point geometry that is a specified percentage of the distance along a link's line string geometry, use the SDO_NET.GET_PT function.

Examples

The following example returns the percentage (as a decimal fraction) of the distance of a specified point along the geometry associated with the link whose link ID is 101 in the network named `ROADS_NETWORK`.

```
SQL> SELECT SDO_NET.GET_PERCENTAGE('ROADS_NETWORK', 101,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(2, 2.5, NULL), NULL, NULL))
  FROM DUAL;  2    3

SDO_NET.GET_PERCENTAGE('ROADS_NETWORK',101,SDO_GEOMETRY(2001,NULL,SDO_POINT_TYPE
--------------------------------------------------------------------------------
                                                                             .25
```

SDO_NET.GET_PT

Format

```
SDO_NET.GET_PT(
   network  IN VARCHAR2,
   link_id  IN NUMBER,
   percentage IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the point geometry that is a specified percentage of the distance along a link's line string geometry.

Parameters

network
Network name.

link_id
ID number of the link for which to return the point geometry at the specified `percentage` distance.

percentage
Percentage value as a decimal fraction between 0 and 1. For example, 0.25 is 25 percent.

Usage Notes

To find the percentage along a link geometry for a specified point, use the SDO_NET.GET_PERCENTAGE function.

Examples

The following example returns the point geometry that is 25 percent of the distance from the start node along the geometry associated with the link whose link ID is 101 in the network named `ROADS_NETWORK`.

```
SELECT SDO_NET.GET_PT('ROADS_NETWORK', 101, 0.25) FROM DUAL;

SDO_NET.GET_PT('ROADS_NETWORK',101,0.25)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z)
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(2, 2.5, NULL), NULL, NULL)
```

SDO_NET.IS_HIERARCHICAL

Format

```
SDO_NET.IS_HIERARCHICAL(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the network has more than one level of hierarchy; returns the string FALSE if the network does not have more than one level of hierarchy.

Parameters

network
Network name.

Usage Notes

For an explanation of network hierarchy, see Section 5.5.

Examples

The following example checks if the network named ROADS_NETWORK has more than one level of hierarchy.

```
SELECT SDO_NET.IS_HIERARCHICAL('ROADS_NETWORK') FROM DUAL;

SDO_NET.IS_HIERARCHICAL('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.IS_LINK_IN_PATH

Format

```
SDO_NET.IS_LINK_IN_PATH(
   network  IN VARCHAR2,
   path_id   IN NUMBER,
   link_id     IN NUMBER,
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the specified link is in the specified path; returns the string `FALSE` if the specified link is not in the specified path.

Parameters

network
Network name.

path_id
ID number of the path.

link_id
ID number of the link.

Usage Notes

You can use this function to check if a specific link is included in a specific path.

Examples

The following example checks if the link with link ID 1 is in the path with path ID 1 in the network named `SDO_NET1`.

```
SELECT SDO_NET.IS_LINK_IN_PATH('SDO_NET1', 1, 1) FROM DUAL;

SDO_NET.IS_LINK_IN_PATH('SDO_NET1',1,1)
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.IS_LOGICAL

Format

```
SDO_NET.IS_LOGICAL(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the network is a logical network; returns the string `FALSE` if the network is not a logical network.

Parameters

network
Network name.

Usage Notes

A network can be a spatial network or a logical network, as explained in Section 5.3.

Examples

The following example checks if the network named `ROADS_NETWORK` is a logical network.

```
SELECT SDO_NET.IS_LOGICAL('ROADS_NETWORK') FROM DUAL;

SDO_NET.IS_LOGICAL('ROADS_NETWORK')
--------------------------------------------------------------------------------
FALSE
```

SDO_NET.IS_NODE_IN_PATH

Format

```
SDO_NET.IS_NODE_IN_PATH(
   network  IN VARCHAR2,
   path_id   IN NUMBER,
   link_id     IN NUMBER,
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the specified node is in the specified path; returns the string `FALSE` if the specified node is not in the specified path.

Parameters

network
Network name.

path_id
ID number of the path.

node_id
ID number of the node.

Usage Notes

You can use this function to check if a specific node is included in a specific path.

Examples

The following example checks if the node with node ID 1 is in the path with path ID 1 in the network named `SDO_NET1`.

```
SELECT SDO_NET.IS_NODE_IN_PATH('SDO_NET1', 1, 1) FROM DUAL;

SDO_NET.IS_NODE_IN_PATH('SDO_NET1',1,1)
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.IS_SPATIAL

Format

```
SDO_NET.IS_SPATIAL(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the network is a spatial network; returns the string FALSE if the network is not a spatial network.

Parameters

network
Network name.

Usage Notes

A network can be a spatial network or a logical network, as explained in Section 5.3.

You can further check for the geometry type of a spatial network by using the following functions: SDO_NET.LRS_GEOMETRY_NETWORK, SDO_NET.SDO_GEOMETRY_NETWORK, and SDO_NET.TOPO_GEOMETRY_NETWORK.

Examples

The following example checks if the network named ROADS_NETWORK is a spatial network.

```
SELECT SDO_NET.IS_SPATIAL('ROADS_NETWORK') FROM DUAL;

SDO_NET.IS_SPATIAL('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.LOAD_CONFIG

Format

```
SDO_NET.LOAD_CONFIG(
   file_directory  IN VARCHAR2,
   file_name       IN VARCHAR2);
```

Description

Loads (or reloads) the load on demand configuration, which is mainly partition cache configuration, from the specified XML file.

Parameters

file_directory
Directory object that identifies the path for the XML file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

file_name
Name of the XML file containing the information to be loaded.

Usage Notes

A default configuration is provided for load on demand. You can use this procedure if you need to change the default configuration.

For information about configuring the load on demand environment, including the partition cache, see Section 5.7.3.

Examples

The following example loads the load on demand configuration from a specified XML file.

```
EXECUTE SDO_NET.LOAD_CONFIG('WORK_DIR', 'netlodcfg.xml');
```

SDO_NET.LOGICAL_PARTITION

Format

```
SDO_NET.LOGICAL_PARTITION(
   network              IN VARCHAR2,
   partition_table_name IN VARCHAR2,
   max_num_nodes        IN NUMBER,
   log_loc              IN VARCHAR2,
   log_file             IN VARCHAR2,
   open_mode            IN VARCHAR2,
   link_level           IN NUMBER);
```

Description

Partitions a logical network, and stores the information in the partition table.

> **Note:** If the logical network is a power law (scale-free) network, do not use this procedure to partition it, but instead use the SDO_NET.LOGICAL_POWERLAW_PARTITION procedure.

Parameters

network
Network name.

link_level
Network link level on which to perform the partitioning. Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in network computations.

partition_table_name
Name of the partition table, which is created by this procedure. (If an existing table with the specified name already exists, it is updated with partition information for the specified link level.) The partition table is described in Section 5.9.6.

max_num_nodes
Maximum number of nodes to include in each partition. For example, if you specify 5000 and if the network contains 50,000 nodes, each partition will have 5000 or fewer nodes, and the total number of partitions will be 10 or higher.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about operations on the logical network, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

After you use this procedure to create the partitions, consider using the SDO_NET.GENERATE_PARTITION_BLOBS procedure, to enable better performance for many network analysis operations, especially with large networks.

Examples

The following example creates partitions for link level 1 in the `MY_LOGICAL_NET` network, and creates the MY_LOGICAL_PART_TAB table. The maximum number of nodes to be placed in any partition is 5000. Information about the operation is added (`open_mode => 'a'`) to the `my_logical_part.log` file, located in the location associated with the directory object named `LOG_DIR`.

```
EXECUTE SDO_NET.LOGICAL_PARTITION(network => 'MY_LOGICAL_NET', -
  partition_table_name => 'my_logical_part_tab',-
  max_num_nodes => 5000,-
  log_loc => 'LOG_DIR', log_file=> 'my_logical_part.log',-
  link_level => 1, open_mode => 'a');
```

SDO_NET.LOGICAL_POWERLAW_PARTITION

Format

```
SDO_NET.LOGICAL_POWERLAW_PARTITION(
   network              IN VARCHAR2,
   partition_table_name IN VARCHAR2,
   max_num_nodes        IN NUMBER,
   log_loc              IN VARCHAR2,
   log_file             IN VARCHAR2,
   open_mode            IN VARCHAR2,
   link_level           IN NUMBER);
```

Description

Partitions a logical power law (also called scale-free) network, and stores the information in the partition table. (In a power law network, the node degree values contain the power law information.)

Parameters

network
Network name.

link_level
Network link level on which to perform the partitioning. Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in network computations.

partition_table_name
Name of the partition table, which is created by this procedure. (If an existing table with the specified name already exists, it is updated with partition information for the specified link level.) The partition table is described in Section 5.9.6.

max_num_nodes
Maximum number of nodes to include in each partition. For example, if you specify 5000 and if the network contains 50,000 nodes, each partition will have 5000 or fewer nodes, and the total number of partitions will be 10 or higher.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about operations on the logical power law network, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

After you use this procedure to create the partitions, consider using the SDO_NET.GENERATE_PARTITION_BLOBS procedure, to enable better performance for many network analysis operations, especially with large networks.

If the logical network is not a power law network, do not use this procedure, but instead use the SDO_NET.LOGICAL_PARTITION procedure.

Examples

The following example creates partitions for link level 1 in the MY_LOGICAL_PLAW_NET network, and creates the MY_LOGICAL_PLAW_PART_TAB table. The maximum number of nodes to be placed in any partition is 5000. Information about the operation is added (open_mode => 'a') to the my_logical_plaw_part.log file, located in the location associated with the directory object named LOG_DIR.

```
EXECUTE SDO_NET.LOGICAL_POWERLAW_PARTITION(network => 'MY_LOGICAL_PLAW_NET', -
  partition_table_name => 'my_logical_plaw_part_tab',-
  max_num_nodes => 5000,-
  log_loc => 'LOG_DIR', log_file=> 'my_logical_plaw_part.log',-
  link_level => 1, open_mode => 'a');
```

SDO_NET.LRS_GEOMETRY_NETWORK

Format

```
SDO_NET.LRS_GEOMETRY_NETWORK(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the network is a spatial network containing LRS geometries; returns the string FALSE if the network is not a spatial network containing LRS geometries.

Parameters

network
Network name.

Usage Notes

A network contains LRS geometries if the GEOMETRY_TYPE column in its entry in the USER_SDO_NETWORK_METADATA view contains the value LRS_GEOMETRY. (The USER_SDO_NETWORK_METADATA view is explained in Section 5.10.1.)

Examples

The following example checks if the network named ROADS_NETWORK is a spatial network containing LRS geometries.

```
SELECT SDO_NET.LRS_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;

SDO_NET.LRS_GEOMETRY_NETWORK('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.NETWORK_EXISTS

Format

```
SDO_NET.NETWORK_EXISTS(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the network exists; returns the string `FALSE` if the network does not exist.

Parameters

network
Network name.

Usage Notes

If you drop a network (using the SDO_NET.DROP_NETWORK procedure), the network no longer exists.

Examples

The following example checks if the network named `ROADS_NETWORK` exists.

```
SELECT SDO_NET.NETWORK_EXISTS('ROADS_NETWORK') FROM DUAL;

SDO_NET.NETWORK_EXISTS('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.REGISTER_CONSTRAINT

Format

```
SDO_NET.REGISTER_CONSTRAINT(
   constraint_name  IN VARCHAR2,
   class_name       IN VARCHAR2,
   directory_name   IN VARCHAR2,
   description      IN VARCHAR2);
```

Description

Loads the compiled Java code for the specified network constraint into the Java class repository in the database, and loads the class name into the CLASS column of the USER_SDO_NETWORK_CONSTRAINTS view (described in Section 5.10.2).

Parameters

constraint_name
Name of the network constraint.

class_name
Fully qualified name (including the name of the package) of the class that implements the network constraint.

directory_name
Name of the directory object (created using the SQL statement CREATE DIRECTORY) that identifies the location of the class file created when you compiled the network constraint.

description
Description of the network constraint.

Usage Notes

Before you call this procedure, you must insert a row into the USER_SDO_NETWORK_CONSTRAINTS view, compile the code for the Java class that implements the network constraint, and use the CREATE DIRECTORY statement to create a directory object identifying the location of the compiled class. For more information about network constraints, see Section 5.6.

To delete the row for the constraint from the USER_SDO_NETWORK_CONSTRAINTS view and thus disable the constraint, use the SDO_NET.DEREGISTER_CONSTRAINT procedure.

This procedure is analogous to using the `registerConstraint` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example registers a network constraint named `GivenProhibitedTurn`.

```
-- Set up the network constraint.
REM
```

```
REM Create the geor_dir on the file system first.
REM
-- Connect as SYSTEM.
DECLARE
  -- This is the directory that contains the CLASS file generated when you
  -- compiled the network constraint.
  geor_dir varchar2(1000) := 'C:\my_data\files81\PROTOTYPES\NETWORK_
CONSTRAINT\PLSQL_EXAMPLE';
BEGIN
  EXECUTE IMMEDIATE 'CREATE OR REPLACE DIRECTORY work_dir AS''' || geor_dir ||
'''';
END;
/
GRANT read,write on directory work_dir to net_con;

-- Connect as the user that will register the constraint.

REM
REM Compile GivenProhibitedTurn before you register the constraint.
REM
BEGIN
  SDO_NET.REGISTER_CONSTRAINT('GivenProhibitedTurn',
     'com/network/constraints/ProhibitedTurn',
     'WORK_DIR', 'This is a network constraint that '||
     'prohibits certain turns');

END;
/
```

SDO_NET.SDO_GEOMETRY_NETWORK

Format

```
SDO_NET.SDO_GEOMETRY_NETWORK(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the network is a spatial network containing SDO geometries (spatial geometries without measure information); returns the string FALSE if the network is not a spatial network containing SDO geometries.

Parameters

network
Network name.

Usage Notes

A network contains SDO geometries if the GEOMETRY_TYPE column in its entry in the USER_SDO_NETWORK_METADATA view contains the value SDO_GEOMETRY. (The USER_SDO_NETWORK_METADATA view is explained in Section 5.10.1.)

Examples

The following example checks if the network named ROADS_NETWORK is a spatial network containing SDO geometries.

```
SELECT SDO_NET.SDO_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;

SDO_NET.SDO_GEOMETRY_NETWORK('ROADS_NETWORK')
--------------------------------------------------------------------------------
FALSE
```

SDO_NET.SET_LOGGING_LEVEL

Format

```
SDO_NET.SET_LOGGING_LEVEL(
   level  IN NUMBER);
```

Description

Sets the minimum level of severity for messages to be displayed for network operations.

Parameters

level
Minimum severity level for messages to be displayed for network operations. Must be one of the numeric constants specified in the Usage Notes.

Usage Notes

All messages at the specified logging level and higher levels will be written. The logging levels, from highest to lowest, are:

```
SDO_NET.LOGGING_LEVEL_FATAL
SDO_NET.LOGGING_LEVEL_ERROR
SDO_NET.LOGGING_LEVEL_WARN
SDO_NET.LOGGING_LEVEL_INFO
SDO_NET.LOGGING_LEVEL_DEBUG
SDO_NET.LOGGING_LEVEL_FINEST
```

The logging level is the Java logging level from the underlying implementation of this function; therefore, to see the Java logging output on the console, execute the following statements beforehand:

```
SET SERVEROUTPUT ON;
EXECUTE DBMS_JAVA.SET_OUTPUT(10000);
```

Examples

The following example sets the logging level at `SDO_NET.LOGGING_LEVEL_ERROR`, which means that only messages with a severity of `SDO_NET.LOGGING_LEVEL_ERROR` or `SDO_NET.LOGGING_LEVEL_FATAL` will be displayed.

```
EXECUTE SDO_NET.SET_LOGGING_LEVEL(SDO_NET.LOGGING_LEVEL_ERROR);
```

SDO_NET.SET_MAX_JAVA_HEAP_SIZE

Format

```
SDO_NET.SET_MAX_JAVA_HEAP_SIZE(
   bytes  IN NUMBER);
```

Description

Sets the Java maximum heap size for an application to run in an Oracle Java virtual machine.

Parameters

bytes
Number of bytes for the Java maximum heap size.

Usage Notes

If you encounter the `java.lang.OutOfMemoryError` exception, you can use this procedure to increase the maximum heap size.

If you specify a value greater than the system limit, the system limit is used.

Examples

The following example sets the Java maximum heap size to 536870912 (512 MB).

```
EXECUTE SDO_NET.SET_MAX_JAVA_HEAP_SIZE(536870912);
```

SDO_NET.SPATIAL_PARTITION

Format

```
SDO_NET.SPATIAL_PARTITION(
   network              IN VARCHAR2,
   partition_table_name IN VARCHAR2,
   max_num_nodes        IN NUMBER,
   log_loc              IN VARCHAR2,
   log_file             IN VARCHAR2,
   open_mode            IN VARCHAR2,
   link_level           IN NUMBER);
```

Description

Partitions a spatial network, and stores the information in the partition table.

Parameters

network
Network name.

link_level
Network link level on which to perform the partitioning. Link level reflects the priority level for the link, and is used for network analysis, so that links with higher priority levels can be considered first in network computations.

partition_table_name
Name of the partition table, which is created by this procedure. (If an existing table with the specified name already exists, it is updated with partition information for the specified link level.) The partition table is described in Section 5.9.6.

max_num_nodes
Maximum number of nodes to include in each partition. For example, if you specify 5000 and if the network contains 50,000 nodes, each partition will have 5000 or fewer nodes, and the total number of partitions will be 10 or higher.

log_loc
Directory object that identifies the path for the log file. To create a directory object, use the SQL*Plus command CREATE DIRECTORY.

log_file
Log file containing information about Spatial network operations, including any possible errors or problems.

open_mode
A one-character code indicating the mode in which to open the log file: `W` for write over (that is, delete any existing log file at the specified location and name, and create a new file), or `A` for append (that is, append information to the existing specified log file). If you specify `A` and the log file does not exist, a new log file is created.

Usage Notes

After you use this procedure to create the partitions, consider using the SDO_NET.GENERATE_PARTITION_BLOBS procedure, to enable better performance for many network analysis operations, especially with large networks.

Examples

The following example creates partitions for link level 1 in the MY_PARTITIONED_NET network, and creates the MY_PARTITIONED_NET_TAB table. The maximum number of nodes to be placed in any partition is 5000. Information about the operation is added (open_mode => 'a') to the my_partitioned_net.log file, located in the location associated with the directory object named LOG_DIR.

```
EXECUTE SDO_NET.SPATIAL_PARTITION(network => 'MY_PARTITIONED_NET', -
  partition_table_name => 'my_partitioned_net_tab',-
  max_num_nodes => 5000,-
  log_loc => 'LOG_DIR', log_file=> 'my_partitioned_net.log',-
  link_level => 1, open_mode => 'a');
```

SDO_NET.TOPO_GEOMETRY_NETWORK

Format

```
SDO_NET.TOPO_GEOMETRY_NETWORK(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the network is a spatial network containing SDO_TOPO_GEOMETRY (topology geometry) objects; returns the string `FALSE` if the network is not a spatial network containing SDO_TOPO_GEOMETRY objects.

Parameters

network
Network name.

Usage Notes

A network contains SDO_TOPO_GEOMETRY objects if the GEOMETRY_TYPE column in its entry in the USER_SDO_NETWORK_METADATA view contains the value `TOPO_GEOMETRY`. (The USER_SDO_NETWORK_METADATA view is explained in Section 5.10.1.)

Examples

The following example checks if the network named `ROADS_NETWORK` is a spatial network containing SDO_TOPO_GEOMETRY objects.

```
SELECT SDO_NET.TOPO_GEOMETRY_NETWORK('ROADS_NETWORK') FROM DUAL;

SDO_NET.TOPO_GEOMETRY_NETWORK('ROADS_NETWORK')
--------------------------------------------------------------------------------
FALSE
```

SDO_NET.VALIDATE_LINK_SCHEMA

Format

```
SDO_NET.VALIDATE_LINK_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the metadata relating to links in a network is valid; returns the string `FALSE` if the metadata relating to links in a network is not valid.

Parameters

network
Network name.

Usage Notes

This function checks the following for validity: table name, geometry column, and cost column for spatial networks; measure-related information for LRS networks; topology-related information for topology networks; and hierarchy-related information for hierarchical networks.

Examples

The following example checks the validity of the metadata related to links in the network named `ROADS_NETWORK`.

```
SELECT SDO_NET.VALIDATE_LINK_SCHEMA('ROADS_NETWORK') FROM DUAL;

SDO_NET.VALIDATE_LINK_SCHEMA('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_LRS_SCHEMA

Format

```
SDO_NET.VALIDATE_LRS_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the metadata relating to LRS information in a network is valid; returns the string `FALSE` if the metadata relating to LRS information in a network is not valid.

Parameters

network
Network name.

Usage Notes

None.

Examples

The following example checks the validity of the metadata related to LRS information in the network named `ROADS_NETWORK`.

```
SELECT SDO_NET.VALIDATE_LRS_SCHEMA('ROADS_NETWORK') FROM DUAL;

SDO_NET.VALIDATE_LRS_SCHEMA('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_NETWORK

Format

```
SDO_NET.VALIDATE_NETWORK(
   network       IN VARCHAR2,
   check_data  IN VARCHAR2 DEFAULT 'FALSE'
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the network is valid; returns the string `FALSE` if the network is not valid.

Parameters

network
Network name.

check_data
`TRUE` performs additional checks on the referential integrity of network data; `FALSE` (the default) performs basic checks, but not additional checks, on the referential integrity of network data.

Usage Notes

This function checks the metadata for the network and any applicable network schema structures (link, node, path, subpath, LRS). It performs basic referential integrity checks on the network data, and it optionally performs additional checks. If any errors are found, the function returns the string `FALSE`.

The checks performed by this function include the following:

- The network exists.
- The node and link tables for the network exist, and they contain the required columns.
- The start and end nodes of each link exist in the node table.
- For an LRS geometry network, the LRS table exists and contains the required columns.
- For a spatial network, columns for the node and path geometries exist and have spatial indexes defined on them.
- If `check_data` is `TRUE`, additional referential integrity checking on the network data is performed. This will take longer, especially if the network is large.

Examples

The following example validates the network named `LOG_NET1`.

```
SELECT SDO_NET.VALIDATE_NETWORK('LOG_NET1') FROM DUAL;

SDO_NET.VALIDATE_NETWORK('LOG_NET1')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_NODE_SCHEMA

Format

```
SDO_NET.VALIDATE_NODE_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the metadata relating to nodes in a network is valid; returns the string FALSE if the metadata relating to nodes in a network is not valid.

Parameters

network
Network name.

Usage Notes

This function checks the following for validity: table name, geometry column, and cost column for spatial networks; measure-related information for LRS networks; topology-related information for topology networks; and hierarchy-related information for hierarchical networks.

Examples

The following example checks the validity of the metadata related to nodes in the network named LOG_NET1.

```
SELECT SDO_NET.VALIDATE_NODE_SCHEMA('LOG_NET1') FROM DUAL;

SDO_NET.VALIDATE_NODE_SCHEMA('LOG_NET1')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_PARTITION_SCHEMA

Format

```
SDO_NET.VALIDATE_PARTITION_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the metadata relating to partitions in a network is valid; returns the string `FALSE` if the metadata relating to partitions in a network is not valid.

Parameters

network
Network name.

Usage Notes

This function checks the validity of information in the partition table, which is described in Section 5.9.6.

Examples

The following example checks the validity of the metadata related to partitions in the network named `SDO_PARTITIONED`.

```
SELECT SDO_NET.VALIDATE_PARTITION_SCHEMA('SDO_PARTITIONED') FROM DUAL;

SDO_NET.VALIDATE_PARTITION_SCHEMA('SDO_PARTITIONED')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_PATH_SCHEMA

Format

```
SDO_NET.VALIDATE_PATH_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string TRUE if the metadata relating to paths in a network is valid; returns the string FALSE if the metadata relating to paths in a network is not valid.

Parameters

network
Network name.

Usage Notes

This function checks the following for validity: table name, geometry column, and cost column for spatial networks; measure-related information for LRS networks; topology-related information for topology networks; and hierarchy-related information for hierarchical networks.

Examples

The following example checks the validity of the metadata related to paths in the network named ROADS_NETWORK.

```
SELECT SDO_NET.VALIDATE_PATH_SCHEMA('ROADS_NETWORK') FROM DUAL;

SDO_NET.VALIDATE_PATH_SCHEMA('ROADS_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

SDO_NET.VALIDATE_SUBPATH_SCHEMA

Format

```
SDO_NET.VALIDATE_SUBPATH_SCHEMA(
   network  IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Returns the string `TRUE` if the metadata relating to subpaths in a network is valid; returns the string `FALSE` if the metadata relating to subpaths in a network is not valid.

Parameters

network
Network name.

Usage Notes

This function checks the validity of information in the subpath table, which is described in Section 5.9.5.

Examples

The following example checks the validity of the metadata related to subpaths in the network named `MY_NETWORK`.

```
SELECT SDO_NET.VALIDATE_SUBPATH_SCHEMA('MY_NETWORK') FROM DUAL;

SDO_NET.VALIDATE_SUBPATH_SCHEMA('MY_NETWORK')
--------------------------------------------------------------------------------
TRUE
```

7

SDO_NET_MEM Package Subprograms

The MDSYS.SDO_NET_MEM package contains subprograms (functions and procedures) that constitute part of the PL/SQL application programming interface (API) for the Spatial network data model. This package, which implements capabilities available through the Java API, contains subprograms related to editing and analyzing networks.

To use the subprograms in this chapter, you must understand the conceptual information about networks in Chapter 5, especially Section 5.8, which explains how to use a network memory object.

The SDO_NET_MEM subprograms are grouped according to their associated object-related class in the oracle.spatial.network interface or class. Except for the SDO_NET_MEM.SET_MAX_MEMORY_SIZE procedure, you must specify a prefix after SDO_NET_MEM for each program, depending on its associated class (for example, SDO_NET_MEM.NETWORK_MANAGER.CREATE_LOGICAL_NETWORK, SDO_NET_MEM.NETWORK.ADD_NODE, and SDO_NET_MEM.NODE.GET_COST).

The SDO_NET_MEM subprogram groupings are as follows:

- SDO_NET_MEM.NETWORK_MANAGER subprograms are related to the `oracle.spatial.network.NetworkManager` Java class. They enable you to create and drop network memory objects and to perform network analysis.
- SDO_NET_MEM.NETWORK subprograms are related to the `oracle.spatial.network.Network` Java interface. They enable you to add and delete nodes, links, and paths.
- SDO_NET_MEM.NODE subprograms are related to the `oracle.spatial.network.Node` Java interface. They enable you to get and set attributes for nodes.
- SDO_NET_MEM.LINK subprograms are related to the `oracle.spatial.network.Link` Java interface. They enable you to get and set attributes for links.
- SDO_NET_MEM.PATH subprograms are related to the `oracle.spatial.network.Path` Java interface. They enable you to get and set attributes for paths.

The associations between SDO_NET_MEM subprograms and methods of the Java API are not necessarily exact. In some cases, a PL/SQL subprogram may combine operations and options from several methods. In addition, some Java methods do not have PL/SQL counterparts. Thus, the Usage Notes for subprograms state only that the function or procedure is analogous to a specific Java method, to indicate a logical relationship between the two. For detailed information about a specific Java method

and others that may be related, see the Javadoc-generated API documentation (briefly explained in Section 5.11.2).

Note that although this manual refers to "the SDO_NET_MEM package," all subprograms except one are actually implemented as methods of several object types. Thus, they are not listed by the statement `DESCRIBE SDO_NET_MEM`. Instead, you can use the DESCRIBE statements listed in Table 7–1 to see the subprograms in each grouping; however, because they are member functions and procedures in an object type, the subprograms in each grouping will not be listed in alphabetical order in the DESCRIBE statement output.

Table 7–1 DESCRIBE Statements for SDO_NET_MEM Subprograms

Subprogram Grouping	DESCRIBE Statement
SDO_NET_MEM.NETWORK_MANAGER	`DESCRIBE sdo_network_manager_t`
SDO_NET_MEM.NETWORK	`DESCRIBE sdo_network_t`
SDO_NET_MEM.NODE	`DESCRIBE sdo_node_t`
SDO_NET_MEM.LINK	`DESCRIBE sdo_link_t`
SDO_NET_MEM.PATH	`DESCRIBE sdo_path_t`

The rest of this chapter provides reference information about the SDO_NET_MEM subprograms, listed in alphabetical order (by grouping, then by name within each grouping), with the SDO_NET_MEM.SET_MAX_MEMORY_SIZE procedure listed first because it does not fit in any grouping.

SDO_NET_MEM.SET_MAX_MEMORY_SIZE

Format

```
SDO_NET_MEM.SET_MAX_MEMORY_SIZE(
   maxsize  IN NUMBER);
```

Description

Sets the Java maximum heap size for an application to run in an Oracle Java virtual machine.

Parameters

maxsize
Number of bytes for the Java maximum heap size.

Usage Notes

If you encounter the `java.lang.OutOfMemoryError` exception, you can use this procedure to increase the maximum heap size.

If you specify a value greater than the system limit, the system limit is used.

Examples

The following example sets the Java maximum heap size to 536870912 (512 MB).

```
EXECUTE SDO_NET_MEM.SET_MAX_MEMORY_SIZE(536870912);
```

SDO_NET_MEM.LINK.GET_CHILD_LINKS

Format

```
SDO_NET_MEM.LINK.GET_CHILD_LINKS(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the child links of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the child links of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getChildLinks` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the child links of the link whose link ID is 1001 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.LINK.GET_CHILD_LINKS(net_mem, 1001);
DBMS_OUTPUT.PUT('Link 1001 has the following child links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Link 1001 has the following child links: 1108 1109
```

SDO_NET_MEM.LINK.GET_CO_LINK_IDS

Format

```
SDO_NET_MEM.LINK.GET_CO_LINK_IDS(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the co-links of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with the co-links of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The **co-link** of a link is that link with its direction reversed. That is, the start node of the co-link is the end node of the original link, and the end node of the co-link is the start node of the original link.

This function is analogous to using the `getCoLinks` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the co-links of the link whose link ID is 9876 in the current network memory object. (This example assumes that a variable named res_array of type SDO_NUMBER_ARRAY has been declared, and that a variable named `net_mem` of type VARCHAR2 contains a network name associated with a network memory object.)

```
res_array := SDO_NET_MEM.LINK.GET_CO_LINK_IDS(net_mem, 9876);
```

SDO_NET_MEM.LINK.GET_COST

Format

```
SDO_NET_MEM.LINK.GET_COST(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the cost value of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the cost value of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getCost` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the cost value of a link, use the SDO_NET_MEM.LINK.SET_COST procedure.

Examples

The following example returns the cost of the link whose link ID is 1104 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.LINK.GET_COST(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The cost of link 1104 is: ' || res_numeric);
. . .
The cost of link 1104 is: 10
```

SDO_NET_MEM.LINK.GET_END_MEASURE

Format

```
SDO_NET_MEM.LINK.GET_END_MEASURE(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the end measure value of a link in an LRS network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the end measure value of a link in an LRS network in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getEndMeasure` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the end measure of the link whose link ID is 104 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.LINK.GET_END_MEASURE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The end measure of link 104 is: ' || res_numeric);
. . .
The end measure of link 104 is: 6
```

SDO_NET_MEM.LINK.GET_END_NODE_ID

Format

```
SDO_NET_MEM.LINK.GET_END_NODE_ID(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID of the end node of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the node ID number of the end node of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getEndNode` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the end node of a link, use the SDO_NET_MEM.LINK.SET_END_NODE procedure.

Examples

The following example returns the end node ID of the link whose link ID is 1104 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.LINK.GET_END_NODE_ID(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The end node of link 1104 is: ' || res_numeric);
. . .
The end node of link 1104 is: 104
```

SDO_NET_MEM.LINK.GET_GEOM_ID

Format

```
SDO_NET_MEM.LINK.GET_GEOM_ID(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the geometry ID of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the geometry ID number of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getGeomID` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the geometry ID of a link, use the SDO_NET_MEM.LINK.SET_GEOM_ID procedure.

Examples

The following example returns the geometry ID of the link whose link ID is 104 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.LINK.GET_GEOM_ID(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The geometry ID of link 104 is: ' || res_numeric);
. . .
The geometry ID of link 104 is: 1003
```

SDO_NET_MEM.LINK.GET_GEOMETRY

Format

```
SDO_NET_MEM.LINK.GET_GEOMETRY(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the spatial geometry object for a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the SDO_GEOMETRY object for a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getGeometry` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the spatial geometry of a link, use the SDO_NET_MEM.LINK.SET_GEOMETRY procedure.

Examples

The following example returns the spatial geometry of the link whose link ID is 9876 in the current network memory object. (This example assumes that a variable named `res_geom` of type SDO_GEOMETRY has been declared, and that a variable named `net_mem` of type VARCHAR2 contains a network name associated with a network memory object.)

```
res_geom := SDO_NET_MEM.LINK.GET_GEOMETRY(net_mem, 9876)
```

SDO_NET_MEM.LINK.GET_LEVEL

Format

```
SDO_NET_MEM.LINK.GET_LEVEL(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the hierarchy level of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the numeric hierarchy level of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getLinkLevel` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the hierarchy level of a link, use the SDO_NET_MEM.LINK.SET_LEVEL procedure.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the hierarchy level of the link whose link ID is 1001 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.LINK.GET_LEVEL(net_mem, 1001);
DBMS_OUTPUT.PUT_LINE('The hierarchy level of link 1001 is: ' || res_numeric);
. . .
The hierarchy level of link 1001 is: 2
```

SDO_NET_MEM.LINK.GET_NAME

Format

```
SDO_NET_MEM.LINK.GET_NAME(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the name of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the name of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getName` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the name of a link, use the SDO_NET_MEM.LINK.SET_NAME procedure.

Examples

The following example returns the name of the link whose link ID is 1104 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.LINK.GET_NAME(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The name of link 1104 is: ' || res_string);
. . .
The name of link 1104 is: N3N4
```

SDO_NET_MEM.LINK.GET_PARENT_LINK_ID

Format

```
SDO_NET_MEM.LINK.GET_PARENT_LINK_ID(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the link ID number of the parent link of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the link ID number of the parent link of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getParentLink` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the parent link of a link, use the SDO_NET_MEM.LINK.SET_PARENT_LINK procedure.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the parent link ID of the link whose link ID is 1108 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.LINK.GET_PARENT_LINK_ID(net_mem, 1108);
DBMS_OUTPUT.PUT_LINE('The parent link of link 1108 is: ' || res_numeric);
. . .
The parent link of link 1108 is: 1001
```

SDO_NET_MEM.LINK.GET_SIBLING_LINK_IDS

Format

```
SDO_NET_MEM.LINK.GET_SIBLING_LINK_IDS(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the sibling links of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the link ID numbers of the sibling links of a link in the specified network memory object. Sibling links are links that have the same parent link. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getSiblingLinkArray` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

For information about parent and child nodes and links in a network hierarchy, see Section 5.5. However, note that parent and child links can be defined without a network hierarchy or in the same level of a network hierarchy.

Examples

The following example returns the sibling links of the link whose link ID is 1108 in the current network memory object. In this case, the only sibling link is the one whose link ID is 1109. Both links are children of the link whose link ID is 1001, which is between nodes `HN1` and `HN2`. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.LINK.GET_SIBLING_LINK_IDS(net_mem, 1108);
DBMS_OUTPUT.PUT('Link 1108 has the following sibling links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Link 1108 has the following sibling links: 1109
```

SDO_NET_MEM.LINK.GET_START_MEASURE

Format

```
SDO_NET_MEM.LINK.GET_START_MEASURE(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the start measure value of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the start measure value of a link in an LRS network in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getStartMeasure` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the start measure value of the link whose link ID is 104 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.LINK.GET_START_MEASURE(net_mem, 104);
DBMS_OUTPUT.PUT_LINE('The start measure of link 104 is: ' || res_numeric);
. . .
The start measure of link 104 is: 0
```

SDO_NET_MEM.LINK.GET_START_NODE_ID

Format

```
SDO_NET_MEM.LINK.GET_START_NODE_ID(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID of the start node of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the node ID value of the start node of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getStartNode` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the start node of a link, use the SDO_NET_MEM.LINK.SET_START_NODE procedure.

Examples

The following example returns the start node of the link whose link ID is 1004 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.LINK.GET_START_NODE_ID(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The start node of link 1104 is: ' || res_numeric);
. . .
The start node of link 1104 is: 103
```

SDO_NET_MEM.LINK.GET_STATE

Format

```
SDO_NET_MEM.LINK.GET_STATE(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the state of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the state of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The state is one of the following string values: `ACTIVE` or `INACTIVE`. The link state determines whether or not the link is considered by network analysis functions, such as SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH. If the state is `ACTIVE`, the link is considered by network analysis functions; if the state is `INACTIVE`, the link is ignored by these functions.

This function is analogous to using the `getState` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the link state, use the SDO_NET_MEM.LINK.SET_STATE procedure.

Examples

The following example returns the state of the link whose link ID is 1104 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.LINK.GET_STATE(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('The state of link 1104 is: ' || res_string);
. . .
The state of link 1104 is: ACTIVE
```

SDO_NET_MEM.LINK.GET_TYPE

Format

```
SDO_NET_MEM.LINK.GET_TYPE(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the type of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the type of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getType` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To set the type of a link, use the SDO_NET_MEM.LINK.SET_TYPE procedure.

Examples

The following example sets and gets the type of the link whose link ID is 1119 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_TYPE(net_mem, 1119, 'Associative');
res_string := SDO_NET_MEM.LINK.GET_TYPE(net_mem, 1119);
DBMS_OUTPUT.PUT_LINE('The type of link 1119 is: ' || res_string);
. . .
The type of link 1119 is: Associative
```

SDO_NET_MEM.LINK.IS_ACTIVE

Format

```
SDO_NET_MEM.LINK.IS_ACTIVE(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a link is active.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the string value `TRUE` if the link in the specified network memory object is active, and the string value `FALSE` if the link is not active. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isActive` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the link whose link ID is 1104 in the current network memory object is active. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.LINK.IS_ACTIVE(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 active?: ' || res_string);
. . .
Is link 1104 active?: TRUE
```

SDO_NET_MEM.LINK.IS_LOGICAL

Format

```
SDO_NET_MEM.LINK.IS_LOGICAL(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a link is in a logical network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the string value `TRUE` if the link in the specified network memory object is in a logical network, and the string value `FALSE` if the link is not in a logical network. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isLogical` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the link whose link ID is 1104 in the current network memory object is in a logical network. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.LINK.IS_LOGICAL(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 a logical link?: ' || res_string);
. . .
Is link 1104 a logical link?: TRUE
```

SDO_NET_MEM.LINK.IS_TEMPORARY

Format

```
SDO_NET_MEM.LINK.IS_TEMPORARY(
   net_mem IN VARCHAR2,
   link_id  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a link is temporary.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

Usage Notes

This function returns the string value `TRUE` if the link in the specified network memory object is temporary, and the string value `FALSE` if the link is not temporary. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

Temporary links, nodes, and paths are not saved in the database when you call the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure.

This function is analogous to using the `isTemporary` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the link whose link ID is 1104 in the current network memory object is temporary. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.LINK.IS_TEMPORARY(net_mem, 1104);
DBMS_OUTPUT.PUT_LINE('Is link 1104 temporary?: ' || res_string);
. . .
Is link 1104 temporary?: FALSE
```

SDO_NET_MEM.LINK.SET_COST

Format

```
SDO_NET_MEM.LINK.SET_COST(
   net_mem  IN VARCHAR2,
   link_id      IN NUMBER,
   cost         IN NUMBER);
```

Description

Sets the cost value of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

cost
Cost value.

Usage Notes

This procedure sets the cost of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setCost` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the cost value of a link, use the SDO_NET_MEM.LINK.GET_COST function.

Examples

The following example sets the cost of the link whose link ID is 1119 in the current network memory object to 40. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_COST(net_mem, 1119, 40);
```

SDO_NET_MEM.LINK.SET_END_NODE

Format

```
SDO_NET_MEM.LINK.SET_END_NODE(
   net_mem      IN VARCHAR2,
   link_id      IN NUMBER,
   end_node_id  IN NUMBER);
```

Description

Sets the end node of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

end_node_id
Node ID number.

Usage Notes

This procedure sets the end node of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setEndNode` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the end node of a link, use the SDO_NET_MEM.LINK.GET_END_NODE_ID function.

Examples

The following example sets the end node of the link whose link ID is 1119 in the current network memory object to the node with the node ID value of 109. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_END_NODE(net_mem, 1119, 109);
```

SDO_NET_MEM.LINK.SET_GEOM_ID

Format

```
SDO_NET_MEM.LINK.SET_GEOM_ID(
   net_mem  IN VARCHAR2,
   link_id      IN NUMBER,
   geom_id   IN NUMBER);
```

Description

Sets the geometry ID number of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

geom_id
Geometry ID number.

Usage Notes

This procedure sets the geometry ID number of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setGeomID` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the geometry ID of a link, use the SDO_NET_MEM.LINK.GET_GEOM_ID function.

Examples

The following example sets the geometry ID of the link whose link ID is 302 in the current network memory object to 9999.

```
SDO_NET_MEM.LINK.SET_GEOM_ID(net_mem, 302, 9999);
```

SDO_NET_MEM.LINK.SET_GEOMETRY

Format

```
SDO_NET_MEM.LINK.SET_GEOMETRY(
   net_mem  IN VARCHAR2,
   link_id  IN NUMBER,
   geom     IN SDO_GEOMETRY);
```

Description

Sets the spatial geometry of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

geom
Spatial geometry object.

Usage Notes

This procedure sets the SDO_GEOMETRY object for a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setGeometry` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the spatial geometry of a link, use the SDO_NET_MEM.LINK.GET_GEOMETRY function.

Examples

The following example sets the spatial geometry of the link whose link ID is 5678 in the current network memory object to a specified line string SDO_GEOMETRY object. (This example assumes that a variable named `net_mem` of type VARCHAR2 contains a network name associated with a network memory object.)

```
SDO_NET_MEM.LINK.SET_GEOMETRY(net_mem, 5678,
   SDO_GEOMETRY(2002, NULL, NULL,
     SDO_ELEM_INFO_ARRAY(1,2,1),
       SDO_ORDINATE_ARRAY(9, 4, 1,1)));
```

SDO_NET_MEM.LINK.SET_LEVEL

Format

```
SDO_NET_MEM.LINK.SET_LEVEL(
   net_mem  IN VARCHAR2,
   link_id  IN NUMBER,
   level    IN NUMBER);
```

Description

Sets the hierarchy level of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

level
Hierarchy level number.

Usage Notes

This procedure sets the hierarchy level of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setLinkLevel` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the hierarchy level of a link, use the SDO_NET_MEM.LINK.GET_LEVEL function.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example sets the hierarchy level of the link whose link ID is 1119 in the current network memory object to 2. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_LEVEL(net_mem, 1119, 2);
```

SDO_NET_MEM.LINK.SET_MEASURE

Format

```
SDO_NET_MEM.LINK.SET_MEASURE(
   net_mem        IN VARCHAR2,
   link_id        IN NUMBER,
   start_measure  IN NUMBER,
   end_measure    IN NUMBER);
```

Description

Sets the start and end measure values of a link in an LRS network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

start_measure
Start measure value.

end_measure
End measure value.

Usage Notes

This procedure sets the start and end measure values of a link in an LRS network in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setMeasure` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the start measure of a link, use the SDO_NET_MEM.LINK.GET_START_MEASURE function. To get the end measure of a link, use the SDO_NET_MEM.LINK.GET_END_MEASURE function.

Examples

The following example sets the measure values of the link whose link ID is 302 in the current network memory object, so that the start measure is 111 and the end measure is 114.16. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
SDO_NET_MEM.LINK.SET_MEASURE(net_mem, 302, 111, 114.16);
```

SDO_NET_MEM.LINK.SET_NAME

Format

```
SDO_NET_MEM.LINK.SET_NAME(
   net_mem   IN VARCHAR2,
   link_id   IN NUMBER,
   link_name IN VARCHAR2);
```

Description

Sets the name of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

link_name
Link name string.

Usage Notes

This procedure sets the name of a link in an LRS network in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setName` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the name of a link, use the SDO_NET_MEM.LINK.GET_NAME function.

Examples

The following example sets the name of the link whose link ID is 1119 in the current network memory object to `My favorite link`. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_NAME(net_mem, 1119, 'My favorite link');
```

SDO_NET_MEM.LINK.SET_PARENT_LINK

Format

```
SDO_NET_MEM.LINK.SET_PARENT_LINK(
   net_mem        IN VARCHAR2,
   link_id        IN NUMBER,
   parent_link_id IN NUMBER);
```

Description

Sets the parent link of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

parent_link_id
Link ID number of the parent link.

Usage Notes

This procedure sets the parent link of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setParentLink` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the parent link of a link, use the SDO_NET_MEM.LINK.GET_PARENT_LINK_ID function.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example sets the parent link of the link whose link ID is 1119 in the current network memory object to the link whose link ID value is 1001. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_PARENT_LINK(net_mem, 1119, 1001);
```

SDO_NET_MEM.LINK.SET_START_NODE

Format

```
SDO_NET_MEM.LINK.SET_START_NODE(
   net_mem       IN VARCHAR2,
   link_id       IN NUMBER,
   srart_node_id IN NUMBER);
```

Description

Sets the start node of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

start_node_id
Node ID number.

Usage Notes

This procedure sets the start node of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setStartNode` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the start node of a link, use the SDO_NET_MEM.LINK.GET_START_NODE_ID function.

Examples

The following example sets the start of the link whose link ID is 1119 in the current network memory object to the node whose node ID is 110. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_START_NODE(net_mem, 1119, 110);
```

SDO_NET_MEM.LINK.SET_STATE

Format

```
SDO_NET_MEM.LINK.SET_STATE(
   net_mem  IN VARCHAR2,
   link_id      IN NUMBER,
   state        IN VARCHAR2);
```

Description

Sets the state value of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

state
State value. Must be one of the following strings: `ACTIVE` or `INACTIVE`.

Usage Notes

This procedure sets the state of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The link state determines whether or not the link is considered by network analysis functions, such as SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH. If the state is `ACTIVE`, the link is considered by network analysis functions; if the state is `INACTIVE`, the link is ignored by these functions.

This procedure is analogous to using the `setState` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the link state, use the SDO_NET_MEM.LINK.GET_STATE function.

Examples

The following example sets the state of the link whose link ID is 1119 in the current network memory object to `INACTIVE`. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.LINK.SET_STATE(net_mem, 1119, 'INACTIVE');
```

SDO_NET_MEM.LINK.SET_TYPE

Format

```
SDO_NET_MEM.LINK.SET_TYPE(
   net_mem  IN VARCHAR2,
   link_id      IN NUMBER,
   type         IN VARCHAR2);
```

Description

Sets the type of a link.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID number.

type
String reflecting a user-determined type for the link.

Usage Notes

This procedure sets the type of a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setType` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

To get the type of a link, use the SDO_NET_MEM.LINK.GET_TYPE function.

Examples

The following example sets the type of the link whose link ID is 302 in the current network memory object to `Normal street`. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
SDO_NET_MEM.LINK.SET_TYPE(net_mem, 302, 'Normal street');
```

SDO_NET_MEM.NETWORK.ADD_LINK

Format

```
SDO_NET_MEM.NETWORK.ADD_LINK(
   net_mem     IN VARCHAR2,
   link_id    IN NUMBER,
   link_name    IN NUMBER,
   start_node_id        IN NUMBER,
   end_node_id        IN NUMBER,
   cost       IN NUMBER);
```

or

```
SDO_NET_MEM.NETWORK.ADD_LINK(
   net_mem     IN VARCHAR2,
   link_id    IN NUMBER,
   link_name    IN NUMBER,
   start_node_id        IN NUMBER,
   end_node_id        IN NUMBER,
   cost       IN NUMBER,
   geom_id       IN NUMBER,
   start_measure       IN NUMBER,
   end_measure       IN NUMBER);
```

or

```
SDO_NET_MEM.ADD_LINK(
   net_mem     IN VARCHAR2,
   link_id    IN NUMBER,
   link_name    IN NUMBER,
   start_node_id        IN NUMBER,
   end_node_id        IN NUMBER,
   geom       IN SDO_GEOMETRY,
   cost       IN NUMBER);
```

Description

Adds a link to a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
ID number of the link to be added.

link_name
Name of the link to be added.

start_node_id
Node ID of the start node of the link to be added.

end_node_id
Node ID of the end node of the link to be added.

cost
Cost value associated with the link.

geom_id
For an LRS geometry, the geometry ID of the geometry object.

start_measure
For an LRS geometry, the start measure value in the geometry object corresponding to the start node for this link.

end_measure
For an LRS geometry, the end measure value in the geometry object corresponding to the end node for this link.

geom
SDO_GEOMETRY object (line or contiguous line string geometry) representing the link to be added.

Usage Notes

This procedure adds a link in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `addLink` method of the `Network` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example adds a link whose link ID is 9901 in the current network memory object. (This example is an excerpt from Example 5–1 in Section 5.8.)

```
-- Add a link with ID=9901, name=N901N1, cost=20 from node N901 to node N1.
sdo_net_mem.network.add_link(net_mem=>'XYZ_NETWORK', link_id=>9901,
  link_name=>'N901N1', start_node_id=>901, end_node_id=>101, cost=>20);
```

SDO_NET_MEM.NETWORK.ADD_LRS_NODE

Format

```
SDO_NET_MEM.NETWORK.ADD_LRS_NODE(
   net_mem              IN VARCHAR2,
   node_id              IN NUMBER,
   node_name            IN VARCHAR2,
   geom_id              IN NUMBER,
   measure              IN NUMBER,
   geom                 IN SDO_GEOMETRY,
   external_network_id  IN NUMBER,
   external_node_id     IN NUMBER);
```

Description

Adds an LRS node (point geometry with measure information) to a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID of the node to be added.

node_name
Name of the node to be added.

geom_id
Geometry ID of the geometry object.

measure
Measure value of the node to be added.

geom
Geometry object of the node to be added. Must be a linear referencing system (LRS) geometry with the measure value for the third dimension value.

external_network_id
If the node is also a node in an external network, the network ID of the external network.

external_node_id
If the node is also a node in an external network, the node ID of the node in the external network.

Usage Notes

This procedure adds an LRS node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

Examples

The following example adds an LRS node whose node ID is 901 in the network memory object for a network named MY_LRS_NETWORK.

```
DECLARE
  res_string  VARCHAR2(100);
BEGIN
-- Add an LRS node with ID=901.
SDO_NET_MEM.NETWORK.ADD_LRS_NODE(net_mem=>'MY_LRS_NETWORK', node_id=>901,
  node_name=>'N901', geom_id=>9901, measure=>8,
  geom=>SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
    SDO_ORDINATE_ARRAY(8,13,9)),
  external_network_id=>0, external_node_id=>0);
-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME('ROADS_NETWORK', 901);
DBMS_OUTPUT.PUT_LINE('The name of node 901 is: ' || res_string);
END;
/
. . .
The name of node 901 is: N901
```

SDO_NET_MEM.NETWORK.ADD_NODE

Format

```
SDO_NET_MEM.NETWORK.ADD_NODE(
   net_mem              IN VARCHAR2,
   node_id              IN NUMBER,
   node_name            IN VARCHAR2,
   geom                 IN SDO_GEOMETRY DEFAULT NULL
   external_network_id  IN NUMBER,
   external_node_id     IN NUMBER);
```

Description

Adds a node (with no associated geometry object) to a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID of the node to be added.

node_name
Name of the node to be added.

geom
Geometry object associated with the node. If this parameter is null, no geometry object is associated with the node.

external_network_id
If the node is also a node in an external network, the network ID of the external network.

external_node_id
If the node is also a node in an external network, the node ID of the node in the external network.

Usage Notes

This procedure adds a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `addNode` method of the `Network` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example adds a node whose node ID is 901 in the current network memory object. (This example is an excerpt from Example 5–1 in Section 5.8.)

```
-- Add a node with ID=901, and set its name to N901.
sdo_net_mem.network.add_node(net_mem=>'XYZ_NETWORK', node_id=>901,
  node_name=>'N901', external_network_id=>0, external_node_id=>0);
```

SDO_NET_MEM.NETWORK.ADD_PATH

Format

```
SDO_NET_MEM.NETWORK.ADD_PATH(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER);
```

or

```
SDO_NET_MEM.NETWORK.ADD_PATH(
   net_mem  IN VARCHAR2,
   path_ids    IN SDO_NUMBER_ARRAY);
```

Description

Adds one or more paths to a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID of the path to be added.

path_ids
SDO_NUMBER_ARRAY object specifying the path IDs of the paths to be added.

Usage Notes

This procedure adds one or more paths in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `addPath` method of the `Network` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example adds a path whose path ID is stored in a variable named `path_id`. (This example is an excerpt from Example 5–1 in Section 5.8.)

```
sdo_net_mem.network.add_path(net_mem=>'XYZ_NETWORK', path_id=>path_id);
```

SDO_NET_MEM.NETWORK.ADD_SDO_NODE

Format

```
SDO_NET_MEM.NETWORK.ADD_SDO_NODE(
   net_mem          IN VARCHAR2,
   node_id          IN NUMBER,
   node_name        IN VARCHAR2,
   x                IN VARCHAR2,
   y                IN VARCHAR2,
   srid             IN NUMBER);
```

Description

Adds a node (with associated spatial coordinates) to a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID of the node to be added.

node_name
Name of the node to be added.

x
X-axis coordinate value for the node to be added.

y
Y-axis coordinate value for the node to be added.

srid
Coordinate system (spatial reference system) associated with the node. Must match the SRID associated with the node table in the USER_SDO_GEOM_METADATA view.

Usage Notes

This procedure adds a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `addNode` method of the `Network` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example adds an SDO node whose node ID is 801 in the network memory object for a network named `ROADS_NETWORK`.

```
DECLARE
  res_string  VARCHAR2(100);
BEGIN
-- Add an SDO node with ID=801.
SDO_NET_MEM.NETWORK.ADD_SDO_NODE(net_mem=>'ROADS_NETWORK',
  node_id=>801, node_name=>'N801', x=>8, y=>12, srid=>null);
-- GET_NAME
res_string := SDO_NET_MEM.NODE.GET_NAME('ROADS_NETWORK', 801);
DBMS_OUTPUT.PUT_LINE('The name of node 801 is: ' || res_string);
END;
/
. . .
The name of node 801 is: N801
```

SDO_NET_MEM.NETWORK.DELETE_LINK

Format

```
SDO_NET_MEM.NETWORK.DELETE_LINK(
   net_mem  IN VARCHAR2,
   link_id      IN NUMBER);
```

Description

Deletes a link from a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

link_id
Link ID of the link to be deleted.

Usage Notes

This procedure deletes a link from the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `deleteLink` method of the `Network` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example deletes the link whose link ID is 302 in the network memory object for a network named `MY_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK.DELETE_LINK('MY_NETWORK', 302);
```

SDO_NET_MEM.NETWORK.DELETE_NODE

Format

```
SDO_NET_MEM.NETWORK.DELETE_NODE(
   net_mem  IN VARCHAR2,
   node_id      IN NUMBER);
```

Description

Deletes a node from a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID of the node to be deleted.

Usage Notes

This procedure deletes a node from the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `deleteNode` method of the `Network` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example deletes the node whose node ID is 8 in the network memory object for a network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK.DELETE_NODE('ROADS_NETWORK', 8);
```

SDO_NET_MEM.NETWORK.DELETE_PATH

Format

```
SDO_NET_MEM.NETWORK.DELETE_PATH(
   net_mem  IN VARCHAR2,
   path_id      IN NUMBER);
```

Description

Deletes a path from a network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID of the path to be deleted.

Usage Notes

This procedure deletes a path from the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

An exception is raised if the specified network memory object is read-only.

This procedure is analogous to using the `deletePath` method of the `Network` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example deletes the path whose path ID is 1 in the network memory object for a network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK.DELETE_PATH('ROADS_NETWORK', 1);
```

SDO_NET_MEM.NETWORK.GET_MAX_LINK_ID

Format

```
SDO_NET_MEM.LINK.GET_MAX_LINK_ID(
   net_mem IN VARCHAR2,
   ) RETURN NUMBER;
```

Description

Returns the link ID with the highest numeric value.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the link ID with the highest numeric value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getMaxLinkId` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the link ID with the highest numeric value in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SELECT SDO_NET_MEM.NETWORK.GET_MAX_LINK_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum link ID = ' || res_numeric);
. . .
Maximum link ID = 1119
```

SDO_NET_MEM.NETWORK.GET_MAX_NODE_ID

Format

```
SDO_NET_MEM.LINK.GET_MAX_NODE_ID(
   net_mem IN VARCHAR2,
   ) RETURN NUMBER;
```

Description

Returns the node ID with the highest numeric value.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the node ID with the highest numeric value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getMaxNodeId` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the node ID with the highest numeric value in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SELECT SDO_NET_MEM.NETWORK.GET_MAX_NODE_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum node ID = ' || res_numeric);
. . .
Maximum node ID = 114
```

SDO_NET_MEM.NETWORK.GET_MAX_PATH_ID

Format

```
SDO_NET_MEM.LINK.GET_MAX_PATH_ID(
   net_mem IN VARCHAR2,
   ) RETURN NUMBER;
```

Description

Returns the path ID with the highest numeric value.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the path ID with the highest numeric value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getMaxPathId` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path ID with the highest numeric value in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SELECT SDO_NET_MEM.NETWORK.GET_MAX_PATH_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum path ID = ' || res_numeric);
. . .
Maximum path ID = 28
```

SDO_NET_MEM.NETWORK.GET_MAX_SUBPATH_ID

Format

```
SDO_NET_MEM.LINK.GET_MAX_SUBPATH_ID(
   net_mem IN VARCHAR2,
   ) RETURN NUMBER;
```

Description

Returns the subpath ID with the highest numeric value.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the subpath ID with the highest numeric value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getMaxSubpathId` method of the `Link` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the subpath ID with the highest numeric value in the current network memory object. In this case the network has no subpaths, and so the returned value is 0 (zero). (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SELECT SDO_NET_MEM.NETWORK.GET_MAX_SUBPATH_ID(net_mem)
  INTO res_numeric FROM DUAL;
DBMS_OUTPUT.PUT_LINE('Maximum subpath ID = ' || res_numeric);
. . .
Maximum subpath ID = 0
```

SDO_NET_MEM.NETWORK_MANAGER.ALL_PATHS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.ALL_PATHS(
   net_mem          IN VARCHAR2,
   start_node_id    IN NUMBER,
   goal_node_id     IN NUMBER,
   depth_limit      IN NUMBER,
   cost_limit       IN NUMBER,
   no_of_solutions  IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns all paths between two nodes.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

start_node_id
Node ID of the start node in the pair of nodes between which to find paths.

goal_node_id
Node ID of the end (goal) node in the pair of nodes between which to find paths.

depth_limit
Maximum number of links in the resultant paths. If this parameter is null, no maximum number of links is applied.

cost_limit
Maximum cost total value of the links in a path. If this parameter is null, no cost limit is applied.

no_of_solutions
Maximum number of paths to be returned. If this parameter is null, all paths that meet the other criteria for this function are returned.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns paths between a start node and an end (goal) node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `allPaths` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns up to 5 paths, each up to a maximum cost value of 200, between the nodes with node ID values 101 and 105 in the current network memory object. It also displays some information about each returned path. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.ALL_PATHS(net_mem,101,105,10,200,5);
DBMS_OUTPUT.PUT_LINE('For each path from node 101 to node 105: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric ||
                       ' has the following properties: ');
  cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
  DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
  res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_array(indx));
  DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);
END LOOP;

For each path from node 101 to node 105:
Path 7 has the following properties:
Path 7 cost: 50
Is path 7 closed? FALSE
Path 8 has the following properties:
Path 8 cost: 70
Is path 8 closed? FALSE
Path 9 has the following properties:
Path 9 cost: 70
Is path 9 closed? FALSE
Path 10 has the following properties:
Path 10 cost: 90
Is path 10 closed? FALSE
Path 11 has the following properties:
Path 11 cost: 120
Is path 11 closed? FALSE
```

SDO_NET_MEM.NETWORK_MANAGER.CREATE_LOGICAL_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.CREATE_LOGICAL_NETWORK(
   network_name           IN VARCHAR2,
   no_of_hierarchy_levels IN NUMBER,
   is_directed            IN VARCHAR2,
   node_table_name        IN VARCHAR2 DEFAULT NULL,
   node_cost_column       IN VARCHAR2 DEFAULT NULL,
   link_table_name        IN VARCHAR2 DEFAULT NULL,
   link_cost_column       IN VARCHAR2 DEFAULT NULL,
   path_table_name        IN VARCHAR2 DEFAULT NULL,
   path_link_table_name   IN VARCHAR2 DEFAULT NULL,
   sub_path_table_name    IN VARCHAR2 DEFAULT NULL,
   is_complex             IN VARCHAR2 DEFAULT 'FALSE');
```

Description

Creates a logical network, creates all necessary tables, and updates the network metadata.

Parameters

network_name
Name of the network.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A string value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If this parameter is null, the node table name will be in the form <*network-name*>_NODE$.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If this parameter is null, the cost column name will be COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If this parameter is null, the link table name will be in the form <*network-name*>_LINK$.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If this parameter is null, the cost column name will be COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If this parameter is null, the path table name will be in the form <*network-name*>_PATH$.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If this parameter is null, the path-link table name will be in the form <*network-name*>_PLINK$.

sub_path_table_name
Name of the subpath table to be created. (The subpath table is explained in Section 5.9.5.) If this parameter is null, the subpath table name will be in the form <*network-name*>_SPATH$.

is_complex
Reserved for future use. Ignored for the current release.

Usage Notes

This procedure creates a logical network in a network memory object, the use of which is explained in Section 5.8.

This procedure provides a convenient way to create a logical network when the node, link, and optional related tables do not already exist. The procedure creates the network in memory. When you save the network objects to the database using the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure, the node, link, path, and path-link tables for the network are created, and the appropriate information is inserted in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

Examples

The following example creates a logical network named `MY_LOGICAL_NET`. The network has two hierarchy levels and is undirected (`'FALSE'` for `is_directed`).

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.CREATE_LOGICAL_NETWORK('MY_LOGICAL_NET', -
  2, 'FALSE', 'MY_NODE_TABLE', 'COST', 'MY_LINK_TABLE', 'COST', -
  'MY_PATH_TABLE', 'MY_PATHLINK_TABLE', 'FALSE');
```

SDO_NET_MEM.NETWORK_MANAGER.CREATE_LRS_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.CREATE_LRS_NETWORK(
   network_name            IN VARCHAR2,
   no_of_hierarchy_levels  IN NUMBER,
   is_directed             IN VARCHAR2,
   srid                    IN NUMBER,
   no_of_dims              IN NUMBER,
   node_table_name         IN VARCHAR2 DEFAULT NULL,
   node_cost_column        IN VARCHAR2 DEFAULT NULL,
   link_table_name         IN VARCHAR2 DEFAULT NULL,
   link_cost_column        IN VARCHAR2 DEFAULT NULL,
   lrs_table_name          IN VARCHAR2,
   lrs_geom_column         IN VARCHAR2,
   path_table_name         IN VARCHAR2 DEFAULT NULL,
   path_geom_column        IN VARCHAR2 DEFAULT NULL,
   path_link_table_name    IN VARCHAR2 DEFAULT NULL,
   sub_path_table_name     IN VARCHAR2 DEFAULT NULL,
   sub_path_geom_column IN VARCHAR2 DEFAULT NULL,
   is_complex              IN VARCHAR2 DEFAULT 'FALSE');
```

Description

Creates a spatial network containing LRS SDO_GEOMETRY objects, creates all necessary tables, and updates the network metadata.

Parameters

network_name
Name of the network.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A string value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

srid
Coordinate system (spatial reference system) associated with the network. Must be specified as either null (no coordinate system is associated) or a value from the SRID column of the SDO_COORD_REF_SYS table (described in *Oracle Spatial Developer's Guide*).

no_of_dims
Number of dimensions for the data, including the LRS measure dimension.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If this parameter is null, the node table name will be in the form <*network-name*>_NODE$.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If this parameter is null, the cost column name will be COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If this parameter is null, the link table name will be in the form <*network-name*>_LINK$.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If this parameter is null, the cost column name will be COST.

lrs_table_name
Name of the table to be created for the spatial LRS geometries

lrs_geom_column
Name of the column of type SDO_GEOMETRY in the table for the spatial LRS geometries.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If this parameter is null, the path table name will be in the form <*network-name*>_PATH$.

path_geom_column
Name of the column of type SDO_GEOMETRY in the path table. If this parameter is null, the geometry column name will be GEOMETRY.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If this parameter is null, the path-link table name will be in the form <*network-name*>_PLINK$.

sub_path_table_name
Name of the subpath table to be created. (The subpath table is explained in Section 5.9.5.) If this parameter is null, the subpath table name will be in the form <*network-name*>_SPATH$.

sub_path_geom_column
Name of the column of type SDO_GEOMETRY in the subpath table. If this parameter is null, the geometry column name will be GEOMETRY.

is_complex
Reserved for future use. Ignored for the current release.

Usage Notes

This procedure creates a spatial LRS network in a network memory object, the use of which is explained in Section 5.8.

This procedure provides a convenient way to create a spatial LRS network when the node, link, and optional related tables do not already exist. The procedure creates the network in memory. When you save the network objects to the database using the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure, the node, link, path, and path-link tables for the network are created, and the appropriate information is inserted in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

Examples

The following example creates an LRS network named `MY_LRS_NET`. The network has one hierarchy level, is undirected (`'FALSE'` for `is_directed`), is based on SRID 8307, and has three-dimensional geometries.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.CREATE_LRS_NETWORK('MY_LRS_NET', -
  1, 'FALSE', 8307, 3, 'MY_NODE_TABLE', 'COST', 'MY_LINK_TABLE', 'COST', -
  'MY_LRS_TABLE', 'GEOM', 'MY_PATH_TABLE', 'GEOM', -
  'MY_PATHLINK_TABLE', 'FALSE');
```

SDO_NET_MEM.NETWORK_MANAGER.CREATE_REF_CONSTRAINTS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.CREATE_REF_CONSTRAINTS(
   network          IN VARCHAR2);
```

Description

Creates referential integrity constraints on the link and path tables.

Parameters

network
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This procedure creates referential integrity constraints on the link and path tables in a network memory object, the use of which is explained in Section 5.8.

When the referential integrity constraints are created, they are automatically enabled. You can disable the referential integrity constrains by calling the SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS procedure, and enable them again by calling the SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS procedure.

This procedure is analogous to using the `createRefConstraints` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example creates referential integrity constraints on the link and path tables in the network memory object for the network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.CREATE_REF_CONSTRAINTS('ROADS_NETWORK');
```

SDO_NET_MEM.NETWORK_MANAGER.CREATE_SDO_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.CREATE_SDO_NETWORK(
   network_name              IN VARCHAR2,
   no_of_hierarchy_levels    IN NUMBER,
   is_directed               IN VARCHAR2,
   srid                      IN NUMBER,
   no_of_dims                IN NUMBER,
   node_table_name           IN VARCHAR2 DEFAULT NULL,
   node_geom_column          IN VARCHAR2 DEFAULT NULL,
   node_cost_column          IN VARCHAR DEFAULT NULL2,
   link_table_name           IN VARCHAR2 DEFAULT NULL,
   link_geom_column          IN VARCHAR2 DEFAULT NULL,
   link_cost_column          IN VARCHAR2 DEFAULT NULL,
   path_table_name           IN VARCHAR2 DEFAULT NULL,
   path_geom_column          IN VARCHAR2 DEFAULT NULL,
   path_link_table_name      IN VARCHAR2 DEFAULT NULL,
   sub_path_table_name       IN VARCHAR2 DEFAULT NULL,
   sub_path_geom_column IN VARCHAR2 DEFAULT NULL,
   is_complex                IN VARCHAR2 DEFAULT 'FALSE');
```

Description

Creates a spatial network containing non-LRS SDO_GEOMETRY objects, creates all necessary tables, and updates the network metadata.

Parameters

network_name
Name of the network.

no_of_hierarchy_levels
Number of hierarchy levels for links in the network. (For an explanation of network hierarchy, see Section 5.5.)

is_directed
A string value. `TRUE` indicates that the links are directed; `FALSE` indicates that the links are undirected (not directed).

srid
Coordinate system (spatial reference system) associated with the network. Must be specified as either null (no coordinate system is associated) or a value from the SRID column of the SDO_COORD_REF_SYS table (described in *Oracle Spatial Developer's Guide*).

no_of_dims
Number of dimensions for the spatial data.

node_table_name
Name of the node table to be created. (The node table is explained in Section 5.9.1.) If this parameter is null, the node table name will be in the form <*network-name*>_NODE$.

node_geom_column
Name of the column of type SDO_GEOMETRY in the node table. (The node table is explained in Section 5.9.1.) If this parameter is null, the geometry column name will be GEOMETRY.

node_cost_column
Name of the cost column in the node table. (The node table is explained in Section 5.9.1.) If this parameter is null, the cost column name will be COST.

link_table_name
Name of the link table to be created. (The link table is explained in Section 5.9.2.) If this parameter is null, the link table name will be in the form <*network-name*>_LINK$.

link_geom_column
Name of the column of type SDO_GEOMETRY in the link table. (The link table is explained in Section 5.9.2.) If this parameter is null, the geometry column name will be GEOMETRY.

link_cost_column
Name of the cost column in the link table. (The link table is explained in Section 5.9.2.) If this parameter is null, the cost column name will be COST.

path_table_name
Name of the path table to be created. (The path table is explained in Section 5.9.3.) If this parameter is null, the path table name will be in the form <*network-name*>_PATH$.

path_geom_column
Name of the column of type SDO_GEOMETRY in the path table. (The path table is explained in Section 5.9.3.) If this parameter is null, the geometry column name will be GEOMETRY.

path_link_table_name
Name of the path-link table to be created. (The path-link table is explained in Section 5.9.4.) If this parameter is null, the path-link table name will be in the form <*network-name*>_PLINK$.

sub_path_table_name
Name of the subpath table to be created. (The subpath table is explained in Section 5.9.5.) If this parameter is null, the subpath table name will be in the form <*network-name*>_SPATH$.

sub_path_geom_column
Name of the column of type SDO_GEOMETRY in the path table. (The path table is explained in Section 5.9.3.) If this parameter is null, the geometry column name will be GEOMETRY.

is_complex
Reserved for future use. Ignored for the current release.

Usage Notes

This procedure creates a spatial (SDO) network in a network memory object, the use of which is explained in Section 5.8.

This procedure provides a convenient way to create a spatial (SDO) network when the node, link, and optional related tables do not already exist. The procedure creates the network in memory. When you save the network objects to the database using the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure, the node, link, path, and path-link tables for the network are created, and the appropriate information is inserted in the xxx_SDO_NETWORK_METADATA views (described in Section 5.10.1).

Examples

The following example creates an SDO network named `MY_SDO_NET`. The network has one hierarchy level, is undirected (`'FALSE'` for `is_directed`), is based on SRID 8307, and has two-dimensional geometries.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.CREATE_SDO_NETWORK('MY_SDO_NET', -
  1, 'FALSE', 8307, 2, -
  'MY_NODE_TABLE', 'GEOM', 'COST', 'MY_LINK_TABLE', 'GEOM', 'COST', -
  'MY_PATH_TABLE', 'GEOM', 'MY_PATHLINK_TABLE', 'FALSE');
```

SDO_NET_MEM.NETWORK_MANAGER.DEREGISTER_CONSTRAINT

Format

```
SDO_NET_MEM.NETWORK_MANAGER.DEREGISTER_CONSTRAINT(
   constraint_name  IN VARCHAR2);
```

Description

Unloads (removes) the class for the specified network constraint from the Java repository in the database, and deletes the row for that constraint from the USER_SDO_NETWORK_CONSTRAINTS view (described in Section 5.10.2).

Note: This procedure is deprecated, and it will not be supported in a future release. You are encouraged to use the SDO_NET.DEREGISTER_CONSTRAINT procedure instead.

Parameters

constraint_name
Name of the network constraint. Must match a value in the CONSTRAINT column of the USER_SDO_NETWORK_CONSTRAINTS view.

Usage Notes

Use this procedure if you want to disable a network constraint that you had previously enabled, such as by using the SDO_NET_MEM.NETWORK_MANAGER.REGISTER_CONSTRAINTprocedure. For more information about network constraints, see Section 5.6.

This procedure is analogous to using the `deregisterConstraint` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example deregisters (disables) a network constraint named `GivenProhibitedTurn`.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.DEREGISTER_CONSTRAINT('GivenProhibitedTurn');
```

SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS(
   network          IN VARCHAR2);
```

Description

Disables referential integrity constraints on the link and path tables.

Parameters

network
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This procedure disables referential integrity constraints (created using the SDO_NET_MEM.NETWORK_MANAGER.CREATE_REF_CONSTRAINTS) on the link and path tables in a network memory object, the use of which is explained in Section 5.8.

When the referential integrity constraints are created, they are automatically enabled. You can disable the referential integrity constrains by calling the SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS procedure, and enable them again by calling the SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS procedure.

This procedure is analogous to using the `disableRefConstraints` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example disables the referential integrity constraints on the link and path tables in the network memory object for the network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS('ROADS_NETWORK');
```

SDO_NET_MEM.NETWORK_MANAGER.DROP_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.DROP_NETWORK(
   net_mem  IN VARCHAR2);
```

Description

Drops (deletes) the network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This procedure deletes the network in a network memory object, the use of which is explained in Section 5.8.

This procedure is analogous to using the `dropNetwork` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example deletes the network in the network memory object for the network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.DROP_NETWORK('ROADS_NETWORK');
```

SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS(
   network          IN VARCHAR2);
```

Description

Enables referential integrity constraints on the link and path tables.

Parameters

network
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This procedure enables referential integrity constraints (that had been disabled using the SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS procedure) on the link and path tables in a network memory object, the use of which is explained in Section 5.8.

When the referential integrity constraints are created, they are automatically enabled. You can disable the referential integrity constrains by calling the SDO_NET_MEM.NETWORK_MANAGER.DISABLE_REF_CONSTRAINTS procedure, and enable them again by calling the SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS procedure.

This procedure is analogous to using the `enableRefConstraints` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example enables the referential integrity constraints on the link and path tables in the network memory object for the network named `ROADS_NETWORK`.

```
EXECUTE SDO_NET_MEM.NETWORK_MANAGER.ENABLE_REF_CONSTRAINTS('ROADS_NETWORK');
```

SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS(
   net_mem IN VARCHAR2
   ) RETURN NUMBER;
```

Description

Returns the number of groups of connected components.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the number of connected components in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

Connected components are a group of nodes, connected by links, such that each node in the group can reach or be reached from all other nodes in the group. For example, the logical network shown in Example 5–5 in Section 5.13.4 contains two groups of connected components: in one group, all nodes in hierarchy level 1 can reach or be reached by all other nodes in that level; and in the other group, the two nodes in hierarchy level 2 can reach each other.

Nodes that belong to the same component have the same component number. To get the component number for a node, use the SDO_NET_MEM.NODE.GET_COMPONENT_NO function. To set the component number for a node, use the SDO_NET_MEM.NODE.SET_COMPONENT_NO procedure.

This function is analogous to using the `findConnectedComponents` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the child links of the link whose link ID is 1001 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS(net_mem);
DBMS_OUTPUT.PUT_LINE('The number of connected components is: ' || res_numeric);
. . .
The number of connected components is: 2
```

SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHABLE_NODES

Format

```
SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHABLE_NODES(
   net_mem          IN VARCHAR2,
   source_node_id   IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the node ID numbers of nodes that can be reached by a path from a specified source node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

source_node_id
Node ID of the source node.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of node ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

If no results can be found that match the criteria, this function returns a null value.

This function is analogous to using the `findReachableNodes` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example nodes that can be reached from the node whose node ID is 101 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHABLE_NODES(net_mem,101);
DBMS_OUTPUT.PUT_LINE('Reachable nodes from 101: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Reachable nodes from 101:
103 102 104 106 105 107 108 109 110 113 111 112 114
```

SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHING_NODES

Format

```
SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHING_NODES(
   net_mem         IN VARCHAR2,
   target_node_id  IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the node ID numbers of nodes that can reach (by a path) a specified target node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

target_node_id
Node ID of the target node.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of node ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

If no results can be found that match the criteria, this function returns a null value.

This function is analogous to using the `findReaching_Nodes` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example nodes from which the node whose node ID is 101 in the current network memory object can be reached. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.FIND_REACHING_NODES(net_mem,101);
DBMS_OUTPUT.PUT_LINE('Nodes from which 101 can be reached: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Nodes from which 101 can be reached:
103 102 104 106 105 107 108 109 110 113 111 112 114
```

SDO_NET_MEM.NETWORK_MANAGER.IS_REACHABLE

Format

```
SDO_NET_MEM.NETWORK_MANAGER.IS_REACHABLE(
   net_mem          IN VARCHAR2,
   source_node_id   IN NUMBER,
   target_node_id   IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN VARCHAR2;
```

Description

Checks if a specified target node can be reached by a path from a specified source node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

source_node_id
Node ID of the source node.

target_node_id
Node ID of the target node.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns the string `TRUE` if the target node can be reached from the source node, and the string `FALSE` if the target node cannot be reached from the source node. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isReachable` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the child links of the link whose link ID is 1001 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NETWORK_MANAGER.IS_REACHABLE(net_mem,101,105);
DBMS_OUTPUT.PUT_LINE('Can node 101 reach node 105? ' || res_string);
. . .
Can node 101 reach node 105? TRUE
```

SDO_NET_MEM.NETWORK_MANAGER.LIST_NETWORKS

Format

SDO_NET_MEM.NETWORK_MANAGER.LIST_NETWORKS() RETURN VARCHAR2;

Description

Returns a list of networks.

Parameters

None.

Usage Notes

This function returns a comma-delimited list of network names in the current network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

Examples

The following example returns the names of all networks with network memory objects in the cache. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_string := SDO_NET_MEM.NETWORK_MANAGER.LIST_NETWORKS;
DBMS_OUTPUT.PUT_LINE('The current in-memory network(s) is/are: ' || res_string);
. . .
The current in-memory network(s) is/are: ROADS_NETWORK
```

SDO_NET_MEM.NETWORK_MANAGER.MCST_LINK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.MCST_LINK(
   net_mem IN VARCHAR2
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the minimum cost spanning tree.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with the link ID values of links that make up the minimum cost spanning tree in the specified network memory object. The minimum cost spanning tree is the path with the lowest total cost that visits all nodes in the network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The Kruskal algorithm is used to determine the minimum cost spanning tree.

This function is analogous to using the `mcstLinkArray` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the links in the minimum cost spanning tree of the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.MCST_LINK(net_mem);
DBMS_OUTPUT.PUT('Network ' || net_mem || ' has the following MCST links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Network XYZ_NETWORK has the following MCST links: 1001 1101 1104 1107 1110 1114
1117 1102 1105 1108 1111 1115 1118 1113
```

SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS

Format

```
SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(
   net_mem          IN VARCHAR2,
   start_node_id    IN NUMBER,
   no_of_neighbors  IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the path ID numbers of paths leading to nodes that are the nearest neighbors (determined by total cost) of a specified start node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

start_node_id
Node ID of the start node.

no_of_neighbors
Maximum number of path IDs to return.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of path ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

If no results can be found that match the criteria, this function returns a null value.

To determine the end node in each path returned, use the SDO_NET_MEM.PATH.GET_END_NODE_ID function. To determine the links in each path returned, use the SDO_NET_MEM.PATH.GET_LINK_IDS function.

This function is analogous to using the `nearestNeighbors` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path IDs of the paths to the three nodes nearest to the node whose node ID is 101 in the current network memory object. It also displays the link IDs for each link in each of the returned paths. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.NEAREST_NEIGHBORS(net_mem,101,3);
```

```
DBMS_OUTPUT.PUT_LINE('Path IDs to the nearest 3 neighbors of node 101 are: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', which contains links: ');
  var1_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
  FOR indx1 IN var1_array.FIRST..var1_array.LAST
  LOOP
    var1_numeric := var1_array(indx1);
    DBMS_OUTPUT.PUT(var1_numeric || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
. . .
Path IDs to the nearest 3 neighbors of node 101 are:
1, which contains links: 1101
2, which contains links: 1102
3, which contains links: 1102 1104
```

SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK(
   net_mem        IN VARCHAR2,
   allow_updates  IN VARCHAR2);
```

or

```
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK(
   net_mem        IN VARCHAR2,
   network        IN VARCHAR2,
   xmin           IN NUMBER,
   ymin           IN NUMBER,
   xmax           IN NUMBER,
   ymax           IN NUMBER,
   allow_updates  IN VARCHAR2);
```

Description

Creates a network memory object in virtual memory cache containing all network objects in a network (first format), or those objects completely inside a specified minimum bounding rectangle (MBR) (second format).

Parameters

net_mem
For the first format: name of the network, all of whose network objects (nodes, links, paths, subpaths) are to be copied into a network memory object that has the same name as the network. For the second format: user-specified name for the network memory object to contain only those network objects completely inside the MBR specified by the `xmin`, `xmax`, `ymin`, and `ymax` parameters for the network specified by the `network` parameter.

network
Name of the network from which to add network objects (nodes, links, paths) into the network memory object. Only those objects in the MBR specified by the `xmin`, `xmax`, `ymin`, and `ymax` parameters are loaded.

xmin
Minimum X coordinate value of the MBR from which objects in the specified network are to be added to the network memory object.

ymin
Minimum Y coordinate value of the MBR from which objects in the specified network are to be added to the network memory object.

xmax
Maximum X coordinate value of the MBR from which objects in the specified network are to be added to the network memory object.

ymax
Maximum Y coordinate value of the MBR from which objects in the specified network are to be added to the network memory object.

allow_updates
TRUE specifies that the network memory object is updatable; that is, you can perform editing operations in the cache and have them written back to the database. FALSE specifies that the network memory object is read-only; that is, you cannot perform editing operations in the cache and have them written back to the database.

Usage Notes

This procedure creates a network memory object, the use of which is explained in Section 5.8.

For better performance, if you only need to retrieve information or to perform network analysis operations, specify the string FALSE for the allow_updates parameter.

If allow_updates is specified as TRUE, rows in the network tables (node, link, path, and path-link) are locked at the database level (through SELECT ... FOR UPDATE NOWAIT statements). This prevents update or delete operations on the locked elements in other SQL sessions; however, insert operations in other sessions are allowed.

This procedure is analogous to using the readNetwork method of the NetworkManager class of the client-side Java API (described in Section 5.11.2).

Examples

The following example places a copy of all network objects in the network named NET_LOGICAL into an updatable ('TRUE' for allow_updates) network memory object.

```
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK('NET_LOGICAL', 'TRUE');
```

The following example places a copy of all network objects inside a specified area of the network named HILLSBOROUGH_NETWORK into a read-only ('FALSE' for allow_updates) network memory object named HILLS_PART1. The objects loaded are in the rectangular area with one corner at longitude/latitude coordinates (-71.64, 43.32) and the other corner at coordinates (-71.32, 43.64). (On a traditional map, these would be the lower-left and upper-right corners, respectively.)

```
SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK('HILLS_PART1', 'HILLSBOROUGH_NETWORK',
-71.64, 43.32, -71.32, 43.64, 'FALSE');
```

SDO_NET_MEM.NETWORK_MANAGER.REGISTER_CONSTRAINT

Format

```
SDO_NET_MEM.NETWORK_MANAGER.REGISTER_CONSTRAINT(
   constraint_name  IN VARCHAR2,
   class_name       IN VARCHAR2,
   directory_name   IN VARCHAR2,
   description      IN VARCHAR2);
```

Description

Loads the compiled Java code for the specified network constraint into the Java class repository in the database, and loads the class name into the CLASS column of the USER_SDO_NETWORK_CONSTRAINTS view (described in Section 5.10.2).

> **Note:** This procedure is deprecated, and it will not be supported in a future release. You are encouraged to use the SDO_NET.REGISTER_CONSTRAINT procedure instead.

Parameters

constraint_name
Name of the network constraint.

class_name
Fully qualified name (including the name of the package) of the class that implements the network constraint.

directory_name
Name of the directory object (created using the SQL statement CREATE DIRECTORY) that identifies the location of the class file created when you compiled the network constraint.

description
Description of the network constraint.

Usage Notes

Before you call this procedure, you must insert a row into the USER_SDO_NETWORK_CONSTRAINTS view, compile the code for the Java class that implements the network constraint, and use the CREATE DIRECTORY statement to create a directory object identifying the location of the compiled class. For more information about network constraints, see Section 5.6.

To delete the row for the constraint from the USER_SDO_NETWORK_CONSTRAINTS view and thus disable the constraint, use the SDO_NET_MEM.NETWORK_MANAGER.DEREGISTER_CONSTRAINT procedure.

This procedure is analogous to using the `registerConstraint` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example registers a network constraint named `GivenProhibitedTurn`.

```
-- Set up the network constraint.
REM
REM Create the geor_dir on the file system first.
REM
-- Connect as SYSTEM.
DECLARE
  -- This is the directory that contains the CLASS file generated when you
  -- compiled the network constraint.
  geor_dir varchar2(1000) := 'C:\my_data\files81\PROTOTYPES\NETWORK_
CONSTRAINT\PLSQL_EXAMPLE';
BEGIN
  EXECUTE IMMEDIATE 'CREATE OR REPLACE DIRECTORY work_dir AS''' || geor_dir ||
'''';
END;
/
GRANT read,write on directory work_dir to net_con;

-- Connect as the user that will register the constraint.

REM
REM Compile GivenProhibitedTurn before you register the constraint.
REM
BEGIN
  SDO_NET_MEM.NETWORK_MANAGER.REGISTER_CONSTRAINT('GivenProhibitedTurn',
     'com/network/constraints/ProhibitedTurn',
     'WORK_DIR', 'This is a network constraint that '||
     'prohibits certain turns');

END;
/
```

SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH

Format

```
SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(
   net_mem          IN VARCHAR2,
   start_node_id    IN NUMBER,
   goal_node_id     IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN NUMBER;
```

Description

Returns the path ID number of the shortest path (based on the A* search algorithm, and considering costs) between a start node and a goal (end) node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

start_node_id
Node ID of the start node.

goal_node_id
Node ID of the goal (end) node.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns a path ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function returns a null value if no path can be made between the specified nodes. For example, if the state of one or more nodes or links is `INACTIVE`, and if this condition causes all possible paths to be ignored, the function will return a null value.

To determine the links in the returned path, use the SDO_NET_MEM.PATH.GET_LINK_IDS function.

This function is analogous to using the `shortestPath` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path ID of the shortest path between the nodes with node ID values 101 and 105 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH(net_mem,101,105);
```

```
DBMS_OUTPUT.PUT_LINE('The shortest path from node 101 to node 105 is path ID: ' ||
res_numeric);
```

SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH_DIJKSTRA

Format

```
SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH_DIJKSTRA(
   net_mem          IN VARCHAR2,
   start_node_id    IN NUMBER,
   goal_node_id     IN NUMBER,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN NUMBER;
```

Description

Returns the path ID number of the shortest path (based on the Dijkstra search algorithm, and considering costs) between a start node and a goal (end) node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

start_node_id
Node ID of the start node.

goal_node_id
Node ID of the goal (end) node.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns a path ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function returns a null value if no path can be made between the specified nodes. For example, if the state of one or more nodes or links is `INACTIVE`, and if this condition causes all possible paths to be ignored, the function will return a null value.

To determine the links in the returned path, use the SDO_NET_MEM.PATH.GET_LINK_IDS function.

This function is analogous to using the `shortestPathDijkstra` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path ID of the shortest path (based on the Dijkstra search algorithm, and considering costs) between the nodes with node ID values 101 and 105 in the current network memory object. It also displays information about the returned path. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH_DIJKSTRA(net_
mem,101,105);
DBMS_OUTPUT.PUT_LINE('The shortest Dijkstra path from node 101 to node 105 is ' ||
res_numeric);

DBMS_OUTPUT.PUT_LINE('The following are characteristics of this shortest path: ');
cost := SDO_NET_MEM.PATH.GET_COST(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Path ' || res_numeric || ' cost: ' || cost);
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, res_numeric);
DBMS_OUTPUT.PUT_LINE('Is path ' || res_numeric || ' closed? ' || res_string);

res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, res_numeric);
DBMS_OUTPUT.PUT('Path ' || res_numeric || ' has nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
The shortest Dijkstra path from node 101 to node 105 is 13
The following are characteristics of this shortest path:
Path 13 cost: 50
Is path 13 closed? FALSE
Path 13 has links: 1102 1104 1105
Path 13 has nodes: 101 103 104 105
```

SDO_NET_MEM.NETWORK_MANAGER.TSP_PATH

Format

```
SDO_NET_MEM.NETWORK_MANAGER.TSP_PATH(
   net_mem          IN VARCHAR2,
   nd_array         IN SDO_NUMBER_ARRAY,
   is_closed        IN VARCHAR2,
   use_exact_cost   IN VARCHAR2,
   constraint       IN VARCHAR2 DEFAULT NULL
   ) RETURN NUMBER;
```

Description

Returns the path ID number of the most efficient (in cost or distance) path that includes all specified nodes, that is, the path that solves the "traveling salesman problem" (TSP) for the specified set of nodes.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

nd_array
An SDO_NUMBER_ARRAY object specifying the node ID numbers of the nodes to be included in the path. The first node specified is always the start node of the returned path. For a closed path, the first node specified is also the last node of the returned path; for an open path, the last node specified is the last node of the returned path.

is_closed
The string value `TRUE` if the path must be closed (that is, start node and end node in returned path are the same node), or the string value `FALSE` if the path must be open (that is, end node in the returned path is different from the start node).

use_exact_cost
The string value `TRUE` if the cost values of links are to be used in calculating the TSP path, or the string value `FALSE` if the Cartesian distances of links are to be used in calculating the TSP path.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns a path ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function returns a null value if no TSP path can be made using the specified nodes. For example, if the state of one or more nodes or links is `INACTIVE`, and if this condition causes all possible paths to be ignored, the function will return a null value.

If multiple possible paths can be constructed that meet all requirements of the request (for example, if two paths have the same lowest total cost), the returned path can be any of these possible paths.

To determine the links in the returned path, use the SDO_NET_MEM.PATH.GET_LINK_IDS function.

This function is analogous to using the `tspPath` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path ID of the open TSP path that starts at node ID 2, ends at node ID 6, and includes node ID 4 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.NETWORK_MANAGER.TSP_PATH(net_mem, sdo_number_array(2,
4, 6), 'FALSE', 'TRUE');
DBMS_OUTPUT.PUT_LINE('Open TSP path ID for N2, N4, N6: ' || res_numeric);
DBMS_OUTPUT.PUT_LINE('which contains these links: ');
var1_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, res_numeric);
FOR indx1 IN var1_array.FIRST..var1_array.LAST
LOOP
  var1_numeric := var1_array(indx1);
  DBMS_OUTPUT.PUT(var1_numeric || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Open TSP path ID for N2, N4, N6: 4
which contains these links:
102 103 104 105
```

SDO_NET_MEM.NETWORK_MANAGER.VALIDATE_NETWORK_SCHEMA

Format

```
SDO_NET_MEM.NETWORK_MANAGER.VALIDATE_NETWORK_SCHEMA(
   net_mem IN VARCHAR2
   ) RETURN VARCHAR2;
```

Description

Validates the network tables.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This function returns the string `TRUE` if the network-related tables in the specified network memory object are valid, and it returns a specific Oracle error message if one or more tables are not valid. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `validateNetworkSchema` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example validates the network tables in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NETWORK_MANAGER.VALIDATE_NETWORK_SCHEMA(net_mem);
 23  DBMS_OUTPUT.PUT_LINE('Is network ' || net_mem || ' valid? ' || res_string);
. . .
Is network XYZ_NETWORK valid? TRUE
```

SDO_NET_MEM.NETWORK_MANAGER.WITHIN_COST

Format

```
SDO_NET_MEM.NETWORK_MANAGER.WITHIN_COST(
   net_mem        IN VARCHAR2,
   start_node_id  IN NUMBER,
   cost_limit     IN NUMBER,
   constraint     IN VARCHAR2 DEFAULT NULL
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array of path IDs of the shortest path to each node that is reachable within a specified cost from a specified start node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

start_node_id
Node ID of the start node.

cost_limit
Maximum total path cost.

constraint
Name of the network constraint to be applied. If this parameter is null, no network constraint is applied. (For information about network constraints, see Section 5.6.)

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of path ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

If no results can be found that match the criteria, this function returns a null value.

To determine the end node in each path returned, use the SDO_NET_MEM.PATH.GET_END_NODE_ID function. To determine the links in each path returned, use the SDO_NET_MEM.PATH.GET_LINK_IDS function.

This function is analogous to using the `withinCost` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the path ID values of the shortest path to each node in the current network memory object that is reachable within a cost of 100 from the node with the node ID value of 102. It also displays the end node for each returned path. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NETWORK_MANAGER.WITHIN_COST(net_mem,102,100);
```

```
DBMS_OUTPUT.PUT('Shortest path IDs to nodes within cost of 100 from node 102: ');
DBMS_OUTPUT.PUT_LINE(' ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  res_numeric := res_array(indx);
  DBMS_OUTPUT.PUT(res_numeric || ', whose end node is: ');
  var1_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, res_numeric);
  DBMS_OUTPUT.PUT(var1_numeric);
  DBMS_OUTPUT.PUT_LINE(' ');
END LOOP;
. . .
Shortest path IDs to nodes within cost of 100 from node 102:
14, whose end node is: 101
15, whose end node is: 103
16, whose end node is: 104
17, whose end node is: 105
18, whose end node is: 106
19, whose end node is: 108
20, whose end node is: 107
```

SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK

Format

```
SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK(
   net_mem  IN VARCHAR2);
```

Description

Saves to the database the network objects in a network memory object.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

Usage Notes

This procedure saves the network objects in a network memory object, the use of which is explained in Section 5.8.

Temporary links, nodes, and paths are not saved in the database when you call this procedure.

This procedure is analogous to using the `writeNetwork` method of the `NetworkManager` class of the client-side Java API (described in Section 5.11.2).

Examples

The following example saves to the database the network objects in the network memory object for a network named `XYZ_NETWORK`.

```
sdo_net_mem.network_manager.write_network(net_mem=>'XYZ_NETWORK');
```

SDO_NET_MEM.NODE.GET_ADJACENT_NODE_IDS

Format

```
SDO_NET_MEM.NODE.GET_ADJACENT_NODE_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the node ID numbers of nodes that are adjacent to a specified node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of node ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getAdjacentNodeArray` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the nodes adjacent to the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_ADJACENT_NODE_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following adjacent nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');

Node 103 has the following adjacent nodes: 102 104 101
```

SDO_NET_MEM.NODE.GET_CHILD_NODE_IDS

Format

```
SDO_NET_MEM.NODE.GET_CHILD_NODE_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the node ID numbers of nodes that are child nodes of a specified node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of node ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getChildNodeArray` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the child nodes of the node whose node ID is 1 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_CHILD_NODE_IDS(net_mem, 1);
DBMS_OUTPUT.PUT('Node 1 has the following child nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
. . .
Node 1 has the following child nodes: 104 103 105 102 106 101
```

SDO_NET_MEM.NODE.GET_COMPONENT_NO

Format

```
SDO_NET_MEM.NODE.GET_COMPONENT_NO(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the component number of a specified node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a numeric component number for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

All nodes in a group of connected components have the same component number. For an explanation of connected components, see the Usage Notes for the SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS function.

This function is analogous to using the `getComponentNo` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the component number for a node, use the SDO_NET_MEM.NODE.SET_COMPONENT_NO procedure.

Examples

The following example returns the component number of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NODE.GET_COMPONENT_NO(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The component number of node 103 is: ' || res_numeric);
. . .
The component number of node 103 is: 1
```

SDO_NET_MEM.NODE.GET_COST

Format

```
SDO_NET_MEM.NODE.GET_COST(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the cost value of a specified node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a numeric cost value for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getCost` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the cost value for a node, use the SDO_NET_MEM.NODE.SET_COST procedure.

Examples

The following example returns the cost of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NODE.GET_COST(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The cost of node 103 is: ' || res_numeric);
. . .
The cost of node 103 is: 0
```

SDO_NET_MEM.NODE.GET_GEOM_ID

Format

```
SDO_NET_MEM.NODE.GET_GEOM_ID(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the geometry ID number of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a numeric geometry ID value for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getGeomID` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the geometry ID value for a node, use the SDO_NET_MEM.NODE.SET_GEOM_ID procedure.

Examples

The following example returns the geometry ID of the node whose node ID is 3 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.NODE.GET_GEOM_ID(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The geometry ID of node 3 is: ' || res_numeric);
. . .
The geometry ID of node 3 is: 1001
```

SDO_NET_MEM.NODE.GET_GEOMETRY

Format

```
SDO_NET_MEM.NODE.GET_GEOMETRY(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the spatial geometry for a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_GEOMETRY object for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getGeometry` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the spatial geometry for a node, use the SDO_NET_MEM.NODE.SET_GEOMETRY procedure.

Examples

The following example returns the spatial geometry of the node whose node ID is 3 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_geom := SDO_NET_MEM.NODE.GET_GEOMETRY(net_mem, 3);
```

SDO_NET_MEM.NODE.GET_HIERARCHY_LEVEL

Format

```
SDO_NET_MEM.NODE.GET_HIERARCHY_LEVEL(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the hierarchy level of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a numeric hierarchy level for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getHierarchyLevel` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the hierarchy level for a node, use the SDO_NET_MEM.NODE.SET_HIERARCHY_LEVEL procedure.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the hierarchy level of the node whose node ID is 1 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NODE.GET_HIERARCHY_LEVEL(net_mem, 1);
DBMS_OUTPUT.PUT_LINE('The hierarchy level of node 1 is: ' || res_numeric);
. . .
The hierarchy level of node 1 is: 2
```

SDO_NET_MEM.NODE.GET_IN_LINK_IDS

Format

```
SDO_NET_MEM.NODE.GET_IN_LINK_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the link ID numbers of links that are inbound links to a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of link ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getInLinks` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the inbound links to the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_IN_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following inbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
. . .
Node 103 has the following inbound links: 1102 1103
```

SDO_NET_MEM.NODE.GET_INCIDENT_LINK_IDS

Format

```
SDO_NET_MEM.NODE.GET_INCIDENT_LINK_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the link ID numbers of links that are to (that is, incident upon) a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of link ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getIncidentLinks` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the incident links of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_INCIDENT_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following incident links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
. . .
Node 103 has the following incident links: 1102 1104 1103
```

SDO_NET_MEM.NODE.GET_MEASURE

Format

```
SDO_NET_MEM.NODE.GET_MEASURE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the measure value of a node in an LRS network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a numeric measure value for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getMeasure` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the measure value for a node, use the SDO_NET_MEM.NODE.SET_MEASURE procedure.

Examples

The following example returns the measure value of the node whose node ID is 3 in the current network memory object. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
res_numeric := SDO_NET_MEM.NODE.GET_MEASURE(net_mem, 3);
DBMS_OUTPUT.PUT_LINE('The measure value of node 3 is: ' || res_numeric);
. . .
The measure value of node 3 is: 8
```

SDO_NET_MEM.NODE.GET_NAME

Format

```
SDO_NET_MEM.NODE.GET_NAME(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the name of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a node name string for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getName` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the name of a node, use the SDO_NET_MEM.NODE.SET_NAME procedure.

Examples

The following example returns the name of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.GET_NAME(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The name of node 103 is: ' || res_string);
. . .
The name of node 103 is: N3
```

SDO_NET_MEM.NODE.GET_OUT_LINK_IDS

Format

```
SDO_NET_MEM.NODE.GET_OUT_LINK_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the link ID numbers of links that are outbound links from a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of link ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getOutLinks` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the outbound links from the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_OUT_LINK_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following outbound links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
. . .
Node 103 has the following outbound links: 1104
```

SDO_NET_MEM.NODE.GET_PARENT_NODE_ID

Format

```
SDO_NET_MEM.NODE.GET_PARENT_NODE_ID(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID number of the parent node of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns numeric node ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getParentNode` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the parent node of a node, use the SDO_NET_MEM.NODE.SET_PARENT_NODE procedure.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the parent node of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_numeric := SDO_NET_MEM.NODE.GET_PARENT_NODE_ID(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The parent node of node 103 is: ' || res_numeric);
. . .
The parent node of node 103 is: 1
```

SDO_NET_MEM.NODE.GET_SIBLING_NODE_IDS

Format

```
SDO_NET_MEM.NODE.GET_SIBLING_NODE_IDS(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the node ID numbers of nodes that are sibling nodes of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object of node ID values in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getSiblingNodeArray` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Sibling nodes are nodes that have the same parent node in a hierarchical network. For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example returns the sibling nodes of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_array := SDO_NET_MEM.NODE.GET_SIBLING_NODE_IDS(net_mem, 103);
DBMS_OUTPUT.PUT('Node 103 has the following sibling nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Node 103 has the following sibling nodes: 104 105 102 106 101
```

SDO_NET_MEM.NODE.GET_STATE

Format

```
SDO_NET_MEM.NODE.GET_STATE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the state of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a state name string for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The state is one of the following string values: `ACTIVE` or `INACTIVE`. The node state determines whether or not the node is considered by network analysis functions, such as SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH. If the state is `ACTIVE`, the node is considered by network analysis functions; if the state is `INACTIVE`, the node is ignored by these functions.

This function is analogous to using the `getState` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the state of a node, use the SDO_NET_MEM.NODE.SET_STATE procedure.

Examples

The following example returns the state of the node whose node ID is 103 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.GET_STATE(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('The state of node 103 is: ' || res_string);
. . .
The state of node 103 is: ACTIVE
```

SDO_NET_MEM.NODE.GET_TYPE

Format

```
SDO_NET_MEM.NODE.GET_TYPE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the type of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns a type name string for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getType` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To set the type of a node, use the SDO_NET_MEM.NODE.SET_TYPE procedure.

Examples

The following example sets the type of the node whose node ID is 114 in the current network memory object, and then returns the type. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
-- SET_TYPE
-- Set the type of node 114 to 'Research'.
SDO_NET_MEM.NODE.SET_TYPE(net_mem, 114, 'Research');
-- GET_TYPE
res_string := SDO_NET_MEM.NODE.GET_TYPE(net_mem, 114);
DBMS_OUTPUT.PUT_LINE('The type of node 114 is: ' || res_string);
. . .
The type of node 114 is: Research
```

SDO_NET_MEM.NODE.IS_ACTIVE

Format

```
SDO_NET_MEM.NODE.IS_ACTIVE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a node is active.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns the string `TRUE` if the node in the specified network memory object is active, or `FALSE` if the node is not active. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isActive` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the node whose node ID is 103 in the current network memory object is active. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.IS_ACTIVE(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 active?: ' || res_string);
. . .
Is node 103 active?: TRUE
```

SDO_NET_MEM.NODE.IS_LOGICAL

Format

```
SDO_NET_MEM.NODE.IS_LOGICAL(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a node is in a logical network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns the string `TRUE` if the node in the specified network memory object is in a logical network, or `FALSE` if the node is not in a logical network. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isLogical` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the node whose node ID is 103 in the current network memory object is in a logical network. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.IS_LOGICAL(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 a logical node?: ' || res_string);
. . .
Is node 103 a logical node?: TRUE
```

SDO_NET_MEM.NODE.IS_TEMPORARY

Format

```
SDO_NET_MEM.NODE.IS_TEMPORARY(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a node is temporary.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This function returns the string `TRUE` if the node in the specified network memory object is temporary, or `FALSE` if the node is not temporary. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

Temporary links, nodes, and paths are not saved in the database when you call the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure.

This function is analogous to using the `isTemporary` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if the node whose node ID is 103 in the current network memory object is temporary. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.IS_TEMPORARY(net_mem, 103);
DBMS_OUTPUT.PUT_LINE('Is node 103 temporary?: ' || res_string);
. . .
Is node 103 temporary?: FALSE
```

SDO_NET_MEM.NODE.LINK_EXISTS

Format

```
SDO_NET_MEM.NODE.LINK_EXISTS(
   net_mem  IN VARCHAR2,
   node_id1  IN NUMBER,
   node_id2  IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a link exists between two nodes.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id1
Node ID number.

node_id2
Node ID number.

Usage Notes

This function returns the string `TRUE` if a link exists between the two nodes in the specified network memory object, or `FALSE` if a link does not exist. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `linkExists` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a link exists between the nodes with node ID values 103 and 104 in the current network memory object. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
res_string := SDO_NET_MEM.NODE.LINK_EXISTS(net_mem, 103, 104);
DBMS_OUTPUT.PUT_LINE('Does a link exist between nodes 103 and 104?: ' || res_
string);
. . .
Does a link exist between nodes 103 and 104?: TRUE
```

SDO_NET_MEM.NODE.MAKE_TEMPORARY

Format

```
SDO_NET_MEM.NODE.MAKE_TEMPORARY(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER);
```

Description

Makes a node temporary.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

Usage Notes

This procedure makes the node in the specified network memory object temporary. (Temporary links, nodes, and paths are not saved in the database when you call the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure.) For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `makeTemporary` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To check if a node in a network memory object is temporary, use the SDO_NET_MEM.NODE.IS_TEMPORARY function.

Examples

The following example makes the node whose node ID is 114 in the current network memory object a temporary node. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.MAKE_TEMPORARY(net_mem, 114);
```

SDO_NET_MEM.NODE.SET_COMPONENT_NO

Format

```
SDO_NET_MEM.NODE.SET_COMPONENT_NO(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   no           IN NUMBER);
```

Description

Sets the component number of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

no
Component number.

Usage Notes

This procedure sets a node component number value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

All nodes in a group of connected components have the same component number. For an explanation of connected components, see the Usage Notes for the SDO_NET_MEM.NETWORK_MANAGER.FIND_CONNECTED_COMPONENTS function.

This procedure is analogous to using the `setComponentNo` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the node component value for a node, use the SDO_NET_MEM.NODE.GET_COMPONENT_NO function.

Examples

The following example sets the component number of the node whose node ID is 114 in the current network memory object to 987. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.SET_COMPONENT_NO(net_mem, 114, 987);
```

SDO_NET_MEM.NODE.SET_COST

Format

```
SDO_NET_MEM.NODE.SET_COST(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   cost         IN NUMBER);
```

Description

Sets the cost value of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

cost
Cost value.

Usage Notes

This procedure sets a numeric node cost value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setCost` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the cost value for a node, use the SDO_NET_MEM.NODE.GET_COST function.

Examples

The following example sets the cost of the node whose node ID is 114 in the current network memory object to 40. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.SET_COST(net_mem, 114, 40);
```

SDO_NET_MEM.NODE.SET_GEOM_ID

Format

```
SDO_NET_MEM.NODE.SET_GEOM_ID(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   geom_id   IN NUMBER);
```

Description

Sets the geometry ID value of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

geom
Geometry ID number.

Usage Notes

This procedure sets a numeric node geometry ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setGeomID` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the geometry ID value for a node, use the SDO_NET_MEM.NODE.GET_GEOM_ID function.

Examples

The following example sets the geometry ID of the node whose node ID is 7 in the current network memory object to 99. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
SDO_NET_MEM.NODE.SET_GEOM_ID(net_mem, 7, 99);
```

SDO_NET_MEM.NODE.SET_GEOMETRY

Format

```
SDO_NET_MEM.NODE.SET_GEOMETRY(
   net_mem  IN VARCHAR2,
   node_id  IN NUMBER,
   geom     IN SDO_GEOMETRY);
```

Description

Sets the geometry for a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

geom
Spatial geometry object.

Usage Notes

This procedure creates an SDO_GEOMETRY object for the node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setGeometry` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the geometry for a node, use the SDO_NET_MEM.NODE.GET_GEOMETRY function.

Examples

The following example sets the geometry of the node whose node ID is 114 in the network memory object for a network named `MY_NETWORK`.

```
SDO_NET_MEM.NODE.SET_GEOMETRY('MY_NETWORK', 114,
  SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(9,4,NULL), NULL, NULL));
```

SDO_NET_MEM.NODE.SET_HIERARCHY_LEVEL

Format

```
SDO_NET_MEM.NODE.SET_HIERARCHY_LEVEL(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   level        IN NUMBER);
```

Description

Sets the hierarchy level of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

level
Hierarchy level number.

Usage Notes

This procedure sets a numeric hierarchy level value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setHierarchyLevel` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the hierarchy level of a node, use the SDO_NET_MEM.NODE.GET_HIERARCHY_LEVEL function.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example sets the hierarchy level whose node ID is 1 in the current network memory object to 2.

```
SDO_NET_MEM.NODE.SET_HIERARCHY_LEVEL(net_mem, 1, 2);
```

SDO_NET_MEM.NODE.SET_MEASURE

Format

```
SDO_NET_MEM.NODE.SET_MEASURE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   measure   IN NUMBER);
```

Description

Sets the measure value of a node in an LRS network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

measure
Measure value.

Usage Notes

This procedure sets a numeric node measure value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setMeasure` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the measure value of a node, use the SDO_NET_MEM.NODE.GET_MEASURE function.

Examples

The following example sets the measure value of the node whose node ID is 7 in the current network memory object to 30. (This example is an excerpt from Example 5–4 in Section 5.13.3.)

```
SDO_NET_MEM.NODE.SET_MEASURE(net_mem, 7, 30);
```

SDO_NET_MEM.NODE.SET_NAME

Format

```
SDO_NET_MEM.NODE.SET_NAME(
   net_mem     IN VARCHAR2,
   node_id     IN NUMBER,
   node_name   IN VARCHAR2);
```

Description

Sets the name of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

node_name
Node name.

Usage Notes

This procedure sets a node name string value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setName` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the name of a node, use the SDO_NET_MEM.NODE.GET_NAME function.

Examples

The following example sets the name of the node whose node ID is 114 in the current network memory object to the string `My favorite node`. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.SET_NAME(net_mem, 114, 'My favorite node');
```

SDO_NET_MEM.NODE.SET_PARENT_NODE

Format

```
SDO_NET_MEM.NODE.SET_PARENT_NODE(
   net_mem        IN VARCHAR2,
   node_id        IN NUMBER,
   parent_node_id IN NUMBER);
```

Description

Sets the parent node of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

parent_node_id
Parent node ID number.

Usage Notes

This procedure specifies the parent node for a node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setParentNode` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the ID value for a parent node, use the SDO_NET_MEM.NODE.GET_PARENT_NODE_ID function.

For information about parent and child nodes and links in a network hierarchy, see Section 5.5.

Examples

The following example sets the parent node of the node whose node ID is 114 in the current network memory object to the node whose node ID is 1. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.SET_PARENT_NODE(net_mem, 114, 1);
```

SDO_NET_MEM.NODE.SET_STATE

Format

```
SDO_NET_MEM.NODE.SET_STATE(
   net_mem  IN VARCHAR2,
   node_id    IN NUMBER,
   state        IN VARCHAR2);
```

Description

Sets the state of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

state
Node state. Must be one of the following string values: `ACTIVE` or `INACTIVE`.

Usage Notes

This procedure sets a node state string value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

The node state determines whether or not the node is considered by network analysis functions, such as SDO_NET_MEM.NETWORK_MANAGER.SHORTEST_PATH. If the state is `ACTIVE`, the node is considered by network analysis functions; if the state is `INACTIVE`, the node is ignored by these functions.

This procedure is analogous to using the `setState` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the node state, use the SDO_NET_MEM.NODE.GET_STATE function.

Examples

The following example sets the state of the node whose node ID is 111 in the current network memory object to the string `INACTIVE`. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
SDO_NET_MEM.NODE.SET_STATE(net_mem, 111, 'INACTIVE');
```

SDO_NET_MEM.NODE.SET_TYPE

Format

```
SDO_NET_MEM.NODE.SET_TYPE(
   net_mem  IN VARCHAR2,
   node_id  IN NUMBER,
   type     IN VARCHAR2);
```

Description

Sets the type of a node.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

node_id
Node ID number.

type
Node type.

Usage Notes

This procedure sets a node type string value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setType` method of the `Node` interface of the client-side Java API (described in Section 5.11.2).

To get the type value for a node, use the SDO_NET_MEM.NODE.GET_TYPE function.

Examples

The following example sets the type of the node whose node ID is 114 in the current network memory object, and then returns the type. (This example is an excerpt from Example 5–5 in Section 5.13.4.)

```
-- SET_TYPE
-- Set the type of node 114 to 'Research'.
SDO_NET_MEM.NODE.SET_TYPE(net_mem, 114, 'Research');
-- GET_TYPE
res_string := SDO_NET_MEM.NODE.GET_TYPE(net_mem, 114);
DBMS_OUTPUT.PUT_LINE('The type of node 114 is: ' || res_string);
. . .
The type of node 114 is: Research
```

SDO_NET_MEM.PATH.COMPUTE_GEOMETRY

Format

```
SDO_NET_MEM.PATH.COMPUTE_GEOMETRY(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER,
   tolerance  IN NUMBER);
```

Description

Sets the spatial geometry for a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

tolerance
Tolerance value associated with geometries in the network. (Tolerance is explained in Chapter 1 of *Oracle Spatial Developer's Guide*.) This value should be consistent with the tolerance values of the geometries in the link table and node table for the network.

Usage Notes

This procedure computes the SDO_GEOMETRY object for the specified path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `computeGeometry` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To get the computed geometry, use the SDO_NET_MEM.PATH.GET_GEOMETRY function.

Examples

The following example computes the spatial geometry of a path in the current network memory object, and then places the computed geometry in a variable (of type SDO_GEOMETRY) named `res_geom`.

```
-- COMPUTE_GEOMETRY
SDO_NET_MEM.PATH.COMPUTE_GEOMETRY(net_mem, path_id, 0.05);
-- GET_GEOMETRY
res_geom := SDO_NET_MEM.PATH.GET_GEOMETRY(net_mem, path_id);
```

SDO_NET_MEM.PATH.GET_COST

Format

```
SDO_NET_MEM.PATH.GET_COST(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the cost value of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns a numeric cost value for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getCost` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the cost of a path in the current network memory object.

```
res_numeric := SDO_NET_MEM.PATH.GET_COST(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The cost of path ' || path_id || ' is: ' || res_numeric);
. . .
The cost of path 21 is: 50
```

SDO_NET_MEM.PATH.GET_END_NODE_ID

Format

```
SDO_NET_MEM.PATH.GET_END_NODE_ID(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID value of the end node of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns a numeric end node ID value for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getEndNodeID` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the node ID of the end node of a path in the current network memory object.

```
res_numeric := SDO_NET_MEM.PATH.GET_END_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The end node ID of path ' || path_id || ' is: ' || res_
numeric);
. . .
The end node ID of path 21 is: 105
```

SDO_NET_MEM.PATH.GET_GEOMETRY

Format

```
SDO_NET_MEM.PATH.GET_GEOMETRY(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER
   ) RETURN SDO_GEOMETRY;
```

Description

Returns the spatial geometry of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns an SDO_GEOMETRY object for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

If you have not previously computed the geometry using the SDO_NET_MEM.PATH.COMPUTE_GEOMETRY procedure, the GET_GEOMETRY function returns a null result. To return the actual geometry, you must first call the SDO_NET_MEM.PATH.COMPUTE_GEOMETRY procedure.

This function is analogous to using the `getPath` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To set the geometry ID value for a path, use the SDO_NET_MEM.PATH.SET_GEOMETRY procedure.

Examples

The following example returns the spatial geometry of a path in the current network memory object.

```
res_geom := SDO_NET_MEM.PATH.GET_GEOMETRY(net_mem, path_id);
```

SDO_NET_MEM.PATH.GET_LINK_IDS

Format

```
SDO_NET_MEM.PATH.GET_LINK_IDS(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array of link ID values of the links in a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with link ID values for links in a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getLinks` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the links in a path in the current network memory object.

```
res_array := SDO_NET_MEM.PATH.GET_LINK_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Path ' || path_id || ' has the following links: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
. . .
Path 21 has the following links:
1102 1104 1105
```

SDO_NET_MEM.PATH.GET_NAME

Format

```
SDO_NET_MEM.PATH.GET_NAME(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the name of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns a path name string for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getName` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To set the name for a path, use the SDO_NET_MEM.PATH.SET_NAME procedure.

Examples

The following example sets the name of a path in the current network memory object to the string `My favorite path`, and then returns the name.

```
-- SET_NAME
-- Set the name of path to 'My favorite path'.
SDO_NET_MEM.PATH.SET_NAME(net_mem, path_id, 'My favorite path');
-- GET_NAME
res_string := SDO_NET_MEM.PATH.GET_NAME(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The name of path ' || path_id || ' is: ' || res_string);
. . .
The name of path 21 is: My favorite path
```

SDO_NET_MEM.PATH.GET_NO_OF_LINKS

Format

```
SDO_NET_MEM.PATH.GET_NO_OF_LINKS(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the number of links in a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the number of links in a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getNoOfLinks` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the number of links in a path in the current network memory object.

```
res_numeric := SDO_NET_MEM.PATH.GET_NO_OF_LINKS(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The number of links in path ' || path_id || ' is: ' || res_
numeric);
. . .
The number of links in path 21 is: 3
```

SDO_NET_MEM.PATH.GET_NODE_IDS

Format

```
SDO_NET_MEM.PATH.GET_NODE_IDS(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns an array of node ID values of the nodes in a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns an SDO_NUMBER_ARRAY object with node ID values for nodes in a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getNodes` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the node IDs of the nodes in a path in the current network memory object.

```
res_array := SDO_NET_MEM.PATH.GET_NODE_IDS(net_mem, path_id);
DBMS_OUTPUT.PUT('Path ' || path_id || ' has the following nodes: ');
FOR indx IN res_array.FIRST..res_array.LAST
LOOP
  DBMS_OUTPUT.PUT(res_array(indx) || ' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
. . .
Path 21 has the following nodes: 101 103 104 105
```

SDO_NET_MEM.PATH.GET_START_NODE_ID

Format

```
SDO_NET_MEM.PATH.GET_START_NODE_ID(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN NUMBER;
```

Description

Returns the node ID value of the start node of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns a numeric start node ID value for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getStartNode` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example returns the start node ID of a path in the current network memory object.

```
res_numeric := SDO_NET_MEM.PATH.GET_START_NODE_ID(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The start node ID of path ' || path_id || ' is: ' || res_
numeric);
. . .
The start node ID of path 21 is: 101
```

SDO_NET_MEM.PATH.GET_TYPE

Format

```
SDO_NET_MEM.PATH.GET_TYPE(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Returns the type of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns a type name string for a path in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `getType` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To set the type for a path, use the SDO_NET_MEM.PATH.SET_TYPE procedure.

Examples

The following example sets the type of a path in the current network memory object to the string `Logical connections`, and then returns the type.

```
-- SET_TYPE
-- Set the type of the path to 'Logical connections'.
SDO_NET_MEM.PATH.SET_TYPE(net_mem, path_id, 'Logical connections');
-- GET_TYPE
res_string := SDO_NET_MEM.PATH.GET_TYPE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The type of path ' || path_id || ' is: ' || res_string);
. . .
The type of path 21 is: Logical connections
```

SDO_NET_MEM.PATH.IS_ACTIVE

Format

```
SDO_NET_MEM.PATH.IS_ACTIVE(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is active.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is active, or `FALSE` if the path is not active. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isActive` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is active.

```
res_string := SDO_NET_MEM.PATH.IS_ACTIVE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' active?: ' || res_string);
. . .
Is path 21 active?: TRUE
```

SDO_NET_MEM.PATH.IS_CLOSED

Format

```
SDO_NET_MEM.PATH.IS_CLOSED(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is closed.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is closed, or `FALSE` if the path is not closed. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isClosed` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is closed.

```
res_string := SDO_NET_MEM.PATH.IS_CLOSED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' closed?: ' || res_string);
. . .
Is path 21 closed?: FALSE
```

SDO_NET_MEM.PATH.IS_CONNECTED

Format

```
SDO_NET_MEM.PATH.IS_CONNECTED(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is connected.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is connected, or `FALSE` if the path is not connected. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isConnected` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is connected.

```
res_string := SDO_NET_MEM.PATH.IS_CONNECTED(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' connected?: ' || res_string);
. . .
Is path 21 connected?: FALSE
```

SDO_NET_MEM.PATH.IS_LOGICAL

Format

```
SDO_NET_MEM.PATH.IS_LOGICAL(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is in a logical network.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is in a logical network, or `FALSE` if the path is not in a logical network. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isLogical` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is a logical path.

```
res_string := SDO_NET_MEM.PATH.IS_LOGICAL(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a logical path?: ' || res_string);
. . .
Is path 21 a logical path?: TRUE
```

SDO_NET_MEM.PATH.IS_SIMPLE

Format

```
SDO_NET_MEM.PATH.IS_SIMPLE(
   net_mem  IN VARCHAR2,
   path_id    IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is simple.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is simple, or `FALSE` if the path is not simple (that is, is a complex path). In a simple path, the links form an ordered list that can be traversed from the start node to the end node with each link visited once. In a complex path, there are multiple options for going from the start node to the end node.

For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This function is analogous to using the `isSimple` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is a simple path.

```
res_string := SDO_NET_MEM.PATH.IS_SIMPLE(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' a simple path?: ' || res_string);
. . .
Is path 21 a simple path?: TRUE
```

SDO_NET_MEM.PATH.IS_TEMPORARY

Format

```
SDO_NET_MEM.PATH.IS_TEMPORARY(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER
   ) RETURN VARCHAR2;
```

Description

Checks if a path is temporary.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

Usage Notes

This function returns the string `TRUE` if the path in the specified network memory object is temporary, or `FALSE` if the path is not temporary. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

Temporary links, nodes, and paths are not saved in the database when you call the SDO_NET_MEM.NETWORK_MANAGER.WRITE_NETWORK procedure.

This function is analogous to using the `isTemporary` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example checks if a path in the current network memory object is temporary.

```
res_string := SDO_NET_MEM.PATH.IS_TEMPORARY(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('Is path ' || path_id || ' temporary?: ' || res_string);
. . .
Is path 21 temporary?: FALSE
```

SDO_NET_MEM.PATH.SET_GEOMETRY

Format

```
SDO_NET_MEM.PATH.SET_GEOMETRY(
   net_mem  IN VARCHAR2,
   path_id     IN NUMBER,
   geom        IN SDO_GEOMETRY);
```

Description

Sets the spatial geometry for a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

geom
Spatial geometry object.

Usage Notes

This procedure creates an SDO_GEOMETRY object for the node in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setGeometry` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To get the geometry for a path, use the SDO_NET_MEM.PATH.GET_GEOMETRY function.

Examples

The following example sets the spatial geometry of a path in the current network memory object.

```
SDO_NET_MEM.PATH.SET_GEOMETRY(net_mem, path_id,
  SDO_GEOMETRY(
    2002, NULL, NULL,
    SDO_ELEM_INFO_ARRAY(1,2,1),
    SDO_ORDINATE_ARRAY(2,2, 2,4, 8,4, 12,4, 12,10, 8,10, 5,14)));
```

SDO_NET_MEM.PATH.SET_NAME

Format

```
SDO_NET_MEM.PATH.SET_NAME(
   net_mem     IN VARCHAR2,
   path_id     IN NUMBER,
   path_name   IN VARCHAR2);
```

Description

Sets the name of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

path_name
Path name.

Usage Notes

This procedure sets a path name string value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setName` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To get the name of a path, use the SDO_NET_MEM.PATH.GET_NAME function.

Examples

The following example sets the name of a path in the current network memory object to the string `My favorite path`, and then returns the name.

```
-- SET_NAME
-- Set the name of path to 'My favorite path'.
SDO_NET_MEM.PATH.SET_NAME(net_mem, path_id, 'My favorite path');
-- GET_NAME
res_string := SDO_NET_MEM.PATH.GET_NAME(net_mem, path_id);
DBMS_OUTPUT.PUT_LINE('The name of path ' || path_id || ' is: ' || res_string);
. . .
The name of path 21 is: My favorite path
```

SDO_NET_MEM.PATH.SET_PATH_ID

Format

```
SDO_NET_MEM.PATH.SET_PATH_ID(
   net_mem      IN VARCHAR2,
   path_id      IN NUMBER,
   new_path_id  IN NUMBER);
```

Description

Sets the path ID value of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

new_path_id
New path ID number.

Usage Notes

This procedure sets a numeric path ID value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setPathID` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

Examples

The following example sets the path ID of a path in the current network memory object to 6789.

```
SDO_NET_MEM.PATH.SET_PATH_ID(net_mem, path_id, 6789);
```

SDO_NET_MEM.PATH.SET_TYPE

Format

```
SDO_NET_MEM.PATH.SET_TYPE(
   net_mem  IN VARCHAR2,
   path_id  IN NUMBER,
   type     IN VARCHAR2);
```

Description

Sets the type of a path.

Parameters

net_mem
Name of the network whose current network memory object (created using the SDO_NET_MEM.NETWORK_MANAGER.READ_NETWORK procedure) is to be used.

path_id
Path ID number.

type
Path type.

Usage Notes

This procedure sets a path type string value in the specified network memory object. For information about using a network memory object for editing and network analysis operations, see Section 5.8.

This procedure is analogous to using the `setType` method of the `Path` interface of the client-side Java API (described in Section 5.11.2).

To get the type value for a path, use the SDO_NET_MEM.PATH.GET_TYPE function.

Examples

The following example sets the path type of a path in the current network memory object to the string `Scenic`.

```
SDO_NET_MEM.PATH.SET_TYPE(net_mem, path_id, 'Scenic');
```

Index

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W