Oracle® Spatial

Spatial Topology and Network Data Model Developer's Guide

23ai
F47001-06
April 2025

ORACLE

Oracle Spatial Spatial Topology and Network Data Model Developer's Guide, 23ai

F47001-06

Primary Author: Lavanya Jayapalan

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

Contents

6 SDO_NET Package Subprograms

7 SDO_NFE Package Subprograms

Index

List of Figures

List of Tables

Preface

Oracle Spatial Topology and Network Data Model Developer's Guide provides usage and reference information about the Topology Data Model and Network Data Model features of Oracle Spatial.

Audience

This guide is intended for those who need 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

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

Access to Oracle Support

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

Related Documents

For more information, see *Oracle Spatial Developer's Guide* and REST API for Oracle Spatial.

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.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I
Topology Data Model

This part covers the topology data model feature of Oracle Spatial.

This document has two main parts:

- Part I provides conceptual, usage, and reference information about the Topology Data Model feature of Oracle Spatial.
- Network Data Model provides conceptual, usage, and reference information about the Network Data Model feature of Oracle Spatial.

Part I contains the following chapters:

- Topology Data Model Overview
  The Topology Data Model feature of Oracle Spatial lets you work with data about nodes, edges, and faces in a topology.
- Editing Topologies
  Node and edge data in a topology can be edited. The operations include adding, moving, and removing nodes and edges, and updating the coordinates of an edge.
- 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 feature. This package mainly contains subprograms for creating and managing topologies.
- 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 feature.

1

Topology Data Model Overview

The Topology Data Model feature of Oracle Spatial lets you work with data about nodes, edges, and faces in a topology.

> **Note:**
>
> Topology Data Model is only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

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 feature. 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 feature, 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 concepts, data types, and operations as documented in *Oracle Spatial Developer's Guide*.

- Main Steps in Using Topology Data
  This topic summarizes the main steps for working with topology data.
- 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*.
- 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.
- 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.

- Topology Data Model Tables
  To use the Oracle 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.
- Topology Data Types
  The main data type associated with the Topology Data Model is SDO_TOPO_GEOMETRY, which describes a topology geometry.
- 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 Oracle Spatial.
- Topology Application Programming Interface
  The Topology Data Model application programming interface (API) consists of the following.
- 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.
- Cross-Schema Topology Usage and Editing
  This topic contains requirements and guidelines for using and editing topologies when multiple database users (schemas) are involved.
- Function-Based Indexes Not Supported
  You cannot create a function-based index on a column of type SDO_TOPO_GEOMETRY.
- Topology Examples (PL/SQL)
  This topic presents simplified PL/SQL examples that perform Topology Data Model operations.
- README File for Spatial and Related Features

1.1 Main Steps in Using Topology Data

This topic summarizes the main steps for working with topology data.

It refers to important concepts, structures, and operations that are described in detail in other topics.

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 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 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 SDO_TOPO_MAP Package Subprograms. The Java API is described in Topology Data Model Java Interface.

- Using a Topology Built from Topology Data
- Using a Topology Built from Spatial Geometries

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 Edge Information Table, Node Information Table, Face Information Table, and History Information Table, 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 Relationship Information Table.)
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 SDO_TOPO_GEOMETRY Constructors.)
7. Query the topology data (for example, using one of topology operators described in Topology Operators).
8. Optionally, edit topology data using the PL/SQL or Java application programming interfaces (APIs).

Topology Built from Topology Data 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 Edge Information Table, Node Information Table, Face Information Table, and History Information Table, respectively.)

2. Create the universe face (F0, defined in Topology Data Model Concepts).
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 Relationship Information Table.)
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. Create a TopoMap object and load the whole topology into cache.
7. Load the feature tables, inserting data from the spatial tables and using the SDO_TOPO_MAP.CREATE_FEATURE function.
8. Query the topology data (using one of topology operators described in Topology Operators).
9. Optionally, edit topology data using the PL/SQL or Java application programming interfaces (APIs).

Topology Built from Spatial Geometries 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).

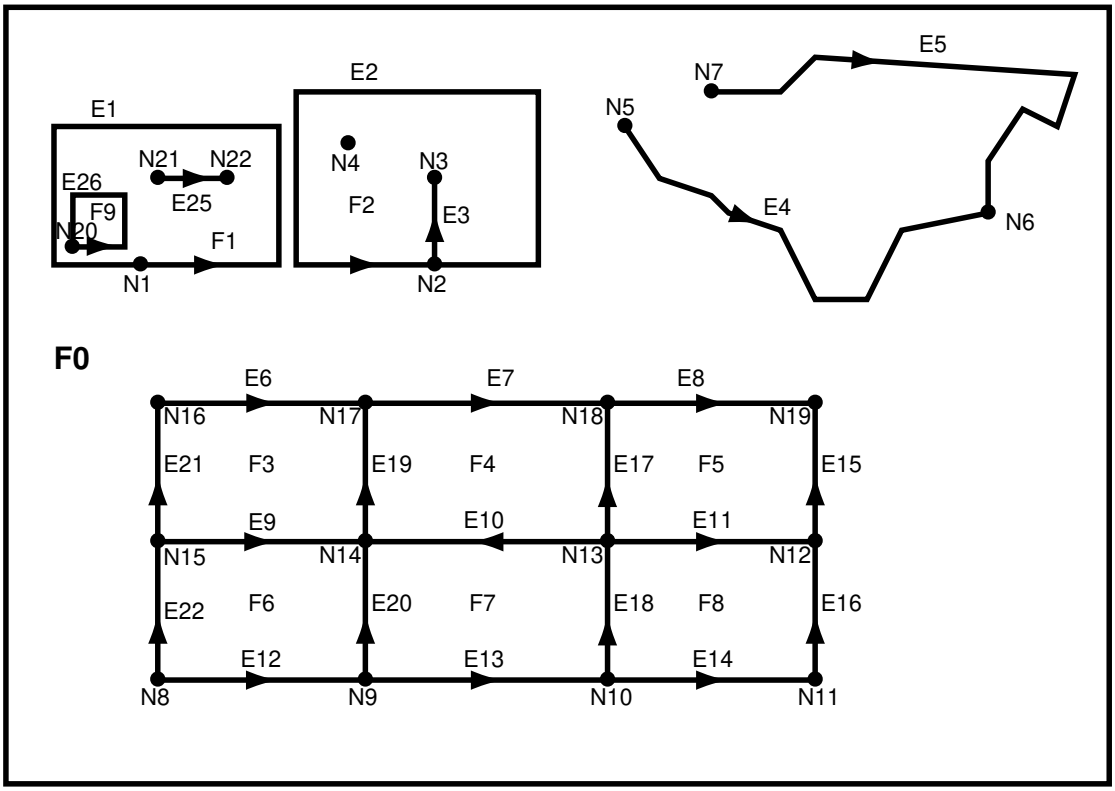
An **edge** is bounded by two nodes: the start (origin) node and the end (terminal) node. An edge has an associated geometric object, usually a coordinate string that describes the spatial representation of the edge. An edge may have several vertices making up a line string. (Circular arcs are not supported 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.

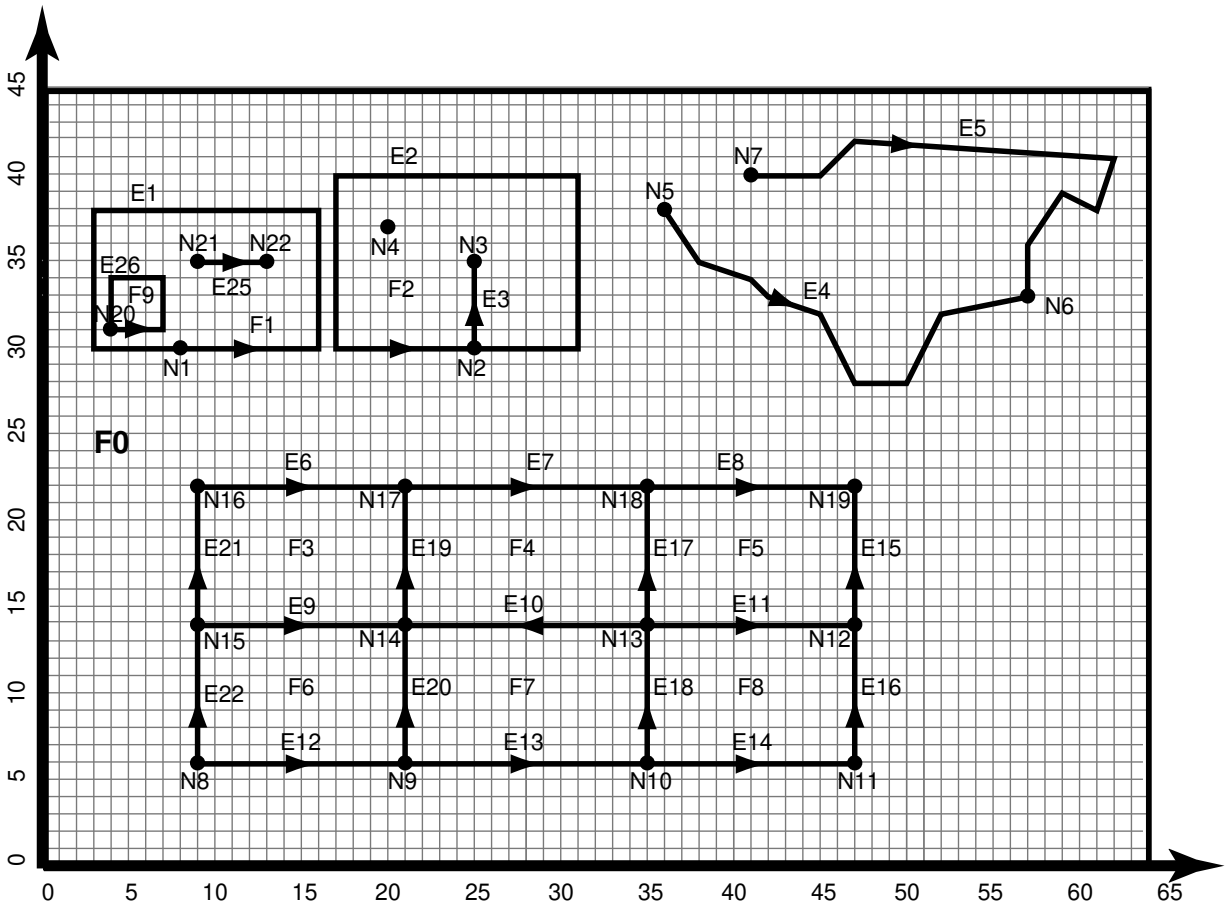
  For edge E9, the start node is N15 and the end node is N14, the next left edge is E19 and the previous left edge is -E21, the next right edge is -E22 and the previous right edge is E20, the left face is F3 and the right face is F6.

  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 Edge Information Table.

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 SDO_TOPO Package Subprograms and SDO_TOPO_MAP Package Subprograms.

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



- Tolerance in the Topology Data Model

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 Collection Layers 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 SDO_TOPO_GEOMETRY Type), 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 xxx_SDO_TOPO_METADATA Views. The USER_SDO_TOPO_INFO and ALL_SDO_TOPO_INFO views (described in xxx_SDO_TOPO_INFO Views) contain a subset of this topology metadata.

- Features
- Collection Layers

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 Topology Data Model Concepts.

Figure 1-3 Features in a Topology

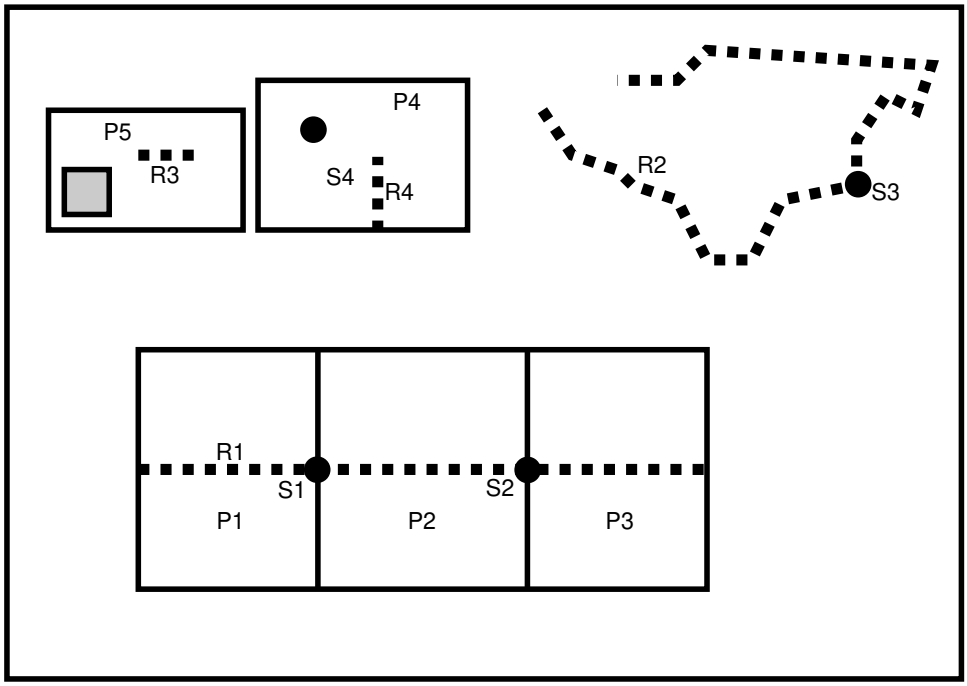


Figure 1-3 shows the following kinds of features in the topology:

- 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 Topology Data Model Concepts.)

Example 1-12 in Topology Built from Topology Data 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 Topology Examples (PL/SQL), 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

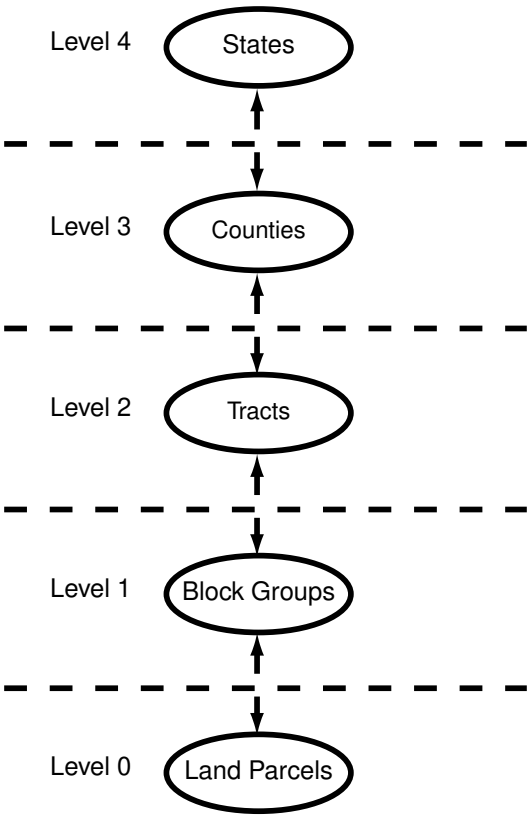
If the topology geometry layers in a topology have this hierarchical relationship, it is far more efficient if you model the layers as hierarchical than if you specify all topology geometry layers at a single level (that is, with no hierarchy). For example, it is more efficient to construct SDO_TOPO_GEOMETRY 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 Network Hierarchy; 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



Example 1-2 Modeling a 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.

```
-- 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 Topology Geometries and Layers, and SDO_TOPO_GEOMETRY constructors are described in SDO_TOPO_GEOMETRY Constructors.

Note that 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 Relationship Information Table.

1.5 Topology Data Model Tables

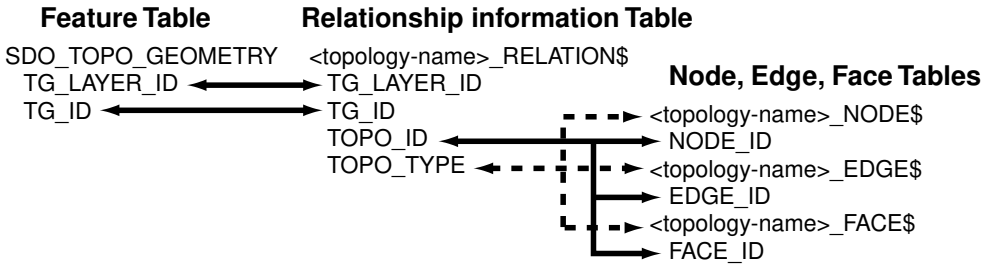
To use the Oracle Spatial topology capabilities, you must first insert data into special edge, node, and face tables, which are created by Spatial when you create a topology.

The edge, node, and face tables are described in Edge Information Table, Node Information Table, and Face Information Table, 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 Relationship Information Table.

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 Topology Metadata Views.

- 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.
- Edge Information Table
- Node Information Table
- Face Information Table
- Relationship Information Table
- History Information Table

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

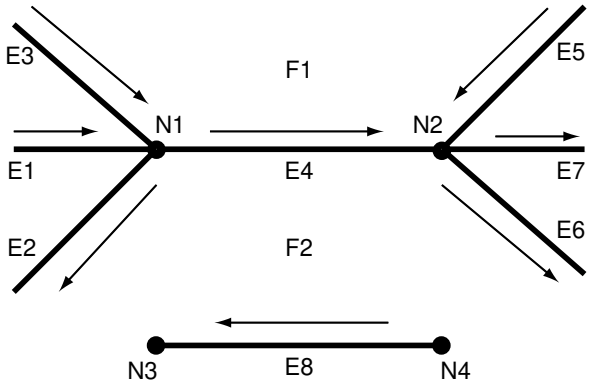


Table 1-2 shows the ID column values in the edge information table for edges E4 and E8 in Figure 1-6. (For clarity, Table 1-2 shows ID column values with alphabetical characters, such as E4 and N1; however, the ID columns actually contain numeric values only, specifically the numeric ID value associated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

Table 1-4 (Cont.) Columns in the <topology-name>_FACE$ Table

Column Name	Data Type	Description
ISLAND_NODE_ID_LIST	SDO_LIST_TYPE	Island nodes (if any) in this face. (The SDO_LIST_TYPE type is described in SDO_LIST_TYPE Type.)
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 Editing Topologies .) 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
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

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

Column Name	Data Type	Description
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.

- SDO_TOPO_GEOMETRY Type
- SDO_TOPO_GEOMETRY Constructors
- GET_GEOMETRY Member Function
- GET_TGL_OBJECTS Member Function
- GET_TOPO_ELEMENTS Member Function
- SDO_LIST_TYPE Type
- SDO_EDGE_ARRAY and SDO_NUMBER_ARRAY Types

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 Topology Examples (PL/SQL).

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 Topology Geometry Layer Hierarchy.)

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.

- Constructors for Insert Operations: Specifying Topological Elements
- Constructors for Insert Operations: Specifying Lower-Level Features
- Constructors for Update Operations: Specifying Topological Elements
- Constructors for Update Operations: Specifying Lower-Level Features

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 Relationship Information Table) 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 Topology Examples (PL/SQL).

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 Constructors for Insert Operations: Specifying Topological Elements.

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 Topology Examples (PL/SQL).

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 Constructors for Insert Operations: Specifying Lower-Level Features.

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 Constructors for Insert Operations: Specifying Lower-Level Features.

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 Constructors for Insert Operations: Specifying Topological Elements.

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 Oracle 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.

- xxx_SDO_TOPO_INFO Views
- xxx_SDO_TOPO_METADATA Views

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 Tolerance in the Topology Data Model.)
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 Topology Geometry Layer Hierarchy.)
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 Topology Geometry Layer Hierarchy.)

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

Column Name	Data Type	Purpose
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
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 Tolerance in the Topology Data Model.)
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 1-9 (Cont.) Columns in the xxx_SDO_TOPO_METADATA Views

Column Name	Data Type	Purpose
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 Topology Geometry Layer Hierarchy.)
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 Topology Geometry Layer Hierarchy.)
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

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

Column Name	Data Type	Purpose
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 SDO_TOPO Package Subprograms) and the SDO_TOPO_MAP package (described in SDO_TOPO_MAP Package Subprograms)
- PL/SQL topology operators (described in Topology Operators)
- Java API (described in Topology Data Model Java Interface)

- Topology Operators
- Topology Data Model Java Interface

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 topic 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 Topology Built from Topology Data.)

```
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 Topology Built from Spatial Geometries.)

```
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 Topology Built from Spatial Geometries.)

```
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 Features 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:

- Usage Notes for the SDO_TOPO.RELATE function

1.8.2 Topology Data Model Java Interface

Note:

Effective with Oracle Database Release 23ai, the Oracle Spatial Topology Data Model APIs are compiled with JDK 11 as the OJVM in the database supports JDK11. However, the APIs will continue to be supported on JDK8 for backwards compatibility. When using the API, ensure that all the related JAR files are consistent with the JDK version (JDK 8 or JDK 11) that is being used. See RDBMS and JDK Version Compatibility for Oracle JDBC Drivers for more information on the JDBC drivers that are supported for the different JDK versions.

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

The Spatial Java class libraries are in .jar files under the `<ORACLE_HOME>/md/jlib/` directory.

> **See Also:**
>
> *Oracle Spatial Java API Reference* for detailed reference information about the Topology Data Model classes, as well as some usage information about the Java API

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.

> **Note:**
>
> The steps are required regardless of whether the topology data is transported using transportable tablespaces. (For detailed information about transportable tablespaces and transporting tablespaces to other databases, see *Oracle Database Administrator's Guide*.)

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 SDO_TOPO Package Subprograms), 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 xxx_SDO_TOPO_INFO Views.)

   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 values in the OWNER and TABLE_SCHEMA columns in all rows of the <topology-name>_EXP$ table to reflect the table owner and schema names 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 topic contains requirements and guidelines for using and editing topologies when multiple database users (schemas) are involved.

- Cross-Schema Topology Usage
- Cross-Schema Topology Editing

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 or READ 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`.

> **See Also:**
>
> - Editing Topologies for information about editing topologies.

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 Development Guide* and *Oracle Database Administrator's Guide*.)

1.12 Topology Examples (PL/SQL)

This topic 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 SDO_TOPO Package Subprograms and SDO_TOPO_MAP Package Subprograms, and the SDO_ANYINTERACT topology operator (see Topology Operators).

Both examples are based on the "city data" topology shown in Figure 1-1 in Topology Data Model Concepts, and the features shown in Figure 1-3 in Features. 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).

- Topology Built from Topology Data

- Topology Built from Spatial Geometries

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. Initialize topology metadata.
-- 6. Create a TopoMap object and load the whole topology into
--     cache for updating.
-- 7. Load feature tables, inserting data from the spatial tables and
--     using SDO_TOPO_MAP.CREATE_FEATURE.
-- 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. Initialize topology metadata.
EXECUTE SDO_TOPO.INITIALIZE_METADATA('CITY_DATA');

-- 6. 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');

-- 7. 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');

-- 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 Model Developer's Guide* (this manual).
This file is located at:

$ORACLE_HOME/md/doc/README.txt

2

Editing Topologies

Node and edge data in a topology can be edited. 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.

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 SDO_TOPO_MAP Package Subprograms. However, you can also perform topology editing operations using the client-side Java API, which is introduced in Topology Data Model Java Interface 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.

- Approaches for Editing Topology Data
- Performing Operations on Nodes
  This topic 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.
- Performing Operations on Edges
  This topic describes the effects of adding, moving, removing, and updating edges, and explains how to perform these operations using the PL/SQL API.

> **See Also:**
>
> - Cross-Schema Topology Editing

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. (This object is described in TopoMap Objects.)

- 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.

- TopoMap Objects
- Specifying the Editing Approach with the Topology Parameter

- Using GET_xxx Topology Functions
- Process for Editing Using Cache Explicitly (PL/SQL API)
- Process for Editing Using the Java API
- Error Handling for Topology Editing

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.

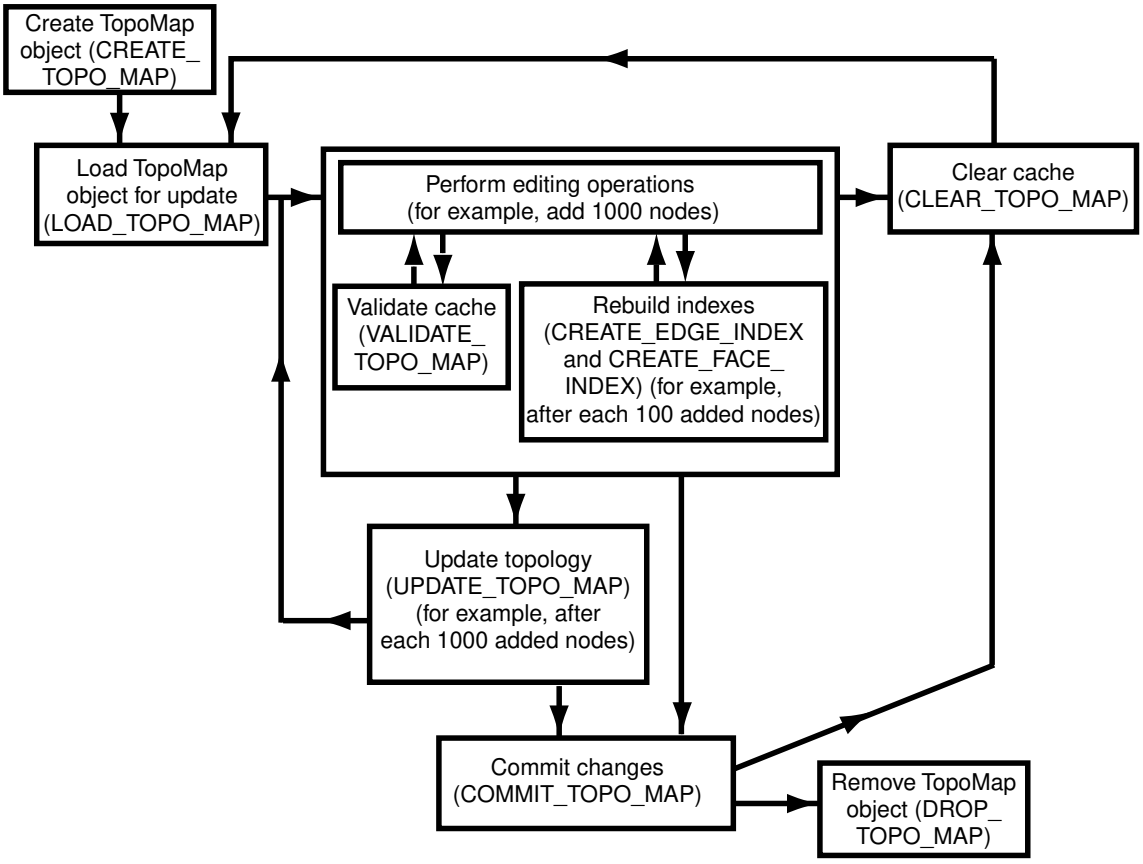
Some SDO_TOPO_MAP package functions that get information about topology editing operations have no parameters. Examples of such functions are SDO_TOPO_MAP.GET_FACE_ADDITIONS and SDO_TOPO_MAP.GET_NODE_CHANGES. These functions use the current updatable TopoMap object. If no updatable TopoMap object exists, an exception is raised. For example, the following statement returns an SDO_NUMBER_ARRAY object (described in SDO_EDGE_ARRAY and SDO_NUMBER_ARRAY Types) 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.

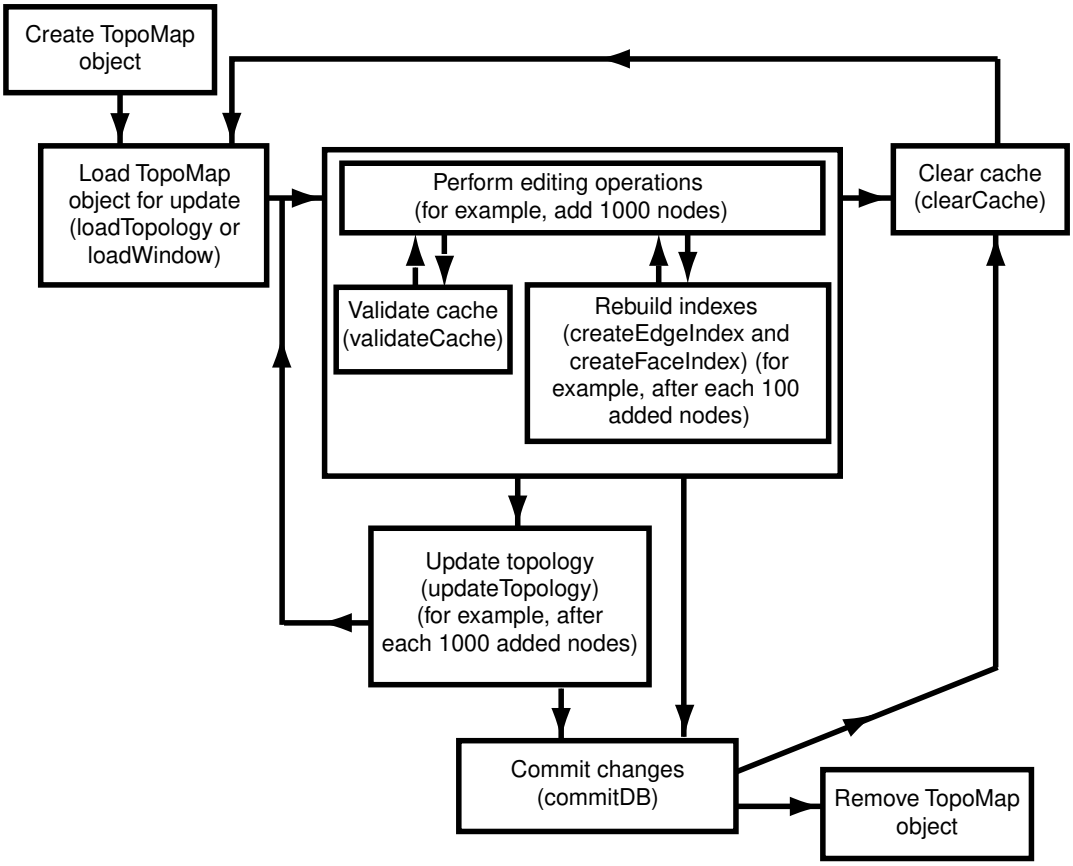
   If the application terminates abnormally, all uncommitted changes to the database will be discarded.

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 Topology Data Model Java Interface 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 topic 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.

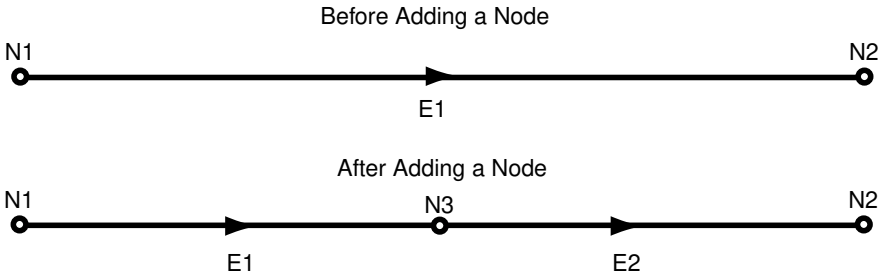
2.2.1 Adding a Node

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

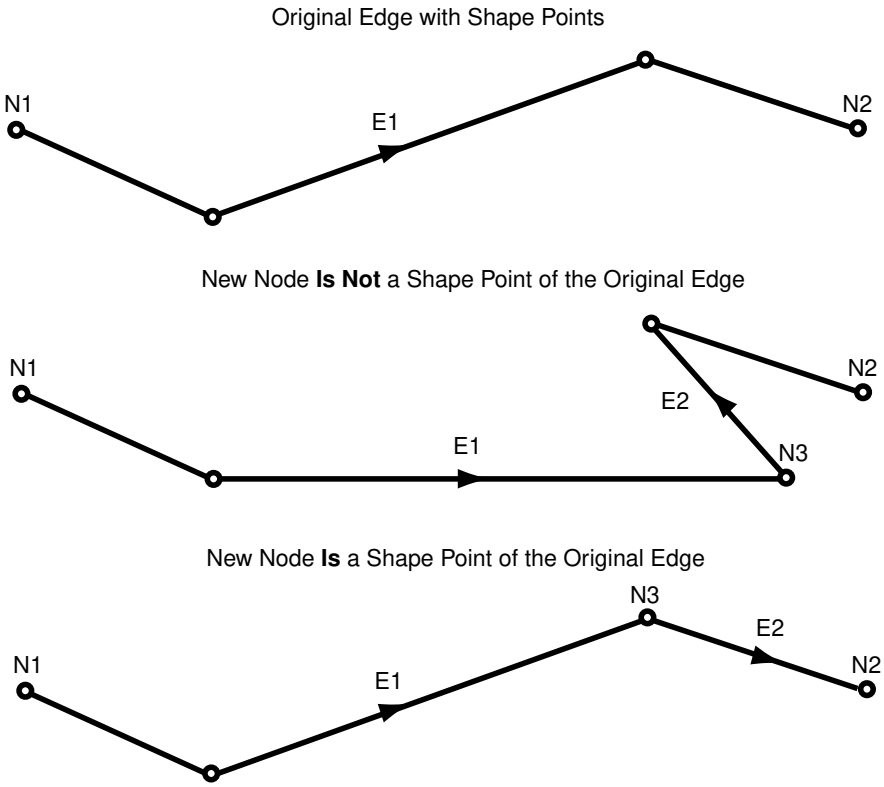


As a result of the operation shown in Figure 2-3:

- 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 History Information Table) 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

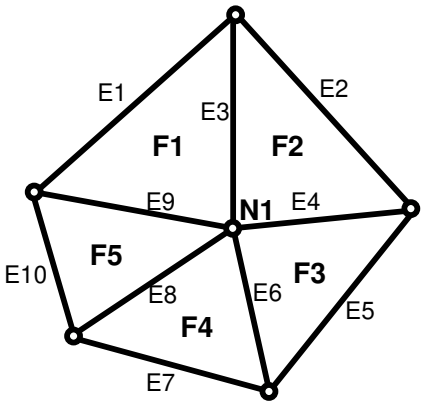
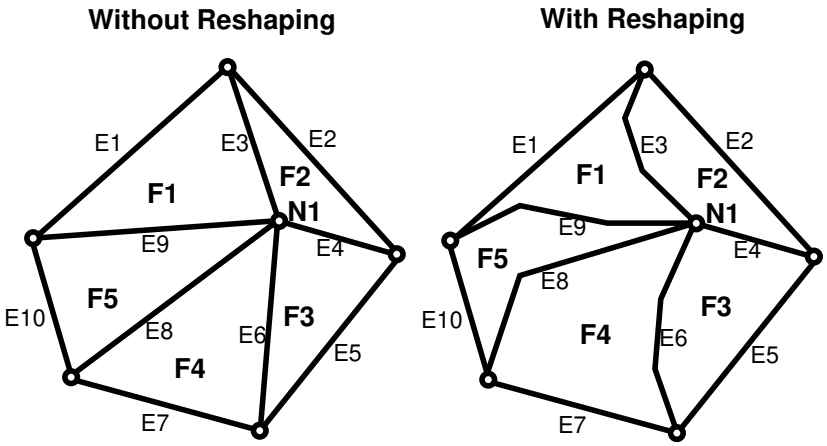


Figure 2-6 shows two cases of the original topology after node N1 is moved. In one case, no reshaping occurs; that is, all edges affected by the movement are specified as straight lines. In the other case, reshaping occurs; that is, one or more affected edges are specified as line segments with multiple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

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



In both cases shown in Figure 2-6:

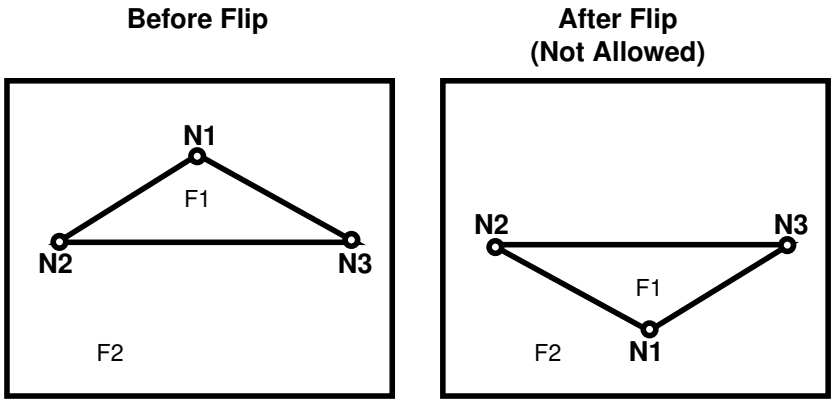
- The topology does not change. That is, the number of nodes, edges, and faces does not change, and the relationships (such as adjacency and connectivity) among the nodes, edges, and faces do not change.
- All features defined on the nodes, edges, and faces retain their definitions.

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

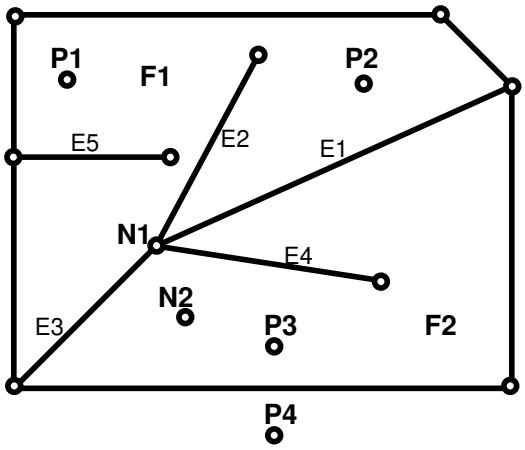


- Additional Examples of Allowed and Disallowed Node Moves

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

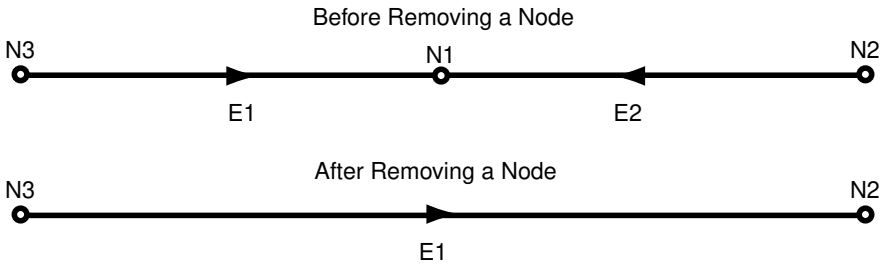
You can remove individual nodes (isolated or non-isolated), as explained in this section, and you can remove all obsolete nodes in a topology, as explained in Removing Obsolete Nodes.

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 History Information Table) 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.

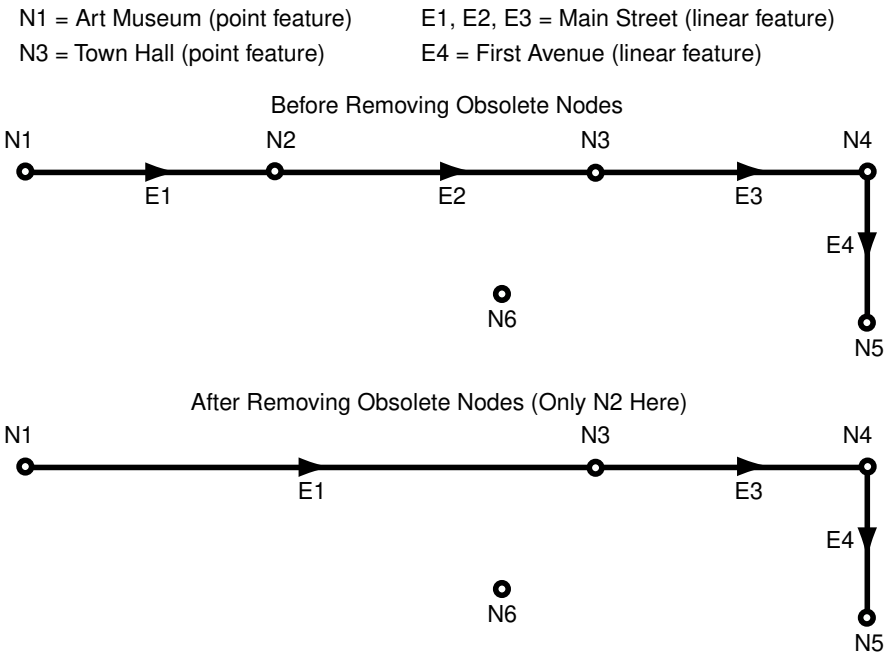
2.2.4 Removing Obsolete Nodes

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

- Adding an Edge
- Moving an Edge
- Removing an Edge
- Updating an Edge

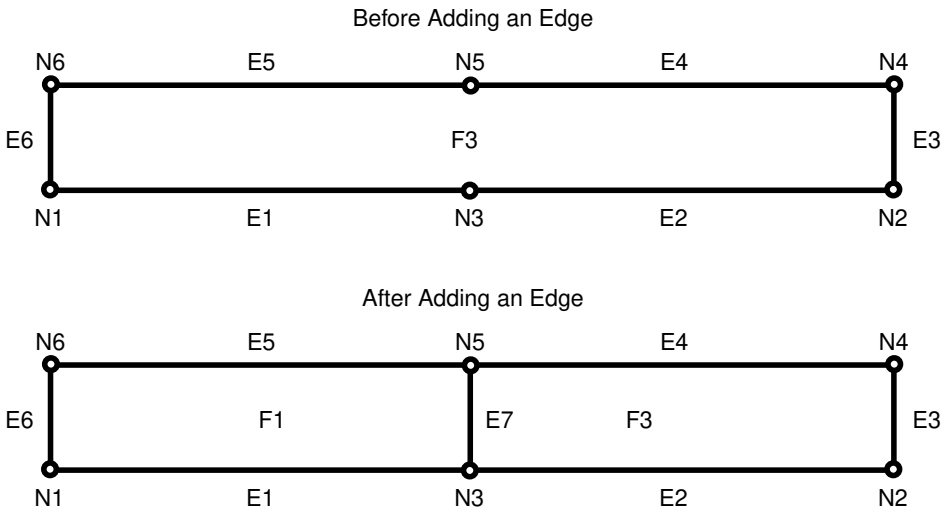
2.3.1 Adding an Edge

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.

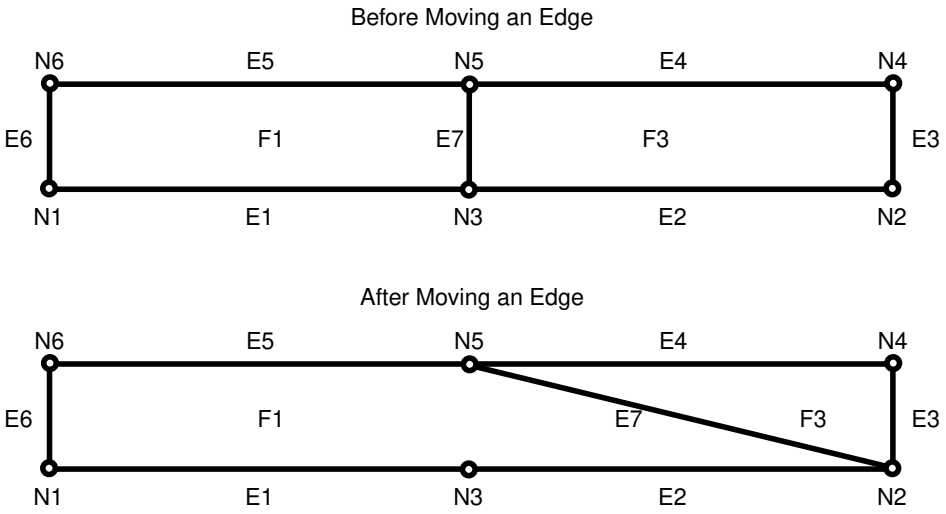
2.3.2 Moving an Edge

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.

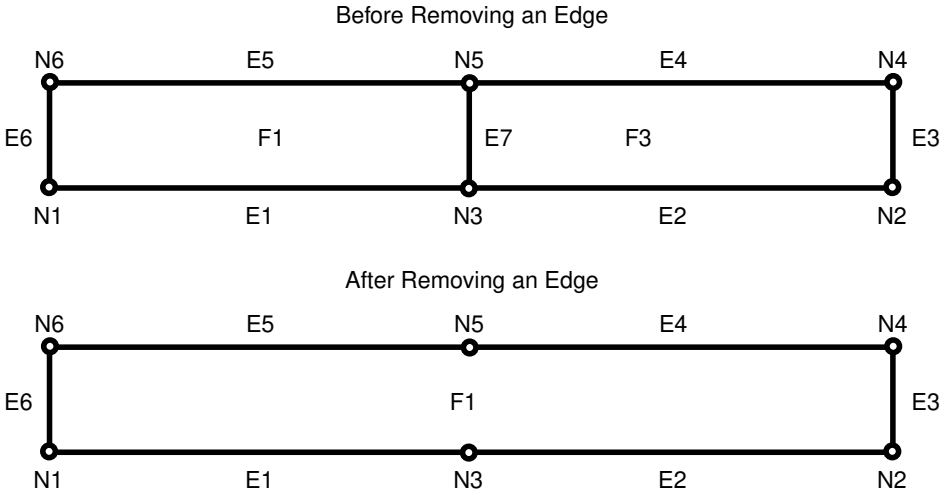
2.3.3 Removing an Edge

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 feature. 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 Topology Data Model Overview.

> **Note:**
>
> SDO_TOPO subprograms are only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

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

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

- SDO_TOPO.ADD_TOPO_GEOMETRY_LAYER
- SDO_TOPO.CREATE_TOPOLOGY
- SDO_TOPO.DELETE_TOPO_GEOMETRY_LAYER
- SDO_TOPO.DROP_TOPOLOGY
- SDO_TOPO.GET_FACE_BOUNDARY
- SDO_TOPO.GET_TOPO_OBJECTS
- SDO_TOPO.INITIALIZE_AFTER_IMPORT
- SDO_TOPO.INITIALIZE_METADATA
- SDO_TOPO.PREPARE_FOR_EXPORT
- SDO_TOPO.RELATE

3.1 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 Relationship Information Table). 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 Topology Geometry Layer Hierarchy.) 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 Relationship Information Table.

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 Topology Built from Topology Data.)

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

3.2 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 Tolerance in the Topology Data Model.)

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 Node Information Table). 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 Edge Information Table). 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 Face Information Table). 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 History Information Table. 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 Topology Data Model Tables, 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 Topology Built from Topology Data.)

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

3.3 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 Relationship Information Table). 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 Topology Built from Topology Data.)

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

3.4 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 Topology Data Model Tables).

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 Topology Built from Topology Data.)

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

3.5 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 Topology Examples (PL/SQL).)

```
-- 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)
```

3.6 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 Constructors for Insert Operations: Specifying Topological Elements.

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 Topology Geometry Layer Hierarchy.)

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 Topology Built from Topology Data.)

```
-- 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))
```

3.7 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 Exporting and Importing Topology Data.

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 Topology Built from Topology Data.)

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

3.8 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 Topology Data Model Tables.)

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 Topology Built from Topology Data.)

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

3.9 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 xxx_SDO_TOPO_INFO Views.

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 Exporting and Importing Topology Data.

Examples

The following example prepares the topology named CITY_DATA for export to a target database. (The example refers to definitions and data from Topology Built from Topology Data.)

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

3.10 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 Constructors for Insert Operations: Specifying Topological Elements).

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

Usage Notes

The topology operators (described in Topology Operators) 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 Topology Examples (PL/SQL). 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 Topology Examples (PL/SQL).) 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 feature.

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 Topology Data Model Overview, as well as the information about editing topologies in Editing Topologies .

> **Note:**
>
> SDO_TOPO_MAP subprograms are only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

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

- SDO_TOPO_MAP.GET_EDGE_COORDS
- SDO_TOPO_MAP.GET_EDGE_DELETIONS
- SDO_TOPO_MAP.GET_EDGE_NODES
- SDO_TOPO_MAP.GET_FACE_ADDITIONS
- SDO_TOPO_MAP.GET_FACE_CHANGES
- SDO_TOPO_MAP.GET_FACE_BOUNDARY
- SDO_TOPO_MAP.GET_FACE_DELETIONS
- SDO_TOPO_MAP.GET_NEAREST_EDGE
- SDO_TOPO_MAP.GET_NEAREST_EDGE_IN_CACHE
- SDO_TOPO_MAP.GET_NEAREST_NODE
- SDO_TOPO_MAP.GET_NEAREST_NODE_IN_CACHE
- SDO_TOPO_MAP.GET_NODE_ADDITIONS
- SDO_TOPO_MAP.GET_NODE_CHANGES
- SDO_TOPO_MAP.GET_NODE_COORD
- SDO_TOPO_MAP.GET_NODE_DELETIONS
- SDO_TOPO_MAP.GET_NODE_FACE_STAR
- SDO_TOPO_MAP.GET_NODE_STAR
- SDO_TOPO_MAP.GET_TOPO_NAME
- SDO_TOPO_MAP.GET_TOPO_TRANSACTION_ID
- SDO_TOPO_MAP.LIST_TOPO_MAPS
- SDO_TOPO_MAP.LOAD_TOPO_MAP
- SDO_TOPO_MAP.MOVE_EDGE
- SDO_TOPO_MAP.MOVE_ISOLATED_NODE
- SDO_TOPO_MAP.MOVE_NODE
- SDO_TOPO_MAP.REMOVE_EDGE
- SDO_TOPO_MAP.REMOVE_NODE
- SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES
- SDO_TOPO_MAP.ROLLBACK_TOPO_MAP
- SDO_TOPO_MAP.SEARCH_EDGE_RTREE_TOPO_MAP
- SDO_TOPO_MAP.SEARCH_FACE_RTREE_TOPO_MAP
- SDO_TOPO_MAP.SET_MAX_MEMORY_SIZE
- SDO_TOPO_MAP.UPDATE_TOPO_MAP
- SDO_TOPO_MAP.VALIDATE_TOPO_MAP
- SDO_TOPO_MAP.VALIDATE_TOPOLOGY

4.1 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addEdge` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.2 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addIsolatedNode` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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.
```

4.3 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addLinearGeometry` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.4 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addLoop` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.5 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addNode` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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.
```

4.6 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 Specifying the Editing Approach with the Topology Parameter). 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 Editing Topologies .

This function is equivalent to using the `addPointGeometry` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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
```

4.7 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 Specifying the Editing Approach with the Topology Parameter). 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 Topology Data Model Java Interface).

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
```

4.8 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 Specifying the Editing Approach with the Topology Parameter). 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 Edge Information Table). (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 Editing Topologies .

This procedure is equivalent to using the `changeEdgeCoords` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)));
```

4.9 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 TopoMap Objects.)

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 Approaches for Editing Topology Data.

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

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

4.10 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 Approaches for Editing Topology Data.

This procedure is equivalent to using the `commitDB` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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;
```

4.11 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

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

4.12 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

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

4.13 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;
or
SDO_TOPO_MAP.CREATE_FEATURE(
  topology    IN VARCHAR2,
  table_name  IN VARCHAR2,
  column_name IN VARCHAR2,
  geom        IN SDO_GEOMETRY,
  snapfeature IN NUMBER
) 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 two formats (with the `geom` parameter and without the `dml_condition` parameter) are 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 third 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.

snapfeature
If set to 1, the specified new feature is snapped to existing edges and nodes in the topology.

dml_condition
For topologies with a topology geometry layer hierarchy (described in Topology Geometry Layer Hierarchy): 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 Main Steps in Using Topology Data, which contains the following subsections:

- Using a Topology Built from Topology Data
- Using a Topology Built from Spatial Geometries (that is, the approach using the CREATE_FEATURE function)

When you use the first or second 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. (That is, Spatial uses an implicitly created TopoMap object to create a new TopoMap object for each call to this function.)

When you use the third 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 Topology Built from Spatial Geometries.)

```
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;
```

4.14 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 TopoMap Objects.)

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 Approaches for Editing Topology Data.

Using this procedure is equivalent to calling the constructor of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
CALL SDO_TOPO_MAP.CREATE_TOPO_MAP('CITY_DATA', 'CITY_DATA_TOPOMAP');
```

4.15 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 TopoMap Objects.)

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 Approaches for Editing Topology Data.

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 Topology Data Model Java Interface.)

Examples

The following example drops the TopoMap object named `CITY_DATA_TOPOMAP`. (The example refers to definitions and data from Topology Built from Topology Data.)

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

4.16 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.17 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 Topology Data Model Java Interface).

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)
```

4.18 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 Topology Data Model Java Interface).

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)
```

4.19 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

edge_id
Edge ID value of the edge.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Using GET_xxx Topology Functions.

This function is equivalent to using the `getEdgeCoords` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.20 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 Topology Data Model Java Interface).

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()
```

4.21 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

edge_id
Edge ID value of the edge.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Using GET_xxx Topology Functions.

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 Topology Data Model Java Interface). 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 Topology Built from Topology Data.)

```
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)
```

4.22 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 Topology Data Model Java Interface).

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)
```

4.23 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 Topology Data Model Java Interface).

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)
```

4.24 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

This function is equivalent to using the `getFaceBoundary` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.25 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 Topology Data Model Java Interface).

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()
```

4.26 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.27 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.28 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.29 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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
```

4.30 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 Topology Data Model Java Interface).

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)
```

4.31 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 Topology Data Model Java Interface).

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)
```

4.32 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

node_id
Node ID value of the node.

Usage Notes

The `topology` or `topo_map` parameter should specify a valid name, as explained in Using GET_xxx Topology Functions.

This function is equivalent to using the `getNodeCoord` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.33 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 Topology Data Model Java Interface).

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()
```

4.34 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

This function is equivalent to using the `getNodeFaceStar` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.35 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 Using GET_xxx Topology Functions. Must not exceed 20 characters.

topo_map
Name of the TopoMap object, or a null value, as explained in Using GET_xxx Topology Functions. (TopoMap objects are explained in TopoMap Objects.)

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 Using GET_xxx Topology Functions.

This function is equivalent to using the `getNodeStar` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.36 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 TopoMap Objects.)

Usage Notes

This function is equivalent to using the `getTopoName` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
SELECT SDO_TOPO_MAP.GET_TOPO_NAME('CITY_DATA_TOPOMAP') FROM DUAL;

SDO_TOPO_MAP.GET_TOPO_NAME('CITY_DATA_TOPOMAP')
--------------------------------------------------------------------------------
CITY_DATA
```

4.37 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 History Information Table.

This function is equivalent to using the `getTopoTransactionId` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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
```

4.38 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 TopoMap Objects.

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 Topology Built from Topology Data.)

```
SELECT SDO_TOPO_MAP.LIST_TOPO_MAPS FROM DUAL;

LIST_TOPO_MAPS
--------------------------------------------------------------------------------
(CITY_DATA_TOPOMAP, CITY_DATA, updatable)
```

4.39 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;
```

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,
  next_load     IN VARCHAR2 DEFAULT 'FALSE'
) 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 TopoMap Objects.)

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.

next_load
`TRUE` allows you to expand the load window area (rectangular portion of the topology) in order to add more data if you discover that the existing window does not cover the area that you want to edit; `FALSE` (the default) does not allow you to add more data to the load window.
The first call to the function on a TopoMap object (`topo_map`) should always have `next_load` as `FALSE` (either by default or explicitly specified). If you need to expand the window area, call the function again but with `next_load` as `TRUE`, then perform any desired additional editing operations.

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, unless you use the function format with `next_load` as `TRUE`. Otherwise, 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).

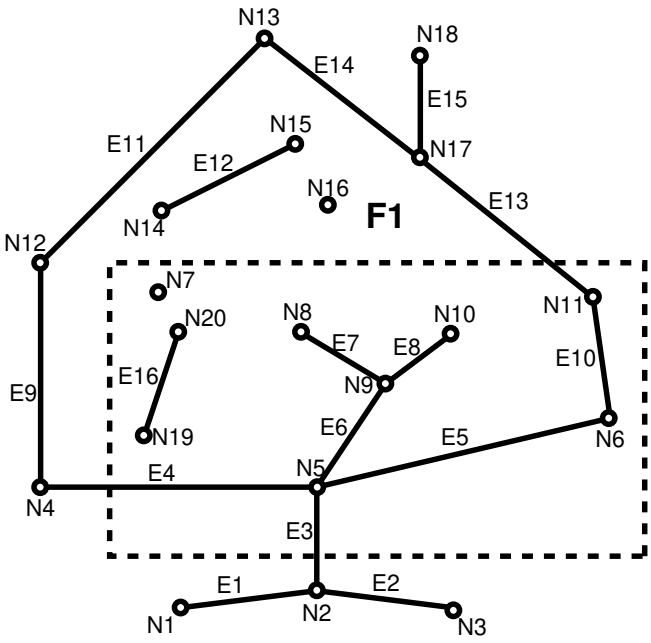
For information about using an in-memory cache to edit topological elements, see Approaches for Editing Topology Data.

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
CALL SDO_TOPO_MAP.LOAD_TOPO_MAP('CITY_DATA_TOPOMAP', 'TRUE') INTO :res_varchar;

Call completed.

PRINT res_varchar;

RES_VARCHAR
--------------------------------------------------------------------------------
TRUE
```

4.40 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 Specifying the Editing Approach with the Topology Parameter). 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 Moving an Edge.

This procedure is equivalent to using the `moveEdge` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Data Model Concepts; 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 Topology Built from Topology Data.)

```
-- 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)
```

4.41 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 Specifying the Editing Approach with the Topology Parameter). 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 Moving a Node.

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 Topology Data Model Java Interface).

Examples

The following example adds an isolated node and then moves it. (The example refers to definitions and data from Topology Built from Topology Data.)

```
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));
```

4.42 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 Specifying the Editing Approach with the Topology Parameter). 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 SDO_EDGE_ARRAY and SDO_NUMBER_ARRAY Types). 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 Moving a Node.

This procedure is equivalent to using the `moveNode` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

Examples

The following example moves node N3 and adjusts the coordinates of the only attached edge. (The example refers to definitions and data from Topology Built from Topology Data.)

```
-- 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)));
```

4.43 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 Specifying the Editing Approach with the Topology Parameter). 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 Removing an Edge.

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);
```

4.44 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 Specifying the Editing Approach with the Topology Parameter). 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 Removing a Node.

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);
```

4.45 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 Removing Obsolete Nodes.)

Parameters

topology
Name of the topology from which to remove obsolete nodes, or null if you are using an updatable TopoMap object (see Specifying the Editing Approach with the Topology Parameter). Must not exceed 20 characters.

Usage Notes

For information about removing obsolete nodes from a topology, see Removing Obsolete Nodes.

Examples

The following example removes all obsolete nodes from the current updatable TopoMap object.

```
CALL SDO_TOPO_MAP.REMOVE_OBSOLETE_NODES(null);
```

4.46 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 Approaches for Editing Topology Data.

This procedure is equivalent to using the `rollbackDB` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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;
```

4.47 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.48 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 TopoMap Objects.)

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 Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
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)
```

4.49 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);
```

4.50 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 Process for Editing Using Cache Explicitly (PL/SQL API). Updates are made, as needed, to the edge, face, and node information tables (described in Topology Data Model Tables). 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 Approaches for Editing Topology Data.

This procedure is equivalent to using the `updateTopology` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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;
```

4.51 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 TopoMap Objects.)

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 Tolerance in the Topology Data Model.

This function is equivalent to using the `validateCache` method of the `TopoMap` class of the client-side Java API (described in Topology Data Model Java Interface).

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 Topology Built from Topology Data.)

```
CALL SDO_TOPO_MAP.VALIDATE_TOPO_MAP('CITY_DATA_TOPOMAP') INTO :res_varchar;

Call completed.

PRINT res_varchar;

RES_VARCHAR
--------------------------------------------------------------------------------
TRUE
```

4.52 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
) 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 TopoMap Objects.)

This function uses a tolerance value of 10E-15 for its internal computations, as explained in Tolerance in the Topology Data Model.

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 Topology Built from Topology Data.)

```
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 part covers the network data model graph feature of Oracle Spatial.

This document has two main parts:

- Topology Data Model provides conceptual, usage, and reference information about the Topology Data Model feature of Oracle Spatial.
- Part II provides conceptual, usage, and reference information about the Network Data Model feature of Oracle Spatial.

Part II contains the following chapters:

- Network Data Model Overview
  This chapter explains the concepts and operations related to the network data model for representing capabilities or objects that are modeled as nodes and links (vertices and edges) in a graph.
- SDO_NET Package Subprograms
  The MDSYS.SDO_NET package contains subprograms (functions and procedures) for managing networks.
- SDO_NFE Package Subprograms
  The MDSYS.SDO_NFE package contains subprograms (functions and procedures) for performing network feature editing.

5

Network Data Model Overview

This chapter explains the concepts and operations related to the network data model for representing capabilities or objects that are modeled as nodes and links (vertices and edges) in a graph.

> **Note:**
>
> Network Data Model is only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

This model is called the Oracle Spatial Network Data Model feature, or simply Network Data Model. This chapter 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.

- 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.
- Main Steps in Using the Network Data Model
  This topic summarizes the main steps for working with the Network Data Model feature in Oracle Spatial. It refers to important concepts, structures, and operations that are described in detail in other topics.
- Network Data Model Concepts
  A network is a type of mathematical graph that captures relationships between objects using connectivity.
- Network Applications
  Networks are used in applications to find how different objects are connected to each other.
- 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.
- Network User Data
  For user data defined through the *xxx*_SDO_NETWORK_USER_DATA views, the default user data I/O implementation (`LODUserDataIOSDO`) is used to access the user data during network analysis. However, some user data is not included in the node or link table, and thus cannot be registered through xxx_SDO_NETWORK_USER_DATA views.
- Feature Modeling
  You can model objects of interest on the network as features.

- Feature Modeling Using Network Feature Editing (NFE)
  Network feature editing (NFE) lets you create and manage an NFE model. An **NFE model** extends the feature modeling capabilities by enabling you to visualize and manipulate features using Java Swing components and a PL/SQL API.
- Network Constraints
  Network constraints are restrictions defined on network analysis computations.
- Network Buffers
  Network Data Model (NDM) introduces the network buffer representation that captures the coverage and cost information on top of the network representation. In contrast to the spatial buffer approach, the cost information of network buffers is accurate and efficient.
- Network Analysis Using Load on Demand
- Network Management and Analysis Using Contraction Hierarchies
- 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.
- Network Data Model and Network Feature Editing (NFE) Model Metadata Views
  Two sets of network metadata views can be created for each schema (user): xxx_SDO_NETWORK_xxxxxx and xxx_SDO_NFE_MODEL_xxxxxx, where the initial *xxx* can be USER or ALL. These views are created, as needed, by Spatial.
- Network Data Model Application Programming Interface
  The Oracle Spatial Network Data Model feature includes two client application programming interfaces (APIs): a PL/SQL interface provided by the SDO_NET package and a Java interface.
- 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 options.
- Network Examples
  This topic presents several Network Data Model examples.
- Network Data Model Tutorial and Other Resources
  Network Data Model learning resources are available.
- README File for Spatial and Related Features

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.

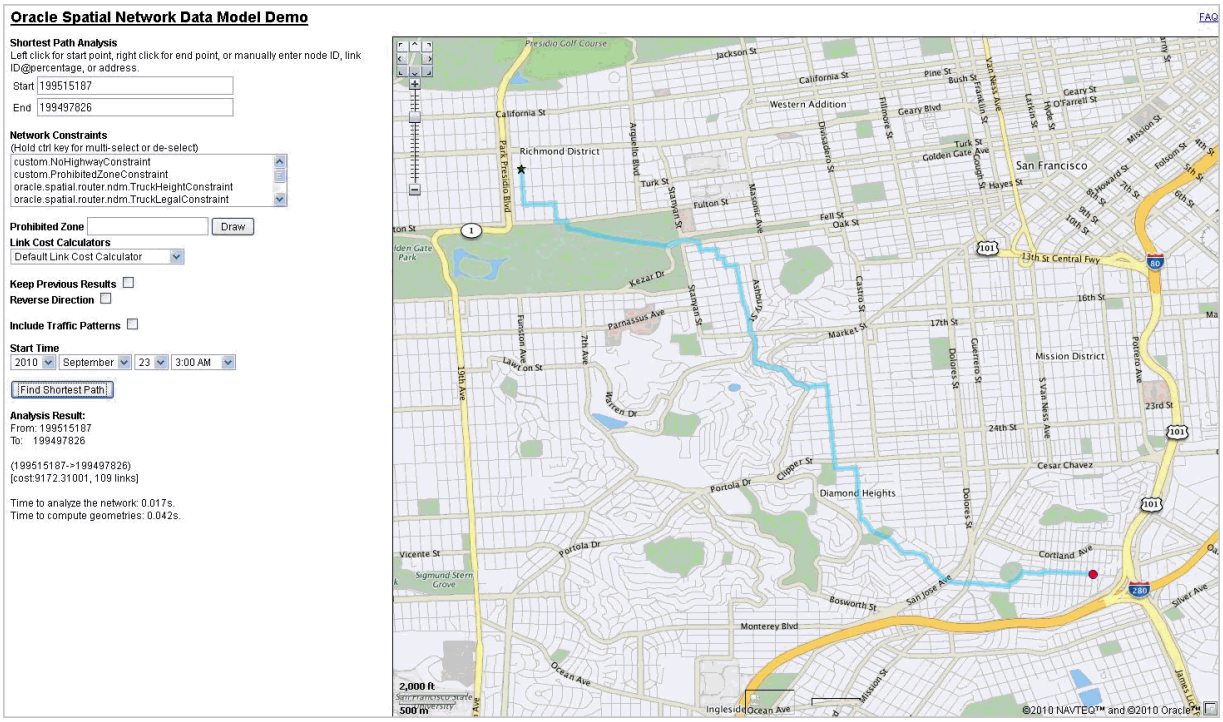
With logical network information, you can analyze a network and answer questions, many of them related to path computing and tracing. For example, for a biochemical pathway, you can find all possible reaction paths between two chemical compounds; or for a road network, you can find the following information:

- 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 addition 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 feature can be used for large, complex networks. For example, Figure 5-1 shows San Francisco and links, which have been defined using the Network Data Model feature, displayed in a demo web-based application for network analysis. (You can install this demo using the NDM tutorial described in Network Data Model Tutorial and Other Resources.)

Figure 5-1 San Francisco 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 package) for creating and managing 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 topic summarizes the main steps for working with the Network Data Model feature in Oracle Spatial. It refers to important concepts, structures, and operations that are described in detail in other topics.

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.

- Perform the operations yourself: create the necessary network tables and update the network metadata.

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 Network Data Model Application Programming Interface.)

- Letting Spatial Perform Most Operations
- Performing the Operations Yourself

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 Network Data Model Tables) and inserts a row with the appropriate network metadata information into the xxx_SDO_NETWORK_METADATA views (described in xxx_SDO_NETWORK_METADATA Views).

   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 Network Data Model Tables.)
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.
2. Insert data into the node table.
3. Create the link table, using the SDO_NET.CREATE_LINK_TABLE procedure.
4. Insert data into the link table.
5. Optionally, create the path table, using the SDO_NET.CREATE_PATH_TABLE procedure.
6. If you created the path table, create the path-link table, using the SDO_NET.CREATE_PATH_LINK_TABLE procedure.
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 xxx_SDO_NETWORK_METADATA Views.)

   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 feature:

- A **node**, also called a vertex, is a point where links can join each other. 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 **network element** is a node or a link.
- 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 a 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 Subpaths.
- 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. Features and feature layer types are explained in Features and Feature Layers
- **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.

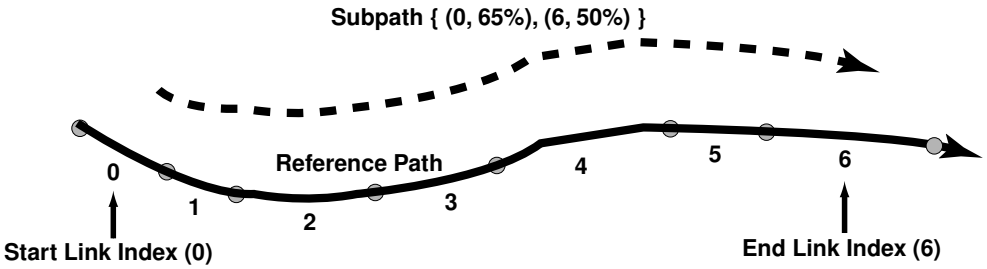
- A **connected component** is a group of network nodes that are directly or indirectly connected. If node A can reach node B, they must belong to the same connected component. If two nodes are not connected, it is concluded that there is no possible path between them. This information can be used as 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.
- Subpaths
- Features and Feature Layers

5.3.1 Subpaths

A **subpath** is a 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.

5.3.2 Features and Feature Layers

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).

A feature consists of one or more feature elements. A **feature element** is a point or line along the network. If it is a point, it can lie on a node or along a line; if it is a line, it can be a full link or a partial link.

Depending on the types of feature elements in the feature, a feature can have any of the **feature types** shown in Table 5-1.

Table 5-1 Feature Types

Type Number	Type Name	Feature Elements Consist of:
1	SDO_NET.FEAT_TYPE_PON	A single point on a node
2	SDO_NET.FEAT_TYPE_POL	A single point on a link
3	SDO_NET.FEAT_TYPE_POINT	A single point, but whether it is on a node or a link is unknown
4	SDO_NET.FEAT_TYPE_LINE	A single line
5	SDO_NET.FEAT_TYPE_MPON	One or more points on one or more nodes
6	SDO_NET.FEAT_TYPE_MPOL	One or more points on one or more links
7	SDO_NET.FEAT_TYPE_MPOINT	One or more points, but the points can be on nodes or links (or a combination)
8	SDO_NET.FEAT_TYPE_MLINE	One or more lines
9	SDO_NET.FEAT_TYPE_COLL	A collection of both points and lines, or the types of the feature elements are unknown

A **feature layer** corresponds to a table containing features that have the same set of attributes. For example, in a roads network, there may be separate feature layers for restaurants and hotels (and perhaps other feature layers for other kinds of things of interest to travelers).

Depending on the types of features in the feature layer, a feature layer can have any of the **feature layer types** shown in Table 5-2, which maps each feature layer type to the associated feature type or types from Table 5-1.

Table 5-2 Feature Layer Types

Layer Type Number	Features in the Layer Are of (type from Table 5-1):
1	Type 1 (SDO_NET.FEAT_TYPE_PON)
2	Type 2 (SDO_NET.FEAT_TYPE_POL)
3	Type 3 (SDO_NET.FEAT_TYPE_POINT)
4	Type 4 (SDO_NET.FEAT_TYPE_LINE)
5	Type 5 (SDO_NET.FEAT_TYPE_MPON) or 1 (SDO_NET.FEAT_TYPE_PON)
6	Type 6 (SDO_NET.FEAT_TYPE_MPOL) or 2 (SDO_NET.FEAT_TYPE_POL)
7	Type 1, 2, 3, 5, 6, or 7
8	Type 8 (SDO_NET.FEAT_TYPE_MLINE) or 4 (SDO_NET.FEAT_TYPE_LINE)
9	Potentially a mixture of any number of feature types

A **parent feature** consists of features from one or more feature layers. For example, in an electrical network, *substation* is a parent feature for the feature layers for all its associated parts, such as *joints*, *switches*, and *cables*.

Features and feature layers can be edited and displayed using the NFE API. This API makes it easier to model certain types of networks, such as electrical, gas, and water utilities.

See Also:

- Feature Modeling Using Network Feature Editing (NFE)

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 topic describes some typical examples of different kinds of network applications.

- Road Network Example
- Subway (Train) Network Example
- Multimodal Network and Temporal Examples
- Utility Network Example
- Biochemical Network Example

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 Subway (Train) 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 Multimodal Network and Temporal Examples

Multimodal networks are networks that consist of multiple modes, such as a network consisting of driving and walking routes. They are usually modeled as individual networks (of the specific mode) and are treated as an aggregate network so that routes of single mode as well as multiple modes can be represented and computed. In general, multimodal networks are "connected" by schedules of different modes, and in such cases they are also *temporal* networks. An example is to compute an itinerary with walking to nearest bus stop, taking the fastest bus route, getting off at the stop that is closest to the destination, then walking to your destination. You could also add modes like driving or flight to be taken into consideration.

Temporal modeling and analysis adds a temporal (time) dimension to network modeling and analysis. The time factor provides cost and/or constraints on top of static (non-temporal) networks. An example is to consider traffic patterns (time-dependent travel time costs) instead of static travel-time costs.

Many metropolitan transportation networks consist of multiple modes such as buses, subways, and commuter rail lines, where transfers across modes are possible (for example, from a bus to the subway). Each transportation mode has a component network within the larger transportation network. The component networks can be modeled using nodes and links, and the transfers across modes can be modeled as links that connect the stops where transfers are possible.

An important feature of such multimodal transportation networks is their schedule-based operation. When performing common network operations such as computing the fastest route from a start point to an end point, the schedule information and possible transfers across modes must be considered. The schedule information at stops can be represented as user-defined data at the nodes representing these stops. Examples of operations that use schedule information in a multimodal network are (A) finding the fastest route (minimum travel time) from a start point to an end point for a specified start time, and (B) finding the latest departure time at a start point to reach an end point by a specified arrival time.

5.4.4 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.

5.4.5 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.

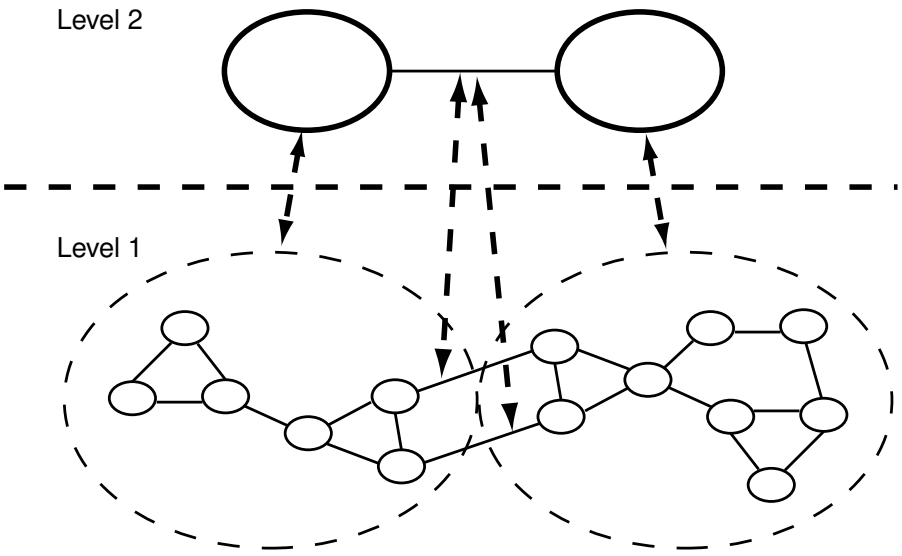
A **network hierarchy** enables you to represent a network with multiple levels of abstraction by assigning a hierarchy level to each node. (Links are not assigned a hierarchy level, and links can be between nodes in the same hierarchy level or in different levels.) The lowest (most detailed) level in 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.

Given certain grouping of nodes in a network, a parent network can be defined. The group IDs in the child network are used as node IDs in the parent network. The aggregated links between groups in the child network represent the links in the parent network, with arbitrary link IDs assigned.

A network can have multiple ways of grouping its nodes based on different criteria; therefore, it can have multiple parent networks. In addition, nodes in a parent network can be further grouped to form a higher-level parent network. For example, in a social network, members can be grouped by city, profession, income, or other criteria. Members grouped by city, for example, can be further grouped into higher-level county, state, or country networks.

The parent-child network relationship is defined through the CHILD_NETWORK and HIERARCHY_TABLE_NAME columns in the network metadata.

> **Note:**
>
> Do not confuse a hierarchical network with a **multilevel network**, which is a network with multiple link levels. A multilevel network does *not necessarily* have parent-child relationships between nodes; that is, a multilevel network may also be a hierarchical network or may also *not* be a hierarchical network. In a multilevel network, a higher-level network (such as level 2) is just a subnetwork of a lower-level network (such as level 1), with link levels greater than or equal to the higher-level link.

5.6 Network User Data

For user data defined through the *xxx*_SDO_NETWORK_USER_DATA views, the default user data I/O implementation (`LODUserDataIOSDO`) is used to access the user data during network analysis. However, some user data is not included in the node or link table, and thus cannot be registered through xxx_SDO_NETWORK_USER_DATA views.

For such user data, users must provide their own custom implementation of the user data I/O interface. A typical way of implementing a custom data I/O interface is to generate BLOBs corresponding to node and link user data, one BLOB for each partition, and then retrieve user data information from the BLOBs during network analysis.

Network Data Model also allows you to associate multiple categories of user-defined data (categorized user data) with a single network. For example, in a multimodal network, if you need to associate driving-related attributes (such as speed limit) with a link in addition to the link's multimodal attributes, user-defined data can be organized in two categories, one for driving-related attributes and the other for multimodal attributes.

See these examples of user-defined data:

- User-Defined Data Example (PL/SQL and Java)
- User-Defined Data Example (Custom User Data I/O Implementation)

5.6.1 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 xxx_SDO_NETWORK_USER_DATA Views) contain information about user-defined data.

To use user-defined data, you must set the USER_DEFINED_DATA column value to `Y` in the appropriate *xxx*_SDO_NETWORK_METADATA views (described in Section xxx_SDO_NETWORK_METADATA Views).

Example 5-1 inserts link-related user-defined data into the network metadata.

Example 5-1 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, category_id)
                  values ('bi_test', 'LINK', 'interaction', 'VARCHAR2', 50, 0) ;
```

```
-- 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,
category_id)
                  values ('bi_test','LINK','PROB','NUMBER', 0) ;
```

After a network or network partition is loaded, user-defined data is available in Java representations. You can access user-defined data through the getCategorizedUserData and setCategorizedUserData methods for the Node, Link, Path, and SubPath interfaces. For example:

```
//The user data index is the sequence number of a user data within a category sorted by
data name.

int interactionUserDataIndex = 0;
int probUserDataIndex = 1;

String interaction = (String)link.getCategorizedUserData().getUserData(0).
                                get(interactionUseDataIndex);

double prob = ((Double)link.getCategorizedUserData().getUserData(0).
                      get(probUserdataIndex)).doubleValue();
```

5.6.2 User-Defined Data Example (Custom User Data I/O Implementation)

This section presents an example of a custom user data I/O implementation (nondefault implementation) of the `LODUserDataIO` interface. In Example 5-2, user data associated with links is written to BLOBs (one BLOB per partition) and read from BLOBs during analysis. It is assumed that the user-defined data BLOB for multimodal data for each partition has the partition ID and number of links associated with the partition, followed by *<Link ID, link route ID>* for each link.

Example 5-2 Implementation of writeUserData method of LODUserDataIO

```
//Method getLinksInPartition(partitionId) computes a vector that
// consists of  the ID and the route ID of each link associated with a partition
// with ID = partitionId
LinkVector = getLinksInPartition(partitionId);

ObjectOutputStream dout = null;

//Insert an empty blob for the partition with ID = partition_id
String insertStr = "INSERT INTO " + MULTIMODAL_USER_DATA +
                           " (partition_id, blob) " + " VALUES " + " (?,
EMPTY_LOB())" ;

PreparedStatement stmt = conn.prepareStatement(insertStr);
stmt.setInt(1,partitionId);
int n = stmt.executeUpdate();
stmt.close();

//lock the row for blob update
 String lockRowStr = "SELECT blob FROM " + MULTIMODAL_USER_DATA +
                                 " WHERE partition_id = ? " + " FOR UPDATE";
 stmt = conn.prepareStatement(lockRowStr);
 stmt.setInt(1,partitionId);
 ResultSet rs = stmt.executeQuery();

 rs.next();
oracle.sql.BLOB userDataBlob = (oracle.sql.BLOB) rs.getBlob(1);
stmt.close();
```

```
OutputStream blobOut = ((oracle.sql.BLOB) userDataBlob).setBinaryStream(1);
dout = new ObjectOutputStream(blobOut);

//write partition ID
dout.writeInt(partitionId);
int numLinks = linkVector.size()

 for (int i=0; i<linkVector.size(); i++) {
       //MultimodalLink is a class with variables link ID and route ID
       MultimodalLink link = (MultimodalLink) linkVector.elementAt(i);
        //write link ID
       dout.writeLong(link.getLinkId());

       // write route ID into file
       dout.writeInt(link.getRouteId());
  }
  dout.close();
   blobOut.close();
   rs.close();
```

The subsections that follow describe the implementations of the `writeUserData` and `readUserData` methods of the `LODUserDataIO` interface.

- Implementation of writeUserData method of LODUserDataIO
- Implementation of readUserData method of LODUserDataIO

5.6.2.1 Implementation of writeUserData method of LODUserDataIO

In the implementation of the writeUserData method of LODUserDataIO, the user-defined data BLOB table name is assumed to be MULTIMODAL_USER_DATA.

```
//Method getLinksInPartition(partitionId) computes a vector that
// consists of  the ID and the route ID of each link associated with a partition
// with ID = partitionId
LinkVector = getLinksInPartition(partitionId);

ObjectOutputStream dout = null;

//Insert an empty blob for the partition with ID = partition_id
String insertStr = "INSERT INTO " + MULTIMODAL_USER_DATA +
                     " (partition_id, blob) " + " VALUES " + " (?, EMPTY_BLOB())" ;

PreparedStatement stmt = conn.prepareStatement(insertStr);
stmt.setInt(1,partitionId);
int n = stmt.executeUpdate();
stmt.close();

//lock the row for blob update
 String lockRowStr = "SELECT blob FROM " + MULTIMODAL_USER_DATA +
                                  " WHERE partition_id = ? " + " FOR UPDATE";
 stmt = conn.prepareStatement(lockRowStr);
 stmt.setInt(1,partitionId);
 ResultSet rs = stmt.executeQuery();

 rs.next();
oracle.sql.BLOB userDataBlob = (oracle.sql.BLOB) rs.getBlob(1);
stmt.close();

 OutputStream blobOut = ((oracle.sql.BLOB) userDataBlob).setBinaryStream(1);
 dout = new ObjectOutputStream(blobOut);
```

```
//write partition ID
dout.writeInt(partitionId);
int numLinks = linkVector.size()

 for (int i=0; i<linkVector.size(); i++) {
       //MultimodalLink is a class with variables link ID and route ID
       MultimodalLink link = (MultimodalLink) linkVector.elementAt(i);
        //write link ID
       dout.writeLong(link.getLinkId());

       // write route ID into file
       dout.writeInt(link.getRouteId());
  }
  dout.close();
   blobOut.close();
   rs.close();
```

5.6.2.2 Implementation of readUserData method of LODUserDataIO

The user-defined data is accessed through the `getCategorizedUserData` and `setCategorizedUserData` methods for the `Node`, `Link`, `Path`, and `SubPath` interfaces and `getUserData` and `setUserData` methods of the `CategorizedUserData` interface.

```
//Read the blob for the required partition from the user data blob table
// In this example,
// MULTIMODAL_USER_DATA  is the name of user -defined data blob table
BLOB multimodalBlob = null;
String queryStr = "SELECT blob FROM " +
MULTIMODAL_USER_DATA
                              " WHERE partition_id = ?";
PreparedStatement stmt = conn.prepareStatement(queryStr);
stmt.setInt(1,partitionId);
ResultSet rs = stmt.executeQuery();
if (rs.next())   {
     multimodalBlob = (oracle.sql.BLOB)rs.getBlob(1);
}

// Materialize the blob value as an input stream
InputStream is = multimodalBlob.getBinaryStream();

//Create an ObjectInputStream that reads from the InputStream is
ObjectInputStream ois = new ObjectInputStream(is);

//Read the values of partition ID and number of links from the blob
int partitionId = ois.readInt();
int numLinks = ois.readInt();

for (int i=0; i<numLinks; i++)  {

    //Read link ID and route ID for each link
     long linkId = ois.readLong();
     int routeId = ois.readInt();

     //MultimodalLinkUserData is an implementation of NDM LOD UserData interface
     //Implementation is provided at the end of the example
     linkUserData = new MultimodalLinkUserData(routeId);

     //Get the link object corresponding to the link ID
     LogicalNetLink link = partition.getLink(linkId);
```

```
    //Get the (categorized) user data associated with the link.
    CategorizedUserData cud = link.getCategorizedUserData();

    // If the link does not have categorized user data associated with it,
    // initialize it to linkUserData
    // Else, set the user data for category USER_DATA_MULTIMODAL
    // to linkUserData
    if (cud == null) {
           UserData [] userDataArray = {linkUserData};
           cud = new CategorizedUserDataImpl(userDataArray);
            link.setCategorizedUserData(cud);
    }
    else {
           cud.setUserData(USER_DATA_MULTIMODAL,linkUserData);
    }
}
```

The following segment shows how to read the user-defined data, specifically the route ID associated with a link during analysis.

```
//info is an instance of LODAnalysisInfo
LogicalLink currentLink = info.getCurrentLink();

//Read the user-defined data (in this case, route ID)
int linkRouteId   = (Integer)currentLink.getCategorizedUserData().

getUserData(USER_DATA_MULTIMODAL).
                           get(INDEX_LINK_ROUTEID);


Implementation of MultimodalLinkUserData :

class MultimodalLinkUserData implements UserData
{
    private int routeId;

    protected MultimodalLinkUserData(int routeId)
   {
        this.routeId = routeId;
   }

  public Object get(int index)
  {
    switch(index)
    {
         case INDEX_LINK_ROUTEID:
              return routeId;
    }
    return null;
  }

  public void set(int index, Object userData)
  {
    switch(index)
    {
         case INDEX_LINK_ROUTEID:
            this.routeId = (Integer) userData;
    }
  }

  public int getNumberOfUserData()
  {
```

```
        return 1;
    }

    public Object clone()
{
      return new MultimodalLinkUserData(routeId);
    }
}
```

5.7 Feature Modeling

You can model objects of interest on the network as features.

A feature consists of one or many feature elements. A feature element can be a point, a link, or a partial link along the network. You can define feature layers on top of a network. For example, restaurants and hotels can each be defined as a feature layer on a road network, and switches can be defined as a feature layer on an electrical network.

The following are the typical steps for using feature modeling.

1. Create a feature layer.

   For example, the points of interest (POIs) on a road network can be modeled as features. Each type of POI (hotels, restaurants, hospitals, schools, and so on) corresponds to one feature layer. The following example adds a feature layer for hotels:

   ```
   sdo_net.add_feature_layer(
      'US_ROAD_NETWORK', --network name
      'HOTEL',           --feature layer name
      2,                 --feature layer type: point on link
      'HOTEL_TAB',       --feature table or view name
      'HOTEL_NET_REL',   --relation table or view name
      null);             --hierarchy table or view name
   ```

2. Register feature user data, if any application-specific feature attributes are potentially useful in feature analysis.

   Feature user data is registered by adding an entry in the XXX_SDO_NETWORK_USER_DATA view, just like registering the user data for network nodes or links, except that the TABLE_TYPE column is set to the name of the feature table. The following example adds hotel name as user data for hotel features:

   ```
   INSERT INTO USER_SDO_NETWORK_USER_DATA(
      network, table_type, data_name, data_type, category_id)
   VALUES(
      'US_ROAD_NETWORK',  --network name
      'HOTEL_TAB',            --feature table or view name
      'NAME',             --user data name, i.e., name of the user data column
      'VARCHAR2',         --user data type
      3);                 --user data category
   ```

3. Add, update, or delete features on the feature layer.

   If the content of feature tables, feature element relationship table, and feature hierarchy table (all described in Feature Layer Tables) -- or any combination of these tables -- is managed by the data provider, then you can skip this step. Otherwise, you can call procedures in the SDO_NET package, such as ADD_FEATURE, UPDATE_FEATURE, DELETE_FEATURES, to add, update or delete features in a feature layer. (The SDO_NET subprograms are described in SDO_NET Package Subprograms.)

4. Perform feature analysis using `NetworkAnalyst`. The feature analysis functions include:

   - Shortest paths between features

- Nearest (reaching) features
- Within (reaching) cost features

- Data Types Used for Feature Modeling

5.7.1 Data Types Used for Feature Modeling

This section describes the following PL/SQL data types that are used for parameters and return values of some SDO_NET package subprograms related to feature modeling:

- SDO_NET_FEAT_ELEM_ARRAY
- SDO_NET_FEAT_ELEM
- SDO_NET_LAYER_FEAT_ARRAY
- SDO_NET_LAYER_FEAT
- SDO_NETWORK_NVP_TAB
- SDO_NETWORK_NVP

SDO_NET_FEAT_ELEM_ARRAY is defined as `VARRAY(1024) OF MDSYS.SDO_NET_FEAT_ELEM`.

SDO_NET_FEAT_ELEM is defined as:

```
FEAT_ELEM_TYPE    NUMBER
NET_ELEM_ID       NUMBER
START_PERCENTAGE  NUMBER
END_PERCENTAGE    NUMBER
```

SDO_NET_LAYER_FEAT_ARRAY is defined as `VARRAY(1024) OF MDSYS.SDO_NET_LAYER_FEAT`.

SDO_NET_LAYER_FEAT is defined as:

```
FEATURE_LAYER_ID  NUMBER
FEATURE_ID        NUMBER
```

SDO_NETWORK_NVP_TAB is defined as `TABLE OF MDSYS.SDO_NETWORK_NVP`.

SDO_NETWORK_NVP is defined as:

```
NAME   VARCHAR2(128)
VALUE  VARCHAR2(1024)
```

5.8 Feature Modeling Using Network Feature Editing (NFE)

Network feature editing (NFE) lets you create and manage an NFE model. An **NFE model** extends the feature modeling capabilities by enabling you to visualize and manipulate features using Java Swing components and a PL/SQL API.

NFE lets you define features on the top of an existing network. For example, restaurants and hotels can be defined as features on the top of a road network. You can also define models that consist just of features (network elements are hidden from you), and where the connectivity can be restricted by rules. Such connectivity restrictions are typically used in utility networks, such as electrical, water, or gas networks, to model the network devices and restrict the connectivity. One example of using connectivity restrictions is to avoid connections between high tension and low tension devices in an electrical network

NFE includes concepts such as feature classes and rules, which can be built into a model. Metadata tables and views are automatically created and maintained when you create and work with an NFE model.

The minimum privileges required for using NFE features are `CREATE TABLE`,`CREATE VIEW`, `CREATE SEQUENCE`, and `CREATE SESSION`. These are in addition to any other privileges that the user might need to perform operations not specifically related to NFE.

- Creation Modes for NFE Models
- NFE Feature Classes
- NFE Rules
- Data Types Used for NFE Connectivity Rules

5.8.1 Creation Modes for NFE Models

You can create a network feature editing (NFE) model in one of two possible modes: from scratch and over an existing network model.

- **From scratch**: In this mode, a network is generated automatically when you create the NFE the model, and the network is assign to the model. You will work only with features, and the underlying network elements will be hidden from you. This mode supports rules whose statements specify what kind of features (feature classes) can be connected with each other kinds of features. One typical use of this mode is to model utility networks, such as electricity or water networks, where you want to restrict the connections among certain components in the network.
- **Over an existing network model**: In this mode, an existing network is used when you create the NFE model. The network is visible to you, and you can add features over the network elements. Rules are *not* supported in this mode.

The feature class relationship table contains information about the relationship between features and feature classes, that is, which feature belongs to which feature class.

5.8.2 NFE Feature Classes

A feature class describes a group of features based on attributes values, shape, and style. Each feature class belongs only to one feature layer, so the group of features in the feature class also belongs to the same feature layer.

For example, an electrical network might include two feature layers: Transformers and Conductors. Each of these feature layers might include two feature classes:

- The Transformers feature layer might include the HT Transformers (high tension transformers) and LT Transformers (low tension transformers), where transformers within each feature type have specified input and output voltages, and tension types. Both feature types are associated with points (their shape). Each feature type has a different associated icon for use in diagrams.
- The Conductors feature layer might include the HT Conductors (high tension conductors) and LT Conductors (low tension conductors), where conductors within each feature type have specified attribute values. Both feature types are associated with lines (their shape). Each feature type has a different associated icon for use in diagrams.

The following table lists the shapes that can be associated with a feature class, and requirements and restrictions depending on the creation mode used for the NFE model..

Table 5-3 Shapes for NFE Feature Classes

Shape Type (Number)	Shape Name	Model Created from Scratch	Model Created over Existing Network
1	Point	Features can have only one PointOnNode feature element.	Features can have one or more feature elements of types PointOnNode and PointOnLine.
2	Line	Features can have only one line feature element. A line supports connections only on its start and end points.	Features can contain one or more line feature elements. It does not matter if the lines are adjacent or not.
3	Complex line	Features can have multiple adjacent (connected) line feature elements. A complex line supports connection on its middle points; when this happens, the complex line is divided into two line feature elements.	(Complex lines not supported for this creation mode.)
4	Path	Feature representing a path. This type of feature class is typically used when analysis is performed.	Feature representing a path. This type of feature class is typically used when analysis is performed.

5.8.3 NFE Rules

Rules in NFE models are constraints related to feature connections. Rules are available only for models created from scratch, that is, not for models created over an existing network. Rules can be customized, and can support any kind of connection constraint.

Rules can generally be classified into cardinality rules and connectivity rules:

- Cardinality rules are used over point features that have specific characteristics (that is, belong to the same feature class), to specify the maximum and minimum number of incoming and outgoing connections that the point feature can accept.
- Connectivity rules are used to indicate whether two features can be connected if they have some specific characteristics (feature layer, feature class, and attributes) and a specific interaction. A connectivity rule will allow the connection with a point feature as long as the point feature has available space to support the corresponding incoming and outgoing connections needed, as dictated by its related cardinality rule.

  Connectivity rules are the only way to connect features in NFE, and they are always positive; that is, they allow connections, but cannot deny them. The absence of connectivity rules between elements indicates that elements cannot be connected.

Line-Point Connectivity Rules

A line-point connectivity rule states that whenever a line feature and a point feature interact in certain ways, they can be connected.

A line-point connectivity rule, along with its cardinality rule, can be expressed as follows: “Any Line Feature from Feature Layer L1 with Feature Class C1 matching the condition P1 can be connected to a Point Feature from Feature Layer L2 with Feature Class C2 matching the condition P2. The cardinality of Point Feature indicates that maximum of X incoming connections and a maximum of Y outgoing connections are allowed.”

The following table shows examples of line-point connectivity rules.

Table 5-4 Examples of Line-Point Connectivity Rules

Line Condition	Point Condition	Cardinality	Description
HT Conductor class from Conductors layer	HT Transformer class from Transformers layer	Unbounded	Any number of high tension conductors can connect to a high tension transformer.
HT Conductor class from Conductors layer	LT Transformer class from Transformers layer	In: unbounded; Out: 0	Any number of incoming high tension conductors can connect to a low tension transformer, but no outgoing high tension conductors can be connected out from a low tension transformer. A low tension transformer steps down the voltage from a high tension conductor to a low tension conductor.
LT Conductor class from Conductors layer	LT Transformer class from Transformers layer	Unbounded	Any number of low tension conductors can connect to a low tension transformer.

Line-Line Connectivity Rules

A line-line connectivity rule states that when two line features interact in some specific way, they can be connected through one specific point feature. The rule can also specify to automatically create the connecting point, if it does not already exist, when the interaction between the feature lines occurs. Another way of looking at a line-line rule is as two line-point rules having in common the same point feature restrictions, so in that sense a line-line rule refers to two line-point rules with the same point feature condition.

An example line-line connectivity rule might be expressed as follows: “If interaction I is true between (Line Feature1 from Feature Layer L1 with Feature Class C1 matching the condition P1) AND (Line Feature2 from Feature Layer L2 with Feature Class C2 matching the condition P2) the lines can be connected using (Point Feature P1) (Optionally: create automatically Point Feature P1 if it does not exist).”

Left Hand Side and Right Hand Size of a Line-Line Connectivity Rule

A line-line connectivity rule specifies interaction between two line features: line feature 1 and line feature 2. These two line features are known as the **left hand side (LHS) of the rule** and the **right hand side (RHS) of the rule**, respectively. Any feature that is on line feature 1 is considered to be on the LHS of the rule, and any feature that is on line feature 2 is considered to be on the RHS of the rule.

The following table shows examples of line-line rules suitable for an electrical network model, identifying the LHS and RHS line feature groups of the rule. Each row in the table describes one line-line rule.

Table 5-5 LHS and RHS for Sample Line-Line Rules

LHS Line Features Group	Interaction Type	RHS Line Features Group	Common Point	Description
HT Conductor class from Conductors layer	Touch end point	HT Conductor class from Conductors layer	HT Transformer class from Transformers layer	High tension conductors are automatically connected to each other by a high tension transformer when they are touching on their end points.

Table 5-5 (Cont.) LHS and RHS for Sample Line-Line Rules

LHS Line Features Group	Interaction Type	RHS Line Features Group	Common Point	Description
LT Conductor class from Conductors layer	Touch end point	LT Conductor class from Conductors layer	LT Transformer class from Transformers layer	Low tension conductors are automatically connected to each other by a low tension transformer when they are touching on their end points.
HT Conductor class from Conductors layer	Touch end point	LT Conductor class from Conductors layer	LT Transformer class from Transformers layer	High tension conductors are automatically connected to low tension conductors by a low tension transformer when they are touching on their end points.

Rule Decision Handlers

Rules can be customized by using rule decision handlers, which can be applied to both line-point and line-line rules. **Decision handlers** are a mechanism that allows a user to determine which elements in an interaction must be connected and how.

A rule implementation always uses a decision handler to manage the connections among the features. NFE provides a default implementation to connect features for line-point and line-line rules. When executing a rule, the default decision handler implementation will try to connect as many elements matching the rule as possible, but there could be scenarios where you want more control over the connections than the rule provides by default, in which case you must modify the default decision handler implementation..

Using a water network example, assume that two valves can interact with pipes at a spatial point. Each valve has four outlets. Either (A) all four pipe ends can be connected to the four outlets of a single valve, or (B) two pipe ends can connect to two opposite outlets of one valve, and the two other pipe ends can connect to two opposite outlets of the other valve. The default decision handler implementation will try the first approach (all four pipe ends connecting to a single valve); but if you wanted to distribute the four pipes among the two available valves, you could implement a custom decision handler tio use the second approach (two pipe ends connecting to the first valve, the two other pipe ends connecting to the second valve).

Rule Instances

When features are connected because of the application of certain rule, the group of connected features is called a **rule instance**, that is, an instance of the rule that allowed the connection. Rule instances are maintained in the Rule Instance Table.

A rule definition can be removed or modified only if no dependent rule instances exist (that is, no connections that depend on any existing rule instances)

If a feature element involved in a rule instance is deleted, the rule instance may or may not be automatically deleted. If the feature element deletion causes the number of elements to drop below the minimum number for the rule instance (which depends on the type of rule), then the rule instance is deleted. However, if the feature element deletion does not cause the number of elements to drop below the minimum number for the rule instance, then the feature element is deleted from the rule instance, but the .the rule instance itself is not deleted (but it is modified).

5.8.4 Data Types Used for NFE Connectivity Rules

This section describes the following PL/SQL data types that are used for parameters and return values of some SDO_NFE package subprograms related to network feature editing (NFE):

- SDO_INTERACT_POINT_FEAT_ARRAY
- SDO_INTERACT_POINT_FEAT
- SDO_INTERACT_LINE_FEAT_ARRAY
- SDO_INTERACT_LINE_FEAT
- SDO_INTERACTION_ARRAY
- SDO_INTERACTION

SDO_INTERACT_POINT_FEAT_ARRAY is defined as `VARRAY(1024) OF MDSYS.SDO_INTERACT_POINT_FEAT`.

SDO_INTERACT_POINT_FEAT is defined as:

```
FEATURE_LAYER_ID       NUMBER
FEATURE_ID             NUMBER
FEATURE_CLASS_ID       NUMBER
NODE_ID                NUMBER
NODE_GEOM              SDO_GEOMETRY
AVAILABLE_IN_CONN      SDO_NUMBER_ARRAYSET
AVAILABLE_OUT_CONN    SDO_NUMBER_ARRAYSET
RUNTIME_CREATED         CHAR(1)
```

SDO_INTERACT_LINE_FEAT_ARRAY is defined as `VARRAY(1024) OF MDSYS.SDO_INTERACT_LINE_FEAT`.

SDO_NET_LAYER_FEAT is defined as:

```
FEATURE_LAYER_ID      NUMBER
FEATURE_ID            NUMBER
FEATURE_CLASS_ID      NUMBER
LINK_ID               NUMBER
LINK_GEOM             SDO_GEOMETRY
START_NODE            SDO_INTERACT_POINT_FEAT
END_NODE              SDO_INTERACT_POINT_FEAT
BIDIRECTED            CHAR(1)
INTERSECTION_LOCATION     NUMBER
RULE_SIDE             CHAR(1)
```

SDO_INTERACTION_ARRAY is defined as `VARRAY (1048576) OF MDSYS.SDO_INTERACTION`.

SDO_INTERACTION is defined as:

```
LINES             SDO_INTERACT_LINE_FEAT_ARRAY
POINTS            SDO_INTERACT_POINT_FEAT_ARRAY
INTERSECT_PT_GEOM SDO_GEOMETRY
```

5.9 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, you must specify that constraint.

Examples of Java classes to implement network constraints are provided in the Network Data Model demo files, which are described in Network Data Model Tutorial and Other Resources. 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.

5.10 Network Buffers

Network Data Model (NDM) introduces the network buffer representation that captures the coverage and cost information on top of the network representation. In contrast to the spatial buffer approach, the cost information of network buffers is accurate and efficient.

You can compute a network buffer with the LOD Java API. Each buffer center can either be a node or a point along a link (`PointOnNet`), and a network buffer can have multiple buffer centers. For large-scale drive time and distance analysis, you can precompute and persist network buffers in the database. Additionally, you can query on coverage, cost, and shortest path on one or multiple network buffers using simple SQL statements and without any spatial operations or network analysis.

The following sections show examples using Java snippets to create and persist a network buffer to the database and then query the coverage and cost information using SQL.

- Creating a Network Buffer
  You can create a network buffer using the Java API.
- Analyzing a Network Buffer Using the Java API
  You can perform network buffer analysis using the NDM Java API.
- Analyzing a Network Buffer Using SQL Queries
  You can perform network buffer analysis using SQL queries directly on the network buffer tables.

5.10.1 Creating a Network Buffer

You can create a network buffer using the Java API.

The following example shows how to create a network buffer that captures the coverage within a ten-minute drive of a single location using the Network Analyst class. The newly created buffer is then stored in the network buffer tables in the database with the prefix `SF`. However, note, the `writeNetworkBuffer()` method creates the network buffer tables only if they are already not existing in the database. Otherwise, the existing tables are used.

Example 5-3 Creating a Network Buffer

```
PointOnNet[] startPoint = {new PointOnNet(linkId, percent)};
double cost = 600; // 600 seconds

// Creates a network buffer with the given start point and cost
NetworkBuffer buffer = analyst.networkBuffer(startPoint, cost, null);
```

```
int bufferId = 0;
String tableNamePrefix = "SF";

// Writes the network buffer tables
networkIO.writeNetworkBuffer(buffer, bufferId, tableNamePrefix);
```

> **See Also:**
>
> Network Buffer Schema

5.10.2 Analyzing a Network Buffer Using the Java API

You can perform network buffer analysis using the NDM Java API.

The following example obtains the cost associated with a network buffer to a given location in `PointOnNet`:

Example 5-4 Retrieving Cost Using Java API

```
// The cost is Double.POSITIVE_INFINITY if not covered
double cost = buffer.getCosts(pt) == null ? Double.POSITIVE_INFINITY :
buffer.getCosts(pointOnNet)[0];

// To get the ids and costs of the covered nodes and links from a network
buffer:

// Finds all covered nodes
LogicalNode[] nodes = buffer.getElements().getNodes();
for (LogicalNode node : nodes) {
    long nodeId = node.getId();
    // Returns the minimum costs to reach the node from a buffer center
    double[] nodeCosts = buffer.getNodeCosts(node.getId());
}

// Coverage and cost information for covered links stored in LinkIntervals
NetworkBuffer.LinkIntervals[] linkIntervals =
buffer.getElements().getLinkIntervals();

// Finds all covered links
for (int i = 0; i < linkIntervals.length; i++) {
    LogicalLink link = linkIntervals[i].getLink();
    long linkId = link.getId(); // link Id
    NetworkBuffer.DoubleInterval[] intervals =
linkIntervals[i].getIntervals();
    // Finds costs for the covered links
    for (NetworkBuffer.DoubleInterval interval : intervals) {
        NetworkBuffer.DoubleInterval costs =
                buffer.getLinkIntervalCosts(linkId, interval);
        // Computes the cost of the mid-point of the covered link
        double percentage = 0.5; // range:[0,1]
        double[] cost = buffer.getCosts(new PointOnNet(linkId, percent));
```

```
    }
}
```

> **Note:**
>
> See Oracle Spatial API package for more examples.

5.10.3 Analyzing a Network Buffer Using SQL Queries

You can perform network buffer analysis using SQL queries directly on the network buffer tables.

Example 5-5 Retrieving Cost Using SQL Queries

The following example obtains the costs of all buffers from a center location point (`link_id=945669955, percentage=0.95`) in an ascending cost order:

```
SELECT buffer_id, MIN(start_cost+(0.95-start_percentage)*(end_cost-
start_cost)/(end_percentage-start_percentage)) cost
FROM SF_NB1$
WHERE link_id=945669955
AND (0.95-start_percentage)*(end_percentage-start_percentage)>=0
GROUP BY buffer_id
ORDER BY cost;
```

Example 5-6 Retrieving Links of the Shortest Path Using SQL Queries

The following example obtains the links of the shortest path from a given location (`link_id =799415310`) to a specific buffer center (buffer_id = 1) using a hierarchical query:

```
SELECT buffer_id, link_id, prev_link_id
FROM SF_NB1$
WHERE buffer_id = 1
START WITH link_id = 799415310
CONNECT BY PRIOR prev_link_id = link_id AND
      PRIOR start_cost = end_cost AND
      PRIOR buffer_id = buffer_id
ORDER BY start_cost;
```

5.11 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 Main Steps in Using the Network Data Model.

2. Partition the network using the SDO_NET.SPATIAL_PARTITION procedure, as explained in Partitioning a Network.
3. Optionally, generate partition BLOBs, as explained in Generating Partition BLOBs.
4. Configure the load on demand environment, including the partition cache, as explained in Configuring the Partition Cache.
5. Analyze the network, as explained in Analyzing the Network.

> **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 Partitioning and Load on Demand Analysis Examples (PL/SQL_ XML_ and Java).

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 Network Data Model Tutorial and Other Resources.

- Partitioning a Network
- Generating Partition BLOBs
- Configuring the Partition Cache
- Analyzing the Network
- Using Link Levels for Priority Modeling
- Precomputed Analysis Results

5.11.1 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.

5.11.2 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.

5.11.3 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 Network Data Model Tutorial and Other Resources.

5.11.4 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 Network Data Model Tutorial and Other Resources).

You can write the analysis results to the database using the load on demand Java API.

5.11.5 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.

5.11.6 Precomputed Analysis Results

Some analysis operations, such as connected component analysis, can be time consuming. To improve runtime 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.

At runtime, 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.

5.12 Network Management and Analysis Using Contraction Hierarchies

Contraction hierarchies can be used to find the shortest path in a graph, potentially providing better performance than traditional shortest path algorithms like Dijkstra or A*.
Thus, effective with Release 21c, Spatial provides two approaches for using the network data model:

- Load on demand (LOD) to support customization with business rules and data.
- Contraction hierarchies, including a REST API for easy web services development and deployment with static networks.

The contraction hierarchies approach does the following:

- Precomputes and stores the node order (importance) and shortcut links in a set of files.
- Finds the shortest path with shortcut links by skipping unimportant nodes during queries.
- Maps the shortcut links back to the original graph representation.

Using contraction hierarchies involves using a contraction hierarchies REST API for model management and analysis with contraction hierarchies.

Subtopics include:

- Contraction Hierarchies REST API
- Contraction Hierarchies REST API Examples
- Contraction Hierarchies Model Management

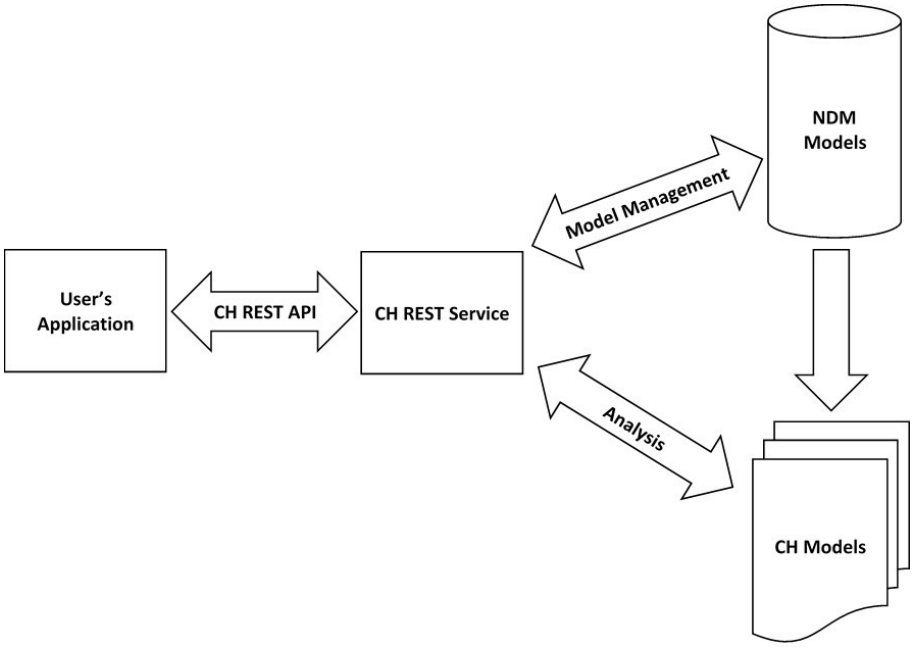
- Contraction Hierarchies Analysis
- Using the Contraction Hierarchies REST API (including the Web Demo)
- REST API Reference Information

Contraction Hierarchies REST API

You can use the contraction hierarchies REST API to manage contraction hierarchies models and perform analysis on the models. You can issue requests and receive responses using HTTP and HTTPS protocols. Both JSON and XML formats are supported.

The following figure shows the contraction hierarchies REST API architecture.

Figure 5-4 Contraction Hierarchies REST API Architecture



In the preceding figure:

- The user's application uses the contraction hierarchies REST API to send requests and receive responses from the contraction hierarchies REST service.
- When the contraction hierarchies REST service receives a model management request, it passes it to the network data models logic for processing, and receives information back.

  When the REST service receives an analysis request, it passes it to the contraction hierarchies models for processing, and receives information back.
- The NDM models logic updates the models as needed, communicates information back to the contraction hierarchies REST service, and updates the file(s) containing the relevant contraction hierarchies models.
- When any updates to contraction hierarchies models are received, the relevant model files are updated.

Several components of the architecture are described in more detail later in this topic.

Contraction Hierarchies REST API Examples

RESTful JSON **request** for creating a contraction hierarchies network model under a preconfigured directory:

```
{"createNetworkRequest":
    {
        "chName":"my_example",
        "networkName":"MY_NETWORK_NAME",
        "dbUrl":"jdbc:oracle:thin:@<host>:<port>:<sid>",
        "dbUser":"my_username",
        "dbPassword":"my_password",
        "processGeometry":true,
        "processTurnRestrictions":false,
        "primaryLinkCostColumn":"LENGTH",
        "primaryCostUnit":"meter",
        "primaryCostScaleFactor":10,
        "secondaryLinkCostColumns":["LENGTH/S"],
        "secondaryCostUnits":["second"],
        "secondaryCostScaleFactors":[10]
    }
}
```

RESTful JSON **request** for loading a contraction hierarchies network model into memory:

```
{"loadNetworkRequest":
    {
        "chName":"my_example",
        "networkName":"MY_NETWORK_NAME",
        "considerTurnRestrictions":false
    }
}
```

RESTful JSON **request** for a shortest path analysis:

```
{
    "chName":"my_example",
    "shortestPathRequest":{
        "startPoints" : { "pointOnNet" : [
            { "linkId" : 238135, "percentage" : 0.28 }
        ] },
        "endPoints" : { "pointOnNet" : [
            { "linkId" : 261315, "percentage" : 0.93 }
        ] },
        "geometry":true
    }
}
```

RESTful JSON **response** for a shortest path analysis (reformatted for readability):

```
{
  "shortestPathResponse" : {
    "cost" : 2906.6,
    "geometry" : "{\"type\":\"LineString\",\"coordinates\":[[-74.00501,40.70583],
```

```
[-74.00457,40.70549],[-74.00447,40.70541],[-74.00418,40.70559],[-74.00386,40.70579],
[-74.00361,40.70595],[-74.00346,40.70605],[-74.00335,40.70611],[-74.00318,40.70621],
[-74.00231,40.7067],[-74.00274,40.70722],[-74.00311,40.70767],[-74.00336,40.708],
[-74.00345,40.70808],[-74.00407,40.70745],[-74.00412,40.70757],[-74.00433,40.70783],
[-74.00477,40.70841],[-74.00505,40.70876],[-74.00513,40.70885],[-74.00524,40.70893],
[-74.00532,40.70899],[-74.00547,40.70909],[-74.00643,40.70956],[-74.00705,40.70987],
[-74.00774,40.71022],[-74.00906,40.71089],[-74.01046,40.71153],[-74.01013,40.71209],
[-74.00967,40.71274],[-74.00927,40.71326],[-74.00902,40.71359],[-74.00885,40.71381],
[-74.0084,40.71437],[-74.00795,40.71494],[-74.00755,40.71544],[-74.00882,40.71602],
[-74.0092,40.71619],[-74.00911,40.71692],[-74.00906,40.71726],[-74.009,40.7176],
[-74.00894,40.71793],[-74.00888,40.71827],[-74.00882,40.71864],[-74.00875,40.71903],
[-74.0087,40.7193],[-74.00858,40.71996],[-74.00847,40.72065],[-74.00842,40.72089],
[-74.00837,40.7212],[-74.00834,40.72133],[-74.00823,40.72198],[-74.00812,40.72264],
[-74.00801,40.72328],[-74.00795,40.72365],[-74.00793,40.72376],[-74.00786,40.72382],
[-74.00777,40.72388],[-74.00773,40.72392],[-74.00771,40.72393],[-74.00745,40.72412],
[-74.00736,40.72417],[-74.00728,40.72424],[-74.00723,40.72429],[-74.0071,40.72441],
[-74.00703,40.7245]]}",
    "linkIds" : [ 238135, 69834, 69856, 187992, 39327, 39328, 18867, 189084, 189085,
189086, 189087, 142716, 142717, 142718, 142719, 142720, 193362, 193363, 54588, 54589,
54657, 153376, 68912, 61331, 61332, 177603, 177604, 177605, 177606, 177607, 106801,
96723, 96724, 96725, 96726, 65028, 176816, 176817, 65012, 65013, 65014, 65015, 261314,
261315 ],
    "nodeIds" : [ 42427254, 42427256, 42440356, 3350498747, 42452620, 42457292,
42444271, 42440270, 42440271, 673008453, 42440278, 42440280, 42440282, 42440284,
42440287, 42428385, 42440290, 42453943, 42453952, 42430004, 42429562, 42449597,
42431611, 42445356, 42445357, 42436322, 42430571, 42430529, 42429833, 42436326,
42436327, 42436330, 42436333, 42436335, 42436336, 42424610, 1104165608, 42436308,
42424408, 42436340, 42436014, 4142105822, 42424619, 42424630, 42423514 ],
    "startIndex" : 0,
    "startPercentage" : 0.28,
    "endIndex" : 43,
    "endPercentage" : 0.93
  },
  "unit" : "meter"
}
```


Contraction Hierarchies Model Management

You can create a contraction hierarchies model from any Spatial network data model in the database. Each model is stored as a set of files under a specific directory.

After creating the model, you can load it into memory and perform analysis.

Contraction Hierarchies Analysis

Several contraction hierarchies analysis functions are supported, including:

- Shortest Path: the shortest path between two points
- Multi-Stop Shortest Path: the shortest paths between a sequence of points
- Cost Matrix: the costs of N start points to M end points in a matrix form (N by M)
- Traveling Salesperson Path: the shortest path that visits all given points
- Alternative Paths: K alternative paths with no more than a specific percentage of path overlapping
- Secondary Cost: get all available secondary costs of a path

Additional analysis information such as path geometry can also be obtained using the REST API.

Using the Contraction Hierarchies REST API

You can use the following steps to develop your application with Contraction Hierarchies REST API:

1. Deploy the contraction hierarchies ear file (chrest.ear) to any J2EE container.

   This deploys the contraction hierarchies service to handle the REST API.
2. Create a contraction hierarchies model from a Spatial Network Data Model model in the database, and store the contraction hierarchies model on the file system.
3. Perform analysis by loading a contraction hierarchies model into memory.

The `chrest.ear` file also comes with a demo that shows how to use the REST API. To use this demo, go to this URL:

`https://<host>:<port>/chrest/`

The **Contraction Hierarchies Web Demo** main page lets you query logical and spatial network data stored in files, to perform analysis functions of the Spatial Network Data Model.

The available links in this demo include:

- Documentation & Schema
- Contraction Hierarchies Network Analysis
- Visualization Demo
- Configuration

REST API Reference Information

For reference information about the REST API for contraction hierarchies support, see REST API for Oracle Spatial.

5.13 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 Network Data Model and Network Feature Editing (NFE) Model Metadata Views.

- 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.
- Network Layer Tables
- Feature Layer Tables
- Network Feature Editing (NFE) Model Tables

5.13.1 Network Layer Tables

The metadata tables in this section are not related to feature modeling.

- Node Table
- Link Table
- Path Table
- Path-Link Table
- Subpath Table
- Partition Table
- Partition BLOB Table
- Connected Component Table
- Node Hierarchy Table (Optional)
- Node Level Table (Optional)
- Network Buffer Schema

5.13.1.1 Node Table

Each network has a node table that can contain the columns described in Table 5-6. (The specific columns depend on the network type and whether the network is hierarchical or not.)

Table 5-6 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 Partition Table.)

Table 5-6 (Cont.) Node Table Columns

Column Name	Data Type	Description
<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. (Network Hierarchy explains network hierarchy.)
PARENT_NODE_ID	NUMBER	For hierarchical networks only: node ID of the parent node of this node. (Network Hierarchy explains network hierarchy.)

5.13.1.2 Link Table

Each network has a link table that contains the columns described in Table 5-7.

Table 5-7 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

Table 5-7 (Cont.) Link Table Columns

Column Name	Data Type	Description
<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. (Network Hierarchy 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).

5.13.1.3 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 Path-Link Table). Each path table contains the columns described in Table 5-8.

Table 5-8 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

Table 5-8 (Cont.) Path Table Columns

Column Name	Data Type	Description
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.

5.13.1.4 Path-Link Table

For each path table (described in Path Table), 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 5-9.

Table 5-9 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.

5.13.1.5 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 Network Data Model Concepts. The subpath table contains the columns described in Table 5-10.

Table 5-10 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 Network Data Model Concepts, the START_LINK_INDEX is 0, and the START_PERCENTAGE is 0.65.
END_LINK_INDEX	NUMBER	Link ID of the link used to define the end of the subpath. For example, in Figure 5-2 in Network Data Model Concepts, the END_LINK_INDEX is 6, and the END_PERCENTAGE is 0.5.
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 Network Data Model Concepts, the START_LINK_INDEX is 0, and the START_PERCENTAGE is 0.65. ("Percentage" values in this case are expressed as between 0 and 1.0, so 0.65 is 65 percent.)

Table 5-10 (Cont.) Subpath Table Columns

Column Name	Data Type	Description
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 Network Data Model Concepts, the END_LINK_INDEX is 6, and the END_PERCENTAGE is 0.5. ("Percentage" values in this case are expressed as between 0 and 1.0, so 0.5 is 50 percent.)
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.

5.13.1.6 Partition Table

Each partitioned network has a partition table. For information about partitioned networks, see Network Analysis Using Load on Demand. Each partition table contains the columns described in Table 5-11.

Table 5-11 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.
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.)

5.13.1.7 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 Network Analysis Using Load on Demand.

> **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 5-12.

Table 5-12 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).

Table 5-12 (Cont.) Partition BLOB Table Columns

Column Name	Data Type	Description
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.

5.13.1.8 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 Precomputed Analysis Results.

Each connected component table contains the columns described in Table 5-13.

Table 5-13 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.

5.13.1.9 Node Hierarchy Table (Optional)

Each network can have a node hierarchy table, which stores parent-child relationships if the network has a hierarchy (explained in Network Hierarchy).

If the node hierarchy table exists, it contains the columns described in Table 5-14.

Table 5-14 Node Hierarchy Table Columns

Column Name	Data Type	Description
PARENT_ID	NUMBER	Parent node ID, that is, the node ID in the parent network, or the group, cluster, or partition ID in the child network.
CHILD_ID	NUMBER	Child ID, that is, the node ID in the child network.
LINK_LEVEL	NUMBER	Link level on which the parent-child relationship is defined. A network on a higher link level is a subnetwork of that on a lower link level. A network on link level *n* consists of only links with link level greater than or equal to *n*.

5.13.1.10 Node Level Table (Optional)

Each network can have a node level table, which stores information on the maximum link level for each higher level node (that is, a node whose maximum link level is greater than 1). The node level table is only useful for multilevel networks; it makes loading partitions from node and link tables faster.

If the node level table exists, it contains the columns described in Table 5-15.

Table 5-15 Node Level Table Columns

Column Name	Data Type	Description
NODE_ID	NUMBER	ID of a node whose maximum link level is greater than 1
LINK_LEVEL	NUMBER	Maximum link level to which the node is connected.

5.13.1.11 Network Buffer Schema

Each network buffer consists of user tables that describe its metadata, cost, and coverage information. These network buffer tables have a table prefix which is user provided. They also contain a set of predefined postfixes.

When you write a network buffer to the database for the first time, the `writeNetworkBuffer()` function will automatically create the following tables. Note, you can also append additional network buffers to the existing network buffer tables.

- <prefix>_NBR$
- <prefix>_NBCL$
- <prefix>_NBCN$
- <prefix>_NBN$
- <prefix>_NBL$

5.13.1.11.1 <prefix>_NBR$

The `<prefix>_NBR$` table provides information of a network buffer and its buffer cost and direction.

Table 5-16 `<prefix>_NBR$` Table Columns

Column Name	Data Type	Description
BUFFER_ID	NUMBER	Network buffer ID
RADIUS	NUMBER	Maximum cost the network buffer covers
DIRECTION	VARCHAR2(1)	Direction of the network buffer. The supported values are: • `F`: Forward • `B`: Backward

5.13.1.11.2 <prefix>_NBCL$

The `<prefix>_NBCL$` table provides information of a network buffer and its buffer center location on a link.

Table 5-17 `<prefix>_NBCL$` Table Columns

Column Name	Data Type	Description
BUFFER_ID	NUMBER	Network buffer ID
LINK_ID	NUMBER	Link ID located by the buffer center
PERCENTAGE	NUMBER	Link percentage located by the buffer center

5.13.1.11.3 <prefix>_NBCN$

The `<prefix>_NBCN$` table provides information of a network buffer and its buffer center location on a node.

Table 5-18 `<prefix>_NBCN$` Table Columns

Column Name	Data Type	Description
BUFFER_ID	NUMBER	Network buffer ID
NODE_ID	NUMBER	Node ID located by the buffer center

5.13.1.11.4 <prefix>_NBN$

The `<prefix>_NBN$` table provides information of a network buffer and the nodes it covers.

Table 5-19 <prefix>_NBN$ Table Columns

Column Name	Data Type	Description
BUFFER_ID	NUMBER	Network buffer ID
NODE_ID	NUMBER	Node ID covered by the network buffer
COST	NUMBER	Minimum cost between the buffer center and the covered node

5.13.1.11.5 <prefix>_NBL$

The <prefix>_NBL$ table provides information of a network buffer and the links it covers.

Table 5-20 <prefix>_NBL$ Table Columns

Column Name	Data Type	Description
BUFFER_ID	NUMBER	Network buffer ID
LINK_ID	NUMBER	Link ID covered by the network buffer
PREV_LINK_ID	NUMBER	Previous shortest path link ID of the covered link
START_PERCENTAGE	NUMBER	Start percentage of the covered link
END_PERCENTAGE	NUMBER	End percentage of the covered link
START_COST	NUMBER	Minimum cost between the buffer center and the start of the covered link
END_COST	NUMBER	Minimum cost between the buffer center and the end of the covered link

5.13.2 Feature Layer Tables

The tables in this section are related to feature modeling (see Feature Modeling). These tables are used to describe each registered feature layer.

In most applications, the tables containing feature entity information, feature to network relationships, or feature hierarchy relationships already exist, although the table schema may be different from that of the NDM tables. In such cases, you can create views to map the existing table schema to the NDM table schema.

- Feature Table
- Feature Element Relationships Table
- Feature Hierarchy Table

5.13.2.1 Feature Table

A feature table contains feature entity information. Each feature table must contain a FEATURE_ID column. Other feature attributes that are potentially useful during feature analysis can be registered as user data.

Each feature table contains the columns described in Table 5-21.

Table 5-21 Feature Table Columns

Column Name	Data Type	Description
FEATURE_ID	NUMBER	ID of the feature.
(Additional columns as needed)	(As appropriate)	(Other feature attributes that are potentially useful during feature analysis)

5.13.2.2 Feature Element Relationships Table

The feature element relationships table contains information about the relationships between feature elements and network elements (nodes and links).

The feature element relationships table contains the columns described in Table 5-22.

Table 5-22 Feature Element Relationships Table Columns

Column Name	Data Type	Description
FEATURE_ID	NUMBER	ID of the feature.
FEAT_ELEM_TYPE	NUMBER	Feature element type. One of the following: 1 for SDO_NET. FEAT_ELEM_TYPE_PON (point on node) ; 2 for SDO_NET. FEAT_ELEM_TYPE_POL (point on link); 3 for SDO_NET. FEAT_ELEM_TYPE_LINE (line).
NET_ELEM_ID	NUMBER	ID of the network element (node or link) associated with this feature element.
START_PERCENTAGE	NUMBER	Start percentage along NET_ELEM_ID for this feature element (ignored for point on node feature elements).
END_PERCENTAGE	NUMBER	End percentage along NET_ELEM_ID for this feature element (ignored for point on node and point on line feature elements).
SEQUENCE	NUMBER	Sequence number of the feature element.

5.13.2.3 Feature Hierarchy Table

The feature hierarchy table contains feature hierarchy information. Child features can belong to different feature layers.

The feature hierarchy table contains the columns described in Table 5-23.

Table 5-23 Feature Hierarchy Table Columns

Column Name	Data Type	Description
PARENT_ID	NUMBER	ID of the parent feature.
CHILD_LAYER_ID	NUMBER	Feature layer ID of the child feature.
CHILD_ID	NUMBER	ID of the child feature.
SEQUENCE	NUMBER	Sequence number of the child feature.

5.13.3 Network Feature Editing (NFE) Model Tables

The tables in this section are related to feature manipulation using Network Feature Editing (NFE) (see Feature Modeling Using Network Feature Editing (NFE)). These tables are used to describe each NFE model.

In most cases, these tables are created or updated automatically when a new NFE Model is created using the PL/SQL function SDO_NFE.CREATE_MODEL_STRUCTURE or the Java API; but you can also manually create the tables or views and register them to the desired model.

- Automatically Created Points Default Attributes Table
  The automatically created points default attributes table contains information about the default values for the attributes of automatically created points as the result of the execution of a connectivity line-line rule.
- Connectivity Line-Line Rules Table
  The connectivity line-line rules table contains the information about the definition of connectivity line-line rules applicable to features involved in an NFE model.
- Connectivity Line-Point Rules Table
  The connectivity line-point rules table contains the information about the definition of connectivity line-point rules applicable to features involved in an NFE model.
- Feature Class Table
  A feature class is related to a feature. A feature class table contains the records of all feature classes from a feature layer, and must contain the columns described in the following table.
- Feature Class Attributes Constraints Table
  The feature class attributes constraints table contains information about user restrictions over feature class attributes, which are the same attributes defined for the feature layer that the feature class belongs to.
- Feature Class Default Predefined Connected Points Table
  The feature class default predefined connected points table contains the information about such default connected point features along a line feature described by a feature class.
- Feature Class Relationship Table
  The feature class relationship table contains information about the relationship between features and feature classes, that is, which feature belongs to which feature class.
- Feature Rule Relationship Table
  The feature rule relationship table contains information that relates each feature element involved in a rule, with the rule that caused its generation or connection.
- Feature User Data Table
  The feature user data table contains the information of feature class attributes of a catalog type.

- Feature User Data Catalog Table
  You can assign a value to a feature class attribute from a catalog; this is, that the attribute type from a feature class can be a catalog type. The feature user data catalog table keeps a register of the catalogs that can be used for that purpose.
- Feature User Data Catalog Values Table
  The feature user data catalog values table contains the list of values held by catalogs defined in feature user data catalog table.
- Point Cardinality Rules Table
  The point cardinality rules table contains the configuration of the maximum and minimum inbound and outbound connections that a specific point feature must support in an NFE model.
- Rule Decision Handlers Table
  The rule decision handlers table contains information about the names of the Java class and/or PL/SQL procedures to be executed as decision handlers when a connectivity rule (line-line or line-point) is executed.
- Rule Instance Table
  The rule instance table contains information about rule instances generated by the application of either line-line or line-point connectivity rules in an NFE model.

5.13.3.1 Automatically Created Points Default Attributes Table

The automatically created points default attributes table contains information about the default values for the attributes of automatically created points as the result of the execution of a connectivity line-line rule.

. The following table describes the columns of the automatically created points default attributes table. When an NFE model is created, the name by default given to this kind of table is *POINT_ATTR_DEF_[model_id]$*.

Table 5-24 Automatically Created Points Default Attributes Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Default value identifier.
LINE_LINE_RULE_ID	NUMBER	Foreign key. An existing line-line rule identifier. Refers to the ID column in the line-line rules table.
ATTRIBUTE_NAME	VARCHAR(50)	A point feature class attribute name. Refers to the DATA_NAME column in the xxx_SDO_NETWORK_USER_DATA view.
DEFAULT_VALUE	VARCHAR(100)	Default value for the point's attribute.

5.13.3.2 Connectivity Line-Line Rules Table

The connectivity line-line rules table contains the information about the definition of connectivity line-line rules applicable to features involved in an NFE model.

This definition depicts how two line features (described in a line-point rule) must interact in order to be connected each other.

The following table describes the columns of a connectivity line-line rules table When an NFE model is created, the name by default given to this kind of table is *LINE_LINE_RULE_[model_id]$*.

Table 5-25 Connectivity Line-Line Rules Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Line-line rule identifier.
LINE_POINT_1_ID	NUMBER	Foreign key. An existing line-point rule identifier. Refers to the ID column from a line point rules table.
LINE_POINT_2_ID	NUMBER	Foreign key. An existing line-point rule identifier. Refers to the ID column from a line point rules table.
INTERACTION	NUMBER	Interaction type between the two lines. Posible values: 1 = Crosses, 2 = Touches end points, 3 = Touches middle points, 4 = Touches any point, 5 = Any interact
DECISION_HANDLER_ID	NUMBER	Foreign key. The ID of the decision handler (if any) associated with the rule. Refrs to the ID column in the rules decision handler table.
CREATE_POINT	VARCHAR2(1)	Specifies whether the connection point should be created automatically or not when the interaction among feature lines occurs. Possible values: 'Y', 'N'.

5.13.3.3 Connectivity Line-Point Rules Table

The connectivity line-point rules table contains the information about the definition of connectivity line-point rules applicable to features involved in an NFE model.

This definition includes specifications of feature layer, feature class, and feature attributes conditions for both line and point interacting features that can be connected.

The following table describes the columns of a connectivity line-point rules table When an NFE model is created, the name by default given to this kind of table is *LINE_POINT_RULE_[model_id]$*.

Table 5-26 Connectivity Line-Point Rules Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Line-point rule identifier.
LINE_FEATURE_LAYER_ID	NUMBER	Feature layer identifier to which the line belongs. Refers to the xxx_SDO_NETWORK_FEATURE table. A value of -1 means all feature layers.
LINE_FEATURE_CLASS_ID	NUMBER	Feature class identifier for the line to which the rule will be applied. A value of -1 means all feature classes.
LINE_ATTRIBUTE_CONDITION	VARCHAR2(200)	Condition to be evaluated in the rule over the feature line attributes. Example: `MATERIAL = 'IRON'`
POINT_FEATURE_LAYER_ID	NUMBER	Feature layer identifier to which allowed connection point belongs. Refers to the xxx_SDO_NETWORK_FEATURE table.
POINT_FEATURE_CLASS_ID	NUMBER	Feature class identifier for the allowed connection point. A value of -1 means all feature classes.
DECISION_HANDLER_ID	NUMBER	Foreign key. The ID of the decision handler (if any) associated with the rule. Refers to the ID column from the rule decision handler table.
MAX_IN_CONN	NUMBER	Maximum number of incoming lines that can be connected to the point.

Table 5-26 (Cont.) Connectivity Line-Point Rules Table Columns

Column Name	Data Type	Description
MAX_OUT_CONN	NUMBER	Maximum number of outgoing lines that can be connected to the point.
MIN_IN_CONN	NUMBER	Minimum number of incoming lines that must be connected to the point
MIN_OUT_CONN	NUMBER	Minimum number of outgoing lines that must be connected to the point
SOURCE	NUMBER	Indicates whether the rule was created by the user or by a line-line rule. Possible values are: 1 = User, 2=Line-Line Rule. The default value is 1.
ID	NUMBER	Primary key. Line-line rule identifier.

5.13.3.4 Feature Class Table

A feature class is related to a feature. A feature class table contains the records of all feature classes from a feature layer, and must contain the columns described in the following table.

When an NFE model is created, the name by default given to this kind of table is *FT_CLASS_[model_id]$*.

Table 5-27 Feature Class Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Feature class identifier
NAME	VARCHAR2(50)	Feature class name
FEATURE_LAYER_ID	NUMBER	Feature layer identifier. Reference to xxx_SDO_NETWORK_FEATURE table.
SHAPE	NUMBER	Feature class shape type. Possible values: SDO_NFE.FT_CLASS_POINT (1), SDO_NFE.FT_CLASS_SIMPLE_LINE (2), SDO_NFE.FT_CLASS_COMPLEX_LINE (3), SDO_NFE.FT_CLASS_PATH (4).
STYLE	VARCHAR2(50)	Feature class style. Reference to xxx_SDO_STYLES table.

5.13.3.5 Feature Class Attributes Constraints Table

The feature class attributes constraints table contains information about user restrictions over feature class attributes, which are the same attributes defined for the feature layer that the feature class belongs to.

This table must contain the columns described in the following table. When an NFE model is created, the name by default given to this kind of table is *FT_CLASS_ATTR_CSTR_[model_id]$*.

Table 5-28 Feature Class Attributes Constraints Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Constraint identifier.
FEATURE_CLASS_ID	NUMBER	Foreign key. Feature class identifier. Reference to feature class table.
ATTRIBUTE_NAME	VARCHAR2(50)	Name of the attribute to be restricted. Reference to DATA_NAME column from the xxx_SDO_NETWORK_USER_DATA view.
DEFAULT_VALUE	VARCHAR2(100)	Default value for the specified attribute (*attribute_name*).
VISIBLE	VARCHAR2(1)	Specifies whether the attribute must be visible or not.
EDITABLE	VARCHAR2(1)	Specifies whether the attribute must be editable or not.

5.13.3.6 Feature Class Default Predefined Connected Points Table

The feature class default predefined connected points table contains the information about such default connected point features along a line feature described by a feature class.

A Line Feature Class can be defined with points connected by default. The following table describes the columns of a feature class default predefined connected points table. When an NFE model is created, the name by default given to this kind of table is *FT_CLASS_DEF_CON_PT_[model_id]$*.

Table 5-29 Feature Class Default Predefined Connected Points Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Default connected point identifier.
LINE_FEATURE_CLASS_ID	NUMBER	Foreign key. Line feature class identifier. Refers to the ID column in the feature class table.
POINT_FEATURE_CLASS_ID	NUMBER	Foreign key. Point feature class identifier. Refers to the ID column in the feature class table.
POSITION_PERCENTAGE	DECIMAL	Percentage of point's location along the line.

5.13.3.7 Feature Class Relationship Table

The feature class relationship table contains information about the relationship between features and feature classes, that is, which feature belongs to which feature class.

The feature class relationship table contains the columns described in the following table. When an NFE model is created, the name by default given to this kind of table is *FT_CLASS_REL_[model_id]$*.

Table 5-30 Feature Class Relationship Table Columns

Column Name	Data Type	Description
FEATURE_ID	NUMBER	Primary key. Feature identifier. Reference to feature table.
FEATURE_CLASS_ID	NUMBER	Foreign key. Feature class identifier. Reference to feature class table.

5.13.3.8 Feature Rule Relationship Table

The feature rule relationship table contains information that relates each feature element involved in a rule, with the rule that caused its generation or connection.

The feature rule relationship table contains the columns described in the following table. When an NFE model is created, the name by default given to this kind of table is *FT_RULE_REL_[model_id]$*.

Table 5-31 Feature Rule Relationship Table Columns

Column Name	Data Type	Description
RULE_INSTANCE_ID	NUMBER	Primary key. Rule instance identifier.
FEATURE_LAYER_ID	NUMBER	Primary key. Feature layer identifier. Refers to the *xxx_SDO_NETWORK_FEATURE* table.
FEATURE_ID	NUMBER	Primary key. Feature identifier. Refers to the FEATURE_ID column from the feature table.
NET_ELEM_ID	NUMBER	Primary key. Network element identifier. Refers to the NET_ELEM_ID column from the feature element relationships table.
FEAT_ELEM_TYPE	NUMBER	Primary key. Feature element type. Refers to the FEAT_ELEM_TYPE column from the feature element relationships table.

5.13.3.9 Feature User Data Table

The feature user data table contains the information of feature class attributes of a catalog type.

The feature user data table is an extension of xxx_SDO_NETWORK_USER_DATA view. The following table describes the columns of a feature user data table. When an NFE model is created, the name by default given to this kind of table is *FT_USR_DATA_[model_id]$*.

Table 5-32 Feature User Data Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Feature class attribute identifier.
FEATURE_LAYER_ID	NUMBER	Feature layer identifier to which the feature class attribute belongs.
ATTRIBUTE_NAME	VARCHAR2(50)	Attribute name. Refers to the DATA_NAME column of the xxx_SDO_NETWORK_USER_DATA view.
CATALOG_ID	NUMBER	Foreign key. Catalog identifier. Refers to the ID column from the feature user data catalog table.

5.13.3.10 Feature User Data Catalog Table

You can assign a value to a feature class attribute from a catalog; this is, that the attribute type from a feature class can be a catalog type. The feature user data catalog table keeps a register of the catalogs that can be used for that purpose.

The following table describes the columns that a feature user data catalog table must contain. When an NFE model is created, the name by default given to this kind of table is *FT_USR_DATA_CATLG_[model_id]$*.

Table 5-33 Feature User Data Catalog Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Catalog identifier.
NAME	VARCHAR2(200)	Catalog name.
DATA_TYPE	VARCHAR2(12)	Catalog data type (for example, Number or Varchar2).

5.13.3.11 Feature User Data Catalog Values Table

The feature user data catalog values table contains the list of values held by catalogs defined in feature user data catalog table.

The following table describes the columns that a feature user data catalog values table must contain. When an NFE model is created, the name by default given to this kind of table is *FT_USR_DATA_CVAL_[model_id]$*.

Table 5-34 Feature User Data Catalog Values Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Catalog entry identifier.
CATALOG_ID	NUMBER	Foreign key. Catalog identifier. Reference to the feature user data catalog table.
VALUE	VARCHAR2(12)	Catalog entry.

5.13.3.12 Point Cardinality Rules Table

The point cardinality rules table contains the configuration of the maximum and minimum inbound and outbound connections that a specific point feature must support in an NFE model.

The point cardinality rules table contains the columns described in the following table. When an NFE model is created, the name by default given to this kind of table is *POINT_CARD_RULE_[model_id]$*.

Table 5-35 Point Cardinality Rules Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Cardinality rule identifier
FEATURE_LAYER_ID	NUMBER	Feature layer of the point
FEATURE_CLASS_ID	NUMBER	Feature class of the point. The shape of the class must be of type POINT
MAX_IN_CONN	NUMBER	Maximum number of incoming lines that can be connected to the point
MAX_OUT_CONN	NUMBER	Maximum number of outgoing lines that can be connected to the point

5.13.3.13 Rule Decision Handlers Table

The rule decision handlers table contains information about the names of the Java class and/or PL/SQL procedures to be executed as decision handlers when a connectivity rule (line-line or line-point) is executed.

The rule decision handlers table contains the columns described in the following table. When an NFE model is created, the name by default given to this kind of table is *RULE_DEC_HANDLER_[model_id]$*.

Table 5-36 Rule Decision Handlers Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Decision handler identifier.
TYPE	VARCHAR2(1)	`SDO_NFE.RULE_TYPE_LINE_LINE` for line-line rule handler, or `SDO_NFE.RULE_TYPE_LINE_POINT` for line-point rule handler.
CLASS_FQNAME	VARCHAR(100)	Handler class fully qualified name. This class must be an implementation of *oracle.spatial.network.nfe.model.rule.DecisionHandler*. The implementation class must be accessible from the classpath of the application that is running the rule engine in the Java API.

Table 5-36 (Cont.) Rule Decision Handlers Table Columns

Column Name	Data Type	Description
PLSQL_SP_GET_CONN_GROUPS	VARCHAR2(50)	Name of the PL/SQL stored procedure used to obtain the different groups of elements participating in an intersection that can be connected through a rule. The name must include the package name. For a handler of type 'L', the default value is ***SDO_NFE.get_ll_conn_intersections***, and the parameters must be: model_id IN NUMBER - Model identifier ll_rule_id IN NUMBER - Line-Line rule identifier *interaction_grp* IN OUT SDO_INTERACTION - Group of lines and points that are interacting rule_lhs_lines_indexes IN dbms_sql.NUMBER_TABLE - Among the line features in the interacting group, indexes of the lines that specifically match the LEFT hand side of the line-line rule. *rule_rhs_lines_indexes* IN dbms_sql.NUMBER_TABLE - Among the line features in the interacting group, indexes of the lines that specifically match the RIGHT hand side of the line-line rule. *rule_points_indexes* IN dbms_sql.NUMBER_TABLE - Among the point features in the interacting group, indexes of the points that specifically match the point feature specification in the line-line rule. These points are the ones to be considered in the conformation of connectable groups. Refer to SDO_NFE.GET_LL_CONN_INTERSECTIONS function documentation in Section 7 for more details. For a handler of type 'P', the default value is ***SDO_NFE.get_lp_conn_intersections***, and the parameters must be: model_id IN NUMBER - Model identifier *lp_rule_id* IN NUMBER - Line-point rule identifier *interaction_grp* IN OUT SDO_INTERACTION - Group of lines and points that are interacting. *rule_lines_indexes* IN dbms_sql.NUMBER_TABLE - Among the line features in the interacting group, indexes of the LINES that specifically match the line-point rule. These lines will be considered in the conformation of connectable groups. *rule_points_indexes* IN dbms_sql.NUMBER_TABLE - Among the point features in the interacting group, indexes of the POINTS that specifically match the point feature specification in the line-point rule. These points are the ones to be considered in the conformation of connectable groups. Refer to SDO_NFE.GET_LP_CONN_INTERSECTIONSfunction documentation in Section 7 for more details.
PLSQL_SP_GET_CONN_POINT	VARCHAR2(50)	Name of the PL/SQL stored procedure used to determine the geometry of the connection point between the participating elements in an intersection. The name must include the package name. Default value is ***SDO_NFE.get_connection_point_geom***. It must accept only one parameter as an object of type SDO_INTERACTION which is the group of interacting features. Refer to SDO_NFE.GET_CONNECTION_POINT_GEOM function documentation for more details.

For using customized rule decision handlers in the Java API, the decision handler Java class to be used must be specified in CLASS_FQNAME. For using customized rule decision

handlers in PL/SQL, the subprogram for calculating the connectable groups of features must be specified in PLSQL_SP_GET_CONN_GROUPS, and the subprogram for calculating the geometry of the connection point among the features to be connected must be specified in PLSQL_SP_GET_CONN_POINT.

5.13.3.14 Rule Instance Table

The rule instance table contains information about rule instances generated by the application of either line-line or line-point connectivity rules in an NFE model.

This definition includes the identifier for the rule instance, the identifier of the rule that generated the instance, and the type of the rule.

The following table describes the columns of a rule instance table When an NFE model is created, the name by default given to this kind of table is *RULE_INSTANCE_[model_id]$*.

Table 5-37 Rule Instance Table Columns

Column Name	Data Type	Description
ID	NUMBER	Primary key. Rule instance identifier
RULE_ID	NUMBER	Rule identifier. Refers to ID column in either the line-line rules table or line-point rules table.
RULE_TYPE	VARCHAR(1)	Must be `SDO_NFE.RULE_TYPE_LINE_LINE` or `SDO_NFE.RULE_TYPE_LINE_POINT`.

5.14 Network Data Model and Network Feature Editing (NFE) Model Metadata Views

Two sets of network metadata views can be created for each schema (user): xxx_SDO_NETWORK_xxxxxx and xxx_SDO_NFE_MODEL_xxxxxx, where the initial *xxx* can be USER or ALL. These views are created, as needed, by Spatial.

The xxx_SDO_NFE_MODEL_xxxxxx metadata views relate to Feature Modeling Using Network Feature Editing (NFE).

- xxx_SDO_NETWORK_METADATA Views
- xxx_SDO_NETWORK_CONSTRAINTS Views
- xxx_SDO_NETWORK_USER_DATA Views
- xxx_SDO_NETWORK_FEATURE Views
- xxx_SDO_NFE_MODEL_FTLAYER_REL Views
- xxx_SDO_NFE_MODEL_METADATA Views
- xxx_SDO_NFE_MODEL_WORKSPACE Views

5.14.1 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 5-38, 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 5-38 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
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 Network Hierarchy for information about network hierarchy.)
NO_OF_PARTITIONS	NUMBER	(Not currently used)

Table 5-38 (Cont.) Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
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 Node Table.)
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 Link Table.)
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`.
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)

Table 5-38 (Cont.) Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
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 Path Table.)
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 Path-Link Table.)
SUBPATH_TABLE_NAME	VARCHAR2(32)	Contains the name of an optional table containing information about subpaths. (The subpath table is described in Subpath Table.)
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 Partition Table.
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 Partition BLOB Table.
COMPONENT_TABLE_NAME	VARCHAR2(32)	The name of the table containing information about precomputed connected components, as explained in Precomputed Analysis Results. (The connected component table is described in Connected Component Table.)

Table 5-38 (Cont.) Columns in the xxx_SDO_NETWORK_METADATA Views

Column Name	Data Type	Purpose
NODE_LEVEL_TABLE_NAME	VARCHAR2(32)	The name of the table containing information about node levels in a multilevel 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)
CHILD_NETWORK	VARCHAR2(32)	Name of the child network, if a network hierarchy is involved.
HIERARCHY_TABLE_NAME	VARCHAR2(32)	Name of the hierarchy table, if a network hierarchy is involved.

5.14.2 xxx_SDO_NETWORK_CONSTRAINTS Views

The following views contain information about network constraints (described in Network Constraints):

- 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 5-39, 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 5-39 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

Table 5-39 (Cont.) Columns in the xxx_SDO_NETWORK_CONSTRAINTS Views

Column Name	Data Type	Purpose
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

5.14.3 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 5-39, 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 5-40 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_USER_DATA view only)
NETWORK	VARCHAR2(32)	Name of the network
TABLE_TYPE	VARCHAR2(12)	Type of the table containing the user-defined data: `NODE`, `LINK`, `PATH`, or `SUBPATH` If feature user data is registered through the xxx_SDO_USER_NETWORK_USER_DATA views. TABLE_TYPE is set to the name of the feature table.
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`

Table 5-40 (Cont.) Columns in the xxx_SDO_NETWORK_USER_DATA Views

Column Name	Data Type	Purpose
DATA_LENGTH	NUMBER(38)	If DATA_TYPE is `VARCHAR2`, the length of the user-defined data
CATEGORY_ID	NUMBER	User data category ID (non-negative number, default 0). The category ID allows for grouping of user data for different applications. Category 0 is reserved for general-purpose user data that is useful for all applications. User data for different purposes can be grouped into different categories, so that during network analysis, only the relevant user data categories are loaded into memory, reducing memory consumption at runtime. For example, for a road network, category 0 user data can include the speed limit and function class of links, and the x, y coordinates of nodes; trucking-related user data might belong to category 1; and traffic-related user data might belong to category 2.

To use user-defined data, you must set the USER_DEFINED_DATA column value to `Y` in the appropriate xxx_SDO_NETWORK_METADATA views (described in xxx_SDO_NETWORK_METADATA Views).

For an example of using user-defined data, see User-Defined Data Examples (PL/SQL and Java).

For user data defined through the xxx_SDO_NETWORK_USER_DATA views, the default user data I/O implementation (`LODUserDataIOSDO`) is used to access the user data during network analysis. However, some user data is not included in the node or link table, and thus cannot be registered through xxx_SDO_NETWORK_USER_DATA views. For such user data, you must provide your own implementation of the user data I/O interface. A typical way of implementing a custom user data I/O interface is to generate BLOBs corresponding to node and link user data , one BLOB for each partition, and then retrieve user data information from the BLOBs during network analysis.

You can also associate multiple categories of user-defined data (categorized user data) with a single network. For example, in a multimodal network (described in Multimodal Network and Temporal Examples), if you must associate driving-related attributes (such as speed limit) with a link in addition to the link's multimodal attributes, you can organize user-defined data in two categories: one for driving-related attributes and the other for multimodal attributes.

5.14.4 xxx_SDO_NETWORK_FEATURE Views

The following views contain information about network feature layers (described in Features and Feature Layers):

- USER_SDO_NETWORK_FEATURE contains information about all network feature layers owned by the user.
- ALL_SDO_NETWORK_FEATURE contains information about all network feature layers on which the user has SELECT permission.

The USER_SDO_NETWORK_FEATURE and ALL_SDO_NETWORK_FEATURE views contain the same columns, as shown Table 5-39, except that the USER_SDO_NETWORK_FEATURE view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5-41 Columns in the xxx_SDO_NETWORK_FEATURE Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the network feature layer (ALL_SDO_NETWORK_FEATURE view only)
NETWORK	VARCHAR2(32)	Name of the network on which the feature layer is defined
FEATURE_LAYER_NAME	VARCHAR2(32)	Name of the feature layer
FEATURE_LAYER_ID	NUMBER	ID of the feature layer (assigned by Oracle Spatial)
FEATURE_LAYER_TYPE	NUMBER	Type of the feature layer (see Table 5-2 in Features and Feature Layers)
FEATURE_TABLE_NAME	VARCHAR2(32)	Name of the feature table (see Feature Table)
RELATION_TABLE_NAME	VARCHAR2(32)	Name of the feature element relationships table, which maps feature elements with network elements (nodes and links) (see Feature Element Relationships Table)
HIERRCHY_TABLE_NAME	VARCHAR2(32)	Name of the feature hierarchy table, which defines parent-child relationships between features (see Feature Hierarchy Table)

To use user-defined data, you must set the USER_DEFINED_DATA column value to `Y` in the appropriate xxx_SDO_NETWORK_METADATA views (described in xxx_SDO_NETWORK_METADATA Views).

For an example of using user-defined data, see User-Defined Data Examples (PL/SQL and Java).

For user data defined through the xxx_SDO_NETWORK_USER_DATA views, the default user data I/O implementation (`LODUserDataIOSDO`) is used to access the user data during network analysis. However, some user data is not included in the node or link table, and thus cannot be registered through xxx_SDO_NETWORK_USER_DATA views. For such user data, you must provide your own implementation of the user data I/O interface. A typical way of implementing a custom user data I/O interface is to generate BLOBs corresponding to node and link user data , one BLOB for each partition, and then retrieve user data information from the BLOBs during network analysis.

You can also associate multiple categories of user-defined data (categorized user data) with a single network. For example, in a multimodal network (described in Multimodal Network and Temporal Examples), if you must associate driving-related attributes (such as speed limit) with a link in addition to the link's multimodal attributes, you can organize user-defined data in two categories: one for driving-related attributes and the other for multimodal attributes.

5.14.5 xxx_SDO_NFE_MODEL_FTLAYER_REL Views

The following views contain information about network feature layers related to NFE models. (This topic assumes you are familiar with the concepts explained in Feature Modeling Using Network Feature Editing (NFE).)

- USER_SDO_NFE_MODEL_FTLAYER_REL contains information about feature layers that are related to all NFE models that are owned by the user.
- ALL_SDO_NFE_MODEL_FTLAYER_REL contains information about feature layers that are related to NFE models on which the user has SELECT permission.

The USER_SDO_NFE_MODEL_FTLAYER_REL and ALL_SDO_NFE_MODEL_FTLAYER_REL views contain the same columns, as shown in Table 5-42, except that the USER_SDO_NFE_MODEL_FTLAYER_REL view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5-42 Columns in the xxx_SDO_NFE_MODEL_FTLAYER_REL Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the NFE model (ALL_SDO_NFE_MODEL_FTLAYER_REL view only)
MODEL_ID	NUMBER	Identifier of the model related to a feature layer.
FEATURE_LAYER_ID	NUMBER	Identifier of the related feature layer.
HIERARCHY_LEVEL	NUMBER	Hierarchical level for the feature layer in the model. The default is 0 (zero). Higher levels are on top of lower levels.
Z_ORDER	NUMBER	Depth of the feature layer among other feature layers in the same hierarchy level. Normally used to determine the order of drawing the feature layer elements on a canvas: the lowest order is the first to be presented.
PATH_LAYER	VARCHAR2(1)	Indicates whether the feature layer is a path generated from an analysis operation. `Y` indicates a path feature layer (generated from an analysis operation); `N` or null indicates a common feature layer.

5.14.6 xxx_SDO_NFE_MODEL_METADATA Views

The following views contain information about NFE models. (This topic assumes you are familiar with the concepts explained in Feature Modeling Using Network Feature Editing (NFE).)

- USER_SDO_NFE_MODEL_METADATA contains information about NFE models that are owned by the user.
- ALL_SDO_NFE_MODEL_METADATA contains information about NFE models on which the user has SELECT permission.

The USER_SDO_NFE_MODEL_METADATA and ALL_SDO_NFE_MODEL_METADATA views contain the same columns, as shown in Table 5-43, except that the USER_SDO_NFE_MODEL_METADATA view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5-43 Columns in the xxx_SDO_NFE_MODEL_METADATA Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the NFE model (ALL_SDO_NFE_MODEL_METADATA view only).
ID	NUMBER	Model identifier (assigned by Oracle Spatial).
NAME	VARCHAR2(100)	Name of the model.
EDITION_MODE	NUMBER	Can be `1` (SDO_NFE.FROM_SCRATCH) for models using a new network, creating features along with underlying network elements, or `2` (SDO_NFE.OVER_EXISTING_NETWORK) for models built on top of a currently existing network (in which network elements cannot be modified).
VERSIONABLE_IND	VARCHAR2(1)	Indicates whether the model will allow different versions or branches. `Y` indicates a they are allowed; `N` indicates they are not allowed.
TABLE_REG_TAB	VARCHAR2(50)	Name of the table in which the names of the Network Feature Editing (NFE) Model Tables are registered. This table is automatically created and maintained, and it has the columns described in Table 5-44.

Table 5-43 (Cont.) Columns in the xxx_SDO_NFE_MODEL_METADATA Views

Column Name	Data Type	Purpose
SEQUENCE_REG_TAB	VARCHAR2(50)	Name of the table in which the sequences associated with the model's tables are registered. This table is automatically created and maintained, and it has the columns described in xxx_SDO_NFE_MODEL_METADATA Views.
NETWORK_NAME	VARCHAR2(50)	Name of the network associated with the model.

Table 5-44 Columns in the TABLE_REG_TAB Table

Column Name	Data Type	Purpose
TABLE_TYPE	VARCHAR2(50)	Primary Key. Type of the table to be registered. Possible values: SDO_NFE.FT_CLASS, SDO_NFE.FT_CLASS_REL, SDO_NFE.FT_CLASS_ATTR_CON, DO_NFE.FT_USR_DATA, SDO_NFE.FT_USR_DATA_CAT, SDO_NFE.FT_USR_DATA_CVAL, SDO_NFE.FT_CLASS_DEF_PTS, SDO_NFE.LINE_LINE_RULES, SDO_NFE.LINE_POINT_RULES, SDO_NFE.RULE_INSTANCE, SDO_NFE.FT_RULE_REL, SDO_NFE.RULE_DEC_HANDLER, SDO_NFE.POINT_CARD_RULES, SDO_NFE.POINT_ATTR_DEF
TABLE_NAME	VARCHAR2(50)	Name assigned to the table. When you use SDO_NET.ADD_CHILD_FEATURE, by default this name is created in the form `[TABLE_TYPE]_[model_id]$`.

Table 5-45 Columns in the SEQUENCE_REG_TAB Table

Column Name	Data Type	Purpose
TABLE_NAME	VARCHAR2(50)	Primary key. Name of the table associated with the sequence..
SEQUENCE_NAME	VARCHAR2(50)	Name of the sequence..

5.14.7 xxx_SDO_NFE_MODEL_WORKSPACE Views

The following views contain information about workspaces related to NFE models. (This topic assumes you are familiar with the concepts explained in Feature Modeling Using Network Feature Editing (NFE).)

- USER_SDO_NFE_MODEL_WORKSPACE contains information about workspaces that are related to all NFE models that are owned by the user.
- ALL_SDO_NFE_MODEL_WORKSPACE contains information about workspaces that are related to NFE models on which the user has SELECT permission.

These views are automatically maintained by Spatial using the NFE Java and PL/SQL interfaces for creating and deleting Workspace Manager workspaces. You should never directly modify the content of these views.

The USER_SDO_NFE_MODEL_WORKSPACE and ALL_SDO_NFE_MODEL_WORKSPACE views contain the same columns, as shown in Table 5-42, except that the USER_SDO_NFE_MODEL_WORKSPACE view does not contain the OWNER column. (The columns are listed in their order in the view definition.)

Table 5-46 Columns in the xxx_SDO_NFE_MODEL_WORKSPACE Views

Column Name	Data Type	Purpose
OWNER	VARCHAR2(32)	Owner of the NFE model (ALL_SDO_NFE_MODEL_WORKSPACE view only)
ID	NUMBER	Identifier of the workspace. Assigned by Oracle Spatial.
MODEL_ID	NUMBER	Identifier of the model to which the workspace belongs.
WORKSPACE_NAME	VARCHAR2(50)	Name of the workspace.
MBR_IND	VARCHAR2(1)	Indicates whether the workspace represents an MBR (minimum bounding rectangle) region in the model. `Y` indicates the workspace represents an MBR; `N` indicates the workspace does not represent an MBR
LOWER_X	NUMBER	If MBR_IND is `Y`, the X value for the lower coordinate of the MBR.
UPPER_X	NUMBER	If MBR_IND is `Y`, the X value for the upper coordinate of the MBR.
LOWER_Y	NUMBER	If MBR_IND is `Y`, the Y value for the lower coordinate of the MBR.
UPPER_Y	NUMBER	If MBR_IND is `Y`, the Y value for the upper coordinate of the MBR.
LOCK_IND	VARCHAR2(1)	Indicates whether the workspace is locked for editing by others (that is,, unable to be edited by others). `Y` indicates the workspace is locked; `N` indicates the workspace is not locked.

5.15 Network Data Model Application Programming Interface

The Oracle Spatial Network Data Model feature includes two client application programming interfaces (APIs): a PL/SQL interface provided by the SDO_NET package and a Java interface.

Both interfaces let you create and update network data, and the Java interface lets you 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 mainly use 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 xxx_SDO_NETWORK_METADATA Views) 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).
- Network Data Model PL/SQL Interface
- Network Data Model Java Interface
- Network Data Model XML Interface

5.15.1 Network Data Model PL/SQL Interface

The SDO_NET package provides subprograms for creating, accessing, and managing networks on a database server. Example 5-9 in Network Examples shows the use of SDO_NET 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 SDO_NET Package Subprograms.

5.15.2 Network Data Model Java Interface

> **Note:**
>
> Effective with Oracle Database Release 23ai, the Oracle Spatial Network Data Model APIs are compiled with JDK 11 as the OJVM in the database supports JDK11. However, the APIs will continue to be supported on JDK8 for backwards compatibility. When using the API, ensure that all the related JAR files are consistent with the JDK version (JDK 8 or JDK 11) that is being used. See RDBMS and JDK Version Compatibility for Oracle JDBC Drivers for more information on the JDBC drivers that are supported for the different JDK versions.

The Network Data Model feature includes the load on demand Java interface. Complete reference information about this interface is provided in *Oracle Spatial Java API Reference*. The classes of the load on demand Java interface are in the `oracle.spatial.network.lod` package and its subpackages.

The Spatial Java class libraries are in .jar files under the `<ORACLE_HOME>/md/jlib/` directory.

- Network Metadata and Data Management
- Network Analysis Using the Load on Demand Approach

5.15.2.1 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

5.15.2.2 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.

5.15.3 Network Data Model XML Interface

> **Note:**
>
> To efficiently and securely implement your customized network analysis, Oracle Spatial is deprecating the NDM XML API. Instead, use the Load On Demand (LOD) API. The LOD API provides customization interfaces that allow you to easily and securely implement your customization. In addition, you can also use the NDM Contraction Hierarchies REST API for high performance 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*.

HTTP requests can be sent to the web service from Java, PLSQL, or .NET programs or simply from a HTML form. The SDO_NET.POST_XML function (described in SDO_NET Package Subprograms) enables PL/SQL users to call the web service.

The XML schema of the Network Data Model XML API is described in the following: `$ORACLE_HOME/md/doc/sdondmxml.zip`

- User-Specified Implementations

5.15.3.1 User-Specified Implementations

The XML API can take user-specified constraints, cost calculators, or even network analysis algorithm settings, by letting you specify the Java class that implements the LOD interfaces. For any implementation that requires input parameters, such as truck weight or height in a trucking constraint implementation, the Java class must implement the `oracle.spatial.network.lod.XMLConfigurable` interface, that is, it must implement the following two methods:

- `void init(Element parameter);`
- `String getXMLSchema();`

The `init` method lets you pass in the input parameter as an XML element, which must follow the schema returned from the `getXMLSchema` method.

The following XML code segment is an example of how to configure the shortest path algorithm for a shortest path analysis request:

```
<startPoint>
  <nodeID>123</nodeID>
</startPoint>
<endPoint>
  <nodeID>456</nodeID>
</endPoint>
<shortestPathAlgorithm>
  <className>oracle.spatial.network.lod.AStar</className>
  <parameters>
    <heuristicCostFunction>
      <className>oracle.spatial.network.lod.GeodeticCostFunction</className>
      <parameters>
        <userDataCategory>0</userDataCategory>
        <xCoordUserDataIndex>0</xCoordUserDataIndex>
        <yCoordUserDataIndex>1</yCoordUserDataIndex>
      </parameters>
    </heuristicCostFunction>
    <linkLevelSelector>
      <className>oracle.spatial.network.lod.DynamicLinkLevelSelector</className>
      <parameters>
        <maxLinkLevel>2</maxLinkLevel>
        <costThreshold linkLevel="1">40000</costThreshold>
        <numHighLevelNeighbors>8</numHighLevelNeighbors>
        <costMultiplier>1.5</costMultiplier>
        <costFunction>
          <className>oracle.spatial.network.lod.GeodeticCostFunction</className>
          <parameters>
            <userDataCategory>0</userDataCategory>
            <xCoordUserDataIndex>0</xCoordUserDataIndex>
            <yCoordUserDataIndex>1</yCoordUserDataIndex>
          </parameters>
        </costFunction>
      </parameters>
    </linkLevelSelector>
  </parameters>
</shortestPathAlgorithm>
```

More examples of the XML API are provided with the NDM tutorial (see Network Data Model Tutorial and Other Resources).

5.16 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 options.

- 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 Cross-Schema Access by Specifying Owner in Network Metadata.
- For each non-owner user, create views on the Network Data Model tables and update the USER_SDO_NETWORK_METADATA view, as explained in Cross-Schema Access by Using Views.

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 Cross-Schema Access by Specifying Owner in Network Metadata or Cross-Schema Access by Using Views.

- Cross-Schema Access by Specifying Owner in Network Metadata
- Cross-Schema Access by Using Views

5.16.1 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 or READ privilege 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 xxx_SDO_NETWORK_METADATA Views). 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.

5.16.2 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 or READ privilege 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 xxx_SDO_NETWORK_METADATA Views). 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.

5.17 Network Examples

This topic presents several Network Data Model examples.

Most are simplified examples. All examples use the PL/SQL API, and some also use other APIs.

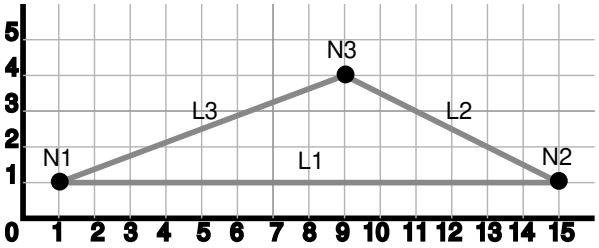
The examples refer to concepts that are explained in this chapter, and they use PL/SQL functions and procedures documented in SDO_NET Package Subprograms.

- Simple Spatial (SDO) Network Example (PL/SQL)
- Simple Logical Network Example (PL/SQL)
- Spatial (LRS) Network Example (PL/SQL)
- Logical Hierarchical Network Example (PL/SQL)
- Partitioning and Load on Demand Analysis Examples (PL/SQL, XML, and Java)
- User-Defined Data Examples (PL/SQL and Java)

5.17.1 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 5-5.

Figure 5-5 Simple Spatial (SDO) Network



As shown in Figure 5-5, 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 5-7 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 5-7 does not show the use of many SDO_NET functions and procedures; these are included in Example 5-9 in Spatial (LRS) Network Example (PL/SQL).

Example 5-7 Simple Spatial (SDO) Network Example (PL/SQL)

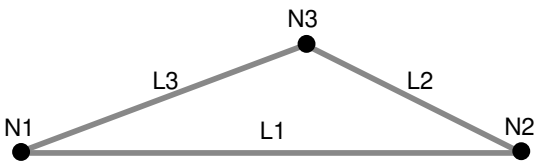
```
-- Create the SDO_NET1 directed network. Also creates the SDO_NET1_NODE$,
-- SDO_NET1_LINK$, SDO_NET1_PATH$, SDO_NET1_PLINK$ tables, and updates
-- USER_SDO_NETWORK_METADATA. All geometry columns are named GEOMETRY.
-- Both the node and link tables contain a cost column named COST.
EXECUTE SDO_NET.CREATE_SDO_NETWORK('SDO_NET1', 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;
```

5.17.2 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 5-6.

Figure 5-6 Simple Logical Network



As shown in Figure 5-6, 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 5-8 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 5-8 does not show the use of many SDO_NET functions and procedures; these are included in the logical hierarchical network example (Example 5-10) in Logical Hierarchical Network Example (PL/SQL).

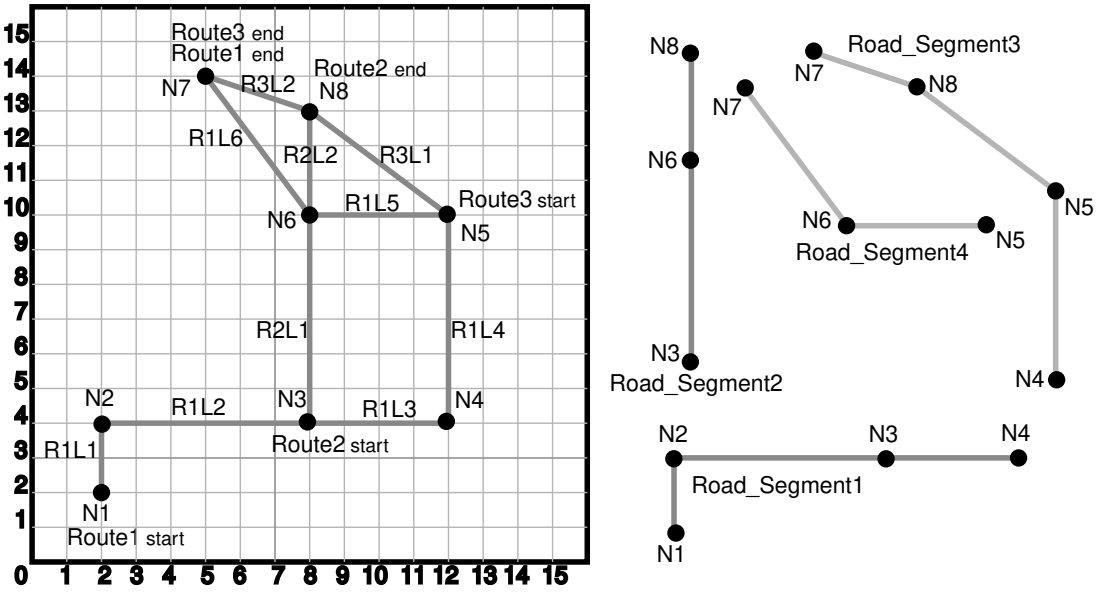
Example 5-8 Simple Logical Network Example (PL/SQL)

```
-- 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.
```

5.17.3 Spatial (LRS) Network Example (PL/SQL)

This section presents an example of a spatial (LRS) network that uses the roads (routes) illustrated in Figure 5-7. 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 5-7 Roads and Road Segments for Spatial (LRS) Network Example



As shown in Figure 5-7:

- `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 5-9 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.

Example 5-9 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.

-- 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
```

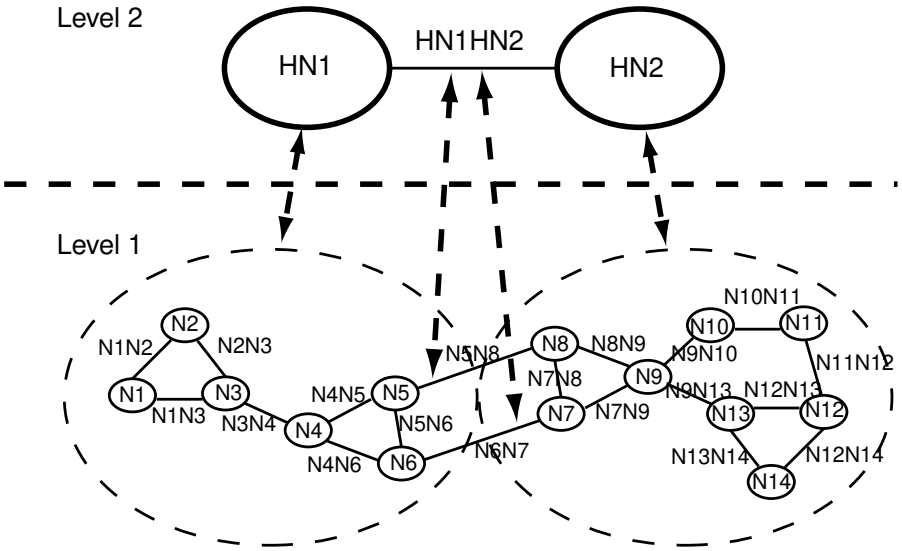
```
    'COST',  -- Link cost column
    'ROADS_PATHS',  -- Path table name
    'PATH_GEOMETRY',  -- Path geometry column
    'ROADS_PATHS_LINKS'  -- Paths and links table
    );

-- 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');
```

5.17.4 Logical Hierarchical Network Example (PL/SQL)

This section presents an example of a logical network that contains the nodes and links illustrated in Figure 5-8. Because it is a logical network, there are no spatial geometries associated with it. (Figure 5-8 is essentially the same as Figure 5-3 in Network Hierarchy, but with the nodes and links labeled.)

Figure 5-8 Nodes and Links for Logical Network Example



As shown in Figure 5-8:

- 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 5-10 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.

Example 5-10 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.

-- 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');
```

5.17.5 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 Network Analysis Using Load on Demand.

Additional examples of using load on demand analysis with partitioned networks are included in the demo files, described in Network Data Model Tutorial and Other Resources.

Example 5-11 Partitioning a Spatial Network

Example 5-11 partitions a spatial network named `NYC_NET`. (Assume that this network already exists and its metadata, node, and link tables are populated.)

Example 5-11 and Example 5-12 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.

```
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 5-12 Generating Partition BLOBs

Example 5-12 generates partition BLOBs for the network.

```
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 5-13 Configuring the Load on Demand Environment, Including Partition Cache

Example 5-13 shows the XML for configuring the load on demand environment, including the partition cache.

```
<?xml version="1.0" encoding="UTF-8" ?>
<LODConfigs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns ="http://xmlns.oracle.com/spatial/network"
            version = "12.1">
<!--The new xml configuration schema takes the version number. If the version attribute
is missing, then we assume it is 11.2 or lower. -->
<!-- default configuration for networks not configured -->
  <LODConfig globalNetworkName="$DEFAULT$" networkName="$DEFAULT$">
    <networkIO>
      <geometryTolerance>0.000001</geometryTolerance>
      <readPartitionFromBlob>false</readPartitionFromBlob>
      <partitionBlobTranslator>
        <className>oracle.spatial.network.lod.PartitionBlobTranslator11gR2</className>
        <parameters></parameters>
      </partitionBlobTranslator>
      <userDataIO categoryId="0">
        <className>oracle.spatial.network.lod.LODUserDataIOSDO</className>
        <parameters></parameters>
      </userDataIO>
      <cachingPolicy linkLevel="1">
        <maxNodes>500000</maxNodes>
        <residentPartitions></residentPartitions>
        <flushRule>
          <className>oracle.spatial.network.lod.LRUCachingHandler</className>
          <parameters></parameters>
        </flushRule>
      </cachingPolicy>
    </networkIO>
    <networkAnalysis>
      <linkLevelSelector>
        <className>oracle.spatial.network.lod.DummyLinkLevelSelector</className>
        <parameters></parameters>
      </linkLevelSelector>
      <withinCostPolygonTolerance>0.05</withinCostPolygonTolerance>
    </networkAnalysis>
  </LODConfig>
  <LODConfig globalNetworkName="SAMPLE_NETWORK" networkName="SAMPLE_NETWORK">
    <networkIO>
      <geometryTolerance>0.000001</geometryTolerance>
      <readPartitionFromBlob>true</readPartitionFromBlob>
      <partitionBlobTranslator>
        <className>oracle.spatial.router.ndm.RouterPartitionBlobTranslator11gR2</
className>
        <parameters></parameters>
      </partitionBlobTranslator>
      <userDataIO categoryId="0">
        <className>oracle.spatial.network.lod.LODUserDataIOSDO</className>
        <parameters></parameters>
      </userDataIO>
      <userDataIO categoryId="1">
```

```
        <className>oracle.spatial.router.ndm.RouterUserDataIO</className>
        <parameters></parameters>
      </userDataIO>
      <cachingPolicy linkLevel="1">
        <maxNodes>200000</maxNodes>
        <residentPartitions></residentPartitions>
        <flushRule>
          <className>oracle.spatial.network.lod.LRUCachingHandler</className>
          <parameters></parameters>
        </flushRule>
      </cachingPolicy>
      <cachingPolicy linkLevel="2">
        <maxNodes>800000</maxNodes>
        <residentPartitions>0</residentPartitions>
        <flushRule>
          <className>oracle.spatial.network.lod.LRUCachingHandler</className>
          <parameters></parameters>
        </flushRule>
      </cachingPolicy>
    </networkIO>
    <networkAnalysis>
    </networkAnalysis>
  </LODConfig>
</LODConfigs>
```

Example 5-14 Reloading the Load on Demand Configuration (Java API)

Example 5-14 and Example 5-15 show the Java and PL/SQL APIs, respectively, for reloading the load on demand configuration.

```
InputStream config = ClassLoader.getSystemResourceAsStream(
                     "netlodcfg.xml");
LODNetworkManager.getConfigManager().loadConfig(config);
```

Example 5-15 Reloading the Load on Demand Configuration (PL/SQL API)

```
EXECUTE SDO_NET.LOAD_CONFIG('WORK_DIR', 'netlodcfg.xml');
```

Example 5-16 Getting Estimated Partition Size

Example 5-16 returns the estimated size in bytes for a specified network partition.

```
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 5-17 Network Analysis: Shortest Path (LOD Java API)

Example 5-17 uses the load on demand Java API (oracle.spatial.network.lod) to issue a shortest-path query on a network.

```
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 5-18 Network Analysis: Shortest Path (XML API)

Example 5-18 uses the XML API (oracle.spatial.network.xml) to issue a shortest-path query on a network. It includes the request and the response.

```
<?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>
```

5.17.6 User-Defined Data Examples (PL/SQL and Java)

This section presents examples 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 xxx_SDO_NETWORK_USER_DATA Views) contain information about user-defined data.

To use user-defined data, you must set the USER_DEFINED_DATA column value to `Y` in the appropriate xxx_SDO_NETWORK_METADATA views (described in xxx_SDO_NETWORK_METADATA Views).

Example 5-19 Inserting User-Defined Data into Network Metadata

Example 5-19 uses the PL/SQL API to insert link-related user-defined data into the 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, category_id)
          values ('bi_test', 'LINK', 'interaction', 'VARCHAR2', 50, 0) ;
-- 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, category_id)
          values ('bi_test','LINK','PROB','NUMBER', 0) ;
```

After a network or network partition is loaded, user-defined data is available in Java representations. You can access user-defined data through the `getCategorizedUserData` and `setCategorizedUserData` methods for the `Node`, `Link`, `Path`, and `SubPath` interfaces. For example:

```
// The user data index is the sequence number of a user data within a category
// sorted by data name.

int interactionUserDataIndex = 0;
int probUserDataIndex = 1;
```

```
String interaction = (String)link.getCategorizedUserData().getUserData(0).
                     get(interactionUseDataIndex);

double prob = ((Double)link.getCategorizedUserData().getUserData(0).
              get(probUserdataIndex)).doubleValue();
```

Example 5-20 Implementation of writeUserData method of LODUserDataIO Interface

Example 5-20 uses the Java API for a custom user data I/O implementation (non-default implementation) of the `LODUserDataIO` interface. User data associated to links is written to BLOBs (one BLOB per partition) and read from BLOBs during analysis. In this example it is assumed that:

- The user-defined data BLOB for multimodal data for each partition has the partition ID and number of links associated with the partition followed by <Link ID, link route ID> for each link
- The user-defined data BLOB table name is MULTIMODAL_USER_DATA

```
 //Method getLinksInPartition(partitionId) computes a vector that
 // consists of  the ID and the route ID of each link associated with a partition
 // with ID = partitionId
 LinkVector = getLinksInPartition(partitionId);

 ObjectOutputStream dout = null;

 //Insert an empty blob for the partition with ID = partition_id
 String insertStr = "INSERT INTO " + MULTIMODAL_USER_DATA +
                             " (partition_id, blob) " + " VALUES " + " (?,
EMPTY_BLOB())" ;

 PreparedStatement stmt = conn.prepareStatement(insertStr);
 stmt.setInt(1,partitionId);
 int n = stmt.executeUpdate();
 stmt.close();

 //lock the row for blob update
  String lockRowStr = "SELECT blob FROM " + MULTIMODAL_USER_DATA +
                                  " WHERE partition_id = ? " + " FOR UPDATE";
  stmt = conn.prepareStatement(lockRowStr);
  stmt.setInt(1,partitionId);
  ResultSet rs = stmt.executeQuery();

  rs.next();
 oracle.sql.BLOB userDataBlob = (oracle.sql.BLOB) rs.getBlob(1);
 stmt.close();

  OutputStream blobOut = ((oracle.sql.BLOB) userDataBlob).setBinaryStream(1);
  dout = new ObjectOutputStream(blobOut);

  //write partition ID
  dout.writeInt(partitionId);
  int numLinks = linkVector.size()

   for (int i=0; i<linkVector.size(); i++) {
         //MultimodalLink is a class with variables link ID and route ID
         MultimodalLink link = (MultimodalLink) linkVector.elementAt(i);
          //write link ID
         dout.writeLong(link.getLinkId());

         // write route ID into file
         dout.writeInt(link.getRouteId());
    }
```

```
    dout.close();
     blobOut.close();
     rs.close();
```

Example 5-21 Implementation of readUserData method of LODUserDataIO Interface

In Example 5-21, the user-defined data is accessed through the `getCategorizedUserData` and `setCategorizedUserData` methods for the `Node`, `Link`, `Path`, and `SubPath` interfaces and the `getUserData` and `setUserData` methods of the `CategorizedUserData` interface.

```
//Read the blob for the required partition from the user data blob table
// In this example,
// MULTIMODAL_USER_DATA  is the name of user -defined data blob table
BLOB multimodalBlob = null;
String queryStr = "SELECT blob FROM " +
MULTIMODAL_USER_DATA
                            " WHERE partition_id = ?";
PreparedStatement stmt = conn.prepareStatement(queryStr);
stmt.setInt(1,partitionId);
ResultSet rs = stmt.executeQuery();
if (rs.next())   {
     multimodalBlob = (oracle.sql.BLOB)rs.getBlob(1);
}

// Materialize the blob value as an input stream
InputStream is = multimodalBlob.getBinaryStream();

//Create an ObjectInputStream that reads from the InputStream is
ObjectInputStream ois = new ObjectInputStream(is);

//Read the values of partition ID and number of links from the blob
int partitionId = ois.readInt();
int numLinks = ois.readInt();

for (int i=0; i<numLinks; i++)  {

    //Read link ID and route ID for each link
     long linkId = ois.readLong();
     int routeId = ois.readInt();

     //MultimodalLinkUserData is an implementation of NDM LOD UserData interface
     //Implementation is provided at the end of the example
     linkUserData = new MultimodalLinkUserData(routeId);

     //Get the link object corresponding to the link ID
     LogicalNetLink link = partition.getLink(linkId);

     //Get the (categorized) user data associated with the link.
     CategorizedUserData cud = link.getCategorizedUserData();

     // If the link does not have categorized user data associated with it,
     // initialize it to linkUserData
     // Else, set the user data for category USER_DATA_MULTIMODAL
     // to linkUserData
     if (cud == null) {
            UserData [] userDataArray = {linkUserData};
            cud = new CategorizedUserDataImpl(userDataArray);
             link.setCategorizedUserData(cud);
     }
     else {
            cud.setUserData(USER_DATA_MULTIMODAL,linkUserData);
```

```
     }
}
```

The following example shows how to read the user-defined data, namely the route ID associated with a link during analysis:

```
//info is an instance of LODAnalysisInfo
LogicalLink currentLink = info.getCurrentLink();

//Read the user-defined data (in this case, route ID)
int linkRouteId   = (Integer)currentLink.getCategorizedUserData().

getUserData(USER_DATA_MULTIMODAL).
                           get(INDEX_LINK_ROUTEID);
```

The implementation of the `MultimodalLinkUserData` interface is as follows:

```
class MultimodalLinkUserData implements UserData
{
    private int routeId;

    protected MultimodalLinkUserData(int routeId)
   {
        this.routeId = routeId;
   }

  public Object get(int index)
  {
    switch(index)
    {
         case INDEX_LINK_ROUTEID:
              return routeId;
    }
    return null;
  }

  public void set(int index, Object userData)
  {
    switch(index)
    {
         case INDEX_LINK_ROUTEID:
            this.routeId = (Integer) userData;
    }
  }

  public int getNumberOfUserData()
  {
       return 1;
  }

  public Object clone()
{
    return new MultimodalLinkUserData(routeId);
  }
}
```

5.18 Network Data Model Tutorial and Other Resources

Network Data Model learning resources are available.

http://www.oracle.com/technetwork/database-options/spatialandgraph on the Oracle Technology Network provides links to valuable resources to help you get started with Oracle Spatial technologies, including the Network Data Model (NDM). The Network Data Model resources include the following:

- A **Network Data Model tutorial** (ndm_tutorial.zip on http://www.oracle.com/technetwork/indexes/samplecode/spatial-1433316.html) outlines the steps to set up and configure a network, and to conduct the analysis. It also includes sample code and a web application that demonstrates how to use Oracle Maps to display analysis results.
- An **NDM technical brief** ("A Load-On-Demand Approach to Handling Large Networks in the Oracle Spatial Network Data Model") provides detailed explanations and examples.
- The **NDM editor**, provided as a demo intended for small and medium-size networks, is a graphical tool to view, browse, and navigate through data that is stored in the network data model. You can also use this tool to perform analysis on networks (shortest path, nearest neighbor, minimum cost spanning tree, and so on), and to add and delete nodes, links, and paths.
- **NFE (Network Feature Editing) Java API documentation** provides examples of NFE capabilities. It also provides electricity and water network examples, and a NFE data dictionary with NFE table descriptions.

  You are encouraged to examine the Java examples before starting development with NFE.
- The **NFE editor** is a sample application to create NFE models and manage them. The models can be created in the *From Scratch* mode or the *Over Existing Network Model* mode. You can create and manage feature layers, feature classes and features. Additionally, a video presentation shows how to create and manage NFE models with the NFE editor.

5.19 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 Model 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 Network Data Model Overview.

> **Note:**
>
> SDO_NET subprograms are only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

For a listing of the subprograms grouped in logical categories, see Network Data Model PL/SQL Interface. The rest of this chapter provides reference information about the subprograms, listed in alphabetical order.

- SDO_NET.DELETE_DANGLING_FEATURES
- SDO_NET.DELETE_DANGLING_LINKS
- SDO_NET.DELETE_DANGLING_NODES
- SDO_NET.DELETE_FEATURE_ELEMENTS
- SDO_NET.DELETE_FEATURE_ELEMENTS_AT
- SDO_NET.DELETE_FEATURES
- SDO_NET.DELETE_LINK
- SDO_NET.DELETE_NODE
- SDO_NET.DELETE_PATH
- SDO_NET.DELETE_PHANTOM_FEATURES
- SDO_NET.DELETE_SUBPATH
- SDO_NET.DEREGISTER_CONSTRAINT
- SDO_NET.DROP_FEATURE_LAYER
- SDO_NET.DROP_NETWORK
- SDO_NET.FIND_CONNECTED_COMPONENTS
- SDO_NET.GENERATE_NODE_LEVELS
- SDO_NET.GENERATE_PARTITION_BLOB
- SDO_NET.GENERATE_PARTITION_BLOBS
- SDO_NET.GET_CHILD_FEATURE_IDS
- SDO_NET.GET_CHILD_LINKS
- SDO_NET.GET_CHILD_NODES
- SDO_NET.GET_DANGLING_FEATURES
- SDO_NET.GET_DANGLING_LINKS
- SDO_NET.GET_DANGLING_NODES
- SDO_NET.GET_FEATURE_ELEMENTS
- SDO_NET.GET_FEATURE_LAYER_ID
- SDO_NET.GET_FEATURES_ON_LINKS
- SDO_NET.GET_FEATURES_ON_NODES
- SDO_NET.GET_GEOMETRY_TYPE
- SDO_NET.GET_IN_LINKS
- SDO_NET.GET_INVALID_LINKS
- SDO_NET.GET_INVALID_NODES
- SDO_NET.GET_INVALID_PATHS
- SDO_NET.GET_ISOLATED_NODES
- 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_LINKS_IN_PATH
- 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_PARENT_FEATURE_IDS
- SDO_NET.GET_PARTITION_SIZE
- SDO_NET.GET_PATH_GEOM_COLUMN
- SDO_NET.GET_PATH_TABLE_NAME
- SDO_NET.GET_PERCENTAGE
- SDO_NET.GET_PHANTOM_FEATURES
- SDO_NET.GET_PT
- SDO_NET.IS_HIERARCHICAL
- SDO_NET.IS_LINK_IN_PATH
- SDO_NET.IS_LOGICAL
- SDO_NET.IS_NODE_IN_PATH
- SDO_NET.IS_SPATIAL
- SDO_NET.LOAD_CONFIG
- SDO_NET.LOGICAL_PARTITION
- SDO_NET.LOGICAL_POWERLAW_PARTITION
- SDO_NET.LRS_GEOMETRY_NETWORK
- SDO_NET.NETWORK_EXISTS
- SDO_NET.POST_XML
- SDO_NET.REGISTER_CONSTRAINT
- SDO_NET.SDO_GEOMETRY_NETWORK
- SDO_NET.SET_LOGGING_LEVEL

6.1 SDO_NET.ADD_CHILD_FEATURE

Format

```
SDO_NET.ADD_CHILD_FEATURE(
     parent_layer_id   IN NUMBER,
     parent_feature_id IN NUMBER,
     child_layer_id    IN NUMBER,
     child_feature_id  IN SDO_NET_LAYER_FEAT,
     sequence_number   IN NUMBER DEFAULT NULL,
     check_integrity   IN BOOLEAN DEFAULT TRUE);
```

Description

Associates a feature as a child feature of a specified parent feature.

Parameters

parent_layer_id
ID of the parent feature layer.

parent_feature_id
ID of the feature that is to become the parent feature of the specified child feature.

child_layer_id
ID of the child feature layer.

child_feature_id
ID of the feature to be associated as a child feature of the specified parent feature. (The SDO_NET_LAYER_FEAT type is described in Data Types Used for Feature Modeling.)

sequence_number
Sequence number of the `child_feature_id` feature in the child feature layer. If this parameter is null, a sequence number after the last current number is assigned.

check_integrity
`TRUE` (the default) checks if the child feature exists; and if it does not exist, an error is generated. `FALSE` does not check if the child feature exists.

Usage Notes

The specified child feature must already exist.

To associate multiple features as child features, use the SDO_NET.ADD_CHILD_FEATURES procedure.

Examples

The following example adds a child feature at sequence number 2.

```
DECLARE
  parent_layer_id NUMBER;
  parent_feature_id NUMBER := 1;
  child_layer_id NUMBER;
  child_feature_id NUMBER := 3;
BEGIN
  parent_layer_id := sdo_net.get_feature_layer_id('GRID', 'PARENT_LAYER');
  child_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  sdo_net.add_child_feature(parent_layer_id, parent_feature_id, child_layer_id,
child_feature_id, 2, true);
END;
/
```

6.2 SDO_NET.ADD_CHILD_FEATURES

Format

```
SDO_NET.ADD_CHILD_FEATURES(
     parent_layer_id   IN NUMBER,
     parent_feature_id IN NUMBER,
     child_feature_ids IN SDO_NET_LAYER_FEAT_ARRAY,
     check_integrity   IN BOOLEAN DEFAULT TRUE);
```

Description

Associates multiple features as child features of a specified parent feature.

Parameters

parent_layer_id
ID of the parent feature layer.

parent_feature_id
ID of the feature that is to become the parent feature of the specified child features.

child_feature_ids
IDs of features to be associated as child features of the specified parent feature. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

check_integrity
`TRUE` (the default) checks if the child features exist; and if any do not exist, an error is generated. `FALSE` does not check if the child features exist.

Usage Notes

The specified child features must already exist.

To associate a single feature as a child feature, use the SDO_NET.ADD_CHILD_FEATURE procedure.

Examples

The following example adds two child features at the end of the parent feature.

```
DECLARE
  parent_layer_id NUMBER;
  parent_feature_id NUMBER := 1;
  child_layer_id NUMBER;
  child_feature_ids SDO_NET_LAYER_FEAT_ARRAY := SDO_NET_LAYER_FEAT_ARRAY();
BEGIN
  parent_layer_id := sdo_net.get_feature_layer_id('GRID', 'PARENT_LAYER');
  child_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  child_feature_ids.extend;
  child_feature_ids(1) := SDO_NET_LAYER_FEAT(child_layer_id, 4);
  child_feature_ids.extend;
  child_feature_ids(2) := SDO_NET_LAYER_FEAT(child_layer_id, 10);
  sdo_net.add_child_features(parent_layer_id, parent_feature_id, child_feature_ids,
true);
END;
/
```

6.3 SDO_NET.ADD_FEATURE

Format

```
SDO_NET.ADD_FEATURE(
     feature_layer_id  IN NUMBER,
     feature_id        IN NUMBER,
     feature_elements  IN SDO_NET_FEAT_ELEM_ARRAY DEFAULT NULL,
     child_feature_ids IN SDO_NET_LAYER_FEAT_ARRAY DEFAULT NULL,
     check_integrity   IN BOOLEAN DEFAULT TRUE);
```

Description

Adds a feature to a feature layer.

Parameters

feature_layer_id
ID of the feature layer to which to add the feature.

feature_id
ID of the feature to be added to the feature layer.

feature_elements
Feature elements of the feature to be added. If this parameter is null, no feature elements are defined for this feature. (The SDO_NET_FEAT_ELEM_ARRAY type is described in Data Types Used for Feature Modeling.)

child_feature_ids
IDs of the child features of the feature that are to be added along with the feature. If this parameter is null, no child features are to be added. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

check_integrity
`TRUE` (the default) checks if the input network elements exist; and if any do not exist, an error is generated. `FALSE` does not check if the input network elements exist.

Usage Notes

To update a feature in a feature layer, use the SDO_NET.UPDATE_FEATURE procedure.

Examples

The following example adds a feature associated with a point at 20% on link 1314.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  elements SDO_NET_FEAT_ELEM_ARRAY := SDO_NET_FEAT_ELEM_ARRAY();
  link_id NUMBER := 1314;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  elements.extend;
  elements(1) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.2, null);
  sdo_net.add_feature(feature_layer_id, feature_id, elements, null);
END;
/
```

6.4 SDO_NET.ADD_FEATURE_ELEMENT

Format

```
SDO_NET.ADD_FEATURE_ELEMENT(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER,
     feature_element  IN SDO_NET_FEAT_ELEM,
     sequence_number  IN NUMBER DEFAULT NULL,
     check_integrity  IN BOOLEAN DEFAULT TRUE);
```

Description

Adds a feature element to a feature.

Parameters

feature_layer_id
ID of the feature layer for the feature.

feature_id
ID of the feature.

feature_element
Feature element to be added to the feature. This feature element is automatically appended to the end of any existing feature elements in the feature. (The SDO_NET_FEAT_ELEM type is described in Data Types Used for Feature Modeling.)

sequence_number
Sequence number of the added feature element in the feature. If this parameter is null, a sequence number after the last current number is assigned.

check_integrity
`TRUE` (the default) checks if the input network elements exist; and if any do not exist, an error is generated. `FALSE` does not check if the input network elements exist.

Usage Notes

To add multiple feature elements to a feature in a single operation, use the SDO_NET.ADD_FEATURE_ELEMENTS procedure.

To update a feature element, use the SDO_NET.UPDATE_FEATURE_ELEMENT procedure.

Examples

The following example adds a point feature for node ID 13 at sequence number 2.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  feature_element SDO_NET_FEAT_ELEM;
  node_id NUMBER := 13;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_element := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_PON, node_id, null, null);
  sdo_net.add_feature_element(feature_layer_id, feature_id, feature_element, 2);
END;
/
```

6.5 SDO_NET.ADD_FEATURE_ELEMENTS

Format

```
SDO_NET.ADD_FEATURE_ELEMENTS(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER,
     feature_elements IN SDO_NET_FEAT_ELEM,_ARRAY,
     check_integrity  IN BOOLEAN DEFAULT TRUE);
```

Description

Adds an array of feature elements to a feature.

Parameters

feature_layer_id
ID of the feature layer for the feature.

feature_id
ID of the feature.

feature_elements
Feature elements to be added to the feature. These feature elements are automatically appended to the end of any existing feature elements in the feature. (The SDO_NET_FEAT_ELEM_ARRAY type is described in Data Types Used for Feature Modeling.)

check_integrity
`TRUE` (the default) checks if the input network elements exist; and if any do not exist, an error is generated. `FALSE` does not check if the input network elements exist.

Usage Notes

To add a single feature element to a feature, use the SDO_NET.ADD_FEATURE_ELEMENT procedure.

Examples

The following example adds two point feature elements at the end of the feature elements associated with feature ID 1.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  elements SDO_NET_FEAT_ELEM_ARRAY := SDO_NET_FEAT_ELEM_ARRAY();
  link_id NUMBER := 1314;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  elements.extend;
  elements(1) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.7, null);
  elements.extend;
  elements(2) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.8, null);
  sdo_net.add_feature_elements(feature_layer_id, feature_id, elements);
END;
/
```

6.6 SDO_NET.ADD_FEATURE_LAYER

Format

```
SDO_NET.ADD_FEATURE_LAYER(
     network_name IN VARCHAR2,
     feature_layer_name IN VARCHAR2,
     feature_layer_type IN VARCHAR2,
     feature_table      IN VARCHAR2,
     relation_table     IN VARCHAR2,
     hierarchy_table    IN VARCHAR2);
```

Description

Adds a feature layer.

Parameters

network_name
Name of the network.

feature_layer_name
Name of the feature layer.

feature_layer_type
Type of features in the layer (from Table 5-1).

feature_table
Name of the feature table (see Feature Table).

relation_table
Name of the feature element relationships table (see Feature Element Relationships Table).

hierarchy_table
Name of the feature hierarchy table (see Feature Hierarchy Table).

Usage Notes

A feature layer ID is automatically generated for the feature layer.

Examples

The following example creates a feature layer named `POI` (points of interest) of multipoints (`SDO_NET.FEAT_TYPE_MPOINT`).

```
BEGIN
  sdo_net.add_feature_layer(
    'GRID',
    'POI',
    SDO_NET.FEAT_TYPE_MPOINT,
    'POI_FEAT$',
    'POI_REL$',
    NULL
  );
END;
/
```

6.7 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 SDO_NET Package Subprograms) 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))
```

6.8 SDO_NET.COPY_NETWORK

Format

```
SDO_NET.COPY_NETWORK(
     source_network     IN VARCHAR2,
     target_network     IN VARCHAR2,
     storage_parameters IN VARCHAR2 DEFAULT NULL);
```

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`.

storage_parameters
Physical storage parameters used internally to create network tables. 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 creates an entry in the xxx_SDO_NETWORK_METADATA views (described in xxx_SDO_NETWORK_METADATA Views) 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');
```

6.9 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,
     add_bidirected_column  IN BOOLEAN DEFALT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);
```

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. (If the `geom_type` value is not spelled correctly, the `geom_column` column is *not* included in the table.)

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 Network Hierarchy.)

add_bidirected_column
`TRUE` adds a column named BIDIRECTED to the link table; `FALSE` (the default) does not add a column named BIDIRECTED to the link table.

storage_parameters
Physical storage parameters used internally to create the link table. 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 link table is described in Link Table.

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);
```

6.10 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,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);

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,
     subpath_table_name     IN VARCHAR2,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);
```

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 Network Hierarchy.)

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 Node Table.) 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 Node Table.) 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 Link Table.) 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 Link Table.) 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 Path Table.) 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 Path-Link Table.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

subpath_table_name
Name of the subpath table to be created. (The subpath table is explained in Subpath Table.)

is_complex
Reserved for future use. Ignored for the current release.

storage_parameters
Physical storage parameters used internally to create network tables. 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 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 xxx_SDO_NETWORK_METADATA Views).

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 as follows: 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);
```

6.11 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,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);

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,
     subpath_table_name     IN VARCHAR2,
     subpath_geom_column    IN VARCHAR2,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);
```

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 Network Hierarchy.)

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 Node Table.) 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 Node Table.) 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 Link Table.) 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 Link Table.) 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 Path Table.) 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 Path Table.) 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 Path-Link Table.) If you use the format that does not specify this parameter, a path-link table named *<network-name>*_PLINK$ is created.

subpath_table_name
Name of the subpath table to be created. (The subpath table is explained in Subpath Table.).

subpath_geom_column
Name of the geometry column in the subpath table. (The subpath table is explained in Subpath Table.)

storage_parameters
Physical storage parameters used internally to create network tables. 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 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 xxx_SDO_NETWORK_METADATA Views).

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 as follows: 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);
```

6.12 SDO_NET.CREATE_LRS_TABLE

Format

```
SDO_NET.CREATE_LRS_TABLE(
     table_name         IN VARCHAR2,
     geom_column        IN VARCHAR2,
     storage_parameters IN VARCHAR2 DEFAULT NULL);
```

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.

storage_parameters
Physical storage parameters used internally to create the LRS table. 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 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
```

6.13 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. (If the `geom_type` value is not spelled correctly, the `geom_column` column is *not* included in the table.)

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 Network Hierarchy.)

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 Node Table). 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 Node Table.

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);
```

6.14 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 Partition Table.

For information about using partitioned networks to perform analysis using the load on demand approach, see Network Analysis Using Load on Demand.

Examples

The following example creates a partition table named MY_PART_TAB.

```
EXECUTE SDO_NET.CREATE_PARTITION_TABLE('MY_PART_TAB');
```

6.15 SDO_NET.CREATE_PATH_LINK_TABLE

Format

```
SDO_NET.CREATE_PATH_LINK_TABLE(
     table_name         IN VARCHAR2,
     storage_parameters IN VARCHAR2 DEFAULT NULL);
```

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.

storage_parameters
Physical storage parameters used internally to create the path-link table. 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 path-link table is described in Path-Link Table.

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');
```

6.16 SDO_NET.CREATE_PATH_TABLE

Format

```
SDO_NET.CREATE_PATH_TABLE(
     table_name         IN VARCHAR2,
     geom_column        IN VARCHAR2,
     storage_parameters IN VARCHAR2 DEFAULT NULL);
```

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.

storage_parameters
Physical storage parameters used internally to create the path table. 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 path table is described in Path Table.

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');
```

6.17 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,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);

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,
     subpath_table_name     IN VARCHAR2,
     subpath_geom_column    IN VARCHAR2,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);
```

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 Network Hierarchy.)

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 Node Table.) 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 Node Table.) 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 Node Table.) 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 Link Table.) 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 Link Table.) 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 Link Table.) 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 Path Table.) 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 Path Table.) 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 Path-Link Table.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

subpath_table_name
Name of the subpath table to be created. (The subpath table is explained in Subpath Table.)

subpath_geom_column
Name of the geometry column in the subpath table. (The subpath table is explained in Subpath Table.).

is_complex
Reserved for future use. Ignored for the current release.

storage_parameters
Physical storage parameters used internally to create network tables. 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 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 xxx_SDO_NETWORK_METADATA Views).

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 as follows: 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);
```

6.18 SDO_NET.CREATE_SUBPATH_TABLE

Format

```
SDO_NET.CREATE_SUBPATH_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 Node Table). 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 Subpath Table.

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');
```

6.19 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,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);

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,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);

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_geom_column       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,
     subpath_table_name     IN VARCHAR2,
     subpath_geom_column    IN VARCHAR2,
     is_complex             IN BOOLEAN DEFAULT FALSE,
     storage_parameters     IN VARCHAR2 DEFAULT NULL);
```

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 Network Hierarchy.)

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 Node Table.) 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 Node Table.) 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 Link Table.) 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 Link Table.) 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 Path Table.) 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 Path Table.) 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 Path-Link Table.) If you use the format that does not specify this parameter, a path-link table named <*network-name*>_PLINK$ is created.

subpath_table_name
Name of the subpath table to be created. (The subpath table is explained in Subpath Table.).

subpath_geom_column
Name of the geometry column in the subpath table. (The subpath table is explained in Subpath Table.)

is_complex
Reserved for future use. Ignored for the current release.

storage_parameters
Physical storage parameters used internally to create network tables. 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 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 xxx_SDO_NETWORK_METADATA Views). 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 as follows: 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);
```

6.20 SDO_NET.DELETE_CHILD_FEATURES

Format

```
SDO_NET.DELETE_CHILD_FEATURES(
     parent_layer_id   IN NUMBER,
     parent_feature_id IN NUMBER,
     child_feature_ids IN SDO_NET_LAYER_FEAT_ARRAY);
```

Description

Removes the parent-child relationship for the input child features.

Parameters

parent_layer_id
ID of the parent feature layer.

parent_feature_id
ID of the parent feature of the specified child features.

child_feature_ids
IDs of the child features. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

Usage Notes

The specified parent and child features must exist.

To delete the child features at specified sequence points, use the SDO_NET.DELETE_CHILD_FEATURES_AT procedure.

Examples

The following example deletes a child feature with feature ID 1 in the `POI` feature layer.

```
DECLARE
  parent_layer_id NUMBER;
  parent_feature_id NUMBER := 1;
  child_layer_id NUMBER;
  child_feature_ids SDO_NET_LAYER_FEAT_ARRAY := SDO_NET_LAYER_FEAT_ARRAY();
BEGIN
  parent_layer_id := sdo_net.get_feature_layer_id('GRID', 'PARENT_LAYER');
  child_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  child_feature_ids.extend;
  child_feature_ids(1) := SDO_NET_LAYER_FEAT(child_layer_id, 1);
  sdo_net.delete_child_features(parent_layer_id, parent_feature_id, child_feature_ids);
END;
/
```

6.21 SDO_NET.DELETE_CHILD_FEATURES_AT

Format

```
SDO_NET.DELETE_CHILD_FEATURES_AT(
     parent_layer_id   IN NUMBER,
     parent_feature_id IN NUMBER,
     sequence_numbers  IN SDO_NUMBER_ARRAY);
```

Description

Removes the parent-child relationship for the child features at the specified sequence numbers.

Parameters

parent_layer_id
ID of the parent feature layer.

parent_feature_id
ID of the parent feature of the specified child features.

child_feature_ids
IDs of the child features. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

Usage Notes

The specified parent and child features must exist.

To delete child features specified by their ID values, use the SDO_NET.DELETE_CHILD_FEATURES procedure.

Examples

The following example deletes a child feature at sequence number 1.

```
DECLARE
  parent_layer_id NUMBER;
  parent_feature_id NUMBER := 1;
  sequence_numbers SDO_NUMBER_ARRAY := SDO_NUMBER_ARRAY(1);
BEGIN
  parent_layer_id := sdo_net.get_feature_layer_id('GRID', 'PARENT_LAYER');
  sdo_net.delete_child_features_at(parent_layer_id, parent_feature_id, sequence_numbers);
END;
/
```

6.22 SDO_NET.DELETE_DANGLING_FEATURES

Format

```
SDO_NET.DELETE_DANGLING_FEATURES(
     feature_layer_id IN NUMBER);
```

Description

Deletes dangling features in a feature layer. A dangling feature is a feature that is not associated with any network elements (nodes or links).

Parameters

feature_layer_id
ID of the feature layer containing the features.

Usage Notes

To find the dangling features in a feature layer, use the SDO_NET.GET_DANGLING_FEATURES function.

Examples

The following example deletes any dangling features in the `POI` feature layer in the `GRID` network.

```
DECLARE
  feature_layer_id NUMBER;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  sdo_net.delete_dangling_features(feature_layer_id);
END;
/
```

6.23 SDO_NET.DELETE_DANGLING_LINKS

Format

```
SDO_NET.DELETE_DANGLING_LINKS(
     network IN VARCHAR2);
```

Description

Deletes links that are not referenced by any feature in any feature layer.

Parameters

network
Name of the network.

Usage Notes

To find the dangling links in a network, use the SDO_NET.GET_DANGLING_LINKS function.

Examples

The following example deletes any dangling links in the `GRID` network.

```
EXECUTE sdo_net.delete_dangling_links('GRID');
```

6.24 SDO_NET.DELETE_DANGLING_NODES

Format

```
SDO_NET.DELETE_DANGLING_NODES(
     network IN VARCHAR2);
```

Description

Deletes nodes that are not referenced by any feature in any feature layer.

Parameters

network
Name of the network.

Usage Notes

To find the dangling nodes in a network, use the SDO_NET.GET_DANGLING_NODES function.

Examples

The following example deletes any dangling nodes in the `GRID` network.

```
EXECUTE sdo_net.delete_dangling_nodes('GRID');
```

6.25 SDO_NET.DELETE_FEATURE_ELEMENTS

Format

```
SDO_NET.DELETE_FEATURE_ELEMENTS(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER,
     feature_elements IN SDO_NET_FEAT_ELEM_ARRAY,
     delete_net_elems IN BOOLEAN DEFAULT FALSE);
```

Description

Deletes feature elements from a feature.

Parameters

feature_layer_id
ID of the feature layer containing the feature.

feature_id
ID of the feature from which to delete the feature elements.

feature_elements
Feature elements to be deleted. (The SDO_NET_FEAT_ELEM_ARRAY type is described in Data Types Used for Feature Modeling.)

delete_net_elems
Controls whether all network elements that are referenced only by the specified features are also deleted: `TRUE` also deletes such elements; `FALSE` (the default) does not also delete such elements.

Usage Notes

Contrast this procedure with SDO_NET.DELETE_FEATURE_ELEMENTS_AT.

Examples

The following example two point feature elements from a specified feature layer.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  elements SDO_NET_FEAT_ELEM_ARRAY := SDO_NET_FEAT_ELEM_ARRAY();
  link_id NUMBER := 1314;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  elements.extend;
  elements(1) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.2, null);
  elements.extend;
  elements(2) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.7, null);
  sdo_net.delete_feature_elements(feature_layer_id, feature_id, elements);
END;
/
```

6.26 SDO_NET.DELETE_FEATURE_ELEMENTS_AT

Format

```
SDO_NET.DELETE_FEATURE_ELEMENTS_AT(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER,
     sequence_numbers IN SDO_NUMBER_ARRAY,
     delete_net_elems IN BOOLEAN DEFAULT FALSE);
```

Description

Deletes the feature elements with specified sequence numbers from a feature.

Parameters

feature_layer_id
ID of the feature layer containing the feature.

feature_id
ID of the feature from which to delete the feature elements.

sequence_numbers
Array of sequence numbers for the feature elements to be deleted.

delete_net_elems
Controls whether all network elements that are referenced only by the specified features are also deleted: `TRUE` also deletes such elements; `FALSE` (the default) does not also delete such elements.

Usage Notes

Contrast this procedure with SDO_NET.DELETE_FEATURE_ELEMENTS

Examples

The following example deletes the feature element at sequence number 1.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  sequence_numbers SDO_NUMBER_ARRAY := SDO_NUMBER_ARRAY();
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  sequence_numbers.extend;
  sequence_numbers(1) := 1;
  sdo_net.delete_feature_elements_at(feature_layer_id, feature_id, sequence_numbers);
END;
/
```

6.27 SDO_NET.DELETE_FEATURES

Format

```
SDO_NET.DELETE_FEATURES(
     feature_layer_id IN NUMBER,
     feature_ids      IN SDO_NUMBER_ARRAY,
     delete_net_elems IN BOOLEAN DEFAULT FALSE,
     delete_children  IN BOOLEAN DEFAULT FALSE);
```

Description

Deletes features.

Parameters

feature_layer_id
ID of the feature layer containing the features

feature_ids
IDs of the features to be deleted.

delete_net_elems
Controls whether all network elements that are referenced only by the specified features are also deleted: `TRUE` also deletes such elements; `FALSE` (the default) does not also delete such elements.

delete_children
Controls whether all child features that are referenced only by the specified features are also deleted: `TRUE` also deletes such features; `FALSE` (the default) does not also delete such features.

Usage Notes

(None.)

Examples

The following example deletes the feature with feature ID 1 from the `POI` feature layer in the `GRID` network.

```
DECLARE
  feature_layer_id NUMBER;
  feature_ids SDO_NUMBER_ARRAY := SDO_NUMBER_ARRAY(1);
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  sdo_net.delete_features(feature_layer_id, feature_ids, false, false);
END;
/
```

6.28 SDO_NET.DELETE_LINK

Format

```
SDO_NET.DELETE_LINK(
     network IN VARCHAR2,
     link_id IN NUMBER);
```

Description

Deletes a link, along with all dependent network elements and all references to the link from features.

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 Link Table), 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);
```

6.29 SDO_NET.DELETE_NODE

Format

```
SDO_NET.DELETE_NODE(
     network IN VARCHAR2,
     node_id IN NUMBER);
```

Description

Deletes a node, along with all dependent network elements and all references to the node from features.

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 Node Table), 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);
```

6.30 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 Path Table), 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);
```

6.31 SDO_NET.DELETE_PHANTOM_FEATURES

Format

```
SDO_NET.DELETE_PHANTOM_FEATURES(
     feature_layer_id IN NUMBER);
```

Description

Deletes phantom features in a feature layer. A phantom feature is a feature that references nonexistent network elements (nodes or links).

Parameters

feature_layer_id
ID of the feature layer containing the features.

Usage Notes

To find the phantom features in a feature layer, use the SDO_NET.GET_PHANTOM_FEATURES function.

Examples

The following example deletes any phantom features in the `POI` feature layer in the `GRID` network.

```
DECLARE
  feature_layer_id NUMBER;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  sdo_net.delete_phantom_features(feature_layer_id);
END;
/
```

6.32 SDO_NET.DELETE_SUBPATH

Format

```
SDO_NET.DELETE_SUBPATH(
     network    IN VARCHAR2,
     subpath_id IN NUMBER);
```

Description

Deletes a subpath.

Parameters

network
Network name.

subpath_id
ID of the subpath to delete.

Usage Notes

This procedure deletes the specified subpath from the path table (described in Path Table). 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);
```

6.33 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 xxx_SDO_NETWORK_CONSTRAINTS Views).

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 Network Constraints.

Examples

The following example deregisters (disables) a network constraint named `GivenProhibitedTurn`.

```
EXECUTE SDO_NET.DEREGISTER_CONSTRAINT('GivenProhibitedTurn');
```

6.34 SDO_NET.DROP_FEATURE_LAYER

Format

```
SDO_NET.DROP_FEATURE_LAYER(
     network_name       IN VARCHAR2,
     feature_layer_name IN VARCHAR2,
     drop_tables        IN BOOLEAN DEFAULT FALSE);
```

Description

Drops (deletes) a feature layer.

Parameters

network_name
Name of the network containing the feature layer to be dropped.

feature_layer_name
Name of the feature layer to be dropped.

drop_tables
Controls whether all relevant tables are deleted along with the feature layer metadata: `TRUE` drops the feature table, feature element relationships table, and feature hierarchy table, and deletes the feature layer metadata; `FALSE` (the default) deletes the feature layer metadata but does not drop the feature table, feature element relationships table, and feature hierarchy table.

Usage Notes

(None.)

Examples

The following example drops the `POI` feature layer in the `GRID` network, and (because `drop_tables` is `true`) drops the feature table, feature element relationships table, and feature hierarchy table, and deletes the feature layer metadata

```
EXECUTE sdo_net.drop_feature_layer('GRID', 'POI', true);
```

6.35 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');
```

6.36 SDO_NET.FIND_CONNECTED_COMPONENTS

Format

```
SDO_NET.FIND_CONNECTED_COMPONENTS(
     network 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 (default = 1). The 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 Connected Component Table.

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` (the default) 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');
```

6.37 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 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` (the default) 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 multilevel network (one with multiple link levels), this procedure does not perform any operation.

This procedure is used internally by the SDO_NET.GENERATE_PARTITION_BLOBS procedure. Therefore, if you have executed SDO_NET.GENERATE_PARTITION_BLOBS, you do not need to execute this procedure. However, you do need to execute this procedure explicitly in these cases:

- When a Java application has been configured to read partitions from the node or link tables instead of from BLOBs, and partition BLOBs have never been generated on the network.
- When a higher-level node has been added or deleted in the network and the node-partition relationship has been updated. Before you execute SDO_NET.GENERATE_PARTITION_BLOB to regenerate the containing partition BLOB or BLOBs, you must manually either update the node level table or execute this procedure (SDO_NET.GENERATE_NODE_LEVELS).

The node level table name is stored in the NODE_LEVEL_TABLE_NAME column of the USER_SDO_NETWORK_METADATA view, which is described in xxx_SDO_NETWORK_METADATA Views.

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');
```

6.38 SDO_NET.GENERATE_PARTITION_BLOB

Format

```
SDO_NET.GENERATE_PARTITION_BLOB(
     network              IN VARCHAR2,
     link_level           IN NUMBER DEFAULT 1,
     partition_id         IN VARCHAR2,
     include_user_data    IN BOOLEAN,
     log_loc              IN VARCHAR2,
     log_file             IN VARCHAR2,
     open_mode            IN VARCHAR2 DEFAULT 'A',
     preform_delta_update IN BOOLEAN DEFAULT FALSE);
```

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 (default = 1). The 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
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 of category 0 (zero) 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` (the default) 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.

perform_delta_update
(Reserved for future use. The only permitted value is `FALSE`, the default.)

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 of category 0 (zero)

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');
```

6.39 SDO_NET.GENERATE_PARTITION_BLOBS

Format

```
SDO_NET.GENERATE_PARTITION_BLOBS(
     network                   IN VARCHAR2,
     link_level                IN NUMBER DEFAULT 1,
     partition_blob_table_name IN VARCHAR2,
     include_user_data         IN BOOLEAN,
     commit_for_each_blob      IN BOOLEAN DEFAULT TRUE,
     log_loc                   IN VARCHAR2,
     log_file                  IN VARCHAR2,
     open_mode                 IN VARCHAR2 DEFAULT 'A',
     perform_delta_update      IN BOOLEAN DEFAULT FALSE,
     regenerate_node_levels    IN BOOLEAN DEFAULT FALSE);
```

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 (default = 1). The 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 Partition BLOB Table.

include_user_data
`TRUE` if each BLOB should include any user data of category 0 (zero) associated with the network elements represented in each BLOB, or `FALSE` if each BLOB should not include any user data.

commit_for_each_blob
`TRUE` (the default) if each partition BLOB should be committed to the database after it is generated, or `FALSE` if each BLOB should not be committed (in which case you must perform one or more explicit commit operations).

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` (the default) 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.

perform_delta_update
(Reserved for future use. The only permitted value is `FALSE`, the default.)

regenerate_node_levels
`TRUE` to regenerate the node level table for multilevel networks, or `FALSE` (the default) not to regenerate the node level table for multilevel networks. You should set this parameter to `TRUE` if higher-level (second level or above) nodes are added or deleted from the network, or if the level of a node is changed. The level of a node is defined as the maximum link level coming into or out of the node.

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.

When this procedure is first executed on a multilevel network, it internally calls SDO_NET.GENERATE_NODE_LEVELS to create and populate the node level table (described in Node Level Table (Optional)). When this procedure is called subsequently on a multilevel network, you can use the `regenerate_node_levels` parameter to specify whether to overwrite the existing node level table.

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 of category 0 (zero) 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');
```

6.40 SDO_NET.GET_CHILD_FEATURE_IDS

Format

```
SDO_NET.GET_CHILD_FEATURE_IDS(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER
) RETURN SDO_NET_LAYER_FEAT_ARRAY;
```

Description

Returns the feature layer ID and child feature IDs for the specified feature. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

Parameters

feature_layer_id
ID of the feature layer for the feature (that is, the parent feature).

feature_id
ID of the feature.

Usage Notes

To get the feature layer ID and feature ID of the parent features for a specified feature, use the SDO_NET.GET_PARENT_FEATURE_IDS function.

For information about features, including parent and child features, see Features and Feature Layers.

Examples

The following example returns and displays the child feature IDs for feature 1 in the `PARENT_LAYER` feature layer.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  feature_ids SDO_NET_LAYER_FEAT_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'PARENT_LAYER');
  feature_ids := sdo_net.get_child_feature_ids(feature_layer_id, feature_id);
  FOR i in 1..feature_ids.count
  LOOP
    --dbms_output.put_line('['||i||']'||' FEATURE_LAYER_ID   = '||
feature_ids(i).feature_layer_id);
    dbms_output.put_line('['||i||']'||' FEATURE_ID         = '||
feature_ids(i).feature_id);
    dbms_output.put_line('---');
  END LOOP;
END;
/
```

6.41 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 Network Hierarchy.

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)
```

6.42 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 Network Hierarchy.

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)
```

6.43 SDO_NET.GET_DANGLING_FEATURES

Format

```
SDO_NET.GET_DANGLING_FEATURES(
     feature_layer_id IN NUMBER
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the IDs of dangling features in a feature layer. A dangling feature is a feature that is not associated with any network elements (nodes or links).

Parameters

feature_layer_id
ID of the feature layer containing the features.

Usage Notes

To delete the dangling features in a feature layer, use the SDO_NET.DELETE_DANGLING_FEATURES procedure.

Examples

The following example gets the dangling features in the `POI` feature layer of the `GRID` network and then displays their feature IDs.

```
DECLARE
  feature_layer_id NUMBER;
  feature_ids SDO_NUMBER_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_ids := sdo_net.get_dangling_features(feature_layer_id);
  dbms_output.put_line('Dangling Features:');
  for i in 1..feature_ids.count loop
    dbms_output.put_line('['||i||'] '||feature_ids(i));
  end loop;
END;
/
```

6.44 SDO_NET.GET_DANGLING_LINKS

Format

```
SDO_NET.GET_DANGLING_LINKS(
     network IN VARCHAR2
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns links that are not referenced by any feature in any feature layer.

Parameters

network
Name of the network.

Usage Notes

To delete the dangling links in a network, use the SDO_NET.DELETE_DANGLING_LINKS procedure.

Examples

The following example gets the dangling links in the `GRID` network and then displays the number (count) of dangling links found.

```
DECLARE
  link_ids SDO_NUMBER_ARRAY;
BEGIN
  link_ids := sdo_net.get_dangling_links('GRID');
  dbms_output.put_line('Number of dangling Links: '||link_ids.count);
END;
/
```

6.45 SDO_NET.GET_DANGLING_NODES

Format

```
SDO_NET.GET_DANGLING_NODES(
     network IN VARCHAR2
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns nodes that are not referenced by any feature in any feature layer.

Parameters

network
Name of the network.

Usage Notes

To delete the dangling nodes in a network, use the SDO_NET.DELETE_DANGLING_NODES procedure.

Examples

The following example gets the dangling nodes in the `GRID` network and then displays the number (count) of dangling nodes found.

```
DECLARE
  node_ids SDO_NUMBER_ARRAY;
BEGIN
  node_ids := sdo_net.get_dangling_nodes('GRID');
  dbms_output.put_line('Number of dangling Nodes: '||node_ids.count);
END;
/
```

6.46 SDO_NET.GET_FEATURE_ELEMENTS

Format

```
SDO_NET.GET_FEATURE_ELEMENTS(
     feature_layer_id IN NUMBER,
     feature_id       IN NUMBER
) RETURN SDO_NET_FEAT_ELEM_ARRAY;
```

Description

Returns the feature elements in a feature layer. (The SDO_NET_FEAT_ELEM_ARRAY type is described in Data Types Used for Feature Modeling.)

Parameters

feature_layer_id
ID of the feature layer for the feature.

feature_id
ID of the feature.

Usage Notes

To add a feature element to a feature, use the SDO_NET.ADD_FEATURE_ELEMENT procedure; to add multiple feature elements in a single operation, use the SDO_NET.ADD_FEATURE_ELEMENTS procedure.

Examples

The following example gets the feature layer ID for a specified feature layer, then gets and displays information about the feature elements for feature 1 in this feature layer.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  elements SDO_NET_FEAT_ELEM_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  elements := sdo_net.get_feature_elements(feature_layer_id, feature_id);
  FOR i in 1..elements.count
  LOOP
    dbms_output.put_line('['||i||']'||' FEAT_ELEM_TYPE   = '||
elements(i).feat_elem_type);
    dbms_output.put_line('['||i||']'||' NET_ELEM_ID      = '||elements(i).net_elem_id);
    dbms_output.put_line('['||i||']'||' START_PERCENTAGE = '||
```

```
elements(i).start_percentage);
    dbms_output.put_line('['||i||']'||' END_PERCENTAGE   = '||
elements(i).end_percentage);
    dbms_output.put_line('---');
  END LOOP;
END;
/
```

6.47 SDO_NET.GET_FEATURE_LAYER_ID

Format

```
SDO_NET.GET_FEATURE_LAYER_ID(
     network_name       IN VARCHAR2
     feature_layer_name IN VARCHAR2
) RETURN NUMBER;
```

Description

Returns the feature layer ID for a specified feature layer.

Parameters

network_name
Network name.

feature_layer_name
Feature layer name.

Usage Notes

This function returns the value of the FEATURE_LAYER_ID column for the network and feature layer combination in the USER_SDO_NETWORK_FEATURE view (see Table 5-41 in xxx_SDO_NETWORK_FEATURE Views).

Examples

The following example gets and displays the feature layer ID for a specified feature layer.

```
DECLARE
  feature_layer_id NUMBER;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  dbms_output.put_line('Feature layer ID for the POI feature layer is '||
feature_layer_id);
END;
/
```

6.48 SDO_NET.GET_FEATURES_ON_LINKS

Format

```
SDO_NET.GET_FEATURES_ON_LINKS
     feature_layer_id IN NUMBER,
     link_ids         IN SDO_NUMBER_ARRAY
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the IDs of features in a feature layer that reference specified links.

Parameters

feature_layer_id
ID of the feature layer containing the features.

link_ids
IDs of the links to check for features.

Usage Notes

To find the IDs of features in a feature layer that reference specified nodes, use the SDO_NET.GET_FEATURES_ON_NODES procedure.

Examples

The following example gets and displays the feature IDs of features on a specified link.

```
DECLARE
  feature_layer_id NUMBER;
  link_ids SDO_NUMBER_ARRAY := SDO_NUMBER_ARRAY(1314);
  feature_ids SDO_NUMBER_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_ids := sdo_net.get_features_on_links(feature_layer_id, link_ids);
  dbms_output.put_line('Features On Link '||link_ids(1)||':');
  for i in 1..feature_ids.count loop
    dbms_output.put_line('['||i||'] '||feature_ids(i));
  end loop;
END;
/
```

6.49 SDO_NET.GET_FEATURES_ON_NODES

Format

```
SDO_NET.GET_FEATURES_ON_NODES
     feature_layer_id IN NUMBER,
     node_ids         IN SDO_NUMBER_ARRAY
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the IDs of features in a feature layer that reference specified nodes.

Parameters

feature_layer_id
ID of the feature layer containing the features.

node_ids
IDs of the nodes to check for features.

Usage Notes

To find the IDs of features in a feature layer that reference specified links, use the SDO_NET.GET_FEATURES_ON_LINKS procedure.

Examples

The following example gets and displays the feature IDs of features on a specified node.

```
DECLARE
  feature_layer_id NUMBER;
  node_ids SDO_NUMBER_ARRAY := SDO_NUMBER_ARRAY(13);
  feature_ids SDO_NUMBER_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_ids := sdo_net.get_features_on_nodes(feature_layer_id, node_ids);
  dbms_output.put_line('Features On Node '||node_ids(1)||':');
  for i in 1..feature_ids.count loop
    dbms_output.put_line('['||i||'] '||feature_ids(i));
  end loop;
END;
/
```

6.50 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.51 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 Network Data Model Concepts.

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)
```

6.52 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;
```

6.53 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;
```

6.54 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;
```

6.55 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 Network Data Model Concepts.

Examples

The following example returns the isolated nodes in the SDO_PARTITIONED network.

```
SELECT SDO_NET.GET_ISOLATED_NODES('SDO_PARTITIONED') FROM DUAL;
```

6.56 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.57 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.58 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.59 SDO_NET.GET_LINK_GEOMETRY

Format

```
SDO_NET.GET_LINK_GEOMETRY(
  network          IN VARCHAR2,
  link_id          IN NUMBER,
  start_percentage IN NUMBER DEFAULT 0,
  end_percentage   IN NUMBER DEFAULT 1.0
) RETURN SDO_GEOMETRY;
```

Description

Returns the entire geometry or a portion of 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.

start_percentage
Percentage of the distance along the link to be used for the start point of the returned geometry. Expressed as a number between 0 and 1.0; for example, 0.5 is 50 percent. The default value is 0; that is, the start of the returned geometry is associated with the start point of the link.

end_percentage
Percentage of the distance along the link to be used for the end point of the returned geometry. Expressed as a number between 0 and 1.0; for example, 0.5 is 50 percent. The default value is 1.0; that is, the end of returned geometry is associated with the end point of the link.

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))
```

6.60 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.61 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 Network Data Model Concepts.

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_LINKS_IN_PATH('XYZ_NETWORK', 1) FROM DUAL;

SDO_NET.GET_LINKS_IN_PATH('XYZ_NETWORK',1)
--------------------------------------------------------------------------------
SDO_NUMBER_ARRAY(1102, 1104, 1105)
```

6.62 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.63 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))
```

6.64 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)
```

6.65 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.66 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.67 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

For an explanation of network hierarchy, see Network Hierarchy.

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
```

6.68 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
```

6.69 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 Network Data Model Concepts.

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
```

6.70 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 Network Data Model Concepts.

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
```

6.71 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.72 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)
```

6.73 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 Network Data Model Concepts.

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
```

6.74 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 Network Data Model Concepts.

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
```

6.75 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.76 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 Network Data Model Concepts.

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)
```

6.77 SDO_NET.GET_PARENT_FEATURE_IDS

Format

```
SDO_NET.GET_PARENT_FEATURE_IDS(
  feature_layer_id  IN NUMBER,
  feature_id        IN NUMBER
) RETURN SDO_NET_LAYER_FEAT_ARRAY;
```

Description

Returns the feature layer ID and parent feature IDs for the specified feature. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

Parameters

feature_layer_id
ID of the feature layer for the feature (that is, the child feature).

feature_id
ID of the feature.

Usage Notes

To get the feature layer ID and feature ID of the child features for a specified feature, use the SDO_NET.GET_CHILD_FEATURE_IDS function.

For information about features, including parent and child features, see Features and Feature Layers.

Examples

The following example returns and displays the parent feature IDs for feature 1 in the POI feature layer.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  feature_ids SDO_NET_LAYER_FEAT_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_ids := sdo_net.get_parent_feature_ids(feature_layer_id, feature_id);
```

```
  FOR i in 1..feature_ids.count
  LOOP
    --dbms_output.put_line('['||i||']'||' FEATURE_LAYER_ID   = '||
feature_ids(i).feature_layer_id);
    dbms_output.put_line('['||i||']'||' FEATURE_ID         = '||
feature_ids(i).feature_id);
    dbms_output.put_line('---');
  END LOOP;
END;
/
```

6.78 SDO_NET.GET_PARTITION_SIZE

Format

```
SDO_NET.GET_PARTITION_SIZE(
  network              IN VARCHAR2,
  partition_id         IN VARCHAR2,
  link_level           IN NUMBER DEFAULT 1,
  include_user_data    IN VARCHAR2 DEFAULT 'FALSE',
  include_spatial_data IN VARCHAR2 DEFAULT 'FALSE'
) 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 (default = 1). The 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.

include_user_data
`TRUE` if the size should include any user data associated with the network elements represented in each BLOB, or `FALSE` (the default) 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` (the default) 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 Network Analysis Using Load on Demand.

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
```

6.79 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.80 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 5-38 in xxx_SDO_NETWORK_METADATA Views).

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
```

6.81 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
```

6.82 SDO_NET.GET_PHANTOM_FEATURES

Format

```
SDO_NET.GET_PHANTOM_FEATURES(
  feature_layer_id  IN NUMBER
) RETURN SDO_NUMBER_ARRAY;
```

Description

Returns the IDs of phantom features in a feature layer. A phantom feature is a feature that references nonexistent network elements (nodes or links).

Parameters

feature_layer_id
ID of the feature layer containing the features.

Usage Notes

To delete the phantom features in a feature layer, use the SDO_NET.DELETE_PHANTOM_FEATURES procedure.

Examples

The following example gets and displays the feature IDs of phantom features in a specified feature layer.

```
DECLARE
  feature_layer_id NUMBER;
  feature_ids SDO_NUMBER_ARRAY;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  feature_ids := sdo_net.get_phantom_features(feature_layer_id);
  dbms_output.put_line('Phantom Features:');
  for i in 1..feature_ids.count loop
    dbms_output.put_line('['||i||'] '||feature_ids(i));
  end loop;
END;
/
```

6.83 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)
```

6.84 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 Network Hierarchy.

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
```

6.85 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
```

6.86 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 Network Data Model Concepts.

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
```

6.87 SDO_NET.IS_NODE_IN_PATH

Format

```
SDO_NET.IS_NODE_IN_PATH(
  network  IN VARCHAR2,
  path_id  IN NUMBER,
  node_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
```

6.88 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 Network Data Model Concepts.

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
```

6.89 SDO_NET.LOAD_CONFIG

Format

```
SDO_NET.LOAD_CONFIG(
  file_directory  IN VARCHAR2,
  file_name       IN VARCHAR2);
```

Description

Loads (or reloads) the configuration for load on demand Java stored procedures from the specified XML file. The load on demand configuration is mainly for partition BLOB translation and partition cache configuration. (The Java stored procedures are classes in the package `oracle.spatial.network.lod`.)

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 Configuring the Partition Cache.

Examples

The following example loads the load on demand configuration from a specified XML file.

```
EXECUTE SDO_NET.LOAD_CONFIG('WORK_DIR', 'netlodcfg.xml');
```

6.90 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 DEFAULT 'A',
  link_level           IN NUMBER DEFAULT 1);
```

or

```
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 DEFAULT 'A',
link_level           IN NUMBER DEFAULT 1,
part_size_tolerance  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.

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 Partition Table.

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` (the default) 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.

link_level
Network link level on which to perform the partitioning (default = 1). The 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.

part_size_tolerance
Allowed tolerance in partition size expressed as a decimal fraction of `max_num_nodes`. Must be from 0 to 1. This parameter allows the partitioning procedure to create partitions with sizes larger than the one specified by `max_num_nodes`, thereby providing the flexibility to generate partitions with reduced inter-connectivity. For example, if `max_num_nodes` is 5000 and

part_size_tolerance is 0.1, a partition can include up to 5500 (5000+500 because 500 is 0.1*5000) nodes.

Usage Notes

The format with the part_size_tolerance parameter enables you to partition logical networks with a primary focus on reducing the inter-connectivity among partitions while keeping the edge-cut small.

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');
```

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 5500 because of the combination of the max_num_nodes and part_size_tolerance values (5000 + 0.1*5000 = 5500). Information about the operation is written (open_mode => 'w') to the my_logical_part.log file located in the location associated with the directory object named LOG_DIR, replacing any existing file with that name in that location.

```
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 => 'w',
  part_size_tolerance => 0.1);
```

6.91 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 DEFAULT 'A',
  link_level           IN NUMBER DEFAULT 1,
  part_size_tolerance  IN NUMBER DEFAULT 0);
```

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.

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 Partition Table.

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.
If the `part_size_tolerance` value is greater than 0, the maximum number of nodes to include in each partition is increased, as explained in the description of that parameter.

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` (the default) 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.

link_level
Network link level on which to perform the partitioning (default = 1). The 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.

part_size_tolerance
Allowed tolerance in partition size expressed as a percentage of `max_num_nodes`. Must be from 0 (the default) to 100.
A `part_size_tolerance` value greater than 0 effectively raises the `max_num_nodes` value. For example, if `max_num_nodes` is 5000 and you specify `part_size_tolerance` as 10, then the actual maximum number of nodes to include in each partition is 5500 (5000 + 500, because 500 is 10 percent of 5000). In deciding whether to use `part_size_tolerance` and what value to specify, consider the cache size and the probability of related nodes being placed in different partitions.

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');
```

6.92 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 xxx_SDO_NETWORK_METADATA Views.)

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
```

6.93 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
```

6.94 SDO_NET.POST_XML

Format

```
SDO_NET.POST_XML(
  url      IN VARCHAR2,
  request  IN XMLTYPE
) RETURN XMLTYPE;
```

Description

Sends an XML request to a URL, and returns the XML response.

Parameters

url
Uniform resource locator (URL) to receive the request.

request
Request in XML form.

Usage Notes

For information about the XML API to the Network Data Model, see Network Data Model XML Interface.

Examples

The following example specifies an XML request, and sends it to a URL and returns the XML response, which it then displays.

```
DECLARE
  xml_request varchar2(4000);
  ndmws_url varchar2(4000);
  xml_response xmltype;
BEGIN
  xml_request :=
'<?xml version="1.0" ?>
<networkAnalysisRequest
   xmlns="http://xmlns.oracle.com/spatial/network"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:gml="http://www.opengis.net/gml">
  <networkName>HILLSBOROUGH_NETWORK2</networkName>
  <shortestPath>
    <startPoint>
      <nodeID>1533</nodeID>
    </startPoint>
    <endPoint>
      <nodeID>10043</nodeID>
    </endPoint>
    <subPathRequestParameter>
      <cost> true </cost>
      <isFullPath> true </isFullPath>
      <startLinkIndex> true </startLinkIndex>
      <startPercentage> true </startPercentage>
      <endLinkIndex> true </endLinkIndex>
      <endPercentage> true </endPercentage>
    <pathRequestParameter>
      <cost> true </cost>
      <isSimple> true </isSimple>
      <startNodeID>true</startNodeID>
      <endNodeID>true</endNodeID>
      <noOfLinks>true</noOfLinks>
      <linksRequestParameter>
        <onlyLinkID>true</onlyLinkID>
      </linksRequestParameter>
      <nodesRequestParameter>
        <onlyNodeID>true</onlyNodeID>
      </nodesRequestParameter>
    </pathRequestParameter>
    </subPathRequestParameter>
  </shortestPath>
</networkAnalysisRequest>';
  ndmws_url := 'http://localhost:7001/SpatialWS-SpatialWS-context-root/
SpatialWSXmlServlet';
  xml_response := sdo_net.POST_XML(ndmws_url, XMLTYPE(xml_request));
  dbms_output.put_line(xml_response.getStringVal());
END;
/
```

6.95 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 xxx_SDO_NETWORK_CONSTRAINTS Views).

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 Network Constraints.

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.

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;
/
```

6.96 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 xxx_SDO_NETWORK_METADATA Views.)

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
```

6.97 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);
```

6.98 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);
```

6.99 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 DEFAULT 'A',
  link_level           IN NUMBER DEFAULT 1);
```

Description

Partitions a spatial network, and stores the information in the partition table.

Parameters

network
Network name.

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 Partition Table.

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` (the default) 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.

link_level
Network link level on which to perform the partitioning (default = 1). The 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.

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');
```

6.100 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 xxx_SDO_NETWORK_METADATA Views.)

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
```

6.101 SDO_NET.UPDATE_FEATURE

Format

```
SDO_NET.UPDATE_FEATURE(
  feature_layer_id   IN NUMBER,
  feature_id         IN NUMBER,
  feature_elements   IN SDO_NET_FEAT_ELEM_ARRAY DEFAULT NULL,
  child_feature_ids  IN SDO_NET_LAYER_FEAT_ARRAY DEFAULT NULL,
  check_integrity    IN BOOLEAN DEFAULT TRUE);
```

Description

Updates a feature in a feature layer.

Parameters

feature_layer_id
ID of the feature layer to which to update the feature.

feature_id
ID of the feature to be updated.

feature_elements
Feature elements of the feature to add to any existing feature elements. If this parameter is null, the existing feature elements are not changed. If this parameter in *empty*, any existing feature elements are removed. (The SDO_NET_FEAT_ELEM_ARRAY type is described in Data Types Used for Feature Modeling.)

child_feature_ids
Child features of the feature that are to add to any existing child features. If this parameter is null, the existing child features are not changed. If this parameter in *empty*, any existing parent relationships for this feature with child features are removed. (The SDO_NET_LAYER_FEAT_ARRAY type is described in Data Types Used for Feature Modeling.)

check_integrity
TRUE (the default) checks if the input network elements exist; and if any do not exist, an error is generated. FALSE does not check if the input network elements exist.

Usage Notes

To add a feature to a feature layer, use the SDO_NET.ADD_FEATURE procedure.

A feature layer ID is automatically generated for the feature layer.

Examples

The following example updates a specified feature by defining two feature elements and adding them.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  elements SDO_NET_FEAT_ELEM_ARRAY := SDO_NET_FEAT_ELEM_ARRAY();
  link_id NUMBER := 1314;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  elements.extend;
  elements(1) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.7, null);
  elements.extend;
  elements(2) := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.8, null);
  sdo_net.update_feature(feature_layer_id, feature_id, elements, null);
END;
/
```

6.102 SDO_NET.UPDATE_FEATURE_ELEMENT

Format

```
SDO_NET.UPDATE_FEATURE_ELEMENT(
  feature_layer_id IN NUMBER,
  feature_id       IN NUMBER,
  sequence_number  IN NUMBER,
  feature_element  IN SDO_NET_FEAT_ELEM,
  check_integrity  IN BOOLEAN DEFAULT TRUE);
```

Description

Updates a feature element.

Parameters

feature_layer_id
ID of the feature layer for the feature.

feature_id
ID of the feature.

sequence_number
Sequence number of the feature element to be updated.

feature_element
Feature element definition to replace the specified feature element. (The SDO_NET_FEAT_ELEM type is described in Data Types Used for Feature Modeling.)

check_integrity
`TRUE` (the default) checks if the input network elements exist; and if any do not exist, an error is generated. `FALSE` does not check if the input network elements exist.

Usage Notes

To add a feature element, use the SDO_NET.ADD_FEATURE_ELEMENT procedure; to add multiple feature elements in a single operation, use the SDO_NET.ADD_FEATURE_ELEMENTS procedure.

Examples

The following example updates the feature element at sequence number 2 on link ID 1314 of feature ID 1.

```
DECLARE
  feature_layer_id NUMBER;
  feature_id NUMBER := 1;
  element SDO_NET_FEAT_ELEM;
  link_id NUMBER := 1314;
BEGIN
  feature_layer_id := sdo_net.get_feature_layer_id('GRID', 'POI');
  element := SDO_NET_FEAT_ELEM(SDO_NET.FEAT_ELEM_TYPE_POL, link_id, 0.2, null);
  sdo_net.update_feature_element(feature_layer_id, feature_id, 1, element);
END;
/
```

6.103 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
```

6.104 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
```

6.105 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
```

6.106 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
```

6.107 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 Partition Table.

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
```

6.108 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
```

6.109 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 Subpath Table.

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_NFE Package Subprograms

The MDSYS.SDO_NFE package contains subprograms (functions and procedures) for performing network feature editing.

> **Note:**
>
> SDO_NFE subprograms are only supported if Oracle JVM is enabled on your Oracle Autonomous Database Serverless deployments. To enable Oracle JVM, see Use Oracle Java in *Using Oracle Autonomous Database Serverless* for more information.

To use these subprograms, you must understand the conceptual information in Network Data Model Graph Overview, and especially Feature Modeling Using Network Feature Editing (NFE).

- SDO_NFE.APPLY_RULE
- SDO_NFE.CLASSIFY_LINES_BY_SIDE
- SDO_NFE.CREATE_MODEL_SEQUENCE
- SDO_NFE.CREATE_MODEL_STRUCTURE
- SDO_NFE.CREATE_MODEL_UNDERLYING_NET
- SDO_NFE.CREATE_MODEL_WORKSPACE
- SDO_NFE.DELETE_ALL_FT_LAYERS
- SDO_NFE.DELETE_ALL_WORKSPACES
- SDO_NFE.DELETE_MODEL_STRUCTURE
- SDO_NFE.DELETE_MODEL_WORKSPACE
- SDO_NFE.DROP_MODEL_SEQUENCE
- SDO_NFE.DROP_MODEL_UNDERLYING_NETWORK
- SDO_NFE.GET_CONNECTION_POINT_GEOM
- SDO_NFE.GET_INTERACTION_GROUPS
- SDO_NFE.GET_LINES_MATCH_LP_RULE
- SDO_NFE.GET_LL_CONN_INTERSECTIONS
- SDO_NFE.GET_LP_CONN_INTERSECTIONS
- SDO_NFE.GET_MODEL_SEQUENCE_NAME
- SDO_NFE.GET_MODEL_TABLE_NAME
- SDO_NFE.GET_MODEL_UNDERLYING_NETWORK
- SDO_NFE.GET_NEXT_SEQUENCE_VALUE
- SDO_NFE.GET_POINTS_MATCH_LP_RULE
- SDO_NFE.IMPORT_NETWORK

7.1 SDO_NFE.APPLY_RULE

Format

```
SDO_NFE.APPLY_RULE(
     model_id  IN NUMBER,
     rule_type IN VARCHAR2,
     rule_id   IN NUMBER);
```

Description

Applies a connectivity rule over all the features contained in a specified NFE model.

Parameters

model_id
ID of the NFE model.

rule_type
Type of connectivity rule to apply: `RULE_TYPE_LINE` or `RULE_TYPE_POINT`.

rule_id
ID of the connectivity rule.

Usage Notes

The specified rule must be registered in the specified model. You can register a connectivity rule in the model tables or through the Java API.

Examples

The following example applies a line-line rule to any interacting lines in an NFE model that meet the connectivity rule identified by the rule ID 1.

```
DECLARE
  model_id  NUMBER := 1;
  rule_type VARCHAR2(1) := sdo_nfe.RULE_TYPE_LINE_LINE;
  rule_id   NUMBER := 1;
BEGIN
  sdo_nfe.apply_rule( model_id, rule_type, rule_id );
END;
/
```

7.2 SDO_NFE.CLASSIFY_LINES_BY_SIDE

Format

```
SDO_NFE.CLASSIFY_LINES_BY_SIDE(
     model_id    IN NUMBER,
     ll_rule_id  IN NUMBER,
     lines       IN NUMBER,
     lhs_indexes OUT DBMS_SQL,NUMBER_TABLE,
     rhs_indexes OUT DBMS_SQL,NUMBER_TABLE);
```

Description

Given a set of line features that match a connectivity Line-Line rule, this procedure classifies which lines lie on the left hand side of the rule and which ones on the right hand side.

Parameters

model_id
ID of the NFE model.

ll_rule_id
Connectivity Line-Line rule identifier.

lines
Set of line features that meet the rule..

lhs_indexes
Associative array where the indexes of the lines lying on the left hand side of the rule will be stored (in the form (index, index)).

rhs_indexes
Associative array where the indexes of the lines lying on the right hand side of the rule will be stored.

Usage Notes

The specified rule must be registered in the specified model. You can register a connectivity rule in the model tables or through the Java API.

Examples

The following example first gets all the interacting groups that meet the rule with ID 1 and then classifies the lines by side. Left hand side lines are output in lhs_indexes while rhs_indexes contain the rule's right hand side lines.

```
DECLARE
  model_id    NUMBER := 1;
  ll_rule_id  NUMBER := 1;
  lines       SDO_INTERACT_LINE_FEAT_ARRAY;
  lhs_indexes dbms_sql.NUMBER_TABLE;
  rhs_indexes dbms_sql.NUMBER_TABLE;
  inter_grps  SDO_INTERACTION_ARRAY;
BEGIN

  -- Get the groups of interacting features that meet the L-L Rule
  inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_LINE,
ll_rule_id );

  FOR i IN 1..inter_grps.count loop
    lines := inter_grps(i).lines;

    -- For each group, classify the lines by rule side.
    sdo_nfe.classify_lines_by_side( model_id, ll_rule_id, lines, lhs_indexes,
rhs_indexes );
  END loop;

END;
```

7.3 SDO_NFE.CREATE_MODEL_SEQUENCE

Format

```
SDO_NFE.CREATE_MODEL_SEQUENCE(
     model_id      IN NUMBER,
     owner_name    IN VARCHAR2,
     aequence_name IN VARCHAR2);
```

Description

Creates and registers a sequence for a model.

Parameters

model_id
ID of the NFE model.

owner_name
Sequence's related table.

aequence_name
Name of the sequence to be created.

Usage Notes

All the sequences for the base tables are created by the SDO_NFE.CREATE_MODEL_STRUCTURE function, but you may need to create other sequences (such as for features).

The NFE model and the sequence’s related table must exist.

Examples

The following example creates a sequence for the NFE model identified by the ID 1 and a table named FEATURES.

```
SDO_NFE.CREATE_MODEL_SEQUENCE('1','features','features_seq')
```

7.4 SDO_NFE.CREATE_MODEL_STRUCTURE

Format

```
SDO_NFE.CREATE_MODEL_STRUCTURE(
     model_name    IN VARCHAR2,
     edition_mode  IN NUMBER,
     versionable   IN VARCHAR2
) RETURN NUMBER;
```

Description

Creates the tables and metadata for an NFE model.

Parameters

model_name
Name to be given to the lNFE model.

edition_mode
Edition mode. Must be `SDO_NFE.FROM_SCRATCH` or `SDO_NFE.OVER_EXIST_NETWORK`.

versionable
The string value `Y` if the model will be versionable, otherwise `N`.

Usage Notes

This function returns the new model's ID value.

Examples

The following example creates a versionable model named `MODEL01` with the `SDO_NFE.FROM_SCRATCH` edition mode.

```
DECLARE
  model_id      NUMBER;
  model_name    VARCHAR2(50) := 'MODEL01';
  edition_mode  NUMBER       := SDO_NFE.FROM_SCRATCH;
  versionable   VARCHAR2(1)  := 'Y';
BEGIN
  model_id := SDO_NFE.create_model_structure( model_name, edition_mode, versionable );
END;
/
```

7.5 SDO_NFE.CREATE_MODEL_UNDERLYING_NET

Format

```
SDO_NFE.CREATE_MODEL_UNDERLYING_NET(
     model_id             IN NUMBER,
     network_name         IN VARCHAR2,
     num_hierarchy_levels IN NUMBER,
     is_directed          IN BOOLDEAN,
     node_with_costs      IN BOOLEAN);
```

Description

Creates a spatial network and associates it to the specified NFE Model. It also creates sequences for its nodes, links, and paths, and registers them in the model's metadata.

Parameters

model_id
ID of the NFE model.

network_name
Name of the network to be created.

num_hierarchy_levels
Number of hierarchical levels for the network.

is_directed
`TRUE` if the network is directed.

node_with_costs
`TRUE` if the network's nodes contain cost values.

Usage Notes

An NFE model with the specified ID must exist. The geometry metadata must be registered for the newly created network's nodes and links tables.

Examples

The following example creates an underlying network for an NFE model and registers the geometry metadata for the network's links and nodes tables.

```
DECLARE
  model_id           NUMBER    := 1;
  network_name          VARCHAR2(50) := 'MODEL01';
  num_hierarchy_levels  NUMBER := 1;
  is_directed           VARCHAR2(10) := 'TRUE';
  node_with_costs       VARCHAR2(10) := 'TRUE';
BEGIN
-- create underlying network
  SDO_NFE.create_model_underlying_net( model_id, network_name, num_hierarchy_levels,
is_directed, node_with_costs );
-- register links and nodes tables geom metadata
  SDO_NET.insert_geom_metadata(network_name, SDO_DIM_ARRAY(SDO_DIM_ELEMENT('LONGITUDE',
-180, 180, 0.5), SDO_DIM_ELEMENT('LATITUDE', -90, 90, 0.5)), 8307);
END;
/
```

7.6 SDO_NFE.CREATE_MODEL_WORKSPACE

Format

```
SDO_NFE.CREATE_MODEL_WORKSPACE(
     model_id              IN NUMBER,
     parent_workspace_name IN VARCHAR2,
     workspace_name        IN VARCHAR2,
     is_mbr    IN VARCHAR2,
     is_locked IN VARCHAR2,
     lower_x   IN NUMBER,
     lower_y   IN NUMBER,
     upper_x   IN NUMBER,
     upper_y   IN NUMBER);
```

Description

Creates a new workspace and relates it to an NFE model.

Parameters

model_id
ID of the NFE model.

parent_workspace_name
Name of the parent workspace.

workspace_name
Name of the workspace.

is_mbr
The string `TRUE` if the workspace is created for a minimum bounding rectangle (MBR) rectangular area of the model.

is_locked
The string `TRUE` if the workspace is locked.

lower_x
The lower x ordinate of the workspace MBR.

lower_y
The lower y ordinate of the workspace MBR.

upper_x
The upper x ordinate of the workspace MBR.

upper_y
The upper y ordinate of the workspace MBR.

Usage Notes

The NFE model must have been created with the `versionable` option enabled.

Examples

The following example creates a workspace for an NFE model.

```
DECLARE
  model_id        NUMBER := 1;
  parent_ws_name  VARCHAR2(30) := 'LIVE';
  workspace_name  VARCHAR2(30) := 'PROJECT_V1';
  is_mbr          VARCHAR2(1) := 'Y';
  is_locked       VARCHAR2(1) := 'N';
  lower_x NUMBER := -15.575;
  lower_y       NUMBER :=  15.575;
  upper_x NUMBER := -12.825;
  upper_y NUMBER :=  28.975;
BEGIN
  SDO_NFE.create_model_workspace(model_id, parent_ws_name, workspace_name, is_mbr,
is_locked, lower_x, lower_y, upper_x, upper_y);
END;
/
```

7.7 SDO_NFE.DELETE_ALL_FT_LAYERS

Format

```
SDO_NFE.DELETE_ALL_FT_LAYERS(
     model_id  IN NUMBER);
```

Description

Drops all content in a specified NFE model.

Parameters

model_id
ID of the NFE model.

Usage Notes

This procedure is mainly used before deleting a model and its structure from the database.

Examples

The following example deletes all content from the model with the ID value 1.

```
EXECUTE SDO_NFE.DELETE_ALL_FT_LAYERS(1);
```

7.8 SDO_NFE.DELETE_ALL_WORKSPACES

Format

```
SDO_NFE.DELETE_ALL_WORKSPACES(
     model_id  IN NUMBER);
```

Description

Dropa all the workspaces related to the specified NFE model, along with their relationship to the model.

Parameters

model_id
ID of the NFE model.

Usage Notes

This procedure is mainly used before deleting a model and its structure from the database.

Examples

The following example deletes all workspaces related to the model with ID value 1.

```
EXECUTE SDO_NFE.DELETE_ALL_WORKSPACES(1);
```

7.9 SDO_NFE.DELETE_MODEL_STRUCTURE

Format

```
SDO_NFE.DELETE_MODEL_STRUCTURE(
     model_id  IN NUMBER);
```

Description

Drops all tables in a specified NFE model, and deletes the metadata records for the model.

Parameters

model_id
ID of the NFE model.

Usage Notes

Before using this procedure, you may need to do the following:

- Delete model's workspaces by executing the SDO_NFE.DELETE_ALL_WORKSPACES procedure.
- Delete the model's feature layers by executing the SDO_NFE.DELETE_ALL_FT_LAYERS procedure.
- If f the model's edition mode is `SDO_NFE.FROM_SCRATCH`, delete the underlying network.

Examples

The following example the structure of the model with the ID value 1.

```
EXECUTE SDO_NFE.DELETE_MODEL_STRUCTURE(1);
```

7.10 SDO_NFE.DELETE_MODEL_WORKSPACE

Format

```
SDO_NFE.DELETE_MODEL_WORKSPACE(
     model_id       IN NUMBER,
     workspace_name IN VARCHAR2);
```

Description

Drops a workspace along with its relationship with the specified NFE model.

Parameters

model_id
ID of the NFE model.

workspace_name
Name of the workspace.

Usage Notes

`workspace_name` must be the name of an existing workspace under the specified NFE model. All branches of the workspace are removed.

The relationship with the model is deleted from xxx_SDO_NFE_MODEL_WORKSPACE views.

Examples

The following example deletes the workspace named `PROJECT_V4` from the NFE model with the ID 1

```
EXECUTE SDO_NFE.DELETE_MODEL_WORKSPACE(1, 'PROJECT_V4');
```

7.11 SDO_NFE.DROP_MODEL_SEQUENCE

Format

```
SDO_NFE.DROP_MODEL_SEQUENCE(
     model_id  IN NUMBER,
     seq_name  IN VARCHAR2);
```

Description

Drops a sequence along with its relationship with the specified NFE model.

Parameters

model_id
ID of the NFE model.

seq_name
Name of the sequence.

Usage Notes

The relationship of the sequence with the model is deleted from the table registered in SEQUENCE_REG_TAB from the xxx_SDO_NFE_MODEL_METADATA views.

Examples

The following example deletes the sequence named `PIPES_FTLAY_ID_SEQ` from the NFE model with the ID 1.

```
EXECUTE SDO_NFE.DROP_MODEL_SEQUENCE(1, 'PIPES_FTLAY_ID_SEQ');
```

7.12 SDO_NFE.DROP_MODEL_UNDERLYING_NETWORK

Format

```
SDO_NFE.DROP_MODEL_UNDERLYING_NETWORK
     network_name   IN VARCHAR2);
```

Description

Drops a network and removes its relationship with any NFE model.

Parameters

network_name
Name of the network.

Usage Notes

The network must be bound to at least one NFE model..

Examples

The following example drops the network named `PIPES` and removes its relationship with any existing NFE model.

```
EXECUTE SDO_NFE.DROP_MODEL_UNDERLYING_NETWORK('PIPES');
```

7.13 SDO_NFE.GET_CONNECTION_POINT_GEOM

Format

```
SDO_NFE.GET_CONNECTION_POINT_GEOM(
     conn_intersection   IN SDO_INTERACTION
     ) RETURN SDO_GEOMETRY;
```

Description

Given a group of interacting features (lines and/or points), calculates and returns the geometry of the point that must connect them.

Parameters

conn_intersection
Interaction group of features. Set of line and/or point features that interact at a common spatial point. (The SDO_INTERACTION type is described in Data Types Used for NFE Connectivity Rules.)

Usage Notes

This function is mainly used over a validated group of features that must be connected because of the requirement of a connectivity rule (see NFE Rules). To get this group of features, use SDO_NFE.GET_LP_CONN_INTERSECTIONS for Line-Point Rules or SDO_NFE.GET_LL_CONN_INTERSECTIONS for Line-Line Rules.

Examples

The following example gets the connection point geometry for each interacting group that meets the given line-point rule.

```
DECLARE
  model_id        NUMBER := 1;
  lp_rule_id      NUMBER := 1;
  inter_grps          SDO_INTERACTION_ARRAY;
  conn_point_geom     SDO_GEOMETRY;
BEGIN
  -- Get the groups of interacting features that meet the L-P Rule in the model
 inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_POINT,
lp_rule_id );
  -- Iterate through the interacting groups
 FOR i IN 1..inter_grps.count loop
  -- Get the connection point geometry for each interacting group
   conn_point_geom := sdo_nfe.get_connection_point_geom( inter_groups(i));
 END loop;
END;
/
```

7.14 SDO_NFE.GET_INTERACTION_GROUPS

Format

```
SDO_NFE.GET_INTERACTION_GROUPS(
     model_id   IN SDO_NUMBER,
     rule_type  IN VARCHAR2,
     rule_id    IN NUMBER
     ) RETURN SDO_INTERACTION_ARRAY;
```

Description

Returns an array of groups of all features that are interacting at spatial points where the specified connectivity rule is being met.

Parameters

model_id
NFE model identifier.

rule_type
Connectivity rule type. Possible values: `SDO_NFE.RULE_TYPE_LINE_LINE` or `SDO_NFE.RULE_TYPE_LINE_POINT`.

rule_id
Rule identifier. Must be a value from the LINE_LINE_RULE or LINE_POINT_RULE table.

Usage Notes

This function returns an object of type SDO_INTERACTION_ARRAY, which is described in Data Types Used for NFE Connectivity Rules.

Each group of the interacting features returned by this function is composed of all the line and point features that interact at a specific spatial point where the specified rule is being met.

By returning the whole group of all interacting features at specific points, this function can help you if you want to create a customized way of connecting features depending on which other features (meeting the rule or not) are taking part in a specified interaction point. (See the discussion of rule decision handlers under NFE Rules.)

Examples

The following example gets the interacting groups which met the given line-point rule.

```
DECLARE
  model_id        NUMBER := 1;
  lp_rule_id      NUMBER := 1;
  inter_grps          SDO_INTERACTION_ARRAY;
BEGIN
 inter_grps := sdo_nfe.get_interaction_groups( model_id,
END;
/
```

7.15 SDO_NFE.GET_LINES_MATCH_LP_RULE

Format

```
SDO_NFE.GET_LINES_MATCH_LP_RULE(
     model_id    IN SDO_NUMBER,
     lp_rule_id  IN NUMBER,
     lines       IN SDO_INTERACT_LINE_FEAT_ARRAY,
     ) RETURN DBMS_SQL.NUMBER_TABLE;
```

Description

Given an set of line features, calculates the group of them that match a connectivity line-point rule. Returns a DBMS_SQL.NUMBER_TABLE object with the indexes of the lines in the input array that match the line-point rule.

Parameters

model_id
NFE model identifier.

lp_rule_id
Connectivity line-point rule identifier. Must exist in the LINE_POINT_RULE table.

lines
Array of line features where the search will take place. (The SDO_INTERACT_LINE_FEAT_ARRAY type is described in Data Types Used for NFE Connectivity Rules.)

Usage Notes

This function is mainly used after the SDO_NFE.GET_INTERACTION_GROUPS function, which returned a group of mixed line features where some line features matched a specific connectivity rule and some did not.

Examples

The following example finds the lines that meet a connectivity line-point rule from interacting groups.

```
DECLARE
  model_id    NUMBER := 1;
  lp_rule_id  NUMBER := 1;
  lines       SDO_INTERACT_LINE_FEAT_ARRAY;
  match_lines dbms_sql.NUMBER_TABLE;
  inter_grps  SDO_INTERACTION_ARRAY;
BEGIN
  -- find interaction groups
  inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_LINE,
1 );

  FOR i IN 1..inter_grps.count loop
    lines := inter_grps(i).lines;
    match_lines := sdo_nfe.get_lines_match_lp_rule( model_id, lp_rule_id, lines );
  END loop;

END;
/
```

7.16 SDO_NFE.GET_LL_CONN_INTERSECTIONS

Format

```
SDO_NFE.GET_LL_CONN_INTERSECTIONS(
     model_id               IN SDO_NUMBER,
     ll_rule_id             IN NUMBER,
     interaction_grp        IN OUT SDO_INTERACTION,
     rule_lhs_lines_indexes IN DBMS_SQL.NUMBER_TABLE,
     rule_rhs_lines_indexes IN DBMS_SQL.NUMBER_TABLE,
     rule_points_indexes    IN DBMS_SQL.NUMBER_TABLE,
     ) RETURN SDO_INTERACTION_ARRAY;
```

Description

Given a group of interacting features (lines and points) this function calculates subgroups of these features that can be connected according to the connectivity line-line rule specified, and returns the set of connectable features groups.

Parameters

model_id
NFE model identifier.

ll_rule_id
Connectivity line-line rule identifier.. Must exist in the LINE_LINE_RULE table.

interaction_grp
Group of interacting features. (The SDO_INTERACTION type is described in Data Types Used for NFE Connectivity Rules.)

rule_lhs_lines_indexes
Among the line features in the interacting group, indexes of the lines that match the left hand side of the line-line rule.

rule_rhs_lines_indexes
Among the line features in the interacting group, indexes of the lines that match the right hand side of the line-line rule.

rule_points_indexes
Among the point features in the interacting group, indexes of the points that match the point feature specification in the line-line rule. These points are the ones to be considered in the conformation of connectable groups.

Usage Notes

This function returns an SDO_INTERACTION_ARRAY object. (The SDO_INTERACTION_ARRAY type is described in Data Types Used for NFE Connectivity Rules.)

The indexes of LHS and RHS lines can be obtained with the SDO_NFE.CLASSIFY_LINES_BY_SIDE procedure. The indexes of the points can be obtained with the SDO_NFE.GET_POINTS_MATCH_LP_RULE function.

This function is registered by default in the Rule Decision Handlers Table when a line-line rule is created in a model (using the Java API). However, this function can be replaced by any other user function that calculates the group of connectable features in a customized way. See the information about Rule Decision Handlers under NFE Rules for information about customizing connections (rule decision handlers).

Examples

The following example gets the set of connectable feature groups for each interacting group that match a given line-line rule.

```
DECLARE
  model_id    NUMBER := 1;
  ll_rule_id  NUMBER := 1;
  rule_lhs_lines_indexes  dbms_sql.NUMBER_TABLE;
  rule_rhs_lines_indexes  dbms_sql.NUMBER_TABLE;
  rule_points_indexes     dbms_sql.NUMBER_TABLE;
  conn_interacs           SDO_INTERACTION_ARRAY;
```

```
  inter_grps              SDO_INTERACTION_ARRAY;
BEGIN
-- Get the groups of interacting features that meet the L-L Rule in the model
  inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_LINE,
ll_rule_id );
  FOR i IN 1..inter_grps.count loop
    -- Classify the line features by side in the L-L rule (LHS, RHS).
    sdo_nfe.classify_lines_by_side( model_id, ll_rule_id, inter_grps(i).lines,
rule_lhs_lines_indexes, rule_rhs_lines_indexes );
    -- Get the specific point features that match the L-L rule.
    rule_points_indexes := sdo_nfe.get_points_match_lp_rule( model_id,  1,
inter_grps(i).points );
    -- Get the group of features that can be connected according the L-L rule.
    conn_interacs := sdo_nfe.get_ll_conn_intersections( model_id, ll_rule_id,
inter_grps(i), rule_lhs_lines_indexes, rule_rhs_lines_indexes, rule_points_indexes);
  END loop;
END;
/
```

7.17 SDO_NFE.GET_LP_CONN_INTERSECTIONS

Format

```
SDO_NFE.GET_LP_CONN_INTERSECTIONS(
     model_id               IN SDO_NUMBER,
     lp_rule_id             IN NUMBER,
     interaction_grp        IN OUT SDO_INTERACTION,
     rule_lhs_lines_indexes IN DBMS_SQL.NUMBER_TABLE,
     rule_rhs_lines_indexes IN DBMS_SQL.NUMBER_TABLE,
     rule_points_indexes    IN DBMS_SQL.NUMBER_TABLE,
     ) RETURN SDO_INTERACTION_ARRAY;
```

Description

Given a group of interacting features (lines and points) this function calculates subgroups of these features that can be connected according to the connectivity line-point rule specified, and returns the set of connectable features groups.

Parameters

model_id
NFE model identifier.

lp_rule_id
Connectivity line-point rule identifier.. Must exist in the LINE_POINT_RULE table.

interaction_grp
Group of interacting features. (The SDO_INTERACTION type is described in Data Types Used for NFE Connectivity Rules.)

rule_lhs_lines_indexes
Among the line features in the interacting group, indexes of the lines that match the left hand side of the line-point rule.

rule_rhs_lines_indexes
Among the line features in the interacting group, indexes of the lines that match the right hand side of the line-point rule.

rule_points_indexes
Among the point features in the interacting group, indexes of the points that match the point feature specification in the line-point rule. These points are the ones to be considered in the conformation of connectable groups.

Usage Notes

This function returns an SDO_INTERACTION_ARRAY object. (The SDO_INTERACTION_ARRAY type is described in Data Types Used for NFE Connectivity Rules.)

The indexes of LHS and RHS lines can be obtained with the SDO_NFE.CLASSIFY_LINES_BY_SIDE procedure. The indexes of the points can be obtained with the SDO_NFE.GET_POINTS_MATCH_LP_RULE function.

This function is registered by default in the Rule Decision Handlers Table when a line-point rule is created in a model (using the Java API). However, this function can be replaced by any other user function that calculates the group of connectable features in a customized way. See the information about Rule Decision Handlers under NFE Rules for information about customizing connections (rule decision handlers).

Examples

The following example gets the group of feature that can be connected according to a given line-point rule for each interacting group.

```
DECLARE
  model_id        NUMBER := 1;
  lp_rule_id      NUMBER := 1;
  rule_lines_indexes  dbms_sql.NUMBER_TABLE;
  rule_points_indexes dbms_sql.NUMBER_TABLE;
  conn_interacs       SDO_INTERACTION_ARRAY;
  inter_grps          SDO_INTERACTION_ARRAY;
BEGIN
  -- Get the groups of interacting features that meet the L-P Rule in the model
  inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_POINT,
lp_rule_id );

  -- For each group:
  FOR i IN 1..inter_grps.count loop
    -- Get the specific line features that match the L-P rule.
    rule_lines_indexes  := sdo_nfe.get_lines_match_lp_rule( model_id, lp_rule_id,
inter_grps(i).lines );

    -- Get the specific point features that match the L-P rule.
    rule_points_indexes := sdo_nfe.get_points_match_lp_rule( model_id,  lp_rule_id,
inter_grps(i).points );

    -- Get the group of features that can be connected according the L-P rule.
    conn_interacs := sdo_nfe.get_lp_conn_intersections( model_id, lp_rule_id,
inter_grps(i), rule_lines_indexes, rule_points_indexes );
  END loop;
END;
/
```

7.18 SDO_NFE.GET_MODEL_SEQUENCE_NAME

Format

```
SDO_NFE.GET_MODEL_SEQUENCE_NAME(
     model_id  IN SDO_NUMBER,
     tab_name  IN VARCHAR2
     ) RETURN VARCHAR2;
```

Description

Returns the sequence name for the specified model's table.

Parameters

model_id
NFE model identifier.

tab_name
Table name for the model.

Usage Notes

The table name must exist in the TABLE_REG_TAB table, and the name of its sequence must exist in the SEQUENCE_REG_TAB table. When a new model is created using SDO_NFE.CREATE_MODEL_STRUCTURE, all the model's tables and sequences are automatically registered in the appropriate views and tables. When SDO_NFE.CREATE_MODEL_SEQUENCEis executed, a sequence for the model's table is registered.

Examples

The following example gets the sequence name defined for the table that holds the feature classes of the NFE model whose ID is 1.

```
SELECT SDO_NFE.GET_MODEL_SEQUENCE_NAME(1, sdo_nfe.get_model_table_name(1,
SDO_NFE.FT_CLASS));
```

7.19 SDO_NFE.GET_MODEL_TABLE_NAME

Format

```
SDO_NFE.GET_MODEL_TABLE_NAME(
     model_id    IN SDO_NUMBER,
     table_type  IN VARCHAR2
     ) RETURN VARCHAR2;
```

Description

Returns the name of the table of a specified type for an NFE model.

Parameters

model_id
NFE model identifier.

table_type
Type of table whose name is to be returned. For example, the value for the feature classes table is `SDO_NFE.FT_CLASS`.

Usage Notes

The table name must exist in the TABLE_REG_TAB table, and the name of its sequence must exist in the SEQUENCE_REG_TAB table. When a new model is created using SDO_NFE.CREATE_MODEL_STRUCTURE, the names of all of the model's tables and sequences are automatically registered in the appropriate views and tables.

Examples

The following example gets the name of the table that holds the feature classes in the NFE model with the ID 1.

```
SELECT SDO_NFE.GET_MODEL_TABLE_NAME(1, SDO_NFE.FT_CLASS);
```

7.20 SDO_NFE.GET_MODEL_UNDERLYING_NETWORK

Format

```
SDO_NFE.GET_MODEL_UNDERLYING_NETWORK(
     model_id    IN SDO_NUMBER
     ) RETURN VARCHAR2;
```

Description

Returns the name of the network that is associated with an NFE model.

Parameters

model_id
NFE model identifier.

Usage Notes

A network is associated with an NFE model during the creation process, either when using the SDO_NFE.CREATE_MODEL_UNDERLYING_NET for models in the SDO_NFE.FROM_SCRATCH mode, or using SDO_NFE.SET_MODEL_UNDERLYING_NETWORK for models in the SDO_NFE.OVER_EXIST_NETWORK mode.

Examples

The following example gets the underlying network associated with an existing NFE model.

```
SELECT SDO_NFE.get_model_underlying_network(1) FROM DUAL;
```

7.21 SDO_NFE.GET_NEXT_SEQUENCE_VALUE

Format

```
SDO_NFE.GET_NEXT_SEQUENCE_VALUE(
     sequence_name        IN VARCHAR2,
     seq_value_increment  IN NUMBER
     ) RETURN NUMBER;
```

Description

Returns the value resulting from adding the value of the second parameter to the current value of the specified sequence.

Parameters

sequence_name
Name of the sequence.

seq_value_increment
Integer value to be added to the current value of `sequence_name`. (If the specified value is negative, it is subtracted from the current value.)

Usage Notes

This function does not change the INCREMENT BY value of the specified sequence or the current value of that sequence.

This function can be used to manage a block of consecutive sequence numbers.

Examples

The following example returns the value that would result from adding 10 to the current value of a sequence named MY_SEQ.

```
SELECT SDO_NFE.GET_NEXT_SEQUENCE_VALUE('my_seq', 10) FROM DUAL;
```

If the current value of MY_SEQ is 100, this example returns the value 110 (100 + 10).

7.22 SDO_NFE.GET_POINTS_MATCH_LP_RULE

Format

```
SDO_NFE.GET_POINTS_MATCH_LP_RULE(
     model_id    IN SDO_NUMBER,
     lp_rule_id  IN NUMBER,
     points      IN SDO_INTERACT_POINT_FEAT_ARRAY,
     ) RETURN DBMS_SQL.NUMBER_TABLE;
```

Description

Given an set of point features, this function calculates the group of them that match a connectivity line-point rule. Returns a DBMS_SQL.NUMBER_TABLE object with the indexes of the points in the input array that match the line-point rule.

Parameters

model_id
NFE model identifier.

lp_rule_id
Connectivity line-point rule identifier. Must exist in the LINE_POINT_RULE table.

points
Array of point features where the search will take place. (The SDO_INTERACT_POINT_FEAT_ARRAY type is described in Data Types Used for NFE Connectivity Rules.)

Usage Notes

This function is mainly used after the SDO_NFE.GET_INTERACTION_GROUPS function, which returned a group of mixed line features where some line features matched a specific connectivity rule and some did not.

Examples

The following example gets the specific point features that match a line-point rule.

```
DECLARE
  model_id        NUMBER := 1;
  lp_rule_id      NUMBER := 1;
  rule_points_indexes dbms_sql.NUMBER_TABLE;
  inter_grps          SDO_INTERACTION_ARRAY;
BEGIN
  -- Get the groups of interacting features that meet the L-P Rule in the model
  inter_grps := sdo_nfe.get_interaction_groups( model_id, sdo_nfe.RULE_TYPE_LINE_POINT,
lp_rule_id );

  -- For each group:
  FOR i IN 1..inter_grps.count loop
    -- Get the specific point features that match the L-P rule.
    rule_points_indexes := sdo_nfe.get_points_match_lp_rule( model_id,  lp_rule_id,
inter_grps(i).points );
  END loop;
END;
/
```

7.23 SDO_NFE.IMPORT_NETWORK

Format

```
SDO_NFE.IMPORT_NETWORK(
     model_id          IN NUMBER,
     model_id          IN NUMBER,
     network_from      IN VARCHAR2,
     line_ft_layer_id  IN NUMBER,
     line_ft_class_id  IN NUMBER,
     point_ft_layer_id IN NUMBER,
     point_ft_class_id IN NUMBER);
```

Description

Copies the network elements from an existing network to the underlying network of an NFE model (created in the SDO_NFE.FROM_SCRATCH mode), translating every link in line features from the line feature class (*line_ft_class_id*), and every node in point features from the point feature class (*point_ft_class_id*)..

Parameters

model_id
NFE model identifier.

network_from
Name of the network to be imported.

line_ft_layer_id
Feature layer ID for the newly created line features (created from the link elements).

line_ft_class_id
Feature class ID for the newly created line features.

point_ft_layer_id
Feature layer ID for the newly created point features (created from the node elements).

point_ft_class_id
Feature class ID for the newly created point features.

Usage Notes

The feature classes for the line and point features must already exist in the NFE model.

Examples

The following example imports a network named `NET01` to a model identified by the ID 1. Lines and point features will be created for every link and node using the feature layers 10 and 11 and the feature classes 5 and 6.

```
EXECUTE SDO_NFE.import_network(1, 'NET01', 10, 5, 11, 6);
```

7.24 SDO_NFE.SET_MODEL_UNDERLYING_NETWORK

Format

```
SDO_NFE.SET_MODEL_UNDERLYING_NETWORK(
     model_id     IN SDO_NUMBER
     network_name IN VARCHAR2);
```

Description

Associates a network as the underlying network of an NFE model. (The model must have been created in the SDO_NFE.OVER_EXIST_NETWORK mode.)

Parameters

model_id
NFE model identifier.

network_name
Name of the network to be associated with the model.

Usage Notes

See also the SDO_NFE.GET_MODEL_UNDERLYING_NETWORK function.

Examples

The following example ....

```
EXECUTE ... ;
```

Index

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X