Oracle® Spatial

User's Guide and Reference

Release 9.0.1

June 2001

Part No. A88805-01

ORACLE®

Oracle Spatial User's Guide and Reference, Release 9.0.1

Part No. A88805-01

Primary Author: Chuck Murray

Contributors: Dan Abugov, Nicole Alexander, Bruce Blackwell, Dan Geringer, Ravi Kothuri, Deborah Owens, Richard Pitts, Siva Ravada, Jack Wang, and Jeffrey Xie

Contents

2 The Object-Relational Schema

3 Loading Spatial Data

4 Indexing and Querying Spatial Data

5 Coordinate Systems (Spatial Reference Systems)

6 Linear Referencing System

7 Generic Geocoding Interface

12 Geometry Functions

13 Spatial Aggregate Functions

14 Coordinate System Transformation Functions

15 Linear Referencing Functions

16 Migration Procedures

17 Tuning Functions and Procedures

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Spatial User's Guide and Reference, Release 9.0.1

Part No. A88805-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter and section or page number (if available). You can send comments to us in the following ways:

- Electronic mail: nedc-doc_us@oracle.com
- FAX: 603.897.3825 Attn: Spatial Documentation
- Postal service:
  Oracle Corporation
  Oracle Spatial Documentation
  One Oracle Drive
  Nashua, NH 03062-2804
  USA

If you would like a reply, please include your name and contact information.

If you have problems with the software, please contact Oracle Support Services.

Preface

The *Oracle Spatial User's Guide and Reference* provides usage and reference information for indexing and storing spatial data and for developing spatial applications.

Spatial requires Oracle9*i* Enterprise Edition. Oracle9*i* and Oracle9*i* Enterprise Edition have the same basic features. However, several advanced features, such as extended data types, are available only with the Enterprise Edition, and some of these features are optional. For example, to use Oracle9*i* table partitioning, you must have the Enterprise Edition and the Partitioning Option.

For information about the differences between Oracle9*i* and Oracle9*i* Enterprise Edition and the features and options that are available to you, see *Oracle9i Database New Features*.

Note: This is the last release of Oracle Spatial that will support the relational geometry model; only the object-relational model will be supported in Oracle9*i* release 2. Information about the relational model has been removed from this guide and placed in a separate manual, *Oracle Spatial Relational Model Guide and Reference*, which is available on the Oracle Technology Network.

If you have not already switched completely to the object-relational model, you should do so immediately.

Audience

This guide is intended for anyone who needs to store spatial data in an Oracle database.

Organization

This guide has two main parts (conceptual and usage information, and reference information) and several appendixes with supplementary information. The first part is organized for efficient learning about Oracle Spatial; it covers basic concepts and techniques first, and proceeds to more advanced material (such as coordinate systems, the linear referencing system, geocoding, and extending spatial indexing).

This guide has the following elements.

Part I	**Conceptual and Usage Information**
Chapter 1	Introduces spatial data concepts.
Chapter 2	Explains the object-relational schema.
Chapter 3	Explains how to load spatial data.
Chapter 4	Explains how to index and query spatial data.
Chapter 5	Provides detailed information about coordinate system (spatial reference system) support.
Chapter 6	Provides conceptual and usage information for using the Oracle Spatial linear referencing system (LRS).
Chapter 7	Describes the Spatial Generic Geocoding Interface.
Chapter 8	Explains how to extend the capabilities of Oracle Spatial indexing.
Part II	**Reference Information**
Chapter 9	Provides the syntax and semantics for SQL indexing statements.
Chapter 10	Provides the syntax and semantics for methods used with the spatial object data type.
Chapter 11	Provides the syntax and semantics for operators used with the spatial object data type.
Chapter 12	Provides the syntax and semantics for the geometric functions and procedures.
Chapter 13	Provides the syntax and semantics for the spatial aggregate functions.
Chapter 14	Provides the syntax and semantics for the coordinate system transformation functions.

Chapter 15	Provides the syntax and semantics for the linear referencing (LRS) functions.
Chapter 16	Provides the syntax and semantics for the migration functions.
Chapter 17	Provides the syntax and semantics for the tuning functions and procedures.
(Other)	**Supplementary Information**
Appendix A	Describes installation, compatibility, and migration issues.
Appendix B	Describes hybrid indexing.
Appendix C	Describes Oracle9*i* Locator.
Glossary	
Index	

Changes for Release 9.0.1

The following are the main changes to this guide for this release:

- With Oracle9*i*, Spatial provides a rational and complete treatment of geodetic (longitude/latitude) coordinates. For more information about geodetic data support, see Section 5.2.
- *Tolerance* has a different meaning for geodetic data than for non-geodetic data. See Section 1.5.5.
- The SDO_GTYPE element of the SDO_GEOMETRY type has a new format that identifies the linear referencing dimension (if any). See Section 2.2.1.
- Methods are provided for the SDO_GEOMETRY type: GET_GTYPE, GET_DIMENSIONS, GET_LRS_DIMS. See Chapter 10.
- Spatial aggregate functions are provided: SDO_AGGR_MBR, SDO_AGGR_UNION, SDO_AGGR_BUFFER, SDO_AGGR_CONVEXHULL. See Chapter 13.
- The SDO_GEOMETRY type can be embedded in a user-defined data type. See Section 8.1.
- Partitioned indexes are supported. See Section 4.1.6, and the CREATE INDEX and ALTER INDEX descriptions in Chapter 9.
- Coordinate system support has been enhanced: storage and conversion of coordinates in any datum and projection; new SDO_CS.VIEWPORT_

TRANSFORM function; local coordinate systems; and user-defined coordinate systems. See Chapter 5 for most of the information, and Chapter 14 for reference information about the SDO_CS.VIEWPORT_TRANSFORM function.

- Linear referencing support enhancements include support for additional kinds of geometric segments (multiple line strings and 2D polygons, in addition to line strings), *_3D* function formats, monotonically decreasing as well as increasing measure, several new functions, and aggregate concatenation. See Chapter 6.
- The new SDO_GEOM.SDO_ARC_DENSIFY function is provided to change arcs to straight lines to create a regular polygon for input to Spatial functions. See Chapter 16.
- The new SDO_MIGRATE.TO_CURRENT procedure is provided to migrate from previous Spatial releases to the current release. See Chapter 16.
- R-tree index administration functions are provided, to check the quality of the index and minimize query performance degradation. See Chapter 17.
- Columns have been added to the spatial index views (xxx_SDO_INDEX_METADATA). See Section 2.5.1.
- New views are provided for retrieving basic information about spatial indexes: xxx_SDO_INDEX_INFO views. See Section 2.5.1.1.
- Unit of measurement support (for example, MILE for distance units in miles) is provided for relevant Spatial functions and operators. See Section 2.6.
- A function to return the minimum bounding rectangle of a geometry is provided. See SDO_GEOM.SDO_MBR in Chapter 12.
- A commit interval can be specified when validating a layer. See the SDO_GEOM.VALIDATE_LAYER description in Chapter 12.
- The SDO_VERSION function returns the Spatial release (version) number. See Section 1.10.
- The 18-character limit for spatial index names is removed.
- Information about the relational model of Oracle Spatial has been removed, because this model is no longer supported. (Only the object-relational model is supported.) The removed material is in a document titled *Oracle Spatial Relational Model Guide and Reference*, which is available on the Oracle Technology Network.
- This guide has been reorganized into two main parts: Part I (Conceptual and Usage Information) and Part II (Reference Information).

- Error messages for coordinate systems and the linear referencing system have been moved to *Oracle9i Database Error Messages*.

Technologies Released Separately

Technologies of interest to spatial application developers, but not officially part of Oracle Spatial, are sometimes made available through the Oracle Technology Network (OTN). To access the OTN, go to

`http://technet.oracle.com`

Related Documents

For more information, see the following documents:

- *Oracle9i Database New Features*
- *Oracle9i Database Administrator's Guide*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Application Developer's Guide - Workspace Manager*
- *Oracle9i Database Error Messages* - Spatial messages are in the range of 13000 to 13499.
- *Oracle9i Database Concepts*
- *Oracle9i Database Performance Guide and Reference*
- *Oracle9i Database Utilities*

For additional information about Oracle Spatial, including white papers and other collateral, go to

`http://www.oracle.com/`

and search for *Spatial*.

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following conventions are used in this guide:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface text indicates a term defined in the text, the glossary, or in both locations.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
%	The percent sign represents the system prompt on a UNIX system.

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility`

Accessibility of Code Examples in Documentation

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

Part I

Conceptual and Usage Information

This document has two main parts:

- Part I provides conceptual and usage information about Oracle Spatial.
- Part II provides reference information about Oracle Spatial methods, operators, functions, and procedures.

Appendixes with supplementary information follow Part II.

Part I is organized for efficient learning about Oracle Spatial. It covers basic concepts and techniques first, and proceeds to more advanced material (such as coordinate systems, the linear referencing system, geocoding, and extending spatial indexing). Part I contains the following chapters:

1

Spatial Concepts

Oracle Spatial is an integrated set of functions and procedures that enables spatial data to be stored, accessed, and analyzed quickly and efficiently in an Oracle9*i* database.

Spatial data represents the essential location characteristics of real or conceptual objects as those objects relate to the real or conceptual space in which they exist.

1.1 What Is Oracle Spatial?

Oracle Spatial, often referred to as Spatial, provides a SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle9*i* database. Spatial consists of the following components:

- A schema (MDSYS) that prescribes the storage, syntax, and semantics of supported geometric data types
- A spatial indexing mechanism
- A set of operators and functions for performing area-of-interest queries, spatial join queries, and other spatial analysis operations
- Administrative utilities

The spatial component of a spatial feature is the geometric representation of its shape in some coordinate space. This is referred to as its **geometry**.

1.2 Object-Relational Model

Spatial supports the **object-relational** model for representing geometries. The object-relational model uses a table with a single column of MDSYS.SDO_ GEOMETRY and a single row per geometry instance. The object-relational model

corresponds to a "SQL with Geometry Types" implementation of spatial feature tables in the OpenGIS ODBC/SQL specification for geospatial features.

Note: This is the last release of Oracle Spatial that will support the relational geometry model; only the object-relational model will be supported in Oracle9*i* release 2. Information about the relational model has been removed from this guide and placed in a separate manual, *Oracle Spatial Relational Model Guide and Reference*, which is available on the Oracle Technology Network.

If you have not already switched completely to the object-relational model, you should do so immediately.

The benefits provided by the object-relational model include:

- Support for many geometry types, including arcs, circles, compound polygons, compound line strings, and optimized rectangles
- Ease of use in creating and maintaining indexes and in performing spatial queries
- Index maintenance by the Oracle9*i* database server
- Geometries modeled in a single row and single column
- Optimal performance

1.3 Introduction to Spatial Data

Oracle Spatial is designed to make spatial data management easier and more natural to users of location-enabled applications and Geographic Information System (GIS) applications. Once this data is stored in an Oracle database, it can be easily manipulated, retrieved, and related to all the other data stored in the database.

A common example of spatial data can be seen in a road map. A road map is a two-dimensional object that contains points, lines, and polygons that can represent cities, roads, and political boundaries such as states or provinces. A road map is a visualization of geographic information. The location of cities, roads, and political boundaries that exist on the surface of the Earth are projected onto a two-dimensional display or piece of paper, preserving the relative positions and relative distances of the rendered objects.

The data that indicates the Earth location (latitude and longitude, or height and depth) of these rendered objects is the spatial data. When the map is rendered, this spatial data is used to project the locations of the objects on a two-dimensional piece of paper. A GIS is often used to store, retrieve, and render this Earth-relative spatial data.

Types of spatial data that can be stored using Spatial other than GIS data include data from computer-aided design (CAD) and computer-aided manufacturing (CAM) systems. Instead of operating on objects on a geographic scale, CAD/CAM systems work on a smaller scale, such as for an automobile engine or printed circuit boards.

The differences among these systems are only in the relative sizes of the data, not the data's complexity. The systems might all actually involve the same number of data points. On a geographic scale, the location of a bridge can vary by a few tenths of an inch without causing any noticeable problems to the road builders, whereas if the diameter of an engine's pistons are off by a few tenths of an inch, the engine will not run. A printed circuit board is likely to have many thousands of objects etched on its surface that are no bigger than the smallest detail shown on a road builder's blueprints.

These applications all store, retrieve, update, or query some collection of features that have both nonspatial and spatial attributes. Examples of nonspatial attributes are name, soil_type, landuse_classification, and part_number. The spatial attribute is a coordinate geometry, or vector-based representation of the shape of the feature.

1.4 Geometry Types

A geometry is an ordered sequence of vertices that are connected by straight line segments or circular arcs. The semantics of the geometry are determined by its type. Spatial supports several primitive types and geometries composed of collections of these types, including 2-dimensional:

- Points and point clusters
- Line strings
- *n*-point polygons
- Arc line strings (All arcs are generated as circular arcs.)
- Arc polygons
- Compound polygons
- Compound line strings

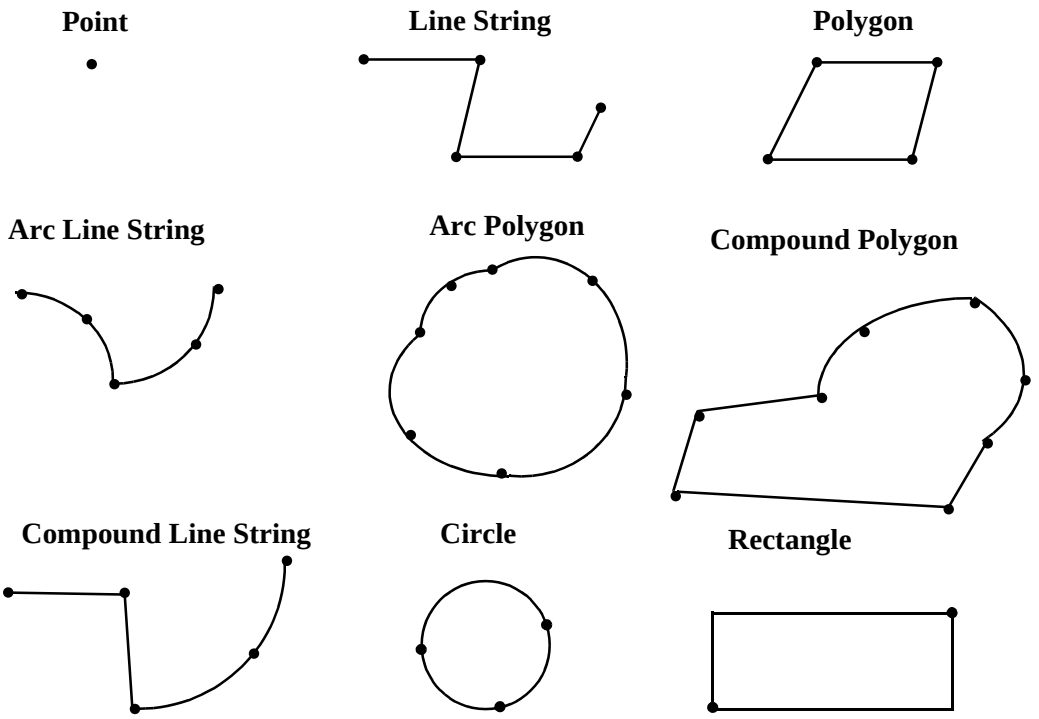
- Circles
- Optimized rectangles

2-dimensional points are elements composed of two ordinates, X and Y, often corresponding to longitude and latitude. **Line strings** are composed of one or more pairs of points that define line segments. **Polygons** are composed of connected line strings that form a closed ring and the interior of the polygon is implied.

Self-crossing polygons are not supported, although self-crossing line strings are supported. If a line string crosses itself, it does not become a polygon. A self-crossing line string does not have any implied interior.

Figure 1–1 illustrates the geometric types.

Figure 1–1 Geometric Types



1.5 Data Model

The Spatial data model is a hierarchical structure consisting of elements, geometries, and layers, which correspond to representations of spatial data. Layers are composed of geometries, which in turn are made up of elements.

For example, a point might represent a building location, a line string might represent a road or flight path, and a polygon might represent a state, city, zoning district, or city block.

1.5.1 Element

An **element** is the basic building block of a geometry. The supported spatial element types are points, line strings, and polygons. For example, elements might model star constellations (point clusters), roads (line strings), and county boundaries (polygons). Each coordinate in an element is stored as an X,Y pair. The exterior ring and the interior ring of a polygon with holes are considered as two distinct elements that together make up a complex polygon.

Point data consists of one coordinate. **Line data** consists of two coordinates representing a line segment of the element. **Polygon data** consists of coordinate pair values, one vertex pair for each line segment of the polygon. Coordinates are defined in order around the polygon (counterclockwise for an exterior polygon ring, clockwise for an interior polygon ring).

1.5.2 Geometry

A **geometry** (or **geometry object**) is the representation of a spatial feature, modeled as an ordered set of primitive elements. A geometry can consist of a single element, which is an instance of one of the supported primitive types, or a homogeneous or heterogeneous collection of elements. A multipolygon, such as one used to represent a set of islands, is a homogeneous collection. A heterogeneous collection is one in which the elements are of different types, for example, a point and a polygon.

An example of a geometry might describe the buildable land in a town. This could be represented as a polygon with holes where water or zoning prevents construction.

1.5.3 Layer

A **layer** is a collection of geometries having the same attribute set. For example, one layer in a GIS might include topographical features, while another describes

population density, and a third describes the network of roads and bridges in the area (lines and points). Each layer's geometries and associated spatial index are stored in the database in standard tables.

1.5.4 Coordinate System

A **coordinate system** (also called a *spatial reference system*) is a means of assigning coordinates to a location and establishing relationships between sets of such coordinates. It enables the interpretation of a set of coordinates as a representation of a position in a real world space.

Any spatial data has a coordinate system associated with it. The coordinate system can be *georeferenced* (related to a specific representation of the Earth) or not georeferenced (that is, Cartesian, and not related to a specific representation of the Earth). If the coordinate system is georeferenced, it has a default *unit of measurement* (such as meters) associated with it, but you can have Spatial automatically return results in another specified unit (such as miles). (For more information about unit of measurement support, see Section 2.6.)

Before Oracle Spatial release 8.1.6, geometries (objects of type SDO_GEOMETRY) were stored as strings of coordinates without reference to any specific coordinate system. Spatial functions and operators always assumed a coordinate system that had the properties of an orthogonal Cartesian system, and sometimes did not provide correct results if Earth-based geometries were stored in latitude and longitude coordinates. With release 8.1.6, Spatial provided support for many different coordinate systems, and for converting data freely between different coordinate systems.

Spatial data can be associated with a Cartesian, geodetic (geographical), projected, or local coordinate system:

- Cartesian coordinates are coordinates that measure the position of a point from a defined origin along axes that are perpendicular in the represented two-dimensional or three-dimensional space.

  If a coordinate system is not explicitly associated with a geometry, a Cartesian coordinate system is assumed.

- Geodetic coordinates (sometimes called *geographic coordinates*) are angular coordinates (longitude and latitude), closely related to spherical polar coordinates, and are defined relative to a particular Earth geodetic datum. (A geodetic datum is a means of representing the figure of the Earth and is the reference for the system of geodetic coordinates.)

- Projected coordinates are planar Cartesian coordinates that result from performing a mathematical mapping from a point on the Earth's surface to a plane. There are many such mathematical mappings, each used for a particular purpose.
- Local coordinates are Cartesian coordinates in a non-Earth (non-georeferenced) coordinate system. Local coordinate systems are often used for CAD applications and local surveys.

When performing operations on geometries, Spatial uses either a Cartesian or curvilinear computational model, as appropriate for the coordinate system associated with the spatial data.

For more information about coordinate system support in Spatial, including geodetic, projected, and local coordinates and coordinate system transformation, see Chapter 5.

1.5.5 Tolerance

Tolerance is used to associate a level of precision with spatial data. The tolerance value must be a non-negative number greater than zero. The range of values and the significance of the value depend on whether or not the spatial data is associated with a geodetic coordinate system. (Geodetic and other types of coordinate systems are described in Section 1.5.4.)

- For geodetic data (such as data identified by longitude and latitude coordinates), the tolerance value is a number of meters. For example, a tolerance value of 100 indicates a tolerance of 100 meters.
- For non-geodetic data, the tolerance value can be up to 1, referring to the decimal fraction of the distance unit in use. (If a coordinate system is specified, the distance unit is the default for that system.) For example, a tolerance value of 0.005 indicates a tolerance of 0.005 (that is, 1/200) of the distance unit.

In both cases, the smaller the tolerance value, the more precision is to be associated with the data.

A tolerance value is specified in two cases:

- In the geometry metadata definition for a layer (see Section 1.5.5.1)
- As an optional input parameter to certain functions (see Section 1.5.5.2)

1.5.5.1 In the Geometry Metadata for a Layer

The dimensional information for a layer includes a tolerance value. Specifically, the DIMINFO column (described in Section 2.4.3) of the xxx_SDO_GEOM_METADATA views includes an SDO_TOLERANCE value.

If a function accepts an optional *tolerance* parameter and this parameter is null or not specified, the SDO_TOLERANCE value of the layer is used. Using the non-geodetic data from the example in Section 2.1, the actual distance between geometries *cola_b* and *cola_d* is 0.846049894. If a query uses the SDO_GEOM.SDO_DISTANCE function to return the distance between *cola_b* and *cola_d* and does not specify a *tolerance* parameter value, the result depends on the SDO_TOLERANCE value of the layer. For example:

- If the SDO_TOLERANCE value of the layer is 0.005, this query returns .846049894.
- If the SDO_TOLERANCE value of the layer is 0.5, this query returns 0.

  The zero result occurs because Spatial first constructs an imaginary buffer of the tolerance value (0.5) around each geometry to be considered, and the buffers around *cola_b* and *cola_d* overlap in this case.

You can therefore take either of two approaches in selecting an SDO_TOLERANCE value for a layer:

- The value can reflect the desired level of precision in queries for distances between objects. For example, if two non-geodetic geometries 0.8 units apart should be considered as separated, specify a small SDO_TOLERANCE value such as 0.05 or smaller.
- The value can reflect the precision of the values associated with geometries in the layer. For example, if all the geometries in a non-geodetic layer are defined using integers and if two objects 0.8 units apart should not be considered as separated, an SDO_TOLERANCE value of 0.5 is appropriate. To have greater precision in any query, you must override the default by specifying the *tolerance* parameter.

With non-geodetic data, the guideline to follow for most instances of the second case (precision of the values of the geometries in the layer) is: take the highest level of precision in the geometry definitions, and use .5 at the next level as the SDO_TOLERANCE value. For example, if geometries are defined using integers (as in the simplified example in Section 2.1), the appropriate value is 0.5. However, if geometries are defined using numbers up to 4 decimal positions (for example, 31.2587), such as with longitude and latitude values, the appropriate value is 0.00005.

Note: This guideline, however, should not be used if the geometries include any polygons that are so narrow at any point that the distance between facing sides is less than the proposed tolerance value. Be sure that the tolerance value is less than the shortest distance between any two sides in any polygon.

Moreover, if you encounter "invalid geometry" errors with inserted or updated geometries, and if the geometries are in fact valid, consider increasing the precision of the tolerance value (for example, changing 0.00005 to 0.000005).

1.5.5.2 As an Input Parameter

Many Spatial functions accept an optional *tolerance* parameter, which (if specified) overrides the default tolerance value for the layer (explained in Section 1.5.5.1). If the distance between two points is less than or equal to the tolerance value, Spatial considers the two points to be a single point. Thus, tolerance is usually a reflection of how accurate or precise users perceive their spatial data to be.

For example, assume that you want to know which restaurants are within 5 kilometers of your house. Assume also that Maria's Pizzeria is 5.1 kilometers from your house. If the spatial data has a geodetic coordinate system and if you ask, *Find all restaurants within 5 kilometers and use a tolerance of 100* (or greater, such as 500), Maria's Pizzeria will be included, because 5.1 kilometers (5100 meters) is within 100 meters of 5 kilometers (5000 meters). However, if you specify a tolerance less than 100 (such as 50), Maria's Pizzeria will not be included.

Tolerance values for Spatial functions are typically very small, although the best value in each case depends on the kinds of applications that use or will use the data.

1.6 Query Model

Spatial uses a *two-tier* query model to resolve spatial queries and spatial joins. The term is used to indicate that two distinct operations are performed to resolve queries. The output of the two combined operations yields the exact result set.

The two operations are referred to as *primary* and *secondary* filter operations.

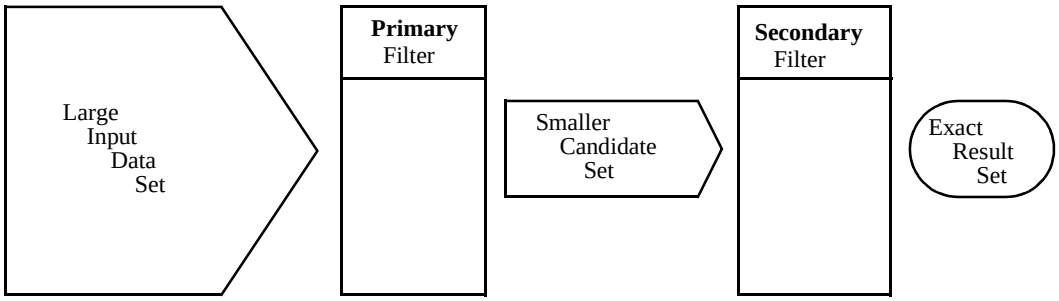
- The **primary filter** permits fast selection of candidate records to pass along to the secondary filter. The primary filter compares geometry approximations to reduce computation complexity and is considered a lower-cost filter. Because

the primary filter compares geometric approximations, it returns a superset of the exact result set.

- The **secondary filter** applies exact computations to geometries that result from the primary filter. The secondary filter yields an accurate answer to a spatial query. The secondary filter operation is computationally expensive, but it is only applied to the primary filter results, not the entire data set.

Figure 1–2 illustrates the relationship between the primary and secondary filters.

Figure 1–2 Query Model



As shown in Figure 1–2, the primary filter operation on a large input data set produces a smaller candidate set, which contains at least the exact result set and may contain more records. The secondary filter operation on the smaller candidate set produces the exact result set.

Spatial uses a spatial index to implement the primary filter. Spatial does not require the use of both the primary and secondary filters. In some cases, just using the primary filter is sufficient. For example, a *zoom* feature in a mapping application queries for data that has any interaction with a rectangle representing visible boundaries. The primary filter very quickly returns a superset of the query. The mapping application can then apply clipping routines to display the target area.

The purpose of the primary filter is to quickly create a subset of the data and reduce the processing burden on the secondary filter. The primary filter therefore should be as efficient (that is, selective yet fast) as possible. This is determined by the characteristics of the spatial index on the data.

1.7 Indexing of Spatial Data

The introduction of spatial indexing capabilities into the Oracle database engine is a key feature of the Spatial product. A spatial index, like any other index, provides a mechanism to limit searches, but in this case based on spatial criteria such as intersection and containment. A spatial index is needed to:

- Find objects within an indexed data space that interact with a given point or area of interest (window query)
- Find pairs of objects from within two indexed data spaces that interact spatially with each other (spatial join)

A spatial index is considered a logical index. The entries in the spatial index are dependent on the location of the geometries in a coordinate space, but the index values are in a different domain. Index entries may be ordered using a linearly ordered domain, and the coordinates for a geometry may be pairs of integer, floating-point, or double-precision numbers.

Oracle Spatial lets you use R-tree indexing (the default) or quadtree indexing, or both. Each index type is appropriate in different situations. You can maintain both an R-tree and quadtree index on the same geometry column, by using the *add_index* parameter with the ALTER INDEX statement (described in Chapter 9), and you can choose which index to use for a query by specifying the *idxtab1* and / or *idxtab2* parameters with certain Spatial operators, such as SDO_RELATE, described in Chapter 11.

In choosing whether to use an R-tree or quadtree index for a spatial application, consider the items in Table 1–1.

Table 1–1 Choosing R-tree or Quadtree Indexing

R-tree Indexing	Quadtree Indexing
The approximation of geometries cannot be fine-tuned. (Spatial uses the minimum bounding rectangles, as described in Section 1.7.1.)	The approximation of geometries can be fine-tuned by setting the tiling level and number of tiles.
Index creation and tuning are easier.	Tuning is more complex, and setting the appropriate tuning parameter values can affect performance significantly.
Less storage is required.	More storage is required.
If your application workload includes nearest-neighbor queries (SDO_NN operator), R-tree indexes are faster.	If your application workload includes nearest-neighbor queries (SDO_NN operator), quadtree indexes are slower.

Table 1–1 Choosing R-tree or Quadtree Indexing (Cont.)

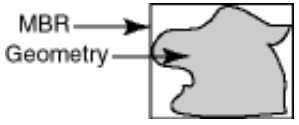
R-tree Indexing	Quadtree Indexing
If there is heavy update activity to the spatial column, an R-tree index may not be a good choice.	Heavy update activity does not affect the performance of a quadtree index.
You can index up to 4 dimensions.	You can only index only 2 dimensions.
An R-tree index is recommended for indexing geodetic data if SDO_WITHIN_DISTANCE queries will be used on it.	
An R-tree index is required for a whole-earth index.	

Testing of R-tree and quadtree indexes with many workloads and operators is ongoing, and results and recommendations will be documented as they become available. However, before choosing an index type for an application, you should understand the concepts and options associated with both R-tree indexing (described in Section 1.7.1) and quadtree indexing (described in Section 1.7.2).

1.7.1 R-tree Indexing

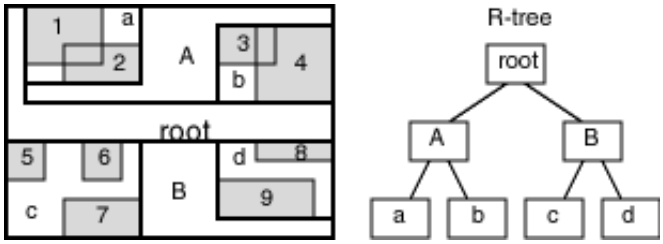
A spatial R-tree index can index spatial data of up to 4 dimensions. An R-tree index approximates each geometry by a single rectangle that minimally encloses the geometry (called the minimum bounding rectangle, or MBR), as shown in Figure 1–3.

Figure 1–3 MBR Enclosing a Geometry



For a layer of geometries, an R-tree index consists of a hierarchical index on the MBRs of the geometries in the layer, as shown in Figure 1–4.

Figure 1–4 R-tree Hierarchical Index on MBRs



In Figure 1–4:

- *1* through *9* are geometries in a layer.
- *a*, *b*, *c*, and *d* are the leaf nodes of the R-tree index, and contain minimum bounding rectangles of geometries, along with pointers to the geometries. For example, *a* contains the MBR of geometries *1* and *2*, *b* contains the MBR of geometries *3* and *4*, and so on.
- *A* contains the MBR of *a* and *b*, and *B* contains the MBR of *c* and *d*.
- The root contains the MBR of *A* and *B* (that is, the entire area shown).

An R-tree index is stored in the spatial index table (SDO_INDEX_TABLE in the USER_SDO_INDEX_METADATA view, described in Section 2.5). The R-tree index also maintains a sequence number generator (SDO_RTREE_SEQ_NAME in the USER_SDO_INDEX_METADATA view) to ensure that simultaneous updates by concurrent users can be made to the index.

1.7.1.1 R-tree Quality

A substantial number of insert and delete operations affecting an R-tree index may degrade the quality of the R-tree structure, which may adversely affect query performance.

The R-tree is a hierarchical tree structure with nodes at different heights of the tree. The performance of an R-tree index structure for queries is roughly proportional to the area and perimeter of the index nodes of the R-tree. The area covered at level 0 represents the area occupied by the minimum bounding rectangles of the data geometries, the area at level 1 indicates the area covered by leaf-level R-tree nodes, and so on. The original ratio of the area at the root (topmost level) to the area at level 0 can change over time based on updates to the table; and if there is a

degradation in that ratio (that is, if it increases significantly), rebuilding the index may help the performance of queries.

Spatial provides several functions and procedures related to the quality of an R-tree index:

- SDO_TUNE.ANALYZE_RTREE provides advice about whether or not an index needs to be rebuilt. It computes the current index quality score and compares it to the quality score when the index was created or most recently rebuilt, and it displays a recommendation.
- SDO_TUNE.RTREE_QUALITY returns the current index quality score.
- SDO_TUNE.QUALITY_DEGRADATION returns the current index quality degradation.

These functions and procedures are described in Chapter 17.

To rebuild an R-tree index, use the ALTER INDEX REBUILD statement, which is described in Chapter 9.

1.7.2 Quadtree Indexing

In the linear quadtree indexing scheme, the coordinate space (for the layer where all geometric objects are located) is subjected to a process called **tessellation**, which defines exclusive and exhaustive cover tiles for every stored geometry. Tessellation is done by decomposing the coordinate space in a regular hierarchical manner. The range of coordinates, the coordinate space, is viewed as a rectangle. At the first level of decomposition, the rectangle is divided into halves along each coordinate dimension generating four tiles. Each tile that interacts with the geometry being tessellated is further decomposed into four tiles. This process continues until some termination criteria, such as size of the tiles or the maximum number of tiles to cover the geometry, is met.

Spatial can use either fixed-size or variable-sized tiles to cover a geometry:

- Fixed-size tiles are controlled by tile resolution. If the resolution is the sole controlling factor, then tessellation terminates when the coordinate space has been decomposed a specific number of times. Therefore, each tile is of a fixed size and shape.
- Variable-sized tiling is controlled by the value supplied for the maximum number of tiles. If the number of tiles per geometry, *n*, is the sole controlling factor, the tessellation terminates when *n* tiles have been used to cover the given geometry.

Fixed-size tile resolution and the number of variable-sized tiles used to cover a geometry are user-selectable parameters called SDO_LEVEL and SDO_NUMTILES, respectively. Smaller fixed-size tiles or more variable-sized tiles provides better geometry approximations. The smaller the number of tiles, or the larger the tiles, the coarser are the approximations.

Spatial supports two quadtree indexing types, reflecting two valid combinations of SDO_LEVEL and SDO_NUMTILES values:

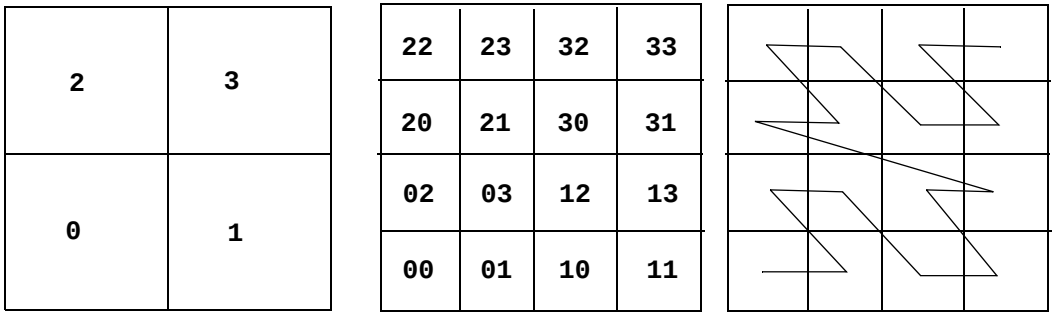
- **Fixed indexing:** a non-null and non-zero SDO_LEVEL value and a null or zero (0) SDO_NUMTILES value, resulting in fixed-sized tiles. Fixed indexing is described in Section 1.7.2.2.
- **Hybrid indexing:** non-null and non-zero values for SDO_LEVEL and SDO_NUMTILES, resulting in two sets of tiles per geometry. One set contains fixed-size tiles and the other set contains variable-sized tiles. Hybrid indexing is *not recommended* for most spatial applications, and is described in Appendix B.

1.7.2.1 Tessellation of a Layer During Indexing

The process of determining which tiles cover a given geometry is called **tessellation**. The tessellation process is a quadtree decomposition, where the two-dimensional coordinate space is broken down into four covering tiles of equal size. Successive tessellations divide those tiles that interact with the geometry down into smaller tiles, and this process continues until the desired level or number of tiles has been achieved. The results of the tessellation process on a geometry are stored in a table, referred to as the SDOINDEX table.

The tiles at a particular level can be linearly sorted by systematically visiting tiles in an order determined by a space-filling curve as shown in Figure 1–5. The tiles can also be assigned unique numeric identifiers, known as Morton codes or z-values. The terms tile and tile code will be used interchangeably in this and other sections related to spatial indexing.

Figure 1–5 Quadtree Decomposition and Morton Codes



1.7.2.2 Fixed Indexing

Fixed spatial indexing uses tiles of equal size to cover a geometry. Because all the tiles are the same size, they all have codes of the same length, and the standard SQL equality operator (=) can be used to compare tiles during a join operation. This results in excellent performance characteristics.

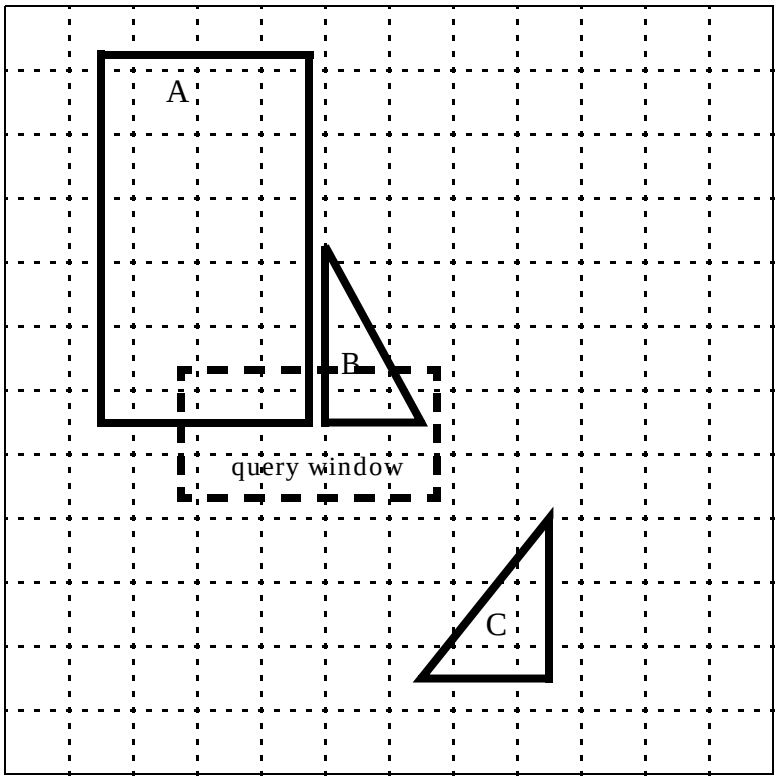
Two geometries are likely to interact, and hence pass the primary filter stage, if they share one or more tiles. The SQL statement for the primary filter stage is:

```
SELECT DISTINCT <select_list for geometry identifiers>
  FROM table1_sdoindex A, table2_sdoindex B
  WHERE A.sdo_code = B.sdo_code
```

The effectiveness and efficiency of this indexing method depends on the tiling level and the variation in size of the geometries in the layer. If you select a small fixed-size tile to cover small geometries and then try to use the same size tile to cover a very large geometry, a large number of tiles would be required. However, if the chosen tile size is large, so that fewer tiles are generated in the case of a large geometry, then the index selectivity suffers because the large tiles do not approximate the small geometries very well. Figure 1–6 and Figure 1–7 illustrate the relationships between tile size, selectivity, and the number of cover tiles.

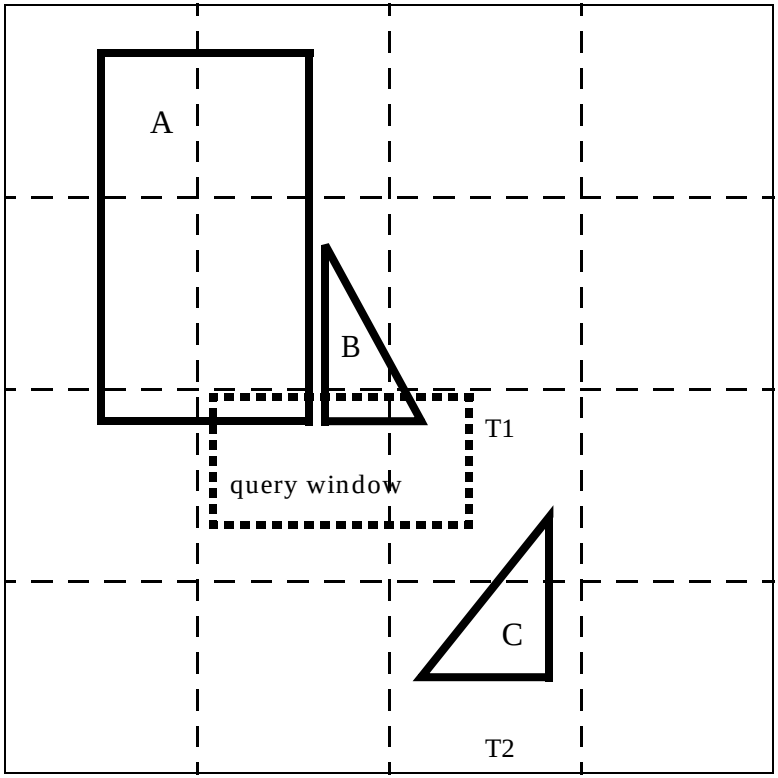
With a small fixed-size tile as shown in Figure 1–6, selectivity is good, but a large number of tiles is needed to cover large geometries. A window query would easily identify geometries A and B, but would reject C.

Figure 1–6 Fixed-Size Tiling with Many Small Tiles



With a large fixed-size tile as shown in Figure 1–7, fewer tiles are needed to cover the geometries, but the selectivity is not as good. The same window query as in Figure 1–6 would probably pick up all three geometries. Any object that shares tile T1 or T2 would identify object C as a candidate, even though the objects may be far apart, such as objects B and C are in Figure 1–7.

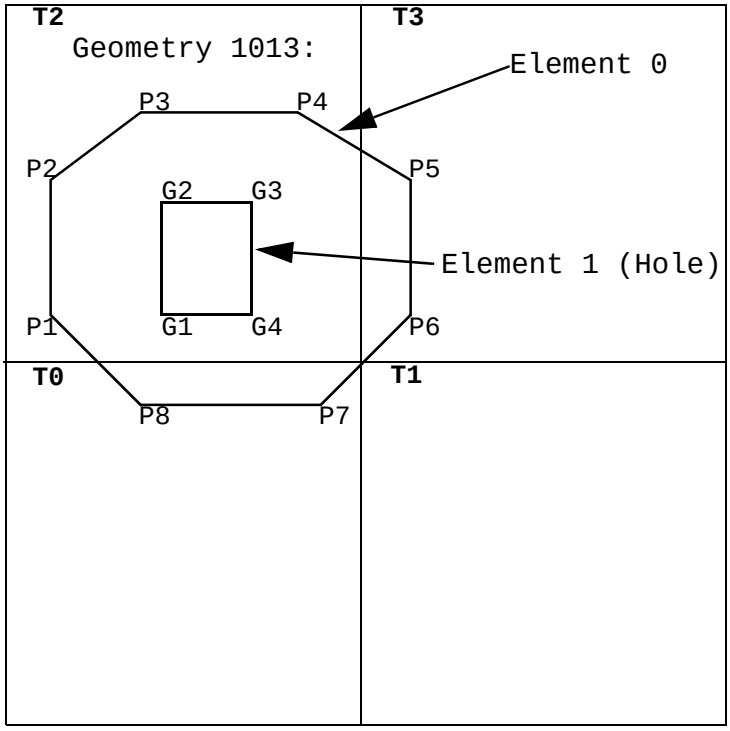
Figure 1–7 Fixed-Size Tiling with Fewer Large Tiles



You can use the SDO_TUNE.ESTIMATE_TILING_LEVEL function or the tiling wizard of the Spatial Index Advisor tool in Oracle Enterprise Manager to help determine an appropriate tiling level for your data set.

Figure 1–8 illustrates geometry 1013 tessellated to three fixed-sized tiles at level 1. The codes for these cover tiles are then stored in an SDOINDEX table.

Figure 1–8 Tessellated Geometry



Only three of the four tiles generated by the first tessellation interact with the geometry. Only those tiles that interact with the geometry are stored in the SDOINDEX table, as shown in Table 1–2. In this example, three fixed-size tiles are used. The table structure is shown for illustrative purposes only, because you should not directly access the index tables.

Table 1–2 SDOINDEX Table Using Fixed-Size Tiles

SDO_GID <number>	SDO_CODE <raw>
1013	T0
1013	T2
1013	T3

All elements in a geometry are tessellated. In a multielement geometry such as 1013, Element 1 is already covered by tile T2 from the tessellation of Element 0. If, however, the specified tiling resolution was such that tile T2 was further subdivided and one of these smaller tiles was completely contained in Element 1, then that tile would be excluded because it would not interact with the geometry.

1.8 Spatial Relations and Filtering

Spatial uses secondary filters to determine the spatial relationship between entities in the database. The spatial relation is based on geometry locations. The most common spatial relations are based on topology and distance. For example, the *boundary* of an area consists of a set of curves that separates the area from the rest of the coordinate space. The *interior* of an area consists of all points in the area that are not on its boundary. Given this, two areas are said to be adjacent if they share part of a boundary but do not share any points in their interior.

The distance between two spatial objects is the minimum distance between any points in them. Two objects are said to be *within a given distance* of one another if their distance is less than the given distance.

To determine spatial relations, Spatial has several secondary filter methods:

- The SDO_RELATE operator evaluates topological criteria.
- The SDO_WITHIN_DISTANCE operator determines if two spatial objects are within a specified distance of each other.
- The SDO_NN operator identifies the nearest neighbors for a spatial object.

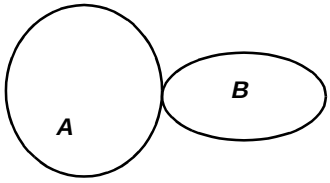
The syntax of these operators is given in Chapter 11.

The SDO_RELATE operator implements a 9-intersection model for categorizing binary topological relations between points, lines, and polygons. Each spatial object has an interior, a boundary, and an exterior. The boundary consists of points or lines that separate the interior from the exterior. The boundary of a line consists of its end points. The boundary of a polygon is the line that describes its perimeter. The interior consists of points that are in the object but not on its boundary, and the exterior consists of those points that are not in the object.

Given that an object A has 3 components (a boundary Ab, an interior Ai, and an exterior Ae), any pair of objects has 9 possible interactions between their components. Pairs of components have an empty (0) or a non-empty (1) set intersection. The set of interactions between 2 geometries is represented by a 9-intersection matrix that specifies which pairs of components intersect and which do not. Figure 1–9 shows the 9-intersection matrix for 2 polygons that are adjacent

to one another. This matrix yields the following bit mask, generated in row-major form: “101001111”.

Figure 1–9 The 9-Intersection Model



A TOUCH B

		B		
		b	i	e
A	b	1	0	1
	i	0	0	1
	e	1	1	1

9-Intersection Matrix

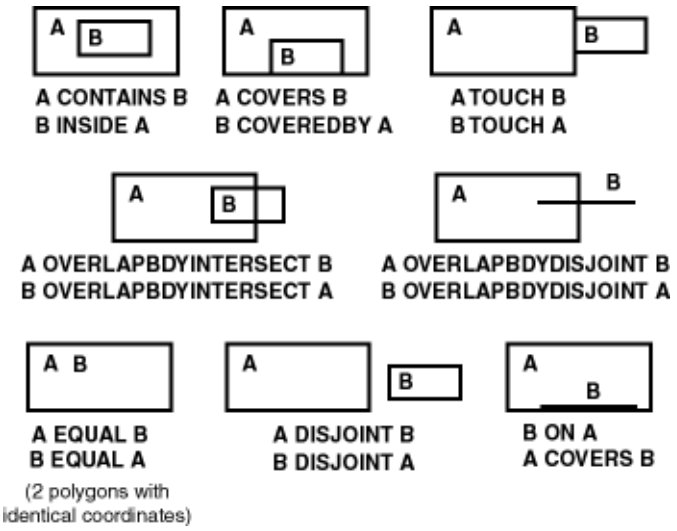
Some of the topological relationships identified in the seminal work by Professor Max Egenhofer (University of Maine, Orono) and colleagues have names associated with them. Spatial uses the following names:

- DISJOINT -- The boundaries and interiors do not intersect.
- TOUCH -- The boundaries intersect but the interiors do not intersect.
- OVERLAPBDYDISJOINT -- The interior of one object intersects the boundary and interior of the other object, but the two boundaries do not intersect. This relationship occurs, for example, when a line originates outside a polygon and ends inside that polygon.
- OVERLAPBDYINTERSECT -- The boundaries and interiors of the two objects intersect.
- EQUAL -- The two objects have the same boundary and interior.
- CONTAINS -- The interior and boundary of one object is completely contained in the interior of the other object.
- COVERS -- The interior of one object is completely contained in the interior of the other object and their boundaries intersect.
- INSIDE -- The opposite of CONTAINS. A INSIDE B implies B CONTAINS A.
- COVEREDBY -- The opposite of COVERS. A COVEREDBY B implies B COVERS A.

- ON -- The interior and boundary of one object is on the boundary of the other object (and the second object covers the first object). This relationship occurs, for example, when a line is on the boundary of a polygon.
- ANYINTERACT -- The objects are non-disjoint.

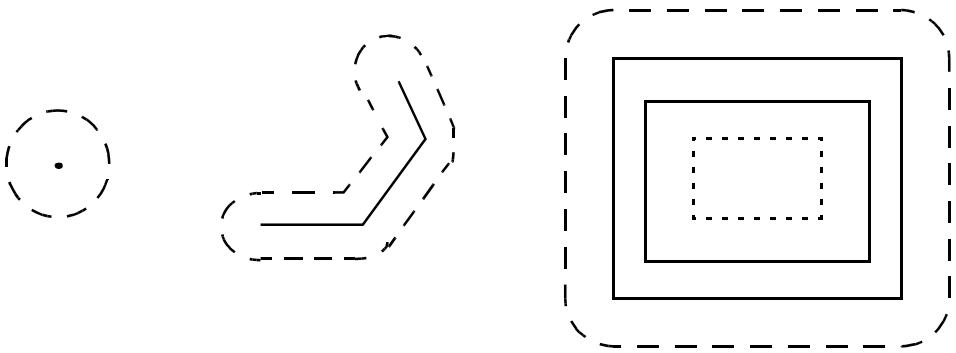
Figure 1–10 illustrates these topological relationships.

Figure 1–10 Topological Relationships



The SDO_WITHIN_DISTANCE operator determines if two spatial objects, A and B, are within a specified distance of one another. This operator first constructs a distance buffer, D_b, around the reference object B. It then checks that A and D_b are non-disjoint. The distance buffer of an object consists of all points within the given distance from that object. Figure 1–11 shows the distance buffers for a point, a line, and a polygon.

Figure 1–11 Distance Buffers for Points, Lines, and Polygons



In the geometries shown in Figure 1–11:

- The dashed lines represent distance buffers. Notice how the buffer is rounded near the corners of the objects.
- The geometry on the right is a polygon with a hole: the large rectangle is the exterior polygon ring and the small rectangle is the interior polygon ring (the hole). The dashed line outside the large rectangle is the buffer for the exterior ring, and the dashed line inside the small rectangle is the buffer for the interior ring.

The SDO_NN operator returns a specified number of objects from a geometry column that are closest to a specified geometry (for example, the five closest restaurants to a city park). In determining how close two geometry objects are, the shortest possible distance between any two points on the surface of each object is used.

1.9 Spatial Aggregate Functions

SQL has long had aggregate functions, which are used to aggregate the results of a SQL query. The following example uses the SUM aggregate function to aggregate employee salaries by department:

```
SELECT SUM(salary), dept
   FROM employees
   GROUP BY dept;
```

Oracle Spatial aggregate functions aggregate the results of SQL queries involving geometry objects. Spatial aggregate functions return a geometry object of type SDO_GEOMETRY. For example, the following statement returns the minimum bounding rectangle of all the geometries in a table (using the definitions and data from Section 2.1):

```
SELECT SDO_AGGR_MBR(shape) FROM cola_markets;
```

The following example returns the union of all geometries except *cola_d*:

```
SELECT SDO_AGGR_UNION(MDSYS.SDOAGGRTYPE(c.shape, 0.005))
  FROM cola_markets c WHERE c.name < 'cola_d';
```

All geometries used with spatial aggregate functions must be defined using 4-digit SDO_GTYPE values (that is, must be in the format used by Oracle Spatial release 8.1.6 or higher). For information about SDO_GTYPE values, see Section 2.2.1.

For reference information about the spatial aggregate functions and examples of their use, see Chapter 13.

1.9.1 SDOAGGRTYPE Object Type

Many spatial aggregate functions accept an input parameter of type MDSYS.SDOAGGRTYPE. Oracle Spatial defines the object type SDOAGGRTYPE as:

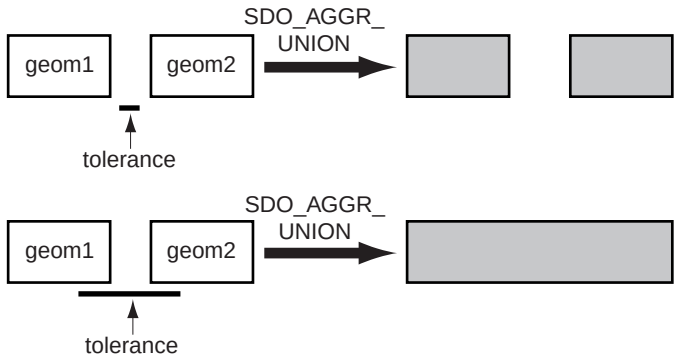
```
CREATE TYPE sdoaggrtype AS OBJECT (
 geometry MDSYS.SDO_GEOMETRY,
 tolerance NUMBER);
```

Note: Do not use SDOAGGRTYPE as the data type for a column in a table. Use this type only in calls to spatial aggregate functions.

The *tolerance* value in the SDOAGGRTYPE definition should be the same as the SDO_TOLERANCE value specified in the DIMINFO in the xxx_SDO_GEOM_METADATA views for the geometries, unless you have a specific reason for wanting a different value. For more information about tolerance, see Section 1.5.5; for information about the xxx_SDO_GEOM_METADATA views, see Section 2.4.

The *tolerance* value in the SDOAGGRTYPE definition can affect the result of a spatial aggregate function. Figure 1–12 shows a spatial aggregate union (SDO_AGGR_UNION) operation of two geometries using two different tolerance values: one smaller and one larger than the distance between the geometries.

Figure 1–12 Tolerance in an Aggregate Union Operation



In the first aggregate union operation in Figure 1–12, where the tolerance is less than the distance between the rectangles, the result is a compound geometry consisting of two rectangles. In the second aggregate union operation, where the tolerance is greater than the distance between the rectangles, the result is a single geometry.

1.10 Performance and Tuning Information

Many factors can affect the performance of Oracle Spatial applications, such as the indexing method (R-tree or quadtree), the SOD_LEVEL value for a quadtree index, and the use of optimizer hints to influence the plan for query execution. This guide contains some information about performance and tuning where it is relevant to a particular topic. For example, Section 1.7 includes performance-related items among the considerations for choosing an R-tree or quadtree index.

In addition, more Spatial performance and tuning information is available in one or more white papers through the Oracle Technology Network (OTN). That information is often more detailed that what is in this guide, and it is periodically updated as a result of internal testing and consultations with Spatial users. To find that information on the OTN, go to

`http://technet.oracle.com`

Search for *Spatial*, and then search for white papers relevant to performance and tuning.

1.11 Spatial Release (Version) Number

To check which release of Spatial you are running, use the SDO_VERSION function. For example:

```
SELECT SDO_VERSION FROM DUAL;

SDO_VERSION
--------------------------------------------------------------------------------
9.0.1
```

The SDO_VERSION function replaces the SDO_ADMIN.SDO_VERSION function, which was available with the deprecated relational model of Oracle Spatial.

1.12 Examples

Oracle Spatial provides examples that you can use to reinforce your learning and to create models for coding certain operations. Several examples are provided in the following directory:

$ORACLE_HOME/md/demos/examples

The following files in that directory are helpful for applications that use the Oracle Call Interface (OCI):

- readgeom.c and readgeom.h
- writegeom.c and writegeom.h

This guide also includes many examples in SQL and PL/SQL. One or more examples are usually provided with the reference information for each function or procedure, and several simplified examples are provided that illustrate table and index creation, as well as several functions and procedures:

- Inserting, indexing, and querying spatial data (Section 2.1)
- Coordinate systems (spatial reference systems) (Section 5.8)
- Linear referencing system (LRS) (Section 6.6)

2

The Object-Relational Schema

The object-relational implementation of Oracle Spatial consists of a set of object data types, type methods, and operators, functions, and procedures that use these types. A geometry is stored as an object, in a single row, in a column of type SDO_GEOMETRY. Spatial index creation and maintenance is done using basic DDL (CREATE, ALTER, DROP) and DML (INSERT, UPDATE, DELETE) statements.

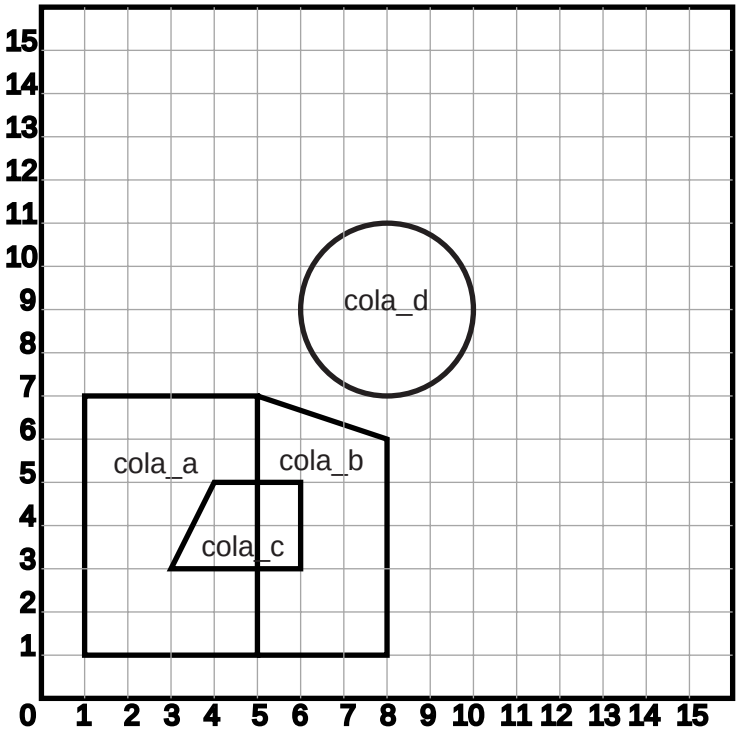
2.1 Simple Example: Inserting, Indexing, and Querying Spatial Data

This section presents a simple example of creating a spatial table, inserting data, creating the spatial index, and performing spatial queries. It refers to concepts that were explained in Chapter 1 and that will be explained in other sections of this chapter.

The scenario is a soft drink manufacturer that has identified geographical areas of marketing interest for several products (colas). The colas could be those produced by the company or by its competitors, or some combination. Each area of interest could represent any user-defined criterion: for example, an area where that cola has the majority market share, or where the cola is under competitive pressure, or where the cola is believed to have significant growth potential. Each area could be a neighborhood in a city, or a part of a state, province, or country.

Figure 2–1 shows the areas of interest for four colas.

Figure 2–1 Areas of Interest for Simple Example



Example 2–1 performs the following operations:

- Creates a table (COLA_MARKETS) to hold the spatial data
- Inserts rows for four areas of interest (*cola_a*, *cola_b*, *cola_c*, *cola_d*)
- Updates the USER_SDO_GEOM_METADATA view to reflect the dimension of the areas
- Creates a spatial index (COLA_SPATIAL_IDX)
- Performs some spatial queries

Many concepts and techniques in Example 2–1 are explained in detail in other sections of this chapter.

Example 2–1 Simple Example: Inserting, Indexing, and Querying Spatial Data

```
-- Create a table for cola (soft drink) markets in a
-- given geography (such as city or state).
-- Each row will be an area of interest for a specific
-- cola (for example, where the cola is most preferred
-- by residents, where the manufacturer believes the
-- cola has growth potential, and so on).

CREATE TABLE cola_markets (
  mkt_id NUMBER PRIMARY KEY,
  name VARCHAR2(32),
  shape MDSYS.SDO_GEOMETRY);

-- The next INSERT statement creates an area of interest for
-- Cola A. This area happens to be a rectangle.
-- The area could represent any user-defined criterion: for
-- example, where Cola A is the preferred drink, where
-- Cola A is under competitive pressure, where Cola A
-- has strong growth potential, and so on.

INSERT INTO cola_markets VALUES(
  1,
  'cola_a',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3), -- one rectangle (1003 = exterior)
    MDSYS.SDO_ORDINATE_ARRAY(1,1, 5,7) -- only 2 points needed to
          -- define rectangle (lower left and upper right) with
          -- Cartesian-coordinate data
  )
);

-- The next two INSERT statements create areas of interest for
-- Cola B and Cola C. These areas are simple polygons (but not
-- rectangles).

INSERT INTO cola_markets VALUES(
  2,
  'cola_b',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
```

```
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon (exterior polygon ring)
    MDSYS.SDO_ORDINATE_ARRAY(5,1, 8,1, 8,6, 5,7, 5,1)
  )
);

INSERT INTO cola_markets VALUES(
  3,
  'cola_c',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon (exterior polygon ring)
    MDSYS.SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
  )
);

-- Now insert an area of interest for Cola D. This is a
-- circle with a radius of 2. It is completely outside the
-- first three areas of interest.

INSERT INTO cola_markets VALUES(
  4,
  'cola_d',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,4), -- one circle
    MDSYS.SDO_ORDINATE_ARRAY(8,7, 10,9, 8,11)
  )
);

--------------------------------------------------------------------------------
-- UPDATE METADATA VIEW --
--------------------------------------------------------------------------------
-- 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: COLA_MARKETS and SHAPE).

INSERT INTO USER_SDO_GEOM_METADATA
  VALUES (
  'cola_markets',
  'shape',
```

```
  MDSYS.SDO_DIM_ARRAY(   -- 20X20 grid
    MDSYS.SDO_DIM_ELEMENT('X', 0, 20, 0.005),
    MDSYS.SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
     ),
  NULL   -- SRID
);

--------------------------------------------------------------------
-- CREATE THE SPATIAL INDEX --
--------------------------------------------------------------------
CREATE INDEX cola_spatial_idx
ON cola_markets(shape)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;
-- Preceding created an R-tree index.
-- Following line was for an earlier quadtree index:
--    PARAMETERS('SDO_LEVEL = 8');

--------------------------------------------------------------------
-- PERFORM SOME SPATIAL QUERIES --
--------------------------------------------------------------------
-- Return the topological intersection of two geometries.
SELECT SDO_GEOM.SDO_INTERSECTION(c_a.shape, c_c.shape, 0.005)
   FROM cola_markets c_a, cola_markets c_c
   WHERE c_a.name = 'cola_a' AND c_c.name = 'cola_c';

-- Do two geometries have any spatial relationship?
SELECT SDO_GEOM.RELATE(c_b.shape, 'anyinteract', c_d.shape, 0.005)
  FROM cola_markets c_b, cola_markets c_d
  WHERE c_b.name = 'cola_b' AND c_d.name = 'cola_d';

-- Return the areas of all cola markets.
SELECT name, SDO_GEOM.SDO_AREA(shape, 0.005) FROM cola_markets;

-- Return the area of just cola_a.
SELECT c.name, SDO_GEOM.SDO_AREA(c.shape, 0.005) FROM cola_markets c
   WHERE c.name = 'cola_a';

-- Return the distance between two geometries.
SELECT SDO_GEOM.SDO_DISTANCE(c_b.shape, c_d.shape, 0.005)
   FROM cola_markets c_b, cola_markets c_d
   WHERE c_b.name = 'cola_b' AND c_d.name = 'cola_d';

-- Is a geometry valid?
SELECT c.name, SDO_GEOM.VALIDATE_GEOMETRY(c.shape, 0.005)
   FROM cola_markets c WHERE c.name = 'cola_c';
```

```
-- Is a layer valid? (First, create the results table.)
CREATE TABLE validation_results (mkt_id number, result varchar2(10));
EXECUTE SDO_GEOM.VALIDATE_LAYER('COLA_MARKETS', 'SHAPE', 'MKT_ID',
  'VALIDATION_RESULTS');
SELECT * from validation_results;
```

2.2 SDO_GEOMETRY Object Type

In the Spatial object-relational model, the geometric description of a spatial object is stored in a single row, in a single column of object type SDO_GEOMETRY in a user-defined table. Any table that has a column of type SDO_GEOMETRY must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes referred to as geometry tables.

Oracle Spatial defines the object type SDO_GEOMETRY as:

```
CREATE TYPE sdo_geometry AS OBJECT (
 SDO_GTYPE NUMBER,
 SDO_SRID NUMBER,
 SDO_POINT SDO_POINT_TYPE,
 SDO_ELEM_INFO MDSYS.SDO_ELEM_INFO_ARRAY,
 SDO_ORDINATES MDSYS.SDO_ORDINATE_ARRAY);
```

The sections that follow describe the semantics of each SDO_GEOMETRY attribute, and then describe some usage considerations (Section 2.2.6).

The SDO_GEOMETRY object type has methods that provide convenient access to some of the attributes. These methods are described in Chapter 10.

2.2.1 SDO_GTYPE

SDO_GTYPE indicates the type of the geometry. Valid geometry types correspond to those specified in the *Geometry Object Model for the OGIS Simple Features for SQL* specification (with the exception of Surfaces.) The numeric values differ from those given in the OGIS specification, but there is a direct correspondence between the names and semantics where applicable.

The SDO_GTYPE value is 4 digits in the format *dltt*, where:

- *d* identifies the number of dimensions (2, 3, or 4)
- *l* identifies the linear referencing measure dimension for a 3-dimensional linear referencing system (LRS) geometry, that is, which dimension (3 or 4) contains the measure value. For a non-LRS geometry, or to accept the Spatial default of

the last dimension as the measure for an LRS geometry, specify 0. For information about the linear referencing system (LRS), see Chapter 6.

- *tt* identifies the geometry type (00 through 07, with 08 through 99 reserved for future use).

Table 2–1 shows the valid SDO_GTYPE values. The Geometry Type and Description values reflect the OGIS specification.

Table 2–1 Valid SDO_GTYPE Values

Value	Geometry Type	Description
*dl*00	UNKNOWN_ GEOMETRY	Spatial ignores this geometry.
*dl*01	POINT	Geometry contains one point.
*dl*02	LINE or CURVE	Geometry contains one line string that can contain straight or circular arc segments, or both. (LINE and CURVE are synonymous in this context.)
*dl*03	POLYGON	Geometry contains one polygon with or without holes.[1]
*dl*04	COLLECTION	Geometry is a heterogeneous collection of elements.[2] COLLECTION is a superset that includes all other types.
*dl*05	MULTIPOINT	Geometry has one or more points. (MULTIPOINT is a superset of POINT.)
*dl*06	MULTILINE or MULTICURVE	Geometry has one or more line strings. (MULTILINE and MULTICURVE are synonymous in this context, and each is a superset of both LINE and CURVE.)
*dl*07	MULTIPOLYGON	Geometry can have multiple, disjoint polygons (more than one exterior boundary). (MULTIPOLYGON is a superset of POLYGON.)

[1] For a polygon with holes, enter the exterior boundary first, followed by any interior boundaries.

[2] Polygons in the collection can be disjoint.

The *d* in the Value column of Table 2–1 is the number of dimensions: 2, 3, or 4. For example, an SDO_GTYPE value of 2003 indicates a 2-dimensional polygon.

Note: The pre-release 8.1.6 format of a 1-digit SDO_GTYPE value is still supported. If a 1-digit value is used, however, Oracle Spatial determines the number of dimensions from the DIMINFO column of the metadata views described in Section 2.4.

Also, if 1-digit SDO_GTYPE values are converted to 4-digit values, any SDO_ETYPE values that end in 3 or 5 in the SDO_ELEM_INFO array (described in Section 2.2.4) must also be converted.

The number of dimensions reflects the number of ordinates used to represent each vertex (for example, *X*,*Y* for 2-dimensional objects). Points and lines are considered 2-dimensional objects. (However, see Section 6.2 for dimension information about LRS points.)

In any given layer (column), all geometries must have the same number of dimensions. For example, you cannot mix 2-dimensional and 3-dimensional data in the same layer.

The following methods are available for returning the individual *dltt* components of the SDO_GTYPE for a geometry object: GET_DIMS, GET_LRS_DIM, and GET_GTYPE. These methods are described in Chapter 10.

2.2.2 SDO_SRID

SDO_SRID can be used to identify a coordinate system (spatial reference system) to be associated with the geometry. If SDO_SRID is null, no coordinate system is associated with the geometry. If SDO_SRID is not null, it must contain a value from the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1), and this value must be inserted into the SRID column of the USER_SDO_GEOM_METADATA view (described in Section 2.4).

All geometries in a geometry column must have the same SDO_SRID value.

For information about coordinate systems, see Chapter 5.

2.2.3 SDO_POINT

SDO_POINT is defined using an object type with attributes X, Y, and Z, all of type NUMBER. If the SDO_ELEM_INFO and SDO_ORDINATES arrays are both null, and the SDO_POINT attribute is non-null, then the X and Y values are considered to be the coordinates for a point geometry. Otherwise, the SDO_POINT attribute is ignored by Spatial. You should store point geometries in the SDO_POINT attribute

for optimal storage; and if you have only point geometries in a layer, it is strongly recommended that you store the point geometries in the SDO_POINT attribute.

Note: Do not use the SDO_POINT attribute in defining a linear referencing system (LRS) point. For information about LRS, see Chapter 6.

2.2.4 SDO_ELEM_INFO

SDO_ELEM_INFO is defined using a varying length array of numbers. This attribute lets you know how to interpret the ordinates stored in the SDO_ORDINATES attribute (described in Section 2.2.5).

Each triplet set of numbers is interpreted as follows:

- SDO_STARTING_OFFSET -- Indicates the offset within the SDO_ORDINATES array where the first ordinate for this element is stored. Offset values start at 1 and not at 0. Thus, the first ordinate for the first element will be at SDO_GEOMETRY.SDO_ORDINATES(1). If there is a second element, its first ordinate will be at SDO_GEOMETRY.SDO_ORDINATES(*n*), where *n* reflects the position within the SDO_ORDINATE_ARRAY definition (for example, 19 for the 19th number, as in Figure 2–3 later in this chapter).
- SDO_ETYPE - Indicates the type of the element. Valid values are shown in Table 2–2.

  SDO_ETYPE values 1, 2, 1003, and 2003 are considered *simple elements*. They are defined by a single triplet entry in the SDO_ELEM_INFO array. For SDO_ETYPE values 1003 and 2003, the first digit indicates *exterior* (1) or *interior* (2):

  1003: exterior polygon ring (must be specified in counterclockwise order)

  2003: interior polygon ring (must be specified in clockwise order)

Note: The use of 3 as an SDO_ETYPE value for polygon ring elements in a single geometry is discouraged. You should specify 3 only if you do not know if the simple polygon is exterior or interior, and you should then migrate the table or layer to the current format using the SDO_MIGRATE.TO_CURRENT procedure, described in Chapter 16.

You cannot mix 1-digit and 4-digit SDO_ETYPE values in a single geometry. If you use 4-digit SDO_ETYPE values, you must use 4-digit SDO_GTYPE values.

SDO_ETYPE values 4, 1005, and 2005 considered *compound elements*. They contain at least one header triplet with a series of triplet values that belong to the compound element. For SDO_ETYPE values 1005 and 2005, the first digit indicates *exterior* (1) or *interior* (2):

1005: exterior polygon ring (must be specified in counterclockwise order)

2005: interior polygon ring (must be specified in clockwise order)

Note: The use of 5 as an SDO_ETYPE value for polygon ring elements in a single geometry is discouraged. You should specify 5 only if you do not know if the compound polygon is exterior or interior, and you should then migrate the table or layer to the current format using the SDO_MIGRATE.TO_CURRENT procedure, described in Chapter 16.

You cannot mix 1-digit and 4-digit SDO_ETYPE values in a single geometry. If you use 4-digit SDO_ETYPE values, you must use 4-digit SDO_GTYPE values.

The elements of a compound element are contiguous. The last point of a subelement in a compound element is the first point of the next subelement. The point is not repeated.

- SDO_INTERPRETATION - Means one of two things, depending on whether or not SDO_ETYPE is a compound element.

  If SDO_ETYPE is a compound element (4, 1005, or 2005), this field specifies how many subsequent triplet values are part of the element.

If the SDO_ETYPE is not a compound element (1, 2, 1003, or 2003), the interpretation attribute determines how the sequence of ordinates for this element is interpreted. For example, a line string or polygon boundary may be made up of a sequence of connected straight line segments or circular arcs.

Descriptions of valid SDO_ETYPE and SDO_INTERPRETATION value pairs are given in Table 2–2.

If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the SDO_ORDINATES varying length array.

For compound elements (SDO_ETYPE values 4 and 5), a set of *n* triplets (one per subelement) is used to describe the element. It is important to remember that subelements of a compound element are contiguous. The last point of a subelement is the first point of the next subelement. For subelements 1 through *n*-1, the end point of one subelement is the same as the starting point of the next subelement. The starting point for subelements 2...*n*-2 is the same as the end point of subelement 1...*n*-1. The last ordinate of subelement *n* is either the starting offset minus 1 of the next element in the geometry, or the last ordinate in the SDO_ORDINATES varying length array.

The current size of a varying length array can be determined by using the function varray_variable.Count in PL/SQL or OCIColSize in the Oracle Call Interface (OCI).

The semantics of each SDO_ETYPE element and the relationship between the SDO_ELEM_INFO and SDO_ORDINATES varying length arrays for each of these SDO_ETYPE elements are given in Table 2–2.

Table 2–2 Values and Semantics in SDO_ELEM_INFO

SDO_ETYPE	SDO_INTERPRETATION	Meaning
0	(any numeric value)	Type 0 (zero) element. Used to model geometry types not supported by Oracle Spatial. For more information, see Section 2.3.5.
1	1	Point type.
1	*n* > 1	Point cluster with *n* points.
2	1	Line string whose vertices are connected by straight line segments.

Table 2–2 Values and Semantics in SDO_ELEM_INFO (Cont.)

SDO_ ETYPE	SDO_ INTERPRETATION	Meaning
2	2	Line string made up of a connected sequence of circular arcs. Each circular arc is described using three coordinates: the arc's start point, any point on the arc, and the arc's end point. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a line string made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc, where point 3 is only stored once.
1003 or 2003	1	Simple polygon whose vertices are connected by straight line segments. Note that you must specify a point for each vertex, and the last point specified must be identical to the first (to close the polygon). For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.
1003 or 2003	2	Polygon made up of a connected sequence of circular arcs that closes on itself. The end point of the last arc is the same as the start point of the first arc. Each circular arc is described using three coordinates: the arc's start point, any point on the arc, and the arc's end point. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a polygon made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc. The coordinates for points 1 and 5 must be the same, and point 3 is not repeated.
1003 or 2003	3	Rectangle type (sometimes called *optimized rectangle*). A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it. Using this type (that is, defining a rectangle using only two points) is not supported for geodetic data; it is supported only for data associated with a Cartesian coordinate system. With geodetic data, define a rectangle using 5 points (with point 5 the same as point 1) and an SDO_INTERPRETATION value of 1. (You can also use the SDO_CS.VIEWPORT_TRANSFORM function to convert optimized rectangles to valid geodetic rectangles for use with the SDO_FILTER operator.)
1003 or 2003	4	Circle type. Described by three points, all on the circumference of the circle.

Table 2–2 Values and Semantics in SDO_ELEM_INFO (Cont.)

SDO_ETYPE	SDO_INTERPRETATION	Meaning
4	*n* > 1	Compound line string with some vertices connected by straight line segments and some by circular arcs. The value, *n*, in the Interpretation column specifies the number of contiguous subelements that make up the line string. The next *n* triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The last point of a subelement is the first point of the next subelement, and must not be repeated. See Section 2.3.3 and Figure 2–4 for an example of a geometry using this type.
1005 or 2005	*n* > 1	Compound polygon with some vertices connected by straight line segments and some by circular arcs. The value, *n*, in the Interpretation column specifies the number of contiguous subelements that make up the polygon. The next *n* triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The end point of a subelement is the start point of the next subelement, and it must not be repeated. The start and end points of the polygon must be the same. See Section 2.3.4 and Figure 2–5 for an example of a geometry using this type.

2.2.5 SDO_ORDINATES

SDO_ORDINATES is defined using a varying length array (1048576) of NUMBER type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the SDO_ELEM_INFO varying length array. The values in the array are ordered by dimension. For example, a polygon whose boundary has four 2-dimensional points is stored as {X1, Y1, X2, Y2, X3, Y3, X4, Y4, X1, Y1}. If the points are 3-dimensional, then they are stored as {X1, Y1, Z1, X2, Y2, Z2, X3, Y3, Z3, X4, Y4, Z4, X1, Y1, Z1}. Spatial index creation, operators, and functions ignore the Z values because this release of the product supports only 2-dimensional spatial objects. The number of dimensions associated with each point is stored as metadata in the xxx_SDO_GEOM_METADATA views, described in Section 2.4.

The values in the SDO_ORDINATES array must all be valid and non-null. There are no special values used to delimit elements in a multielement geometry. The start and end points for the sequence describing a specific element are determined by the

STARTING_OFFSET values for that element and the next element in the SDO_ELEM_INFO array as explained previously. The offset values start at 1. SDO_ORDINATES(1) is the first ordinate of the first point of the first element.

2.2.6 Usage Considerations

You should use the SDO_GTYPE values as shown in Table 2–1; however, Spatial does not check or enforce all geometry consistency constraints. Spatial does check the following:

- For SDO_GTYPE values *d*001 and *d*005, any subelement not of SDO_ETYPE 1 is ignored.
- For SDO_GTYPE values *d*002 and *d*006, any subelement not of SDO_ETYPE 2 or 4 is ignored.
- For SDO_GTYPE values *d*003 and *d*007, any subelement not of SDO_ETYPE 3 or 5 is ignored. (This includes SDO_ETYPE variants 1003, 2003, 1005, and 2005, which are explained in Section 2.2.4).

The SDO_GEOM.VALIDATE_GEOMETRY function can be used to evaluate the consistency of a single geometry object or all the instances of SDO_GEOMETRY in a specified feature table.

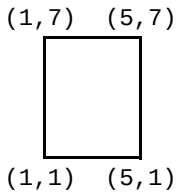
2.3 Geometry Examples

This section contains examples of several geometry types.

2.3.1 Rectangle

Figure 2–2 illustrates the rectangle that represents *cola_a* in the example in Section 2.1.

Figure 2–2 Rectangle



In the SDO_GEOMETRY definition of the geometry illustrated in Figure 2–2:

- SDO_GTYPE = 2003. The *2* indicates two-dimensional, and the *3* indicates a polygon.
- SDO_SRID = NULL.
- SDO_POINT = NULL.
- SDL_ELEM_INFO = (1, 1003, 3). The final *3* in 1,1003,3 indicates that this is a rectangle. Because it is a rectangle, only two ordinates are specified in SDO_ORDINATES (lower-left and upper-right).
- SDO_ORDINATES = (1,1, 5,7). These identify the lower-left and upper-right ordinates of the rectangle.

Example 2–2 shows a SQL statement that inserts the geometry illustrated in Figure 2–2 into the database.

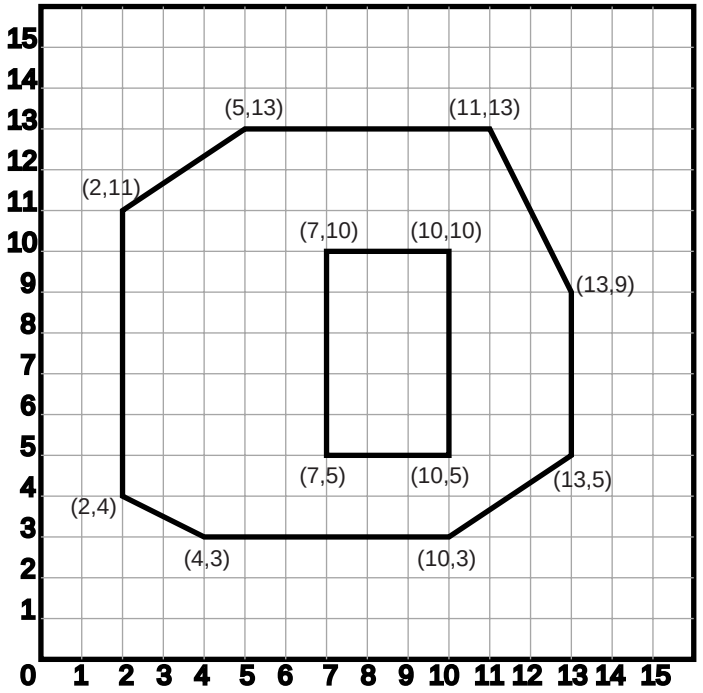
Example 2–2 SQL Statement to Insert a Rectangle

```
INSERT INTO cola_markets VALUES(
  1,
  'cola_a',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3), -- one rectangle (1003 = exterior)
    MDSYS.SDO_ORDINATE_ARRAY(1,1, 5,7) -- only 2 points needed to
          -- define rectangle (lower left and upper right) with
          -- Cartesian-coordinate data
  )
);
```

2.3.2 Polygon with a Hole

Figure 2–3 illustrates a polygon consisting of two elements: an exterior polygon ring and an interior polygon ring. The inner element in this example is treated as a void (a hole).

Figure 2–3 Polygon with a Hole



In the SDO_GEOMETRY definition of the geometry illustrated in Figure 2–3:

- SDO_GTYPE = 2003. The *2* indicates two-dimensional, and the *3* indicates a polygon.
- SDO_SRID = NULL.
- SDO_POINT = NULL.
- SDO_ELEM_INFO = (1,1003,1, 19,2003,1). There are two triplet elements: 1,1003,1 and 19,2003,1.

  1003 indicates that the element is an exterior polygon ring; *2003* indicates that the element is an interior polygon ring.

  19 indicates that the second element (the interior polygon ring) ordinate specification starts at the 19th number in the SDO_ORDINATES array (that is, 7, meaning that the first point is 7,5).

- SDO_ORDINATES = (2,4, 4,3, 10,3, 13,5, 13,9, 11,13, 5,13, 2,11, 2,4, 7,5, 7,10, 10,10, 10,5, 7,5).
- The area (SDO_GEOM.SDO_AREA function) of the polygon is the area of the exterior polygon minus the area of the interior polygon. In this example, the area is 84 (99 - 15).
- The perimeter (SDO_GEOM.SDO_LENGTH function) of the polygon is the perimeter of the exterior polygon plus the perimeter of the interior polygon. In this example, the perimeter is 52.9193065 (36.9193065 + 16).

Example 2–3 shows a SQL statement that inserts the geometry illustrated in Figure 2–3 into the database.

Example 2–3 SQL Statement to Insert a Polygon with a Hole

```
INSERT INTO cola_markets VALUES(
  10,
  'polygon_with_hole',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1, 19,2003,1), -- polygon with hole
    MDSYS.SDO_ORDINATE_ARRAY(2,4, 4,3, 10,3, 13,5, 13,9, 11,13, 5,13, 2,11, 2,4,
        7,5, 7,10, 10,10, 10,5, 7,5)
  )
);
```

An example of such a "polygon with a hole" might be a land mass (such as a country or an island) with a lake inside it. Of course, an actual land mass might have many such interior polygons: each one would require a triplet element in SDO_ELEM_INFO, plus the necessary ordinate specification.

Exterior and interior rings cannot be nested. For example, if a country has a lake and there is an island in the lake (and perhaps a lake on the island), a separate polygon must be defined for the island; the island cannot be defined as an interior polygon ring within the interior polygon ring of the lake.

In a **multipolygon** (polygon collection), rings must be grouped by polygon, and the first ring of each polygon must be the exterior ring. For example, consider a polygon collection that contains two polygons (A and B):

- Polygon A (one interior "hole"): exterior ring A0, interior ring A1

- Polygon B (two interior "holes"): exterior ring B0, interior ring B1, interior ring B2

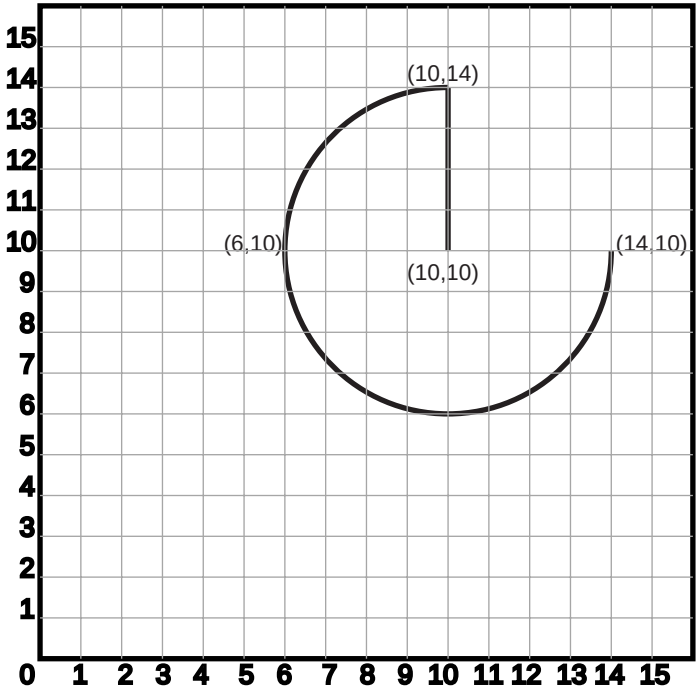
The elements in SDO_ELEM_INFO and SDO_ORDINATES must be in one of the following orders (depending on whether you specify Polygon A or Polygon B first):

- A0, A1; B0, B1, B2
- B0, B1, B2; A0, A1

2.3.3 Compound Line String

Figure 2–4 illustrates a crescent-shaped object represented as a compound line string made up of one straight line segment and one circular arc. Four points are required to represent this shape: points (10,10) and (10,14) describe the straight line segment, and points (10,14), (10,6), and (14,10) describe the circular arc.

Figure 2–4 Compound Line String



In the SDO_GEOMETRY definition of the geometry illustrated in Figure 2–4:

- SDO_GTYPE = 2002. The first *2* indicates two-dimensional, and the second *2* indicates one or more line segments.
- SDO_SRID = NULL.
- SDO_POINT = NULL.
- SDO_ELEM_INFO = (1,4,2, 1,2,1, 3,2,2). There are three triplet elements: 1,4,2, 1,2,1, and 3,2,2.

  The first triplet indicates that this element is a compound line string made up of two subelement line strings, which are described with the next two triplets.

  The second triplet indicates that the line string is made up of straight line segments and that the ordinates for this line string start at offset 1. The end point of this line string is determined by the starting offset of the second line string, 3 in this instance.

  The third triplet indicates that the second line string is made up of circular arcs with ordinates starting at offset 3. The end point of this line string is determined by the starting offset of the next element or the current length of the SDO_ORDINATES array, if this is the last element.
- SDO_ORDINATES = (10,10, 10,14, 6,10, 14,10).

Example 2–4 shows a SQL statement that inserts the geometry illustrated in Figure 2–4 into the database.

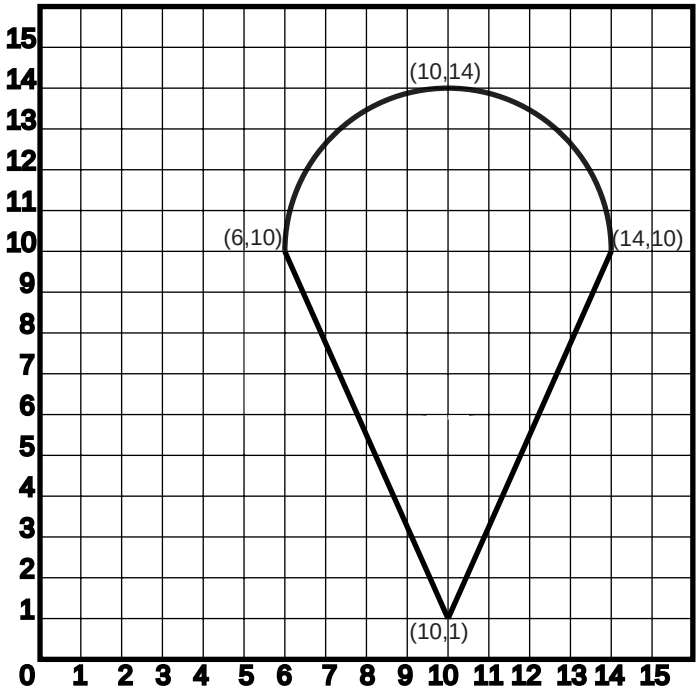
Example 2–4 SQL Statement to Insert a Compound Line String

```
INSERT INTO cola_markets VALUES(
  11,
  'compound_line_string',
  MDSYS.SDO_GEOMETRY(
    2002,
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,4,2, 1,2,1, 3,2,2), -- compound line string
    MDSYS.SDO_ORDINATE_ARRAY(10,10, 10,14, 6,10, 14,10)
  )
);
```

2.3.4 Compound Polygon

Figure 2–5 illustrates an ice cream cone-shaped object represented as a compound polygon made up of one straight line segment and one circular arc. Five points are required to represent this shape: points (6,10), (10,1), and (14,10) describe one acute angle-shaped line string, and points (14,10), (10,14), and (6,10) describe the circular arc. The starting point of the line string and the ending point of the circular arc are the same point (6,10). The SDO_ELEM_INFO array contains three triplets for this compound line string. These triplets are {(1,1005,2), (1,2,1), (5,2,2)}.

Figure 2–5 Compound Polygon



In the SDO_GEOMETRY definition of the geometry illustrated in Figure 2–5:

- SDO_GTYPE = 2003. The *2* indicates two-dimensional, and the *3* indicates a polygon.
- SDO_SRID = NULL.

- SDO_POINT = NULL.
- SDO_ELEM_INFO = (1,1005,2, 1,2,1, 5,2,2). There are three triplet elements: 1,1005,2, 1,2,1, and 5,2,2.

  The first triplet indicates that this element is a compound polygon made up of two subelement line strings, which are described using the next two triplets.

  The second triplet indicates that the first subelement line string is made up of straight line segments and that the ordinates for this line string start at offset 1. The end point of this line string is determined by the starting offset of the second line string, 5 in this instance. Because the vertices are 2-dimensional, the coordinates for the end point of the first line string are at ordinates 5 and 6.

  The third triplet indicates that the second subelement line string is made up of a circular arc with ordinates starting at offset 5. The end point of this line string is determined by the starting offset of the next element or the current length of the SDO_ORDINATES array, if this is the last element.
- SDO_ORDINATES = (6,10, 10,1, 14,10, 10,14, 6,10).

Example 2–5 shows a SQL statement that inserts the geometry illustrated in Figure 2–5 into the database.

Example 2–5 SQL Statement to Insert a Compound Polygon

```
INSERT INTO cola_markets VALUES(
  12,
  'compound_polygon',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1005,2, 1,2,1, 5,2,2), -- compound polygon
    MDSYS.SDO_ORDINATE_ARRAY(6,10, 10,1, 14,10, 10,14, 6,10)
  )
);
```

2.3.5 Type 0 (Zero) Element

Type 0 (zero) elements are used to model geometry types that are not supported by Oracle Spatial, such as curves and splines. A type 0 element has an SDO_ETYPE value of 0. (See Section 2.2.4 for information about the SDO_ETYPE.) Type 0 elements are not indexed by Oracle Spatial, and they are ignored by Spatial functions and procedures.

Geometries with type 0 elements must contain at least one nonzero element, that is, an element with an SDO_ETYPE value that is not 0. The nonzero element should be an approximation of the unsupported geometry, and therefore it must have both:

- An SDO_ETYPE value associated with a geometry type supported by Spatial
- An SDO_INTERPRETATION value that is valid for the SDO_ETYPE value (see Table 2–2)

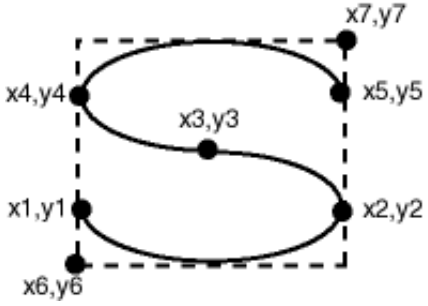
  (The SDO_INTERPRETATION value for the type 0 element can be any numeric value, and applications are responsible for determining the validity and significance of the value.)

The nonzero element is indexed by Spatial, and it will be returned by the spatial index.

The SDO_GTYPE value for a geometry containing a type 0 element must be set to the value for the geometry type of the nonzero element.

Figure 2–6 shows a geometry with two elements: a curve (unsupported geometry) and a rectangle (the nonzero element) that approximates the curve. The curve looks like the letter *S*, and the rectangle is represented by the dashed line.

Figure 2–6 Geometry with Type 0 (Zero) Element



In the example shown in Figure 2–6:

- The SDO_GTYPE value for the geometry is 2003 (for a 2-dimensional polygon).
- The SDO_ELEM_INFO array contains two triplets for this compound line string. For example, the triplets might be {(1,0,57), (11,1003,3)}. That is:

Ordinate Starting Offset (SDO_STARTING_OFFSET)	Element Type (SDO_ETYPE)	Interpretation (SDO_INTERPRETATION)
1	0	57
11	1003	3

In this example:

- The type 0 element has an SDO_ETYPE value of 0.
- The nonzero element (rectangle) has an SDO_ETYPE value of 1003, indicating an exterior polygon ring.
- The nonzero element has an SDO_STARTING_OFFSET value of 11 because ordinate x6 is the eleventh ordinate in the geometry.
- The type 0 element has an SDO_INTERPRETATION value whose significance is application-specific. In this example, the SDO_INTERPRETATION value is 57.
- The nonzero element has an SDO_INTERPRETATION value that is valid for the SDO_ETYPE of 1003. In this example, the SDO_INTERPRETATION value is 3, indicating a rectangle defined by two points (lower-left and upper-right).

Example 2–6 shows a SQL statement that inserts the geometry with a type 0 element (similar to the geometry illustrated in Figure 2–6) into the database. In the SDO_ORDINATE_ARRAY structure, the curve is defined by points (6,6), (12,6), (9,8), (6,10), and (12,10), and the rectangle is defined by points (6,4) and (12,12).

Example 2–6 SQL Statement to Insert a Geometry with a Type 0 Element

```
INSERT INTO cola_markets VALUES(
  13,
  'type_zero_element_geom',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,0,57, 11,1003,3), -- 1st is type 0 element
    MDSYS.SDO_ORDINATE_ARRAY(6,6, 12,6, 9,8, 6,10, 12,10, 6,4, 12,12)
  )
);
```

2.4 Geometry Metadata Structure

The geometry metadata describing the dimensions, lower and upper bounds, and tolerance in each dimension is stored in a global table owned by MDSYS (which users should never directly update). Each Spatial user has the following views available in the schema associated with that user:

- USER_SDO_GEOM_METADATA contains metadata information for all spatial tables owned by the user (schema). This is the only view that you can update, and it is the one in which Spatial users must insert metadata related to spatial tables.
- ALL_SDO_GEOM_METADATA contains metadata information for all spatial tables on which the user has SELECT permission.
- DBA_SDO_GEOM_METADATA contains metadata information for all spatial tables on which the user has SELECT permission if the user has the DBA role.

Spatial users are responsible for populating these views. For each spatial column, you must insert an appropriate row into the USER_SDO_GEOM_METADATA view. Oracle Spatial ensures that the other two views (ALL_SDO_GEOM_METADATA and DBA_SDO_GEOM_METADATA) are also updated to reflect the rows that you insert into USER_SDO_GEOM_METADATA.

Note: These views were new for release 8.1.6. If you are migrating from an earlier release of Spatial, see Appendix A and the information about the SDO_MIGRATE.TO_CURRENT procedure in Chapter 16.

Each metadata view has the following definition:

```
(
  TABLE_NAME   VARCHAR2(32),
  COLUMN_NAME  VARCHAR2(32),
  DIMINFO      MDSYS.SDO_DIM_ARRAY,
  SRID         NUMBER
);
```

In addition, the ALL_SDO_GEOM_METADATA and DBA_SDO_GEOM_METADATA views have an OWNER column identifying the schema that owns the table specified in TABLE_NAME.

2.4.1 TABLE_NAME

The TABLE_NAME column contains the name of a feature table, such as ROADS or PARKS, that has a column of type SDO_GEOMETRY.

2.4.2 COLUMN_NAME

The COLUMN_NAME column contains the name of the column of type SDO_GEOMETRY. For the tables ROADS and PARKS, this column is called THEGEOMETRY, and therefore the xxx_SDO_GEOM_METADATA views should contain rows with values *(ROADS, THEGEOMETRY, SOMEDIMINFO1, NULL)* and *(PARKS, THEGEOMETRY, SOMEDIMINFO2, NULL)*.

2.4.3 DIMINFO

The DIMINFO column is a varying length array of an object type, ordered by dimension, and has one entry per dimension. The SDO_DIM_ARRAY type is defined as follows:

```
Create Type SDO_DIM_ARRAY as VARRAY(4) of SDO_DIM_ELEMENT;
```

The SDO_DIM_ELEMENT type is defined as:

```
Create Type SDO_DIM_ELEMENT as OBJECT (
  SDO_DIMNAME VARCHAR2(64),
  SDO_LB NUMBER,
  SDO_UB NUMBER,
  SDO_TOLERANCE NUMBER);
```

The SDO_DIM_ARRAY instance is of size *n* if there are *n* dimensions. That is, DIMINFO contains 2 SDO_DIM_ELEMENT instances for 2-dimensional geometries, 3 instances for 3-dimensional geometries, and 4 instances for 4-dimensional geometries. Each SDO_DIM_ELEMENT instance in the array must have valid (not null) values for the SDO_LB, SDO_UB, and SDO_TOLERANCE attributes.

Note: The number of dimensions reflected in the DIMINFO information must match the number of dimensions of each geometry object in the layer.

For an explanation of tolerance and how to determine the appropriate SDO_TOLERANCE value, see Section 1.5.5, especially Section 1.5.5.1.

Spatial assumes that the varying length array is ordered by dimension, and therefore, in the ROADS and PARKS tables, *SomeDimInfo1* is the SDO_DIM_ELEMENT for the first dimension and *SomeDimInfo2* is the SDO_DIM_ELEMENT for the second dimension. The DIMINFO varying length array must be ordered by dimension in the same way the ordinates for the points in SDO_ORDINATES varying length array are ordered. For example, if the SDO_ORDINATES varying length array contains {X1, Y1, ..., Xn, Yn}, then *SomeDimInfo1* must define the X dimension and *SomeDimInfo2* must define the Y dimension.

Section 3.2 contains examples that show the use of the SDO_GEOMETRY and SDO_DIM_ARRAY types. These examples demonstrate how various geometry objects are represented, and how a feature table and the USER_SDO_GEOM_METADATA view are populated with the data for those objects.

2.4.4 SRID

The SRID column should contain either of the following: the SRID value for the coordinate system (see Chapter 5) for all geometries in the column, or NULL if no specific coordinate system should be associated with the geometries.

2.5 Spatial Index-Related Structures

This section describes the structure of the tables containing the spatial index data and metadata. Concepts and usage notes for spatial indexing are explained in Section 1.7. The spatial index data and metadata are stored in tables that are created and maintained by the Spatial indexing routines. These tables are created in the schema of the owner of the feature (underlying) table that has a spatial index created on a column of type SDO_GEOMETRY.

2.5.1 Spatial Index Views

There are two sets of spatial index metadata views per schema (user): xxx_SDO_INDEX_INFO and xxx_SDO_INDEX_METADATA, where *xxx* can be USER, DBA, or ALL. These views are read-only to users; they are created and maintained by the Spatial indexing routines.

2.5.1.1 xxx_SDO_INDEX_INFO Views

The following views contain basic information about spatial indexes:

- USER_SDO_INDEX_INFO contains index information for all spatial tables owned by the user.

- ALL_SDO_INDEX_INFO contains index information for all spatial tables on which the user has SELECT permission.
- DBA_SDO_INDEX_INFO contains index information for all spatial tables on which the user has SELECT permission if the user has the DBA role.

The USER_SDO_INDEX_INFO, ALL_SDO_INDEX_INFO, and DBA_SDO_INDEX_INFO views contain the same columns, as shown Table 2–3. (The columns are listed in their order in the view definition.)

Table 2–3 Columns in the xxx_SDO_INDEX_INFO Views

Column Name	Data Type	Purpose
INDEX_NAME	VARCHAR2	Name of the index.
TABLE_NAME	VARCHAR2	Name of the table containing the column on which this index is built.
COLUMN_NAME	VARCHAR2	Name of the column on which this index is built.
SDO_INDEX_TYPE	VARCHAR2	Contains QTREE (for a quadtree index) or RTREE (for an R-tree index).
SDO_INDEX_TABLE	VARCHAR2	Name of the spatial index table (described in Section 2.5.2).

2.5.1.2 xxx_SDO_INDEX_METADATA Views

The following views contain detailed information about spatial index metadata:

- USER_SDO_INDEX_METADATA contains index information for all spatial tables owned by the user. (USER_SDO_INDEX_METADATA is the same as SDO_INDEX_METADATA, which was the only metadata view for Oracle Spatial release 8.1.5.)
- ALL_SDO_INDEX_METADATA contains index information for all spatial tables on which the user has SELECT permission.
- DBA_SDO_INDEX_METADATA contains index information for all spatial tables on which the user has SELECT permission if the user has the DBA role.

> **Note:** These views were new for release 8.1.6. If you are migrating from an earlier release of Spatial, see Appendix A.

The USER_SDO_INDEX_METADATA, ALL_SDO_INDEX_METADATA, and DBA_SDO_INDEX_METADATA views contain the same columns, as shown Table 2–4. (The columns are listed in their order in the view definition.)

Table 2–4 Columns in the xxx_SDO_INDEX_METADATA Views

Column Name	Data Type	Purpose
SDO_INDEX_OWNER	VARCHAR2	Owner of the index.
SDO_INDEX_TYPE	VARCHAR2	Contains QTREE (for a quadtree index) or RTREE (for an R-tree index).
SDO_INDEX_NAME	VARCHAR2	Name of the index.
SDO_INDEX_TABLE	VARCHAR2	Name of the spatial index table (described in Section 2.5.2).
SDO_INDEX_PRIMARY	NUMBER	Indicates if this is a primary or secondary index. 1 = primary, 2 = secondary.
SDO_INDEX_PARTITION	VARCHAR2	For a partitioned index, name of the index partition.
SDO_PARTITIONED	NUMBER	Contains 0 if the index is not partitioned or 1 if the index is partitioned.
SDO_TSNAME	VARCHAR2	Schema name of the SDO_INDEX_TABLE.
SDO_COLUMN_NAME	VARCHAR2	Name of the column on which this index is built.
SDO_INDEX_DIMS	NUMBER	Number of dimensions of the geometry objects in the column on which this index is built.
SDO_RTREE_HEIGHT	NUMBER	Height of the R-tree for an R-tree index.
SDO_RTREE_NUM_NODES	NUMBER	Number of nodes in the R-tree for an R-tree index.
SDO_RTREE_DIMENSIONALITY	NUMBER	Number of dimensions indexed for an R-tree index.
SDO_RTREE_FANOUT	NUMBER	Maximum number of children in each R-tree node for an R-tree index.
SDO_RTREE_ROOT	VARCHAR2	Rowid corresponding to the root node of the R-tree in the index table for an R-tree index.
SDO_RTREE_SEQ_NAME	VARCHAR2	Sequence name associated with the R-tree for an R-tree index.

Table 2–4 Columns in the xxx_SDO_INDEX_METADATA Views (Cont.)

Column Name	Data Type	Purpose
SDO_RTREE_PCTFREE	VARCHAR2	Minimum percentage of slots in each index tree node to be left empty when an R-tree index is created.
SDO_LAYER_GTYPE	VARCHAR2	Contains DEFAULT if the layer can contain both point and polygon data, or a value from the Geometry Type column of Table 2–1 in Section 2.2.1.
SDO_LEVEL	NUMBER	The fixed tiling level at which to tile all objects in the geometry column for a quadtree index.
SDO_NUMTILES	NUMBER	Suggested number of tiles per object that should be used to approximate the shape for a quadtree index.
SDO_MAXLEVEL	NUMBER	Maximum level for any tile for any object for a quadtree index. It will always be greater than the SDO_LEVEL value.
SDO_COMMIT_INTERVAL	NUMBER	Number of geometries (rows) to process, during index creation, before committing the insertion of spatial index entries into the SDOINDEX table.
SDO_FIXED_META	RAW	If applicable, this column contains the metadata portion of the SDO_GROUPCODE or SDO_CODE for a fixed-level index.
SDO_TABLESPACE	VARCHAR2	Same as in the SQL CREATE TABLE statement. Tablespace in which to create the SDOINDEX table.
SDO_INITIAL_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_NEXT_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_PCTINCREASE	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_MIN_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_MAX_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_RTREE_QUALITY	NUMBER	Quality score for an R-tree index. Do not attempt to interpret this value directly; instead, use the SDO_TUNE.ANALYZE_RTREE procedure and the SDO_TUNE.QUALITY_DEGRADATION function, which are described in Chapter 17.

Table 2–4 Columns in the xxx_SDO_INDEX_METADATA Views (Cont.)

Column Name	Data Type	Purpose
SDO_INDEX_VERSION	NUMBER	Internal version number of the index.

2.5.2 Spatial Index Table Definition

The information in each quadtree spatial index table (each SDO_INDEX_TABLE entry as described in Table 2–4 in Section 2.5.1) depends on whether the index is an R-tree index or a quadtree index.

For an R-tree index, the spatial index table contains the columns shown in Table 2–5.

Table 2–5 Columns in an R-tree Spatial Index Data Table

Column Name	Data Type	Purpose
NODE_ID	NUMBER	Unique ID number for this node of the tree.
NODE_LEVEL	NUMBER	Level of the node in the tree. Leaf nodes (nodes whose entries point to data items in base table) are at level 1, their parent nodes are at level 2, and so on.
INFO	BLOB	Other information in a node. Includes an array of <*child_mbr, child_rowid*> pairs (maximum of fanout value, or number of children in each R-tree node, such pairs), where *child_rowid* is the rowid of a child node, or the rowid of a data item from the base table.

For a quadtree index, the spatial index table contains the columns shown in Table 2–6.

Table 2–6 Columns in a Quadtree Spatial Index Data Table

Column Name	Data Type	Purpose
SDO_CODE	RAW	Index entry for the object in the row identified by SDO_ROWID.
SDO_ROWID	ROWID	Rowid of a row in a feature table containing the indexed object.
SDO_STATUS	VARCHAR2	Contains *I* if the tile is inside the geometry, or contains *B* if the tile is on the boundary of the geometry.
SDO_GROUPCODE	RAW	Index entry at level SDO_LEVEL (hybrid indexes only).

For a quadtree index, the SDO_CODE, SDO_ROWID, and SDO_STATUS columns are always present. The SDO_GROUPCODE column is present only when the selected index type is HYBRID.

2.5.3 R-Tree Index Sequence Object

Each R-tree spatial index table has an associated sequence object (SDO_RTREE_SEQ_NAME in the USER_SDO_INDEX_METADATA view, described in Table 2–4 in Section 2.5.1). The sequence is used to ensure that simultaneous updates can be performed to the index by multiple concurrent users.

The sequence name is the index table name with the letter *S* as a suffix.

2.6 Unit of Measurement Support

Geometry functions that involve measurement allow an optional *unit* parameter to specify the unit of measurement for a specified distance or area, if a georeferenced coordinate system (SDO_SRID value) is associated with the input geometry or geometries. The *unit* parameter is not valid for geometries with a null SDO_SRID value (that is, an orthogonal Cartesian system). For information about support for coordinate systems, see Chapter 5.

The default unit of measure is the one associated with the georeferenced coordinate system. The unit of measure for most coordinate systems is the meter, and in these cases the default unit for distances is meter and the default unit for areas is square meter. By using the *unit* parameter, however, you can have Spatial automatically convert and return results that are more meaningful to application users, for example, displaying the distance to a restaurant in miles.

The *unit* parameter must be enclosed in single quotation marks and contain the string *unit*= and a valid SDO_UNIT value from the MDSYS.SDO_DIST_UNITS or MDSYS.SDO_AREA_UNITS table. For example, 'unit=KM' in the following example specifies kilometers as the unit of measurement:

```
SELECT c.name, SDO_GEOM.SDO_LENGTH(c.shape, m.diminfo, 'unit=KM')
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE';
```

Spatial uses the information in the MDSYS.SDO_DIST_UNITS and MDSYS.SDO_AREA_UNITS tables to determine which unit names are valid and what ratios to use in comparing or converting between different units.

The MDSYS.SDO_DIST_UNITS table contains the columns shown in Table 2–7.

Table 2–7 Columns in the SDO_DIST_UNITS Table

Column Name	Data Type	Purpose
SDO_UNIT	VARCHAR2	Unit string to be specified with the *unit* parameter. Examples: *M*, *KM*, *CM*, *MM*, *MILE*, *NAUT_MILE*, *FOOT*, *INCH*
UNIT_NAME	VARCHAR2	Descriptive name of the unit. Examples: *Meter*, *Kilometer*, *Centimeter*, *Millimeter*, *Mile*, *Nautical Mile*, *Foot*, *Inch*
CONVERSION_ FACTOR	NUMBER	Ratio of the unit to 1 meter. For example, the conversion factor for a meter is 1.0, and the conversion factor for a mile is 1609.344.

The MDSYS.SDO_AREA_UNITS table contains the columns shown in Table 2–8.

Table 2–8 Columns in the SDO_AREA_UNITS Table

Column Name	Data Type	Purpose
SDO_UNIT	VARCHAR2	Unit string to be specified with the *unit* parameter. Examples: SQ_*M*, SQ_*KM*, SQ_*CM*, SQ_*MM*, SQ_ *MILE*, SQ_*FOOT*, SQ_*INCH*
UNIT_NAME	VARCHAR2	Descriptive name of the unit. Examples: *Square Meter*, *Square Kilometer*, *Square Centimeter*, *Square Millimeter*, *Square Mile*, *Square Foot*, *Square Inch*
CONVERSION_ FACTOR	NUMBER	Ratio of the unit to 1 square meter. For example, the conversion factor for a square meter is 1.0, and the conversion factor for a square mile is 2589988.

For a complete list of supported unit strings, unit names, and conversion factors, view the contents of the MDSYS.SDO_DIST_UNITS and MDSYS.SDO_AREA_ UNITS tables. For example:

```
SELECT * from MDSYS.SDO_DIST_UNITS;
SELECT * from MDSYS.SDO_AREA_UNITS;
```

3

Loading Spatial Data

This chapter describes how to load spatial data into a database, including storing the data in a table with a column of type SDO_GEOMETRY. After you have loaded spatial data, you can create a spatial index for it and perform queries on it, as described in Chapter 4.

The process of loading data can be classified into two categories:

- Bulk loading of data

  This process is used to load large volumes of data into the database and uses the SQL*Loader utility to load the data.

- Transactional insert operations

  This process is used to insert relatively small amounts of data into the database using the INSERT statement in SQL.

3.1 Bulk Loading

Bulk loading can import large amounts of ASCII data into an Oracle database. Bulk loading is accomplished with the SQL*Loader utility. (For information about SQL*Loader, see *Oracle9i Database Utilities*.)

3.1.1 Bulk Loading SDO_GEOMETRY Objects

Example 3–1 is the SQL*Loader control file for loading four geometries. When this control file is used with SQL*Loader, it loads the same cola market geometries that are inserted using SQL statements in Example 2–1 in Section 2.1.

Example 3–1 Control File for Bulk Load of Cola Market Geometries

```
LOAD DATA
```

```
INFILE *
TRUNCATE
CONTINUEIF NEXT(1:1) = '#'
INTO TABLE COLA_MARKETS
FIELDS TERMINATED BY '|'
TRAILING NULLCOLS (
mkt_id INTEGER EXTERNAL,
name CHAR,
shape COLUMN OBJECT
(
SDO_GTYPE INTEGER EXTERNAL,
SDO_ELEM_INFO VARRAY TERMINATED BY '|/'
(elements FLOAT EXTERNAL),
SDO_ORDINATES VARRAY TERMINATED BY '|/'
(ordinates FLOAT EXTERNAL)
)
)
begindata
 1|cola_a|
#2003|1|1003|3|/
#1|1|5|7|/
 2|cola_b|
#2003|1|1003|1|/
#5|1|8|1|8|6|5|7|5|1|/
 3|cola_c|
#2003|1|1003|1|/
#3|3|6|3|6|5|4|5|3|3|/
 4|cola_d|
#2003|1|1003|4|/
#8|7|10|9|8|11|/
```

Notes on Example 3–1:

- The `EXTERNAL` keyword in the definition `mkt_id INTEGER EXTERNAL` means that each value to be inserted into the MKT_ID column (1, 2, 3, and 4 in this example) is an integer in human-readable form, not binary format.
- In the data after `begindata`, each MKT_ID value is preceded by one space, because the `CONTINUEIF NEXT(1:1) = '#'` specification causes the first position of each data line to be ignored unless it is the asterisk (#) continuation character.

Example 3–2 assumes that a table named POLY_4PT was created as follows:

```
CREATE TABLE POLY_4PT (GID      VARCHAR2(32),
                       GEOMETRY  MDSYS.SDO_GEOMETRY);
```

Assume that the ASCII data consists of a file with delimited columns and separate rows fixed by the limits of the table with the following format:

```
geometry rows:    GID, GEOMETRY
```

The coordinates in the GEOMETRY column represent polygons. Example 3–2 shows the control file for loading the data.

Example 3–2 Control File for Bulk Load of Polygons

```
LOAD DATA
 INFILE *
 TRUNCATE
 CONTINUEIF NEXT(1:1) = '#'
 INTO TABLE POLY_4PT
 FIELDS TERMINATED BY '|'
 TRAILING NULLCOLS (
  GID  INTEGER EXTERNAL,
  GEOM COLUMN OBJECT
   (
     SDO_GTYPE       INTEGER EXTERNAL,
     SDO_ELEM_INFO   VARRAY TERMINATED BY '|/'
       (elements     FLOAT EXTERNAL),
     SDO_ORDINATES   VARRAY TERMINATED BY '|/'
       (ordinates    FLOAT EXTERNAL)
   )
)
begindata
 1|2003|1|1003|1|/
#-122.4215|37.7862|-122.422|37.7869|-122.421|37.789|-122.42|37.7866|
#-122.4215|37.7862|/
 2|2003|1|1003|1|/
#-122.4019|37.8052|-122.4027|37.8055|-122.4031|37.806|-122.4012|37.8052|
#-122.4019|37.8052|/
 3|2003|1|1003|1|/
#-122.426|37.803|-122.4242|37.8053|-122.42355|37.8044|-122.4235|37.8025|
#-122.426|37.803|/
```

3.1.2 Bulk Loading Point-Only Data in SDO_GEOMETRY Objects

Example 3–3 shows a control file for loading a table with point data.

Example 3–3 Control File for a Bulk Load of Point-Only Data

```
LOAD DATA
 INFILE *
 TRUNCATE
 CONTINUEIF NEXT(1:1) = '#'
 INTO TABLE POINT
 FIELDS TERMINATED BY '|'
 TRAILING NULLCOLS (
  GID      INTEGER EXTERNAL,
  GEOMETRY COLUMN OBJECT
   (
     SDO_GTYPE       INTEGER EXTERNAL,
     SDO_POINT COLUMN OBJECT
       (X            FLOAT EXTERNAL,
        Y            FLOAT EXTERNAL)
   )
)

BEGINDATA
 1| 2001| -122.4215| 37.7862|
 2| 2001| -122.4019| 37.8052|
 3| 2001| -122.426| 37.803|
 4| 2001| -122.4171| 37.8034|
 5| 2001| -122.416151| 37.8027228|
```

3.2 Transactional Insert Operations Using SQL

Oracle Spatial uses standard Oracle9*i* tables that can be accessed or loaded with standard SQL syntax. This section contains examples of transactional inserts into columns of type SDO_GEOMETRY. Note that the INSERT statement in Oracle SQL has a limit of 999 arguments. Therefore, you cannot create a variable-length array of more than 999 elements using the SDO_GEOMETRY constructor inside a transactional INSERT statement; however, you can insert a geometry using a host variable, and the host variable can be built using the SDO_GEOMETRY constructor with more than 999 values in the SDO_ORDINATE_ARRAY specification. (The host variable is an OCI, PL/SQL, or Java program variable.)

To perform transactional insertions of geometries, you can create a procedure to insert a geometry, and then invoke that procedure on each geometry to be inserted. Example 3–4 creates a procedure to perform the insert operation.

Example 3–4 Procedure to Perform Transactional Insert Operation

```
CREATE OR REPLACE PROCEDURE
```

```
        INSERT_GEOM(GEOM MDSYS.SDO_GEOMETRY)
IS

BEGIN
  INSERT INTO TEST_1 VALUES (GEOM);
  COMMIT;
END;
/
```

Using the procedure created in Example 3–4, you can insert data by using a PL/SQL block, such as the one in Example 3–5, which loads a geometry into the variable named *geom* and then invokes the INSERT_GEOM procedure to insert that geometry.

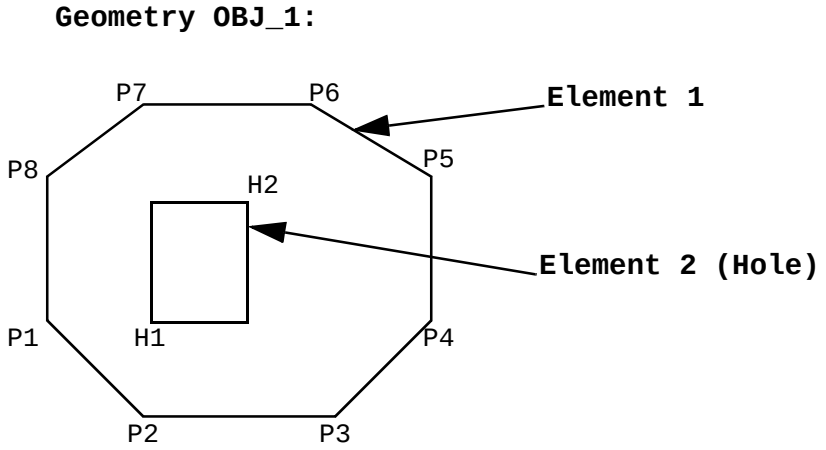
Example 3–5 PL/SQL Block Invoking Procedure to Insert a Geometry

```
DECLARE
geom mdsys.sdo_geometry :=
  mdsys.sdo_geometry (2003, null, null,
          mdsys.sdo_elem_info_array (1,1003,3),
          mdsys.sdo_ordinate_array (-109,37,-102,40));
BEGIN
  INSERT_GEOM(geom);
  COMMIT;
END;
/
```

3.2.1 Polygon with Hole

The geometry to be stored can be a polygon with a hole, as shown in Figure 3–1.

Figure 3–1 Polygon with a Hole



The coordinate values for Element 1 and Element 2 (the hole), shown in Figure 3–1, are:

```
Element 1= [P1(6,15), P2(10,10), P3(20,10), P4(25,15), P5(25,35), P6(19,40),
    P7(11,40), P8(6,25), P1(6,15)]
Element 2= [H1(12,15),  H2(15,24)]
```

The following example assumes that a table named PARKS was created as follows:

```
CREATE TABLE PARKS (NAME VARCHAR2(32),
                    SHAPE MDSYS.SDO_GEOMETRY);
```

The SQL statement for inserting the data for geometry OBJ_1 is:

```
INSERT INTO PARKS
  VALUES ('OBJ_1', MDSYS.SDO_GEOMETRY(2003, NULL,NULL,
                     MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1, 19,2003,3),
                     MDSYS.SDO_ORDINATE_ARRAY(6,15, 10,10, 20,10, 25,15, 25,35,
                         19,40, 11,40, 6,25, 6,15, 12,15, 15,24)));
```

The SDO_GEOMETRY object type takes values and constructors for its attributes SDO_GTYPE, SDO_ELEM_INFO, and SDO_ORDINATES. The SDO_GTYPE is 2003, and the SDO_ELEM_INFO has 2 triplet values because there are 2 elements. Element 1 starts at offset 1, is of ETYPE 1003, and its interpretation value is 1 because the points are connected by straight line segments. Element 2 starts at offset 19, is of ETYPE 2003, and has an interpretation value of 3 (a rectangle). The SDO_

ORDINATES varying length array has 22 values with SDO_ORDINATES(1...18) describing element 1 and SDO_ORDINATES(19...22) describing element 2.

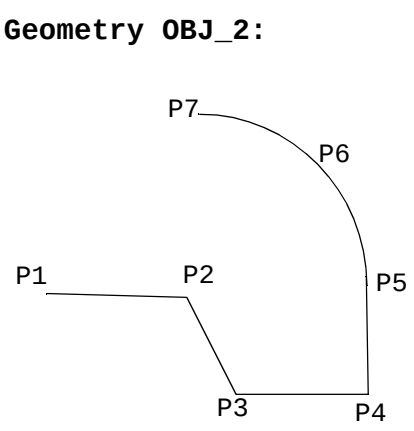
Assume that two dimensions are named X and Y, their bounds are 0 to 100, and the tolerance for both dimensions is 0.005. The SQL statement for loading the USER_SDO_GEOM_METADATA metadata view is:

```
INSERT INTO USER_SDO_GEOM_METADATA
  VALUES ('PARKS', 'SHAPE',
          MDSYS.SDO_DIM_ARRAY(MDSYS.SDO_DIM_ELEMENT('X', 0, 100, 0.005),
                 MDSYS.SDO_DIM_ELEMENT('Y', 0, 100, 0.005)),
          NULL);
```

3.2.2 Compound Line String

A compound line string is a connected sequence of straight line segments and circular arcs. Figure 3–2 is an example of a compound line string.

Figure 3–2 Line String Consisting of Arcs and Straight Line Segments



In Figure 3–2, the coordinate values for points P1..P7 that describe the line string OBJ_2 are:

```
OBJ_2 = [P1(15,10), P2(25,10), P3(30,5), P4(38,5), P5(38,10),
         P6(35,15), P7(25,20)]
```

The SQL statement for inserting this compound line string in a feature table defined as ROADS(GID Varchar2(32), Shape MDSYS.SDO_GEOMETRY) is:

```
INSERT INTO ROADS VALUES ('OBJ_2', MDSYS.SDO_GEOMETRY(2002, NULL, NULL,
   MDSYS.SDO_ELEM_INFO_ARRAY(1,4,2, 1,2,1, 9,2,2),
   MDSYS.SDO_ORDINATE_ARRAY(15,10, 25,10, 30,5, 38,5, 38,10, 35,15, 25,20)));
```

The SDO_GEOMETRY object type takes values and constructors for its attributes SDO_GTYPE, SDO_ELEM_INFO, and SDO_ORDINATES. The SDO_GTYPE is 2002, and the SDO_ELEM_INFO_ARRAY has 9 values because there are 2 subelements for the compound line string. The first subelement starts at offset 1, is of SDO_ETYPE 2, and its interpretation value is 1 because the points are connected by straight line segments. Similarly, subelement 2 has a starting offset of 9. That is, the first ordinate value is SDO_ORDINATES(9), is of SDO_ETYPE 2, and has an interpretation value of 2 because the points describe a circular arc. The SDO_ORDINATES_ARRAY varying length array has 14 values, with SDO_ORDINATES(1..10) describing subelement 1, and SDO_ORDINATES(9..14) describing subelement 2.

Assume that two dimensions are named X and Y, their bounds are 0 to 100, and tolerance for both dimensions is 0.005. The SQL statement to insert the metadata into the USER_SDO_GEOM_METADATA view is:

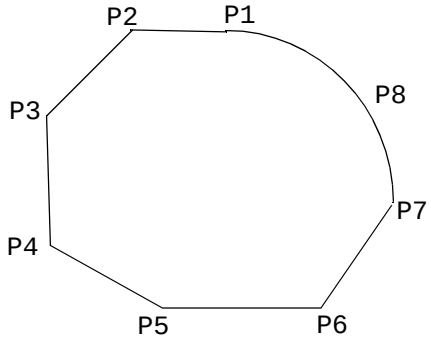
```
INSERT INTO USER_SDO_GEOM_METADATA VALUES ('ROADS', 'SHAPE',
  MDSYS.SDO_DIM_ARRAY(MDSYS.SDO_DIM_ELEMENT('X', 0, 100, 0.005),
                      MDSYS.SDO_DIM_ELEMENT('Y', 0, 100, 0.005)),
  NULL);
```

3.2.3 Compound Polygon

A compound polygon's boundary is a connected sequence of straight line segments and circular arcs, whose first point is equal to its last point. Figure 3–3 is an example of a compound polygon.

Figure 3–3 Compound Polygon

Geometry OBJ_3:



In Figure 3–3, the coordinate values for points P1 to P8 that describe the polygon OBJ_3 are:

```
OBJ_3 = [P1(20,30), P2(11,30), P3(7,22), P4(7,15), P5(11,10), P6(21,10),
   P7(27,30), P8(25,27), P1(20,30)]
```

The following example assumes that a table named PARKS was created as follows:

```
CREATE TABLE PARKS (GID VARCHAR2(32), SHAPE MSSYS.SDO_GEOMETRY);
```

The SQL statement for inserting this compound polygon is:

```
INSERT INTO PARKS VALUES ('OBJ_3', MDSYS.SDO_GEOMETRY(2003, NULL,NULL,
  MDSYS.SDO_ELEM_INFO_ARRAY(1,1005,2, 1,2,1, 13,2,2),
  MDSYS.SDO_ORDINATE_ARRAY(20,30, 11,30, 7,22, 7,15, 11,10, 21,10, 27,30,
    25,27, 20,30)));
```

The SDO_GEOMETRY object type takes values and constructors for its attributes SDO_GTYPE, SDO_ELEM_INFO, and SDO_ORDINATES. The SDO_GTYPE is 2003, the SDO_ELEM_INFO has 3 triplet values. The first triplet (1,1005,2) identifies the element as a compound polygon (ETYPE 1005) with two subelements. The first subelement starts at offset 1, is of ETYPE 2, and its interpretation value is 1 because the points are connected by straight line segments. Subelement 2 has a starting offset of 13, is of ETYPE 2, and has an interpretation value of 2 because the points describe a circular arc. The SDO_ORDINATES varying length array has 18 values, with SDO_ORDINATES(1...14) describing subelement 1, and SDO_ ORDINATES(13...18) describing subelement 2.

This example assumes the PARKS table was created as follows:

```
CREATE TABLE PARKS (GID VARCHAR2(32), SHAPE MSSYS.SDO_GEOMETRY);
```

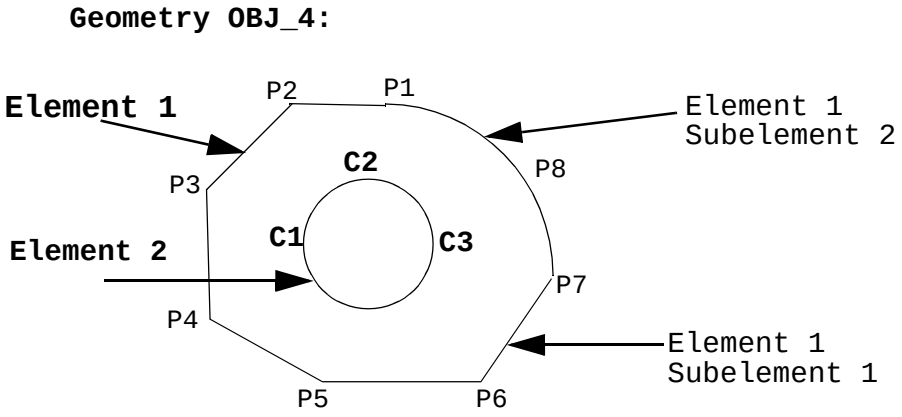
Assume that two dimensions are named X and Y, their bounds are 0 to 100, and tolerance for both dimensions is 0.005. The SQL statement to insert the metadata into the USER_SDO_GEOM_METADATA view is:

```
INSERT INTO USER_SDO_GEOM_METADATA VALUES ('PARKS', 'SHAPE',
  MDSYS.SDO_DIM_ARRAY(MDSYS.SDO_DIM_ELEMENT('X', 0, 100, 0.005),
                      MDSYS.SDO_DIM_ELEMENT('Y', 0, 100, 0.005)),
  NULL);
```

3.2.4 Compound Polygon with Holes

A compound polygon's boundary is a connected sequence of straight line segments and circular arcs. Figure 3–4 is an example of a geometry that contains a compound polygon with a hole (or void).

Figure 3–4 Compound Polygon with a Hole



In Figure 3–4, the coordinate values for points P1 to P8 (Element 1) and C1 to C3 (Element 2) that describe the geometry OBJ_4 are:

```
Element 1 = [P1(20,30), P2(11,30), P3(7,22), P4(7,15), P5(11,10), P6(21,10),
             P7(27,30), P8(25,27), P1(20,30)]
Element 2 = [C1(10,17), C2(15,22), C3(20,17)]
```

The following example assumes that a table named PARKS was created as follows:

```
CREATE TABLE PARKS (GID VARCHAR2(32), SHAPE MSSYS.SDO_GEOMETRY);
```

The SQL statement for inserting this compound polygon with a hole is:

```
INSERT INTO Parks VALUES ('OBJ_4', MDSYS.SDO_GEOMETRY(2003, NULL,NULL,
  MDSYS.SDO_ELEM_INFO_ARRAY(1,1005,2, 1,2,1, 13,2,2, 19,2003,4),
  MDSYS.SDO_ORDINATE_ARRAY(20,30, 11,30, 7,22, 7,15, 11,10, 21,10, 27,30,
                           25,27, 20,30, 10,17, 15,22, 20,17)));
```

The SDO_GEOMETRY object type takes values and constructors for its attributes SDO_GTYPE, SDO_ELEM_INFO, and SDO_ORDINATES. The SDO_GTYPE is 2003, the SDO_ELEM_INFO has 4 triplet values. The first 3 triplet values represent element 1. The first triplet (1,1005,2) identifies this element as a compound element with two subelements. The values in SDO_ELEM_INFO(1...9) pertain to element 1, while SDO_ELEM_INFO(10...12) are for element 2.

The first subelement starts at offset 1, is of ETYPE 2, and its interpretation is 1 because the points are connected by straight line segments. Subelement 2 has a starting offset of 13, is of ETYPE 2, and has an interpretation value of 2 because the points describe a circular arc. The fourth triplet (19,2003,4) represents element 2. Element 2 starts at offset 19, is of ETYPE 2003, and its interpretation value is 4, indicating that it is a circle. The SDO_ORDINATES varying length array has 24 values, with SDO_ORDINATES(1...14) describing subelement 1, SDO_ORDINATES(13...18) describing subelement 2, and SDO_ORDINATES(19...24) describing element 2.

Assume that two dimensions are named X and Y, their bounds are 0 to 100, and tolerance for both dimensions is 0.005. The SQL statement to insert the metadata into the USER_SDO_GEOM_METADATA view is:

```
INSERT INTO USER_SDO_GEOM_METADATA VALUES ('PARKS', 'SHAPE',
  MDSYS.SDO_DIM_ARRAY(MDSYS.SDO_DIM_ELEMENT('X', 0, 100, 0.005),
                      MDSYS.SDO_DIM_ELEMENT('Y', 0, 100, 0.005)),
  NULL);
```

3.2.5 Transactional Insertion of Point-Only Data

A point-only geometry can be inserted with the following statement:

```
INSERT INTO PARKS VALUES ('OBJ_PT',
                          MDSYS.SDO_GEOMETRY(2001,NULL,
                                             MDSYS.SDO_POINT_TYPE(20,30,NULL),
                                             NULL, NULL)
                          );
```

4

Indexing and Querying Spatial Data

After you have loaded spatial data (discussed in Chapter 3), you should create a spatial index on it to enable efficient query performance using the data. This chapter describes how to:

- Create a spatial index
- Query spatial data efficiently, based on an understanding of the Oracle Spatial query model and primary and secondary filtering

4.1 Creating a Spatial Index

Once data has been loaded into the spatial tables through either bulk or transactional loading, a spatial index must be created on the tables for efficient access to the data. Each spatial index can be an R-tree index or a quadtree index. To decide which type of index to use for a spatial application, you must understand the concepts and guidelines discussed in Section 1.7.

If the index creation does not complete for any reason, the index is invalid and must be deleted with the DROP INDEX <index_name> [FORCE] statement.

4.1.1 Creating R-Tree Indexes

If you create a spatial index without specifying any quadtree-specific parameters, an R-tree index is created. For example, the following statement creates a spatial R-tree index named *territory_idx* using default values for parameters that apply to R-tree indexes:

```
CREATE INDEX territory_idx ON territories (territory_geom)
   INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

For detailed information about options when creating a spatial index, see the documentation for the CREATE INDEX statement in Chapter 9.

If the rollback segment is not large enough, an attempt to create an R-tree index will fail. The rollback segment should be 100*n* bytes, where *n* is the number of rows of data to be indexed. For example, if the table contains 1 million (1,000,000) rows, the rollback segment size should be 100,000,000 (100 million bytes).

To ensure an adequate rollback segment, or if you have tried to create an R-tree index and received an error that a rollback segment cannot be extended, review (or have a DBA review) the size and structure of the rollback segments that are available to the schema that owns the table with the geometries. Create a public rollback segment of the appropriate size, and place that rollback segment online. In addition, ensure that any small inappropriate rollback segments are placed offline during large spatial index operations. For information about performing these operations on a rollback segment, see the *Oracle9i Database Administrator's Guide*.

The system parameter SORT_AREA_SIZE affects the amount of time required to create the index. The SORT_AREA_SIZE value is the maximum amount, in bytes, of memory to use for a sort operation. The optimal value depends on the database size, but a good guideline is to make it at least 1 million bytes when you create an R-tree index. To change the SORT_AREA_SIZE value, use the ALTER SESSION statement. For example, to change the value to 20 million bytes:

```
ALTER SESSION SET SORT_AREA_SIZE = 20000000;
```

The tablespace specified with the *tablespace* keyword in the CREATE INDEX statement (or the default tablespace if the *tablespace* keyword is not specified) is used to hold both the index data table and some transient tables that are created for internal computations.

- The R-tree index data table requires approximately 70*n* bytes (where *n* is the number of rows in the table).
- The transient tables require up to approximately 200*n* bytes (where *n* is the number of rows in the table); however, this space is freed up after the R-tree index is created.

For large databases (over 1 million rows), a temporary tablespace may be needed to perform internal sorting operations. The recommended size for this temporary tablespace is 100*n* bytes, where *n* is the number of rows in the table.

4.1.2 Determining Index Creation Behavior (Quadtree Indexes)

With a quadtree index, the tessellation algorithm used by the CREATE INDEX statement and by index maintenance routines on insert or update operations is determined by the SDO_LEVEL and SDO_NUMTILES values, which are supplied in the PARAMETERS clause of the CREATE INDEX statement. They are interpreted as follows:

SDO_LEVEL	SDO_NUMTILES	Action
Not specified or 0	Not specified or 0	R-tree index.
>= 1	Not specified or 0	Fixed indexing (indexing with fixed-size tiles).
>= 1	>= 1	Hybrid indexing with fixed-size and variable-sized tiles. The SDO_LEVEL column defines the fixed tile size. The SDO_NUMTILES column defines the number of variable tiles to generate per geometry.
Not specified or 0	>= 1	Not supported (error).

An explicit commit operation is executed after the tessellation of all the geometries in a geometry column.

By default, spatial index creation requires a sizable amount of rollback space. To reduce the amount of rollback space required, you can supply the SDO_COMMIT_INTERVAL parameter in the CREATE INDEX statement. This will perform a database commit after every *n* geometries are indexed, where *n* is a user-defined value.

4.1.3 Spatial Indexing with Fixed-Size Tiles (Quadtree Indexes)

If you choose quadtree indexing for a spatial index, you should use fixed indexing for most applications, except for the rare circumstances where hybrid indexing should be considered. (See Appendix B for information about hybrid indexing. However, you should also consider using R-tree indexing before deciding on hybrid indexing.)

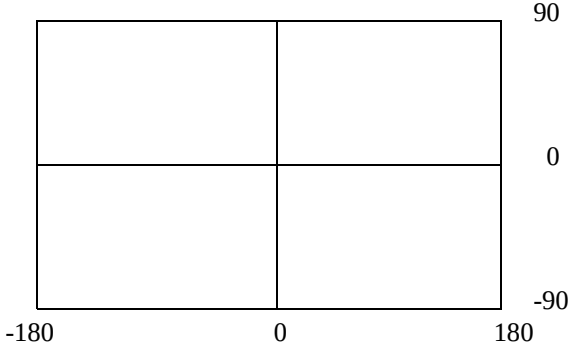
The fixed-size tile algorithm is expressed as a level referring to the number of tessellations performed. To use fixed-size tile indexing, omit the SDO_NUMTILES parameter and set the SDO_LEVEL value to the desired tiling level. The relationship between the tiling level and the resulting size of the tiles depends on the domain of the layer.

The domain used for indexing is defined by the upper and lower boundaries of each dimension stored in the DIMINFO column of the USER_SDO_GEOM_METADATA view, which contains an entry for the table and geometry column to spatially index. A typical domain could be -180 to 180 degrees for longitude, and -90 to 90 degrees for latitude, as represented in Figure 4–1. (The transference of the domain onto a sphere or other projection is left up to an application, unless a coordinate system is specified, as explained in Chapter 5.)

Figure 4–1 Sample Domain

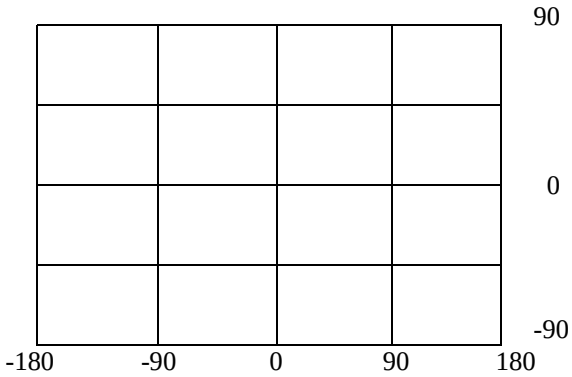
If the SDO_LEVEL column is set to 1, then the tiles created by the indexing mechanism are the same size as tiles at the first level of tessellation. Each tile would be 180 degrees by 90 degrees as shown in Figure 4–2.

Figure 4–2 Fixed-Size Tiling at Level 1



The formula for the number of fixed-size tiles in a domain is 4^n where *n* is the number of tessellations, stored in the SDO_LEVEL column. In reality, tiles are only generated where geometries exist, and not for the whole domain. Figure 4–3 shows fixed-size tiling at level 2. In this figure, each tile is 90 degrees by 45 degrees.

Figure 4–3 Fixed-Size Tiling at Level 2



The size of a tile can be determined by applying the following formula to each dimension:

length = (upper_bound - lower_bound) / 2 ^ sdo_level

The length refers to the length of the tile along the specified dimension. Applying this formula to the tiling shown in Figure 4–3 yields the following sizes:

```
length for dimension X = (180 - (-180) ) / 2^2
                       = (360)  / 4
                       = 90
length for dimension Y = (90 - (-90) ) / 2^2
                       = (180) / 4
                       = 45
```

At level 2, the tiles are 90 degrees by 45 degrees in size. As the number of levels increases, the tiles become smaller and smaller. Smaller tiles provide a more precise fit of the tiles over the geometry being indexed. However, because the number of tiles generated is unbounded, you must take into account the performance implications of using higher levels.

Note: The Spatial Index Advisor component of Oracle Enterprise Manager can be used to determine an appropriate level for indexing with fixed-size tiles. The SDO_TUNE.ESTIMATE_TILING_LEVEL function, described in Chapter 17, can also be used for this purpose; however, this function performs less analysis than the Spatial Index Advisor.

Besides the performance aspects related to selecting a fixed-size tile, tessellating the geometry into fixed-size tiles might have benefits related to the type of data being stored, such as using tiles sized to represent 1-acre farm plots, city blocks, or individual pixels on a display. Data modeling, an important part of any database design, is essential in a spatial database where the data often represents actual physical locations.

In Example 4–1, assume that data has been loaded into a table called ROADS, and the USER_SDO_GEOM_METADATA view has an entry for ROADS.SHAPE. You can use the following SQL statement to create a fixed index named ROADS_FIXED.

Example 4–1 Creating a Fixed Index

```
CREATE INDEX ROADS_FIXED ON ROADS(SHAPE) INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('SDO_LEVEL=8');
```

The SDO_LEVEL value is used while tessellating objects. Increasing the level results in smaller tiles and better geometry approximations.

4.1.4 Constraining Data to a Geometry Type

When you create or rebuild a spatial index, you can ensure that all geometries that are in the table or that are inserted later are of a specified geometry type. To constrain the data to a geometry type in this way, use the *layer_gtype* keyword in the PARAMETERS clause of the CREATE INDEX or ALTER INDEX REBUILD statement, and specify a value from the Geometry Type column of Table 2–1 in Section 2.2.1. For example, to constrain spatial data in a layer to polygons:

```
CREATE INDEX cola_spatial_idx
ON cola_markets(shape)
INDEXTYPE IS MDSYS.SPATIAL_INDEX
PARAMETERS ('layer_gtype=POLYGON');
```

The geometry types in Table 2–1 are considered as a hierarchy when data is checked:

- The *MULTI* forms include the regular form also. For example, specifying `'layer_gtype=MULTIPOINT'` allows the layer to include both POINT and MULTIPOINT geometries.
- COLLECTION allows the layer to include all types of geometries.

4.1.5 Creating a Cross-Schema Index

You can create a spatial index on a table that is not in your schema. Assume that user B wants to create a spatial index on column GEOMETRY in table T1 under user A's schema. User B must perform the following steps:

1. Connect as user A (or have user A connect) and execute the following statement:

   ```
   GRANT select on T1 to B;
   ```

2. Connect as user B and execute a statement such as the following:

   ```
   GRANT create table to A;
   CREATE INDEX t1_spatial_idx on A.T1(geometry)
     INDEXTYPE IS mdsys.spatial_index;
   ```

4.1.6 Using Partitioned Spatial Indexes

You can create a partitioned spatial index on a partitioned table. This section describes usage considerations specific to Oracle Spatial. For a detailed explanation of partitioned tables and partitioned indexes, see the *Oracle9i Database Administrator's Guide*.

A partitioned spatial index can provide the following benefits:

- Reduced response times for long-running queries, because partitioning reduces disk I/O operations
- Reduced response times for concurrent queries, because I/O operations run concurrently on each partition
- Easier index maintenance, because of partition-level create and rebuild operations

  Indexes on partitions can be rebuilt without affecting the queries on other partitions, and storage parameters for each local index can be changed independent of other partitions.

The following restrictions apply to spatial index partitioning:

- The partition key for spatial tables must be a scalar value, and must not be a spatial column.
- Only range partitioning is supported on the underlying table. Hash and composite partitioning are not currently supported for partitioned spatial indexes.
- ALTER TABLE partitioning statements for splitting, merging, and exchanging partitions are not supported for Release 9.0.1. (However, ALTER INDEX partitioning statements are supported.) For information about the ALTER TABLE statement, see the *Oracle9i SQL Reference*.

To create a partitioned spatial index, you must specify the LOCAL keyword. For example:

```
CREATE INDEX counties_idx ON counties(geometry)
   INDEXTYPE IS MDSYS.SPATIAL_INDEX LOCAL;
```

In this example, the default values are used for the number and placement of index partitions, namely:

- Index partitioning is based on the underlying table partitioning. For each table partition, a corresponding index partition is created.
- Each index partition is placed in the default tablespace.

If you do specify parameters for individual partitions, the following considerations apply:

- The storage characteristics for each partition can be the same or different for each partition. If they are different, it may enable parallel I/O (if the tablespaces are on different disks) and may improve performance.
- Any Oracle Spatial parameters (relating to R-tree or quadtree indexing) should be the same for each partition.

To override the default partitioning values, use a CREATE INDEX statement with the following general format:

```
CREATE INDEX <indexname> ON <table>(<column>)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
     [PARAMETERS ('<spatial-params>, <storage-params>')] LOCAL
     [( PARTITION <index_partition>
        PARAMETERS ('<spatial-params>, <storage-params>')
     [, PARTITION <index_partition>
        PARAMETERS ('<spatial-params>, <storage-params>')]
     )]
```

For example, if the COUNTIES table has two partitions, P1 and P2, you can create a quadtree index as follows:

```
CREATE INDEX counties_idx ON counties(geometry)
 INDEXTYPE IS MDSYS.SPATIAL_INDEX
    PARAMETERS ('sdo_level=6 tablespace=def_tbs')
LOCAL
    (PARTITION ip1 PARAMETERS ('sdo_level=6 tablespace=local_tbs1'),
     PARTITION ip2 PARAMETERS ('sdo_level=6 tablespace=local_tbs2'),
     PARTITION ip3);
```

In the preceding example:

- IP1 is the index partition that corresponds to partition P1 of the COUNTIES table, and IP2 is the index partition that corresponds to partition P2.
- The tablespace parameters are specified for each of the local index partitions as LOCAL_TBS1 and LOCAL_TBS2, respectively.
- If you omit the `PARTITION ip2 PARAMETERS...` clause (as is done for partition IP3), the default parameters specified before LOCAL are used. Specifically, the DEF_TBS tablespace is used for storing the index partition (which will have the same name as the second partition of the COUNTIES table, that is, P2).

Queries can operate on partitioned tables to perform the query on only one partition. For example:

```
SELECT * FROM counties PARTITION(p1)
    WHERE ...<some-spatial-predicate>;
```

Querying on a selected partition may speed up the query and also improve overall throughput when multiple queries operate on different partitions concurrently.

When queries use a partitioned spatial index, the semantics (meaning or behavior) of spatial operators and functions is the same with partitioned and nonpartitioned indexes, except in the case of SDO_NN (nearest neighbor). With SDO_NN, the requested number of geometries is returned for each partition that is affected by the query. For example, if you request the 5 closest restaurants to a point and the spatial index has 4 partitions, SDO_NN returns up to 20 (5*4) geometries. In this case, you must use the ROWNUM pseudocolumn (here, `WHERE ROWNUM <=5`) to return the 5 closest restaurants. See the description of the SDO_NN in Chapter 11 for more information.

4.2 Querying Spatial Data

This section describes how the structures of a Spatial layer are used to resolve spatial queries and spatial joins.

4.2.1 Query Model

Spatial uses a two-tier query model to resolve spatial queries and spatial joins. The term *two-tier* is used to indicate that two distinct operations are performed in order to resolve queries. If both operations are performed, the exact result set is returned.

The two operations are referred to as primary filter and secondary filter operations.

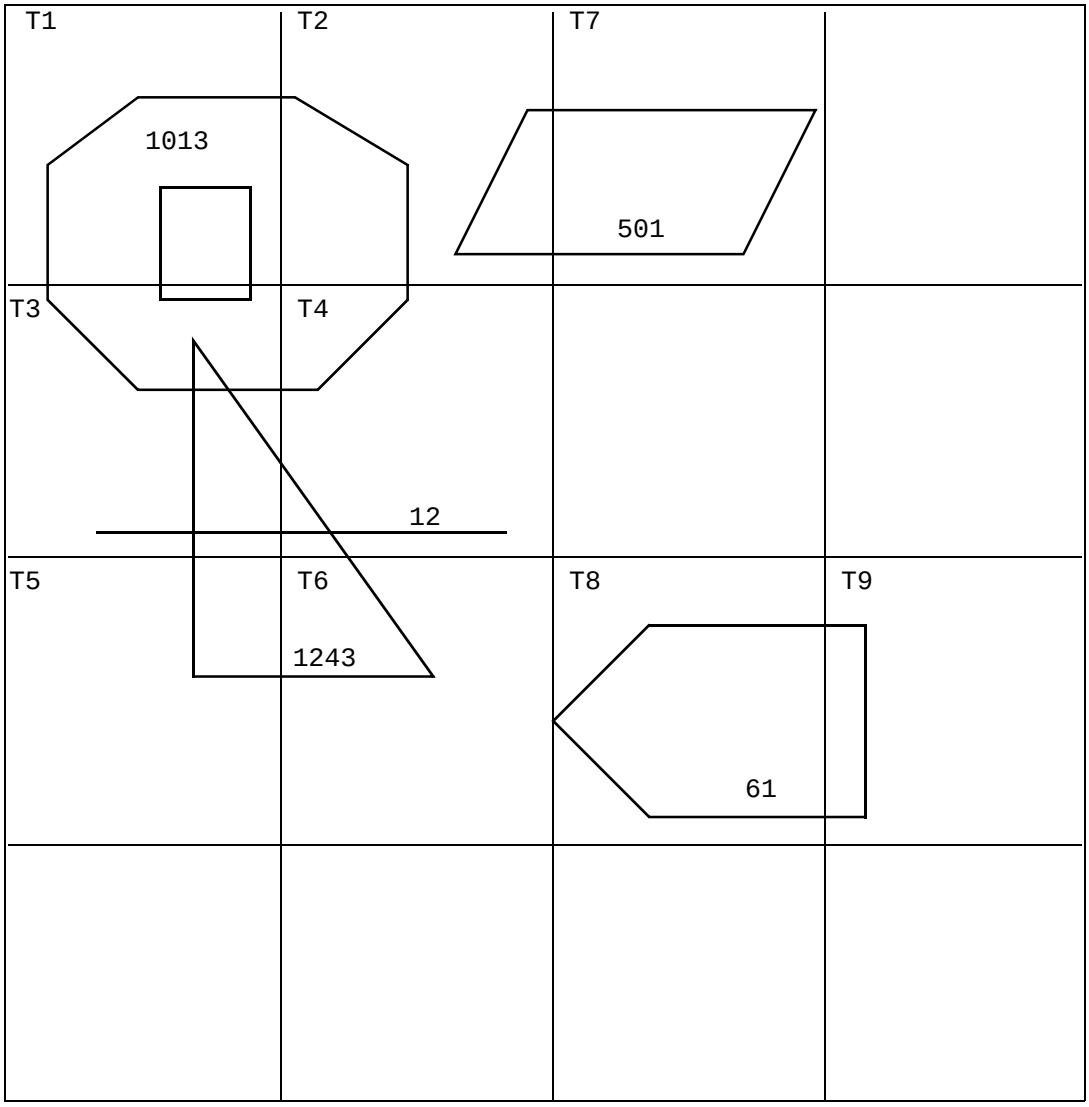
- The **primary filter** permits fast selection of candidate records to pass along to the secondary filter. The primary filter uses geometry approximations (or index tiles) to reduce computational complexity and is considered a lower-cost filter.
- The **secondary filter** applies exact computational geometry to the result set of the primary filter. These exact computations yield the exact answer to a query. The secondary filter operations are computationally more expensive, but they are applied only to the relatively small result set returned from the primary filter.

4.2.2 Spatial Query

An important concept in the spatial data model is that each geometry is represented by a set of exclusive and exhaustive tiles. This means that no tiles overlap each other (**exclusive**), and the tiles fully cover the object (**exhaustive**).

Consider the following layer containing several objects in Figure 4–4. Each object is labeled with its SDO_GID. The relevant tiles are labeled with T*n*.

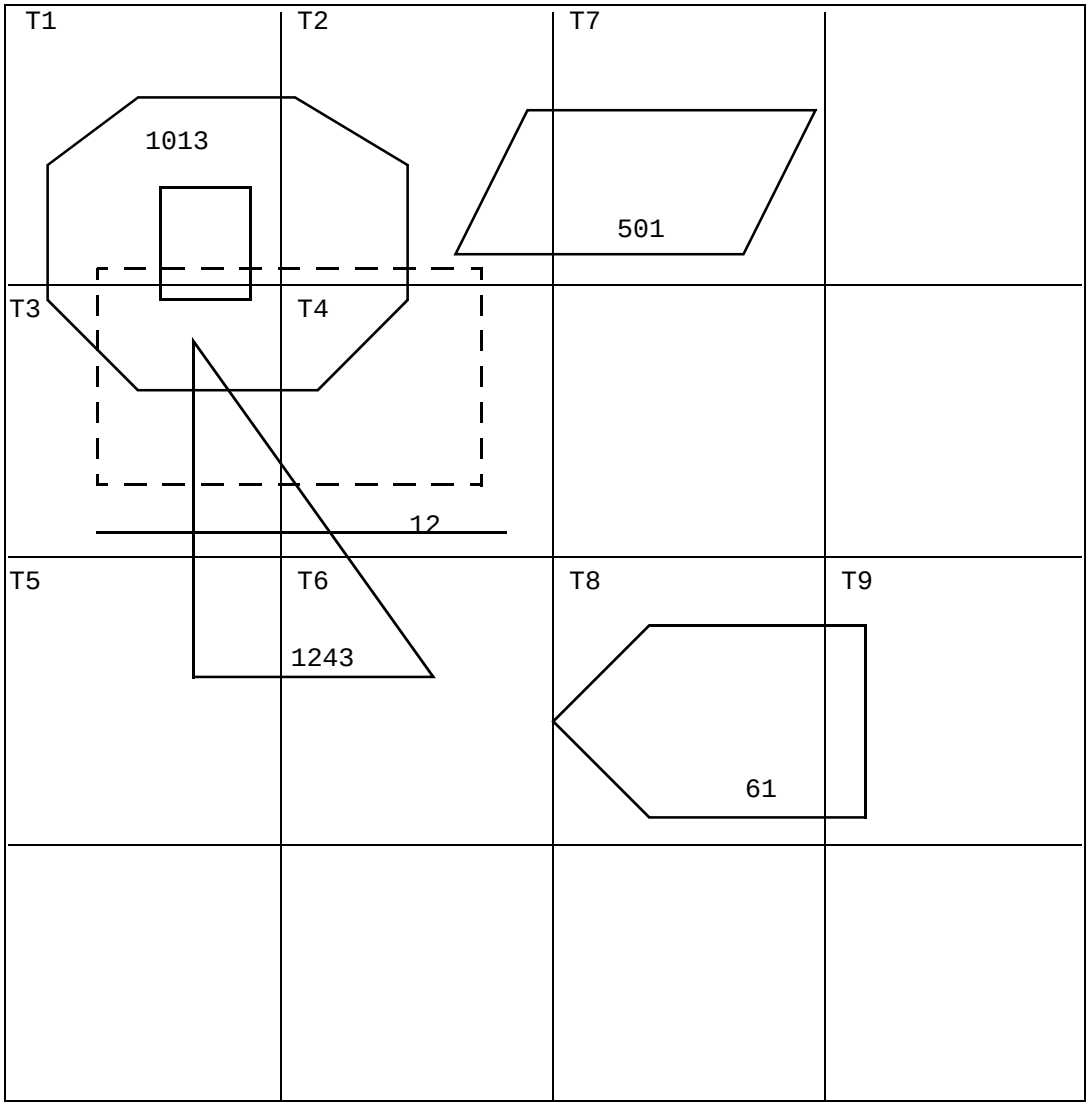
Figure 4–4 Tessellated Layer with Multiple Objects



A typical spatial query is to request all objects that lie within a defined fence or window. A **query window** is shown in Figure 4–5 by the dotted-line box. A

dynamic query window refers to a fence that is not defined in the database, but that must be defined before it is used.

Figure 4–5 Tessellated Layer with a Query Window



4.2.2.1 Primary Filter

Spatial provides an operator named SDO_FILTER. This implements the primary filter portion of the two-step process involved in the product's query processing model. The primary filter uses the index data only to determine a set of candidate object pairs that may interact. The syntax is as follows:

```
SDO_FILTER(geometry1 MDSYS.SDO_GEOMETRY, geometry2 MDSYS.SDO_GEOMETRY,
  params VARCHAR2)
```

Where:

- *geometry1* is a column of type MDSYS.SDO_GEOMETRY in a table. *geometry1* must be spatially indexed.
- *geometry2* is an object of type MDSYS.SDO_GEOMETRY. *geometry2* may or may not come from a table. If it comes from a table, it may or may not be spatially indexed.
- *params* is a quoted string of keyword value pairs that determine the behavior of the operator. See the SDO_FILTER operator in Chapter 11 for a list of parameters.

The following examples perform a primary filter operation only. They will return all the geometries shown in Figure 4–5 that have an index tile in common with one of the index tiles that approximates the query window: tiles T1, T2, T3, and T4. The result of the following examples are geometries with IDs 1013, 1243, 12, and 501.

Example 4–2 performs a primary filter operation without inserting the query window into a table. The window will be indexed in memory and performance will be very good.

Example 4–2 Primary Filter with a Temporary Query Window

```
SELECT A.Feature_ID FROM TARGET A
 WHERE sdo_filter(A.shape, mdsys.sdo_geometry(2003,NULL,NULL,
                                      mdsys.sdo_elem_info_array(1,1003,3),
                                      mdsys.sdo_ordinate_array(x1,y1, x2,y2)),
                        'querytype=window') = 'TRUE';
```

Note that *(x1,y1)* and *(x2,y2)* are the lower-left and upper-right corners of the query window.

In Example 4–3, a transient instance of type SDO_GEOMETRY was constructed for the query window instead of specifying the window parameters in the query itself.

Example 4–3 Primary Filter with a Transient Instance of the Query Window

```
SELECT A.Feature_ID FROM TARGET A
 WHERE sdo_filter(A.shape, :theWindow,'querytype=window') = 'TRUE';
```

Example 4–4 assumes the query window was inserted into a table called WINDOWS, with an ID of WINS_1.

Example 4–4 Primary Filter with a Stored Query Window

```
SELECT A.Feature_ID FROM TARGET A, WINDOWS B
 WHERE B.ID = 'WINS_1' AND
  sdo_filter(A.shape, B.shape,'querytype=window') = 'TRUE';
```

If the B.SHAPE column is not spatially indexed, the SDO_FILTER operator indexes the query window in memory and performance is very good.

If the B.SHAPE column is spatially indexed with the same SDO_LEVEL value as the A.SHAPE column, the SDO_FILTER operator reuses the existing index, and performance is very good or better.

If the B.SHAPE column is spatially indexed with a different SDO_LEVEL value than the A.SHAPE column, the SDO_FILTER operator reindexes B.SHAPE in the same way as if there were no index on the column originally, and then performance is very good.

4.2.2.2 Primary and Secondary Filters

The SDO_RELATE operator performs both the primary and secondary filter stages when processing a query. The syntax of the operator is as follows:

```
SDO_RELATE(geometry1  MDSYS.SDO_GEOMETRY,
           geometry2  MDSYS.SDO_GEOMETRY,
           params     VARCHAR2)
```

Where:

- *geometry1* is a column of type MDSYS.SDO_GEOMETRY in a table. *geometry1* must be spatially indexed.
- *geometry2* is an object of type MDSYS.SDO_GEOMETRY. *geometry2* may or may not come from a table. If it comes from a table, it may or may not be spatially indexed.
- *params* is a quoted string of keyword value pairs that determine the behavior of the operator. See the SDO_RELATE operator in Chapter 11 for a list of parameters.

The following examples perform both primary and secondary filter operations. They return all the geometries in Figure 4–5 that lie within or overlap the query window. The result of these examples is objects 1243 and 1013.

Example 4–5 performs both primary and secondary filter operations without inserting the query window into a table. The window will be indexed in memory and performance will be very good.

Example 4–5 Secondary Filter Using a Temporary Query Window

```
SELECT A.Feature_ID FROM TARGET A
   WHERE sdo_relate(A.shape, mdsys.sdo_geometry(2003,NULL,NULL,
                                        mdsys.sdo_elem_info_array(1,1003,3),
                                       mdsys.sdo_ordinate_array(x1,y1, x2,y2)),
                         'mask=anyinteract querytype=window') = 'TRUE';
```

Note that *(x1,y1)* and *(x2,y2)* are the lower-left and upper-right corners of the query window.

Example 4–6 assumes the query window was inserted into a table called WINDOWS, with an ID of WINS_1.

Example 4–6 Secondary Filter Using a Stored Query Window

```
SELECT A.Feature_ID FROM TARGET A, WINDOWS B
 WHERE B.ID= 'WINS_1' AND
        sdo_relate(A.shape, B.shape,
          'mask=anyinteract querytype=window') = 'TRUE';
```

If the B.SHAPE column is not spatially indexed, the SDO_RELATE operator indexes the query window in memory and performance is very good.

If the B.SHAPE column is spatially indexed with the same SDO_LEVEL value as the A.SHAPE column, the SDO_RELATE operator reuses the existing index, and performance is very good or better.

If the B.SHAPE column is spatially indexed with a different SDO_LEVEL value than the A.SHAPE column, the SDO_FILTER operator reindexes B.SHAPE in the same way as if there were no index on the column originally, and then performance is very good.

4.2.2.3 Within-Distance Operator

The SDO_WITHIN_DISTANCE operator is used to determine the set of objects in a table that are within *n* distance units from a reference object. The reference object

may be a transient or persistent instance of MDSYS.SDO_GEOMETRY (such as a temporary query window or a permanent geometry stored in the database). The syntax is as follows:

```
SDO_WITHIN_DISTANCE(geometry1  MDSYS.SDO_GEOMETRY,
                    aGeom      MDSYS.SDO_GEOMETRY,
                    params     VARCHAR2);
```

Where:

- *geometry1* is a column of type MDSYS.SDO_GEOMETRY in a table. *geometry1* must be spatially indexed.
- *aGeom* is an instance of type MDSYS.SDO_GEOMETRY.
- *params* is a quoted string of keyword value pairs that determines the behavior of the operator. See the SDO_WITHIN_DISTANCE operator in Chapter 11 for a list of parameters.

The following example selects any objects within 1.35 distance units from the query window:

```
SELECT A.Feature_ID
  FROM TARGET A
  WHERE SDO_WITHIN_DISTANCE( A.shape, :theWindow, 'distance=1.35') = 'TRUE';
```

The distance units are based on the geometry coordinate system in use. The distance units are those specified in the UNIT field of the well-known text (WKTEXT) associated with the coordinate system. (The WKTEXT is explained in Section 5.4.1.1.) If you are using a geodetic coordinate system, the units are meters. If no coordinate system is used, the units are the same as for the stored data.

The SDO_WITHIN_DISTANCE operator is not suitable for performing spatial joins. That is, a query such as *Find all parks that are within 10 distance units from coastlines* will not be processed as an index-based spatial join of the COASTLINES and PARKS tables. Instead, it will be processed as a nested loop query in which each COASTLINES instance is in turn a reference object that is buffered, indexed, and evaluated against the PARKS table. Thus, the SDO_WITHIN_DISTANCE operation is performed *n* times if there are *n* rows in the COASTLINES table.

For non-geodetic data, there is an efficient way to accomplish a spatial join that involves buffering all the geometries of a layer. This method does not use the SDO_WITHIN_DISTANCE operator. First, create a new table COSINE_BUFS as follows:

```
CREATE TABLE cosine_bufs UNRECOVERABLE AS
   SELECT SDO_BUFFER (A.SHAPE, B.DIMINFO, 1.35)
     FROM COSINE A, USER_SDO_GEOM_METADATA B
```

```
    WHERE TABLE_NAME='COSINES' AND COLUMN_NAME='SHAPE';
```

Next, create a spatial index on the SHAPE column of COSINE_BUFS. Then you can perform the following query:

```
SELECT a.gif, b.gid FROM parks A cosine_bufs B
  WHERE SDO_Relate(A.shape, B.shape, 'mask=ANYINTERACT querytype=JOIN') ='TRUE';
```

4.2.2.4 Nearest Neighbor Operator

The SDO_NN operator is used to identify the nearest neighbors for a geometry. The syntax is as follows:

```
SDO_NN(geometry1  MDSYS.SDO_GEOMETRY,
       geometry2  MDSYS.SDO_GEOMETRY,
       param      VARCHAR2
       [, number  NUMBER]);
```

Where:

- *geometry1* is a column of type MDSYS.SDO_GEOMETRY in a table. *geometry1* must be spatially indexed.
- *geometry2* is an instance of type MDSYS.SDO_GEOMETRY.
- *param* is a quoted string of a keyword value pair that determines how many nearest neighbor geometries are returned by the operator. See the SDO_NN operator in Chapter 11 for information about this parameter.
- *number* is the same number used in the call to SDO_NN_DISTANCE. Use this only if the SDO_NN_DISTANCE ancillary operator is included in the call to SDO_NN. See the SDO_NN operator in Chapter 11 for information about this parameter.

The following example finds the two objects from the SHAPE column in the COLA_MARKETS table that are closest to a specified point (10,7). (Note the use of the optimizer hint in the SELECT statement, as explained in the Usage Notes for the SDO_NN operator in Chapter 11.)

```
SELECT /*+ INDEX(cola_markets cola_spatial_idx) */
 c.mkt_id, c.name  FROM cola_markets c  WHERE SDO_NN(c.shape,
   mdsys.sdo_geometry(2001, NULL, mdsys.sdo_point_type(10,7,NULL), NULL,
   NULL),  'sdo_num_res=2') = 'TRUE';
```

4.2.3 Spatial Join

A **spatial join** is the same as a regular join except that the predicate involves a spatial operator. In Spatial, a spatial join takes place when you compare all the geometries of one layer to all the geometries of another layer. This is unlike a query window that only compares a single geometry to all geometries of a layer.

In a spatial join, all tables must have the same type of spatial index (that is, R-tree or quadtree) defined on the geometry column; and if they have quadtree indexes, the SDO_LEVEL value must be the same for all the indexes.

Spatial joins can be used to answer questions such as, *Which highways cross national parks?*

The following table structures illustrate how the join would be accomplished for this example:

```
PARKS(    GID VARCHAR2(32), SHAPE MDSYS.SDO_GEOMETRY)
HIGHWAYS( GID VARCHAR2(32), SHAPE MDSYS.SDO_GEOMETRY)
```

The primary filter would identify pairs of *GID* values from the PARKS and HIGHWAYS tables that interact in their index entries. The query that performs the primary filter join is:

```
SELECT A.GID, B.GID
  FROM PARKS A, HIGHWAYS B
  WHERE sdo_filter(A.shape, B.shape, 'querytype=join') = 'TRUE';
```

The original question, asking about highways that cross national parks, requires the secondary filter operator to find the exact relationship between highways and parks.

The query that performs this join using both primary and secondary filters is:

```
SELECT A.GID, B.GID
  FROM parks A, highways B
  WHERE sdo_relate(A.shape, B.shape,
           'mask=ANYINTERACT querytype=join');
```

4.2.4 Cross-Schema Operator Invocation

You can invoke spatial operators on an indexed table that is not in your schema. Assume that user A has a spatial table T1 (with index table IDX_TAB1) with a spatial index defined, that user B has a spatial table T2 (with index table IDX_TAB2) with a spatial index defined, and that user C wants to invoke operators on tables in one or both of the other schemas.

If user C wants to invoke an operator only on T1, user C must perform the following steps:

1. Connect as user A and execute the following statements:

```
GRANT select on T1 to C;
GRANT select on idx_tab1 to C;
```

2. Connect as user C and execute a statement such as the following:

```
SELECT a.gid
  FROM T1 a
  WHERE sdo_filter(a.geometry, :theGeometry, 'querytype=WINDOW') = 'TRUE';
```

If user C wants to invoke an operator on both T1 and T2, user C must perform the following steps:

1. Connect as user A and execute the following statements:

```
GRANT select on T1 to C;
GRANT select on idx_tab1 to C;
```

2. Connect as user B and execute the following statements:

```
GRANT select on T2 to C;
GRANT select on idx_tab2 to C;
```

3. Connect as user C and execute a statement such as the following:

```
SELECT a.gid
  FROM T1 a, T2 b
  WHERE b.gid = 5 AND
        sdo_filter(a.geometry, b.geometry, 'querytype=WINDOW') = 'TRUE';
```

5

Coordinate Systems (Spatial Reference Systems)

This chapter describes in greater detail the Oracle Spatial coordinate system support, which was introduced in Section 1.5.4. You can store and manipulate SDO_GEOMETRY objects in a variety of coordinate systems.

For reference information about coordinate system transformation functions and procedures, see Chapter 14.

5.1 Terms and Concepts

This section explains important terms and concepts related to coordinate system support in Oracle Spatial.

5.1.1 Coordinate System (Spatial Reference System)

A **coordinate system** (also called a *spatial reference system*) is a means of assigning coordinates to a location and establishing relationships between sets of such coordinates. It enables the interpretation of a set of coordinates as a representation of a position in a real world space.

5.1.2 Cartesian Coordinates

Cartesian coordinates are coordinates that measure the position of a point from a defined origin along axes that are perpendicular in the represented two-dimensional or three-dimensional space.

5.1.3 Geodetic Coordinates (Geographic Coordinates)

Geodetic coordinates (sometimes called *geographic coordinates*) are angular coordinates (longitude and latitude), closely related to spherical polar coordinates, and are defined relative to a particular Earth geodetic datum (described in Section 5.1.6). For more information about geodetic coordinate system support, see Section 5.2.

5.1.4 Projected Coordinates

Projected coordinates are planar Cartesian coordinates that result from performing a mathematical mapping from a point on the Earth's surface to a plane. There are many such mathematical mappings, each used for a particular purpose.

5.1.5 Local Coordinates

Local coordinates are Cartesian coordinates in a non-Earth (non-georeferenced) coordinate system. Section 5.3 describes local coordinate system support in Spatial.

5.1.6 Geodetic Datum

A **geodetic datum** is a means of representing the figure of the Earth, usually as an oblate ellipsoid of revolution, that approximates the surface of the Earth locally or globally, and is the reference for the system of geodetic coordinates.

5.1.7 Authalic Sphere

An **authalic sphere** is a sphere that has the same surface area as a particular oblate ellipsoid of revolution representing the figure of the Earth.

5.1.8 Transformation

Transformation is the conversion of coordinates from one coordinate system to another coordinate system.

If the coordinate system is georeferenced, transformation can involve datum transformation: the conversion of geodetic coordinates from one geodetic datum to another geodetic datum, usually involving changes in the shape, orientation, and center position of the reference ellipsoid.

5.2 Geodetic Coordinate Support

With Oracle9*i*, Spatial provides a rational and complete treatment of geodetic coordinates. Before Oracle 9*i*, Spatial computations were based solely on flat (Cartesian) coordinates, regardless of the coordinate system specified for the layer of geometries. Consequently, computations for data in geodetic coordinate systems were inaccurate, because they always treated the coordinates as if they were on a flat surface, and they did not consider the curvature of the surface.

With the current release, ellipsoidal surface computations consider the curvatures of arcs in the specified geodetic coordinate system and return correct, accurate results. In other words, with the current release, Spatial queries return the right answers all the time.

5.2.1 Geodesy and Two-Dimensional Geometry

A two-dimensional geometry is a surface geometry, but the important question is: What is the *surface*? A flat surface (plane) is accurately represented by Cartesian coordinates. However, Cartesian coordinates are not adequate for representing the surface of a solid. A commonly used surface for spatial geometry is the surface of the Earth, and the laws of geometry there are different than they are in a plane. For example, on the Earth's surface there are no parallel lines: lines are geodesics, and all geodesics intersect. Thus, closed curved surface problems cannot be done accurately with Cartesian geometry.

With Oracle9*i*, Spatial provides accurate results regardless of the coordinate system or the size of the area involved, without requiring that the data be projected to a flat surface. The results are accurate regardless of where on the Earth's surface the query is focused, even in "special" areas such as the poles. Thus, you can store coordinates in any datum and projections that you choose, and you can perform accurate queries regardless of the coordinate system.

5.2.2 Choosing a Geodetic or Projected Coordinate System

For applications that deal with the Earth's surface, the data can be represented using a geodetic coordinate system or a projected plane coordinate system. In deciding which approach to take with the data, consider any needs related to accuracy and performance:

- Accuracy

  For many spatial applications, the area is sufficiently small to allow adequate computations on Cartesian coordinates in a local projection. For example, the

New Hampshire State Plane local projection provides adequate accuracy for most spatial applications that use data for that state.

However, Cartesian computations on a plane projection will never give accurate results for a large area such as Canada or Scandinavia. For example, a query asking if Stockholm, Sweden and Helsinki, Finland are within a specified distance may return an incorrect result if the specified distance is close to the actual measured distance. Computations involving large areas -- or requiring very precise accuracy -- must account for the curvature of the Earth's surface.

- Performance

  Spherical computations use more computing resources than Cartesian computations, and take longer to complete. In general, a Spatial operation using geodetic coordinates will take two to three times longer than the same operation using Cartesian coordinates.

5.2.3 Other Considerations and Requirements with Geodetic Data

The following geometries are not permitted if a geodetic coordinate system is used:

- Circles
- Circular arcs
- Optimized rectangles (rectangles defined specifying only two points)

  However, you can use the SDO_CS.VIEWPORT_TRANSFORM function to convert optimized longitude/latitude rectangles to valid geodetic polygons for use with the SDO_FILTER operator.

Geodetic coordinate system support is provided only for geometries that consist of points or geodesics (lines on the ellipsoid). If you have geometries containing circles or circular arcs in a projected coordinate system, you can densify them using the SDO_GEOM.SDO_ARC_DENSIFY function (documented in Chapter 12) before transforming them to geodetic coordinates, and then perform Spatial operations on the resulting geometries.

The following size limits apply with geodetic data:

- No polygon element can have an area larger than one-half the surface of the Earth.
- No line element can have a length longer than half the perimeter (a great circle) of the Earth.

If you need to work with larger elements, first break these elements into multiple smaller elements and work with them. For example, you cannot create an element representing all the ocean surface of the Earth; however, you can create multiple elements, each representing part of the overall ocean surface.

Geodetic layers must be indexed using an R-tree index. (You can create a quadtree index on geodetic data by specifying `'geodetic=FALSE'` in the PARAMETERS clause of the CREATE INDEX statement; however, this is not recommended. See the Usage Notes for the CREATE INDEX statement in Chapter 9 for more information.) In addition, for Spatial release 9.0.1 you must delete (DROP INDEX) and re-create all spatial indexes on geodetic data from a previous release.

Tolerance is specified as meters for geodetic layers. Note that if you use tolerance values typical for non-geodetic data, these values are interpreted as meters for geodetic data. For example, if you specify a tolerance value of 0.005 for geodetic data, this is interpreted as precise to 5 millimeters. If this value is more precise than your applications need, performance may be affected because of the internal computational steps taken to implement the specified precision. (For more information about tolerance, see Section 1.5.5.)

See Section 5.7 for additional notes and restrictions relating to geodetic data.

5.3 Local Coordinate Support

With Oracle9*i*, Spatial provides a level of support for local coordinate systems. Local coordinate systems are often used in CAD systems, and they can also be used in local surveys where the relationship between the surveyed site and the rest of the world is not important.

Several local coordinate systems are predefined and included with Spatial in the MDSYS.CS_SRS table (described in Section 5.4.1). These supplied local coordinate systems, whose names start with *Non-Earth*, define non-Earth Cartesian coordinate systems based on different units of measurement (*Meter*, *Millimeter*, *Inch*, and so on). In the current release, you can use these local coordinate systems only to convert coordinates in a local coordinate system from one unit of measurement to another (for example, inches to millimeters) by transforming a geometry or a layer of geometries.

5.4 Coordinate Systems Data Structures

The coordinate systems functions and procedures use information provided in the following tables supplied with Oracle Spatial:

- MDSYS.CS_SRS (see Section 5.4.1) defines the valid coordinate systems. It associates each coordinate system with its well-known text description, which is in conformance with the standard published by the OpenGIS Consortium (`http://www.opengis.org`).
- MDSYS.SDO_ANGLE_UNITS (see Section 5.4.2) defines the valid angle units. The angle unit is part of the well-known text description.
- MDSYS.SDO_DIST_UNITS (see Section 5.4.3) defines the valid distance units. The distance unit is included in the well-known text description.
- MDSYS.SDO_DATUMS (see Section 5.4.4) defines the valid datums. The datum is part of the well-known text description.
- MDSYS.SDO_ELLIPSOIDS (see Section 5.4.5) defines the valid ellipsoids. The ellipsoid is part of the well-known text description.
- MDSYS.SDO_PROJECTIONS (see Section 5.4.6) defines the valid map projections. The map projection is part of the well-known text description.

Note: You should not modify or delete any Oracle-supplied information in any of the tables that are used for coordinate system support.

You should not add any information to the MDSYS.CS_SRS or MDSYS.SDO_DATUMS table unless you are creating a user-defined coordinate system. (Do not add information to the MDSYS.SDO_ELLIPSOIDS or MDSYS.PROJECTIONS tables.) Section 5.5 describes how to create a user-defined coordinate system.

5.4.1 MDSYS.CS_SRS Table

The MDSYS.CS_SRS reference table contains over 900 rows, one for each valid coordinate system.

Note: You should probably not modify, delete, or add any information in the MDSYS.CS_SRS table. If you plan to add any user-defined coordinate systems, be sure to use SRID values of 1000000 (1 million) or higher, and follow the guidelines in Section 5.5.

The MDSYS.CS_SRS table contains the columns shown in Table 5–1.

Table 5–1 MDSYS.CS_SRS Table

Column Name	Data Type	Description
CS_NAME	VARCHAR2(68)	A well-known name, often mnemonic, by which a user can refer to the coordinate system.
SRID	INTEGER	The unique ID number (Spatial Reference ID) for a coordinate system. Currently, SRID values 1-999999 are reserved for use by Oracle Spatial, and values 1000000 (1 million) and higher are available for user-defined coordinate systems.
AUTH_SRID	INTEGER	An optional ID number that can be used to indicate how the entry was derived; it might be a foreign key into another coordinate table, for example.
AUTH_NAME	VARCHAR2(256)	An authority name for the coordinate system. Contains 'Oracle' in the supplied table. Users can specify any value in any rows that they add.
WKTEXT	VARCHAR2(2046)	The well-known text (WKT) description of the SRS, as defined by the OpenGIS Consortium. For more information, see Section 5.4.1.1.
CS_BOUNDS	MDSYS.SDO_GEOMETRY	Optional SDO_GEOMETRY object that is a polygon with WGS-84 longitude and latitude vertices, representing the spheroidal polygon description of the zone of validity for a projected coordinate system. Must be null for a geographic or non-Earth coordinate system. Is null in all supplied rows.

5.4.1.1 Well-Known Text (WKTEXT)

The WKTEXT column of the MDSYS.CS_SRS table contains the well-known text (WKT) description of the SRS, as defined by the OpenGIS Consortium.

The following is the WKT EBNF syntax. All user-defined coordinate systems must strictly comply with this syntax.

```
<coordinate system> ::=
     <horz cs> | <local cs>

<horz cs> ::=
     <geographic cs> | <projected cs>
```

```
<projected cs> ::=
     PROJCS [ "<name>", <geographic cs>, <projection>,
          <parameter>,}* <linear unit> ]

<projection> ::=
     PROJECTION [ "<name>" ]

<parameter> ::=
     PARAMETER [ "name", <number> ]

<geographic cs> ::=
     GEOGCS [ "<name>", <datum>, <prime meridian>, <angular unit> ]

<datum> ::=
     DATUM [ "<name>", <spheroid>
     {, <shift-x>, <shift-y>, <shift-z>
       , <rot-x>, <rot-y>, <rot-z>, <scale_adjust>}
     ]

<spheroid> ::=
     SPHEROID ["<name>", <semi major axis>, <inverse flattening> ]

<prime meridian> ::=
     PRIMEM ["<name>", <longitude> ]

<longitude> ::=
     <number>

<semi-major axis> ::=
     <number>

<inverse flattening> ::=
     <number>

<angular unit> ::= <unit>

<linear unit> ::= <unit>

<unit> ::=
     UNIT [ "<name>", <conversion factor> ]

<local cs> ::=
     LOCAL_CS [ "<name>", <local datum>, <linear unit>,
          <axis> {, <axis>}* ]
```

```
<local datum> ::=
     LOCAL_DATUM [ "<name>", <datum type>
          {, <shift-x>, <shift-y>, <shift-z>
           , <rot-x>, <rot-y>, <rot-z>, <scale_adjust>}
          ]

<datum type> ::=
     <number>

<axis> ::=
     AXIS [ "<name>", NORTH | SOUTH | EAST |
           WEST | UP | DOWN | OTHER ]
```

An example of the WKT for a geodetic (geographic) coordinate system is:

```
'GEOGCS [ "Longitude / Latitude (Old Hawaiian)", DATUM ["Old Hawaiian", SPHEROID
["Clarke 1866", 6378206.400000, 294.978698]], PRIMEM [ "Greenwich", 0.000000 ],
UNIT ["Decimal Degree", 0.01745329251994330]]'
```

The WKT definition of the coordinate system is hierarchically nested. The Old Hawaiian geographic coordinate system (GEOGCS) is composed of a named datum (DATUM), a prime meridian (PRIMEM), and a unit definition (UNIT). The datum is in turn composed of a named spheroid and its parameters of semimajor axis and inverse flattening.

An example of the WKT for a projected coordinate system (a Wyoming state plane) is:

```
'PROJCS["Wyoming 4901, Eastern Zone (1983, meters)", GEOGCS [ "GRS 80", DATUM
["GRS 80", SPHEROID ["GRS 80", 6378137.000000, 298.257222]], PRIMEM [
"Greenwich", 0.000000 ], UNIT ["Decimal Degree", 0.01745329251994330]],
PROJECTION ["Transverse Mercator"], PARAMETER ["Scale_Factor", 0.999938],
PARAMETER ["Central_Meridian", -105.166667], PARAMETER ["Latitude_Of_Origin",
40.500000], PARAMETER ["False_Easting", 200000.000000], UNIT ["Meter",
1.000000000000]]'
```

The projected coordinate system contains a nested geographic coordinate system as its basis, as well as parameters that control the projection.

Oracle Spatial supports all the common geodetic datums and map projections.

An example of the WKT for a local coordinate system is:

```
LOCAL_CS [ "Non-Earth (Meter)", LOCAL_DATUM ["Local Datum", 0], UNIT ["Meter",
1.0], AXIS ["X", EAST], AXIS["Y", NORTH]]
```

Local coordinate systems are described in Section 5.3.

5.4.2 MDSYS.SDO_ANGLE_UNITS Table

The MDSYS.SDO_ANGLE_UNITS reference table contains one row for each valid UNIT specification in the well-known text (WKT) description in the coordinate system definition. The WKT is described in Section 5.4.1.1.

The MDSYS.SDO_ANGLE_UNITS table contains the columns shown in Table 5–2.

Table 5–2 MDSYS.SDO_ANGLE_UNITS Table

Column Name	Data Type	Description
SDO_UNIT	VARCHAR2(32)	(Reserved for future use by Oracle Spatial.)
UNIT_NAME	VARCHAR2(100)	Name of the angle unit. Specify a value from this column in the UNIT specification of the WKT for any user-defined coordinate system. Examples: *Decimal Degree, Radian, Decimal Second, Decimal Minute, Gon, Grad*
CONVERSION_FACTOR	NUMBER	The ratio of the specified unit to one *Radian*. For example, the ratio of *Decimal Degree* to *Radian* is 0.017453293.

5.4.3 MDSYS.SDO_DIST_UNITS Table

The MDSYS.SDO_DIST_UNITS reference table contains one row for each valid distance unit specification in the well-known text (WKT) description in the coordinate system definition. The WKT is described in Section 5.4.1.1.

The MDSYS.SDO_DIST_UNITS table contains the columns shown in Table 5–3.

Table 5–3 MDSYS.SDO_DIST_UNITS Table

Column Name	Data Type	Purpose
SDO_UNIT	VARCHAR2	Unit string identifier. Examples: *M, KM, CM, MM, MILE, NAUT_MILE, FOOT, INCH*. Do not use this in the WKT definition; instead, use a value from UNIT_NAME.

Table 5–3 MDSYS.SDO_DIST_UNITS Table (Cont.)

Column Name	Data Type	Purpose
UNIT_NAME	VARCHAR2	Descriptive name of the unit, to be used in the WKT specification. Examples: *Meter, Kilometer, Centimeter, Millimeter, Mile, Nautical Mile, Foot, Inch*
CONVERSION_ FACTOR	NUMBER	Ratio of the unit to 1 meter. For example, the conversion factor for a meter is 1.0, and the conversion factor for a mile is 1609.344.

5.4.4 MDSYS.SDO_DATUMS Table

The MDSYS.SDO_DATUMS reference table contains one row for each valid DATUM specification in the well-known text (WKT) description in the coordinate system definition. The WKT is described in Section 5.4.1.1.

The MDSYS.SDO_DATUMS table contains the columns shown in Table 5–4.

Table 5–4 MDSYS.SDO_DATUMS Table

Column Name	Data Type	Description
NAME	VARCHAR2(64)	Name of the datum. Specify a value (Oracle-supplied or user-defined) from this column in the DATUM specification of the WKT for any user-defined coordinate system. Examples: *Adindan, Afgooye, Ain el Abd 1970, Anna 1 Astro 1965, Arc 1950, Arc 1960, Ascension Island 1958.*
SHIFT_X	NUMBER	Number of meters to shift the ellipsoid center relative to the center of the WGS 84 ellipsoid on the x-axis.
SHIFT_Y	NUMBER	Number of meters to shift the ellipsoid center relative to the center of the WGS 84 ellipsoid on the y-axis.
SHIFT_Z	NUMBER	Number of meters to shift the ellipsoid center relative to the center of the WGS 84 ellipsoid on the z-axis.
ROTATE_X	NUMBER	Number of arc-seconds of rotation about the x-axis.
ROTATE_Y	NUMBER	Number of arc-seconds of rotation about the y-axis.

Table 5–4 MDSYS.SDO_DATUMS Table (Cont.)

Column Name	Data Type	Description
ROTATE_Z	NUMBER	Number of arc-seconds of rotation about the z-axis.
SCALE_ ADJUST	NUMBER	A value to be used in adjusting the X, Y, and Z values after any shifting and rotation, according to the formula: 1.0 + (SCALE_ADJUST * 10^{-6})

5.4.5 MDSYS.SDO_ELLIPSOIDS Table

The MDSYS.SDO_ELLIPSOIDS reference table contains one row for each valid SPHEROID specification in the well-known text (WKT) description in the coordinate system definition. The WKT is described in Section 5.4.1.1.

The MDSYS.SDO_ELLIPSOIDS table contains the columns shown in Table 5–5.

Table 5–5 MDSYS.SDO_ELLIPSOIDS Table

Column Name	Data Type	Description
NAME	VARCHAR2(64)	Name of the ellipsoid (spheroid). Specify a value from this column in the SPHEROID specification of the WKT for any user-defined coordinate system. Examples: *Clarke 1866, WGS 72, Australian, Krassovsky, International 1924.*
SEMI_MAJOR_ AXIS	NUMBER	Radius in meters along the semi-major axis (one-half of the long axis of the ellipsoid).
INVERSE_ FLATTENING	NUMBER	Inverse flattening of the ellipsoid. That is, 1 / f, where f = (a-b) / a, and a = semi-major axis and b = semi-minor axis.

5.4.6 MDSYS.SDO_PROJECTIONS Table

The MDSYS.SDO_PROJECTIONS reference table contains one row for each valid PROJECTION specification in the well-known text (WKT) description in the coordinate system definition. The WKT is described in Section 5.4.1.1.

The MDSYS.SDO_PROJECTIONS table contains the column shown in Table 5–6.

Table 5–6 MDSYS.SDO_PROJECTIONS Table

Column Name	Data Type	Description
NAME	VARCHAR2(64)	Name of the map projection. Specify a value from this column in the PROJECTION specification of the WKT for any user-defined coordinate system. Examples: *Geographic (Lat/Long)*, *Universal Transverse Mercator*, *State Plane Coordinates*, *Albers Conical Equal Area*.

5.5 Creating a User-Defined Coordinate System

To create a user-defined coordinate system, add a row to the MDSYS.CS_SRS table. See Section 5.4.1 for information about this table, including the requirements for values in each column.

To specify the WKTEXT column in the MDSYS.CS_SRS table, follow the syntax specified in Section 5.4.1.1. See also the examples in that section.

When you specify the WKTEXT column entry, use valid values from several Spatial reference tables:

- MDSYS.SDO_ANGLE_UNITS (see Section 5.4.2) in a UNIT specification for angle units
- MDSYS.SDO_DIST_UNITS (see Section 5.4.3) in a UNIT specification for distance units
- MDSYS.SDO_DATUMS (see Section 5.4.4) in the DATUM specification, or a user-defined datum not in MDSYS.SDO_DATUMS

  If you supply a user-defined datum, the datum name must be different from any datum name in the MDSYS.SDO_DATUMS table, and the WKT must specify the datum parameters in the order listed in Section 5.4.1.1. Any parameters not specified in a user-defined datum default to zero.
- MDSYS.SDO_ELLIPSOIDS (see Section 5.4.5) in the SPHEROID specification
- MDSYS.SDO_PROJECTIONS (see Section 5.4.6) in the PROJECTION specification

Use meters as the linear unit and decimal degrees as the angular unit for semi-major axis, prime meridian, and projection parameter specifications.

The name in each PARAMETER specification must be one of the following, depending on the projection that you use:

- *Standard_Parallel_1* (in decimal degrees)
- *Standard_Parallel_2* (in decimal degrees)
- *Central_Meridian* (in decimal degrees)
- *Latitude_of_Origin* (in decimal degrees)
- *Azimuth* (in decimal degrees)
- *False_Easting* (in meters)
- *False_Northing* (in meters)
- *Perspective_Point_Height* (in meters)
- *Landsat_Number* (must be 1, 2, 3, 4, or 5)
- *Path_Number*
- *Scale_Factor*

Some of these parameters are appropriate for several projections. They are not all appropriate for every projection.

5.6 Coordinate System Transformation Functions

The current release of Oracle Spatial includes the following functions and procedures for data transformation using coordinate systems:

- SDO_CS.TRANSFORM function: Transforms a geometry representation using a coordinate system (specified by SRID or name).
- SDO_CS.TRANSFORM_LAYER procedure: Transforms an entire layer of geometries (that is, all geometries in a specified column in a table).
- SDO_CS.VIEWPORT_TRANSFORM procedure: Transforms an optimized rectangle into a valid geodetic polygon for use with Spatial operators and functions.

Reference information about these functions and procedures is in Chapter 14.

Support for additional functions and procedures is planned for future releases of Oracle Spatial.

5.7 Notes and Restrictions with Coordinate Systems Support

The following notes and restrictions apply to coordinate systems support in the current release of Spatial.

If you have geodetic data, see also Section 5.2 for considerations, guidelines, and additional restrictions.

5.7.1 Functions Not Supported with Geodetic Data

In the current release, the following functions are not supported with geodetic data:

- SDO_AGGR_MBR
- SDO_GEOM.SDO_MBR
- SDO_GEOM.SDO_MAX_MBR_ORDINATE
- SDO_GEOM.SDO_MIN_MBR_ORDINATE
- All *3D* formats of LRS functions (explained in Section 6.4)

5.7.2 Functions Supported by Approximations with Geodetic Data

In the current release, the following functions are supported by approximations with geodetic data:

- SDO_GEOM.SDO_BUFFER
- SDO_GEOM.SDO_CENTROID
- SDO_GEOM.SDO_CONVEXHULL

When these functions are used on data with geodetic coordinates, they internally perform the operations in an implicitly generated local-tangent-plane Cartesian coordinate system and then transform the results to the geodetic coordinate system. For SDO_GEOM.SDO_BUFFER, generated arcs are approximated by line segments before the back-transform.

5.8 Example of Coordinate System Transformation

This section presents a simplified example that uses coordinate system transformation functions and procedures. It refers to concepts that are explained in this chapter and uses functions documented in Chapter 14.

Example 5–1 uses mostly the same geometry data (cola markets) as in Section 2.1, except that instead of null SDO_SRID values, the SDO_SRID value 8307 is used. That is, the geometries are defined as using the coordinate system whose SRID is 8307 and whose well-known name is "Longitude / Latitude (WGS 84)". This is probably the most widely used coordinate system, and it is the one used for global positioning system (GPS) devices. The geometries are then transformed using the

coordinate system whose SRID is 8199 and whose well-known name is "Longitude / Latitude (Arc 1950)".

Example 5–1 uses the geometries illustrated in Figure 2–1 in Section 2.1, except that *cola_d* is a rectangle (here, a square) instead of a circle, because arcs are not supported with geodetic coordinate systems.

Example 5–1 does the following:

- Creates a table (COLA_MARKETS_CS) to hold the spatial data
- Inserts rows for four areas of interest (*cola_a*, *cola_b*, *cola_c*, *cola_d*), using the SDO_SRID value 8307
- Updates the USER_SDO_GEOM_METADATA view to reflect the dimension of the areas, using the SDO_SRID value 8307
- Creates a spatial index (COLA_SPATIAL_IDX_CS)
- Performs some transformation operations (single geometry and entire layer)

Example 5–2 includes the output of the SELECT statements in Example 5–1.

Example 5–1 Simplified Example of Coordinate System Transformation

```
-- Create a table for cola (soft drink) markets in a
-- given geography (such as city or state).
-- Each row will be an area of interest for a specific
-- cola (for example, where the cola is most preferred
-- by residents, where the manufacturer believes the
-- cola has growth potential, etc.

CREATE TABLE cola_markets_cs (
  mkt_id NUMBER PRIMARY KEY,
  name VARCHAR2(32),
  shape MDSYS.SDO_GEOMETRY);

-- Note re. areas of interest: cola_a (rectangle) and
-- cola_b (4-sided polygon) are side by side (share 1 border).
-- cola_c is a small 4-sided polygon that overlaps parts of
-- cola_a and cola_b. A rough sketch:
--     ---------+-
--     |    a    |  b  \
--     |     +------+    |
--     |   /___c _|     |
--     |         |          |
--     ---------+---------|
```

```
-- The next INSERT statement creates an area of interest for
-- Cola A. This area happens to be a rectangle.
-- The area could represent any user-defined criterion: for
-- example, where Cola A is the preferred drink, where
-- Cola A is under competitive pressure, where Cola A
-- has strong growth potential, and so on.

INSERT INTO cola_markets_cs VALUES(
  1,
  'cola_a',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    8307,  -- SRID for 'Longitude / Latitude (WGS 84)' coordinate system
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- polygon
    MDSYS.SDO_ORDINATE_ARRAY(1,1, 5,1, 5,7, 1,7, 1,1) -- All vertices must
              -- be defined for rectangle with geodetic data.
  )
);

-- The next two INSERT statements create areas of interest for
-- Cola B and Cola C. These areas are simple polygons (but not
-- rectangles).

INSERT INTO cola_markets_cs VALUES(
  2,
  'cola_b',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    8307,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon (exterior polygon ring)
    MDSYS.SDO_ORDINATE_ARRAY(5,1, 8,1, 8,6, 5,7, 5,1)
  )
);

INSERT INTO cola_markets_cs VALUES(
  3,
  'cola_c',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    8307,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), --one polygon (exterior polygon ring)
    MDSYS.SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
```

```
  )
);

-- Insert a rectangle (here, square) instead of a circle as in the original,
-- because arcs are not supported with geodetic coordinate systems.
INSERT INTO cola_markets_cs VALUES(
  4,
  'cola_d',
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    8307,  -- SRID for 'Longitude / Latitude (WGS 84)' coordinate system
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- polygon
    MDSYS.SDO_ORDINATE_ARRAY(10,9, 11,9, 11,10, 10,10, 10,9) -- All vertices
must
              -- be defined for rectangle with geodetic data.
  )
);

--------------------------------------------------------------------------
-- UPDATE METADATA VIEW --
--------------------------------------------------------------------------
-- Update the USER_SDO_GEOM_METADATA view. This is required
-- before the Spatial index can be created. Do this only once for each
-- layer (i.e., table-column combination; here: cola_markets_cs and shape).

INSERT INTO USER_SDO_GEOM_METADATA
  VALUES (
  'cola_markets_cs',
  'shape',
  MDSYS.SDO_DIM_ARRAY(
    MDSYS.SDO_DIM_ELEMENT('Longitude', -180, 180, 10),  -- 10 meters tolerance
    MDSYS.SDO_DIM_ELEMENT('Latitude', -90, 90, 10)  -- 10 meters tolerance
     ),
  8307   -- SRID for ''Longitude / Latitude (WGS 84)' coordinate system
);

-----------------------------------------------------------------
-- CREATE THE SPATIAL INDEX --
-----------------------------------------------------------------
-- Must be R-tree; quadtree not supported for geodetic data.
CREATE INDEX cola_spatial_idx_cs
ON cola_markets_cs(shape)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

```
-----------------------------------------------------------------
-- TEST COORDINATE SYSTEM TRANSFORMATION --
-----------------------------------------------------------------

-- Return the transformation of cola_c using to_srid 8199
-- ('Longitude / Latitude (Arc 1950)')
SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 8199)
  FROM cola_markets_cs c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

-- Same as preceding, but using to_srname parameter.
SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 'Longitude / Latitude (Arc
1950)')
  FROM cola_markets_cs c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

-- Transform the entire SHAPE layer and put results in the table
-- named cola_markets_cs_8199, which the procedure will create.
EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS_CS','SHAPE','COLA_MARKETS_CS_
8199',8199);

-- Select all from the old (existing) table.
SELECT * from cola_markets_cs;

-- Select all from the new (layer transformed) table.
SELECT * from cola_markets_cs_8199;

-- Show metadata for the new (layer transformed) table.
DESCRIBE cola_markets_cs_8199;

-- Viewport_Transform
SELECT c.name FROM cola_markets_cs c WHERE
   SDO_FILTER(c.shape, SDO_CS.VIEWPORT_TRANSFORM(
       MDSYS.SDO_GEOMETRY(
           2003,
           0,    -- SRID = 0 (special case)
           NULL,
           MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
           MDSYS.SDO_ORDINATE_ARRAY(-180,-90,180,90)),
       8307), 'querytype=window') = 'TRUE';
```

Example 5–2 shows the output of the SELECT statements in Example 5–1. Notice the slight differences between the coordinates in the original geometries (SRID 8307)

and the transformed coordinates (SRID 8199) -- for example, (1, 1, 5, 1, 5, 7, 1, 7, 1, 1) and (1.00078604, 1.00274579, 5.00069354, 1.00274488, 5.0006986, 7.00323528, 1.00079179, 7.00324162, 1.00078604, 1.00274579) for *cola_a*.

Example 5–2 Output of SELECT Statements in Coordinate System Transformation Example

```
SQL> -- Return the transformation of cola_c using to_srid 8199
SQL> -- ('Longitude / Latitude (Arc 1950)')
SQL> SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 8199)
  2    FROM cola_markets_cs c, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  4    AND c.name = 'cola_c';

NAME
--------------------------------
SDO_CS.TRANSFORM(C.SHAPE,M.DIMINFO,8199)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z)
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074114, 3.00291482, 6.00067068, 3.00291287, 6.0006723, 5.00307625, 4.0007
1961, 5.00307838, 3.00074114, 3.00291482))


SQL>
SQL> -- Same as preceding, but using to_srname parameter.
SQL> SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 'Longitude / Latitude
(Arc 1950)')
  2    FROM cola_markets_cs c, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  4    AND c.name = 'cola_c';

NAME
--------------------------------
SDO_CS.TRANSFORM(C.SHAPE,M.DIMINFO,'LONGITUDE/LATITUDE(ARC1950)')(SDO_GTYPE, SDO
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074114, 3.00291482, 6.00067068, 3.00291287, 6.0006723, 5.00307625, 4.0007
1961, 5.00307838, 3.00074114, 3.00291482))


SQL>
SQL> -- Transform the entire SHAPE layer and put results in the table
SQL> -- named cola_markets_cs_8199, which the procedure will create.
```

```
SQL> EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS_CS','SHAPE','COLA_MARKETS_CS_
8199',8199);

PL/SQL procedure successfully completed.

SQL>
SQL> -- Select all from the old (existing) table.
SQL> SELECT * from cola_markets_cs;

    MKT_ID NAME
---------- --------------------------------
SHAPE(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
         1 cola_a
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(1, 1, 5, 1, 5, 7, 1, 7, 1, 1))

         2 cola_b
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(5, 1, 8, 1, 8, 6, 5, 7, 5, 1))

         3 cola_c

    MKT_ID NAME
---------- --------------------------------
SHAPE(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3, 3, 6, 3, 6, 5, 4, 5, 3, 3))

         4 cola_d
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(10, 9, 11, 9, 11, 10, 10, 10, 10, 9))


SQL>
SQL> -- Select all from the new (layer transformed) table.
SQL> SELECT * from cola_markets_cs_8199;

SDO_ROWID
------------------
GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
AAABZzAABAAAOa6AAA
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
```

```
AY(1.00078604, 1.00274579, 5.00069354, 1.00274488, 5.0006986, 7.00323528, 1.0007
9179, 7.00324162, 1.00078604, 1.00274579))

AAABZzAABAAAOa6AAB
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(5.00069354, 1.00274488, 8.00062191, 1.00274427, 8.00062522, 6.00315345, 5.000
6986, 7.00323528, 5.00069354, 1.00274488))

SDO_ROWID
------------------
GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------

AAABZzAABAAAOa6AAC
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074114, 3.00291482, 6.00067068, 3.00291287, 6.0006723, 5.00307625, 4.0007
1961, 5.00307838, 3.00074114, 3.00291482))

AAABZzAABAAAOa6AAD
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(10.0005802, 9.00337775, 11.0005553, 9.00337621, 11.0005569, 10.0034478, 10.00

SDO_ROWID
------------------
GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
05819, 10.0034495, 10.0005802, 9.00337775))


SQL>
SQL> -- Show metadata for the new (layer transformed) table.
SQL> DESCRIBE cola_markets_cs_8199;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SDO_ROWID                                          ROWID
 GEOMETRY                                           MDSYS.SDO_GEOMETRY

SQL>
SQL> -- Viewport_Transform
SQL> SELECT c.name FROM cola_markets_cs c WHERE
  2  SDO_FILTER(c.shape, SDO_CS.VIEWPORT_TRANSFORM(
  3      MDSYS.SDO_GEOMETRY(
  4  2003,
  5  0,    -- SRID = 0 (special case)
  6  NULL,
```

```
  7  MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
  8  MDSYS.SDO_ORDINATE_ARRAY(-180,-90,180,90)),
  9      8307), 'querytype=window') = 'TRUE';

NAME
--------------------------------
cola_a
cola_c
cola_b
cola_d
```

6

Linear Referencing System

Linear referencing is a natural and convenient means to associate attributes or events to locations or portions of a linear feature. It has been widely used in transportation applications (such as for highways, railroads, and transit routes) and utilities applications (such as for gas and oil pipelines). The major advantage of linear referencing is its capability of locating attributes and events along a linear feature with only one parameter (usually known as *measure*) instead of two (such as *latitude/longitude* or *x/y* in Cartesian space). Sections of a linear feature can be referenced and created dynamically by indicating the start and end locations along the feature without explicitly storing them.

The linear referencing system (LRS) application programming interface (API) in Oracle Spatial provides server-side LRS capabilities at the cartographic level. The linear measure information is directly integrated into the Oracle Spatial geometry structure. The Oracle Spatial LRS API provides support for dynamic segmentation, and it serves as a groundwork for third-party or middle-tier application development virtually for any linear referencing methods and models in any coordinate systems.

For an example of LRS, see Section 6.6. However, you may want to read the rest of this chapter first, to understand the concepts that the example illustrates.

For reference information about LRS functions, see Chapter 15.

If you have LRS data from a previous release of Spatial, see Section A.5 for information about migrating LRS data.

6.1 Terms and Concepts

This section explains important terms and concepts related to linear referencing support in Oracle Spatial.

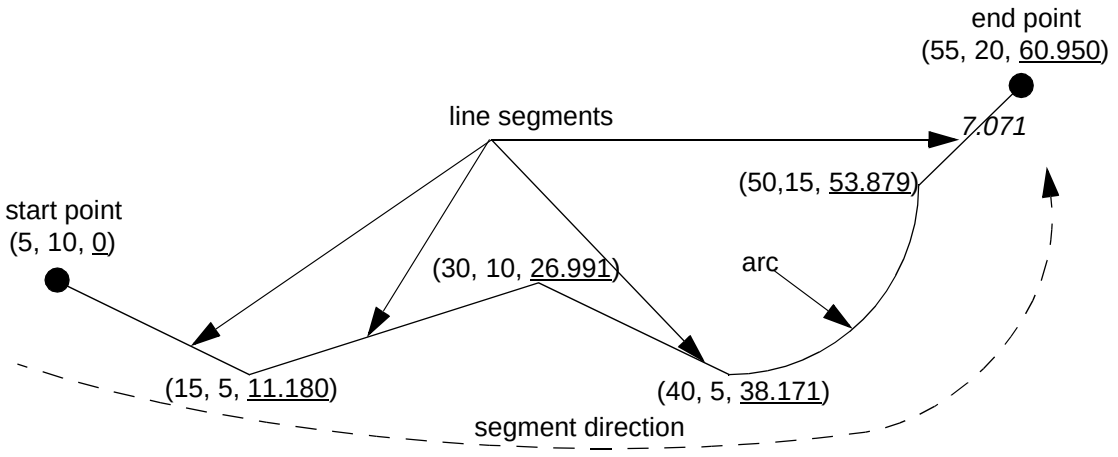
6.1.1 Geometric Segments (LRS Segments)

Geometric segments are basic LRS elements in Oracle Spatial. A geometric segment can be any of the following:

- Line string: an ordered, non-branching, and continuous geometry (for example, a simple road)
- Multiline string: non-connected line strings (for example, a highway with a gap caused by a lake or a bypass road)
- Polygon (for example, a racetrack or a scenic tour route that starts and ends at the same point)

A geometric segment must contain at least start and end measures for its start and end points. Measures of points of interest (such as highway exits) on the geometric segments can also be assigned. These measures are either assigned by users or derived from existing geometric segments. Figure 6–1 shows a geometric segment with four line segments and one arc. Points on the geometric segment are represented by triplets (x, y, m), where *x* and *y* describe the location and *m* denotes the measure (with each measure value underlined in Figure 6–1).

Figure 6–1 Geometric Segment



6.1.2 Shape Points

Shape points are points that are specified when an LRS segment is constructed, and that are assigned measure information. In Oracle Spatial, a line segment is

represented by its start and end points, and an arc is represented by three points: start, middle, and end points of the arc. You must specify these points as shape points, but you can also specify other points as shape points if you need measure information stored for these points (for example, an exit in the middle of a straight part of the highway).

Thus, shape points can serve one or both of the following purposes: to indicate the direction of the segment (for example, a turn or curve), and to identify a point of interest for which measure information is to be stored.

Shape points might not directly relate to mileposts or reference posts in LRS; they are used as internal reference points. The measure information of shape points is automatically populated when you define the LRS segment using the SDO_LRS.DEFINE_GEOM_SEGMENT procedure.

6.1.3 Direction of a Geometric Segment

The **direction** of a geometric segment is indicated from the start point of the geometric segment to the end point. The direction is determined by the order of the vertices (from start point to end point) in the geometry definition. Measures of points on a geometric segment always either increase or decrease along the direction of the geometric segment.

6.1.4 Measure (Linear Measure)

The **measure** of a point along a geometric segment is the linear distance (in the measure dimension) measured from the start point (for increasing values) or end point (for decreasing values) of the geometric segment. The measure information does not necessarily have to be of the same scale as their Euclidean distance. However, the linear mapping relationship between measure and distance is always preserved.

Some LRS functions use *offset* instead of measure to represent measured distance along linear features. Although some other linear referencing systems might use offset to mean what the Oracle Spatial LRS refers to as measure, offset has a different meaning in Oracle Spatial from measure, as explained in Section 6.1.5.

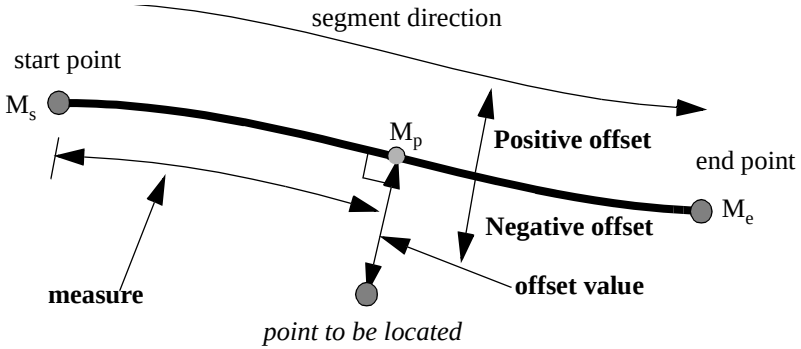
6.1.5 Offset

The **offset** of a point along a geometric segment is the perpendicular distance between the point and the geometric segment. Offsets are positive if points are on the left side along the segment direction and are negative if they are on the right side. Points are on a geometric segment if their offsets to the segment are zero.

The unit of measurement for an offset is the same as for the coordinate system associated with the geometric segment. For geodetic data, the default unit of measurement is meters.

Figure 6–2 shows how a point can be located along a geometric segment with measure and offset information. By assigning an offset together with a measure, it is possible to locate not only points that are on the geometric segment, but also points that are perpendicular to the geometric segment.

Figure 6–2 Describing a Point Along a Segment with a Measure and an Offset



6.1.6 Measure Populating

Any unassigned measures of a geometric segment are automatically populated based upon their distance distribution. This is done before any LRS operations for geometric segments with unknown measures (NULL in Oracle Spatial). The resulting geometric segments from any LRS operations return the measure information associated with geometric segments. The measure of a point on the geometric segment can be obtained based upon a linear mapping relationship between its previous and next known measures or locations. See the algorithm representation in Figure 6–3 and the example in Figure 6–4.

Figure 6–3 Measures, Distances, and Their Mapping Relationship

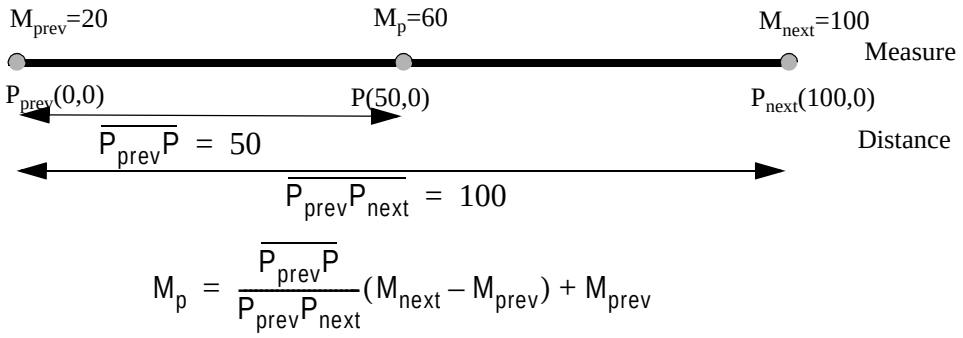
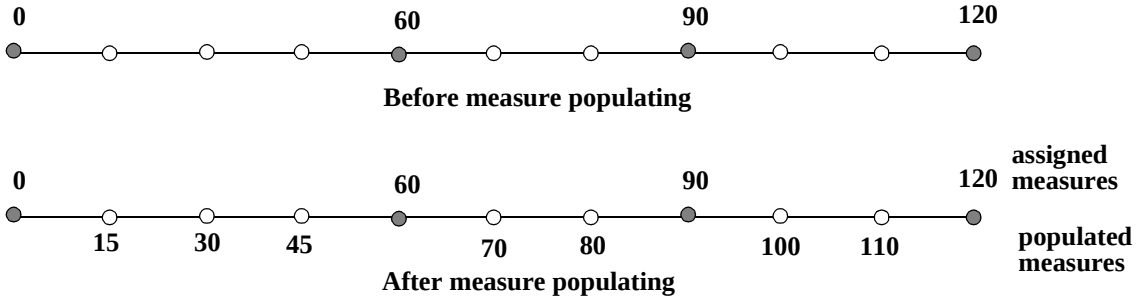


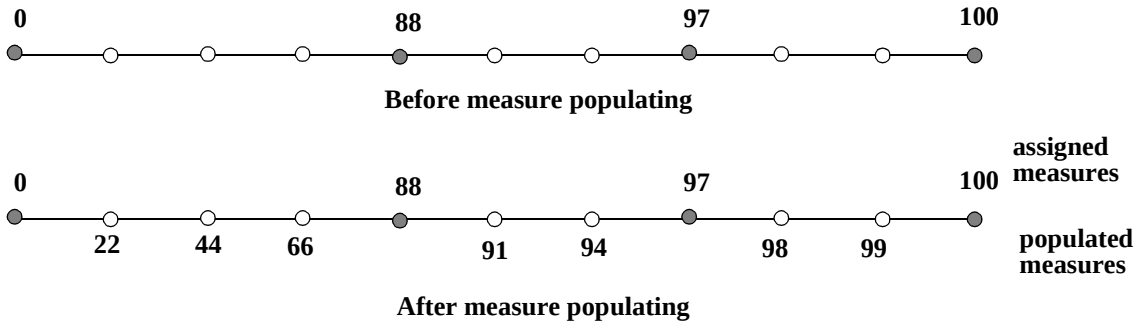
Figure 6–4 Measure Populating of a Geometric Segment



Measures are evenly spaced between assigned measures. However, the assigned measures for points of interest on a geometric segment do not need to be evenly spaced. This could eliminate the problem of error accumulation and account for inaccuracy of data source.

Moreover, the assigned measures do not even need to reflect actual distances; they can be any valid values within the measure range. For example, Figure 6–5 shows the measure population that results when assigned measure values are not proportional and reflect widely varying gaps.

Figure 6–5 Measure Populating With Disproportional Assigned Measures



In all cases, measure populating is done in an incremental fashion along the segment direction. This improves the performance of current and subsequent LRS operations.

6.1.7 Measure Range of a Geometric Segment

The start and end measures of a geometric segment define the linear **measure range** of the geometric segment. Any valid LRS measures of a geometric segment must fall within its linear measure range.

6.1.8 Projection

The **projection** of a point along a geometric segment is the point on the geometric segment with the minimum distance to the point. The measure information of the resulting point is also returned in the point geometry.

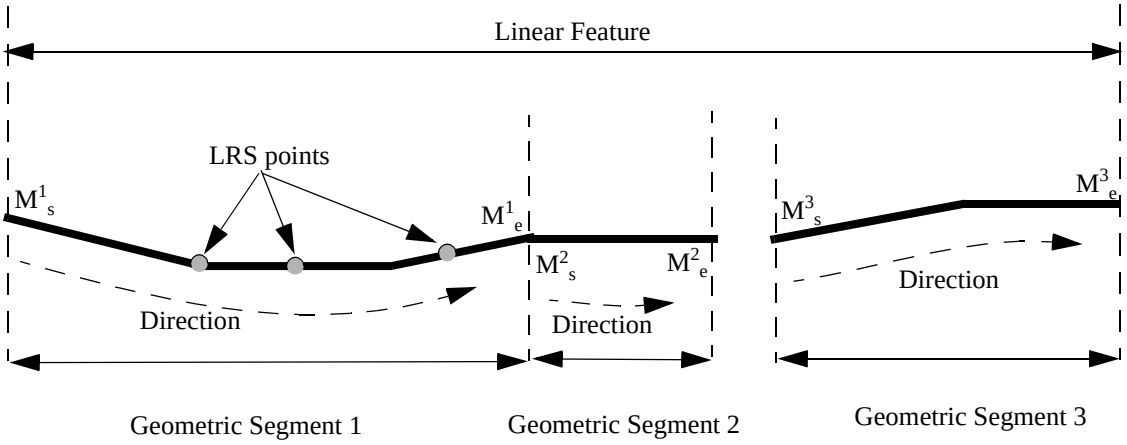
6.1.9 LRS Point

LRS points are points with linear measure information along a geometric segment. A valid LRS point is a point geometry with measure information.

6.1.10 Linear Features

Linear features are any spatial objects that can be treated as a logical set of linear segments. Examples of linear features are highways in transportation applications and pipelines in utility industry applications. The relationship of linear features, geometric segments, and LRS points is shown in Figure 6–6.

Figure 6–6 Linear Feature, Geometric Segments, and LRS Points



6.2 LRS Data Model

The Oracle Spatial LRS data model incorporates measure information into its geometry representation at the point level. The measure information is directly integrated into the Oracle Spatial model. To accomplish this, an additional *measure* dimension must be added to the Oracle Spatial metadata.

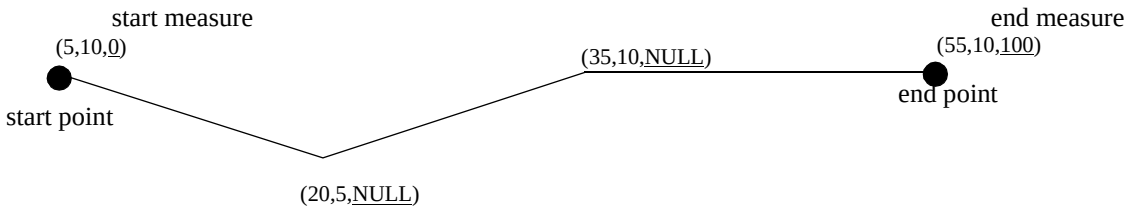
Oracle Spatial LRS support affects the Spatial metadata and data (the geometries). Example 6–1 shows how a measure dimension can be added to 2-dimensional geometries in the Spatial metadata. The measure dimension must be the last element of the SDO_DIM_ARRAY in a spatial object definition (shown in bold in Example 6–1).

Example 6–1 Including LRS Measure Dimension in Spatial Metadata

```
INSERT INTO user_sdo_geom_metadata VALUES(
  'LRS_ROUTES',
  'GEOMETRY',
  MDSYS.SDO_DIM_ARRAY (
    MDSYS.SDO_DIM_ELEMENT('X', 0, 100, 0.005),
    MDSYS.SDO_DIM_ELEMENT('Y', 0, 100, 0.005),
    MDSYS.SDO_DIM_ELEMENT('M', 0, 100, 0.005)),
  NULL);
```

After adding the new measure dimension, geometries with measure information such as geometric segments and LRS points can be represented. An example of creating a geometric segment with three line segments is shown in Figure 6–7.

Figure 6–7 Creating a Geometric Segment



In Figure 6–7, the geometric segment has the following definition (with measure values underlined):

```
SDO_GEOMETRY(3302, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1,2,1),
     MDSYS.SDO_ORDINATE_ARRAY(5,10,0, 20,5,NULL, 35,10,NULL, 55,10,100))
```

Whenever a geometric segment is defined, its start and end measures must be defined or derived from some existing geometric segment. The unsigned measures of all shape points on a geometric segment will be automatically populated.

The LRS API works with geometries in formats of Oracle Spatial before release 8.1.6, but the resulting geometries will be converted to the Oracle Spatial release 8.1.6 or higher format, specifically with 4-digit SDO_GTYPE and SDO_ETYPE values. For example, in Oracle Spatial release 8.1.6 and higher, the geometry type (SDO_GTYPE) of a spatial object includes the number of dimensions of the object as the first digit of the SDO_GTYPE value. Thus, the SDO_GTYPE value of a point is 1 in the pre-release 8.1.6 format but 2001 in the release 8.1.6 format (the number of dimensions of the point is 2). However, an LRS point (which includes measure information) has 3 dimensions, and thus the SDO_GTYPE of any point geometry used with an LRS function must be 3301.

6.3 Indexing of LRS Data

When LRS data is indexed using a spatial quadtree index, only the first two dimensions are indexed; the measure dimension and values are not indexed.

When LRS data is indexed using a spatial R-tree index, you must use the SDO_INDX_DIMS keyword in the CREATE INDEX statement in order to limit the

number of dimensions to be indexed (for example, SDO_INDX_DIMS=2 to index only the X and Y dimensions and not the measure dimension, or SDO_INDX_DIMS=3 to index only the X, Y, and Z dimensions and not the measure dimension). There is no benefit to including the measure dimension in a spatial index, and there is additional processing overhead; therefore, you should use the SDO_INDX_DIMS keyword when spatially indexing LRS data.

Example 6–2 creates a spatial R-tree index on LRS data. It specifies SDO_INDX_DIMS=2 to index only the X and Y dimensions.

Example 6–2 Creating an Index on LRS Data

```
CREATE INDEX lrs_routes_idx ON lrs_routes(route_geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('SDO_INDX_DIMS=2');
```

Information about the CREATE INDEX statement and its parameters and keywords is in Chapter 9.

6.4 3D Formats of LRS Functions

Most LRS functions have formats that end in *_3D*: for example, DEFINE_GEOM_SEGMENT_3D, CLIP_GEOM_SEGMENT_3D, FIND_MEASURE_3D, and LOCATE_PT_3D. If a function has a *3D* format, it is identified in the Usage Notes for the function in Chapter 15.

The *3D* formats should be used only when the geometry object has 4 dimensions and the fourth dimension is the measure (for example, X, Y, Z, and M), and only when you want the function to consider the first 3 dimensions (for example, X, Y, and Z). If the standard format of a function (that is, without the *_3D*) is used on a geometry with 4 dimensions, the function considers only the first 2 dimensions (for example, X and Y).

For example, the following format considers the X, Y, and Z dimensions of the specified GEOM object in performing the clip operation:

```
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT_3D(a.geom, m.diminfo, 5, 10)
  FROM routes r, user_sdo_geom_metadata m
  WHERE m.table_name = 'ROUTES' AND m.column_name = 'GEOM'
    AND r.route_id = 1;
```

However, the following format considers only the X and Y dimensions, and ignores the Z dimension, of the specified GEOM object in performing the clip operation:

```
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(a.geom, m.diminfo, 5, 10)
  FROM routes r, user_sdo_geom_metadata m
  WHERE m.table_name = 'ROUTES' AND m.column_name = 'GEOM'
    AND r.route_id = 1;
```

The parameters for the standard and *3D* formats of any function are the same, and the usage notes apply to both formats.

The *3D* formats are not supported with geodetic data.

6.5 LRS Operations

This section describes several linear referencing operations supported by the Oracle Spatial LRS API.

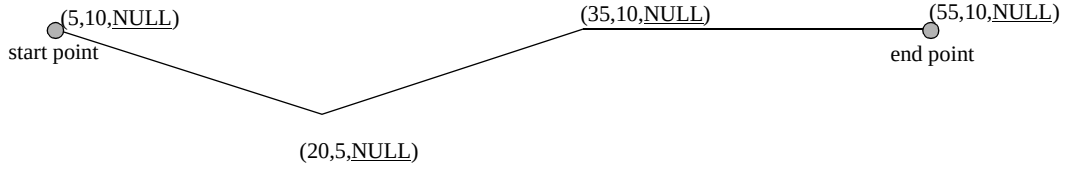
6.5.1 Defining a Geometric Segment

There are two ways to create a geometric segment with measure information:

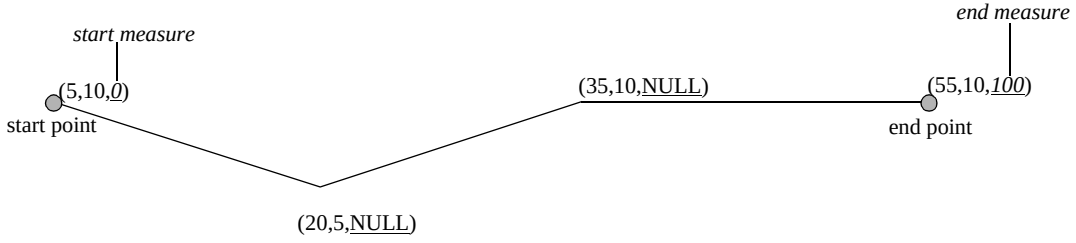
- Construct a geometric segment and assign measures explicitly.
- Define a geometric segment with specified start and end, and / or any other measures, in an ascending or descending order. Measures of shape points with unknown (unassigned) measures (null values) in the geometric segment will be automatically populated according to their locations and distance distribution.

Figure 6–8 shows different ways of defining a geometric segment.

Figure 6–8 Defining a Geometric Segment



a. Geometric segment with no measures assigned



b. Geometric segment with start/end measures

c. Populating measures of shape points in a geometric segment

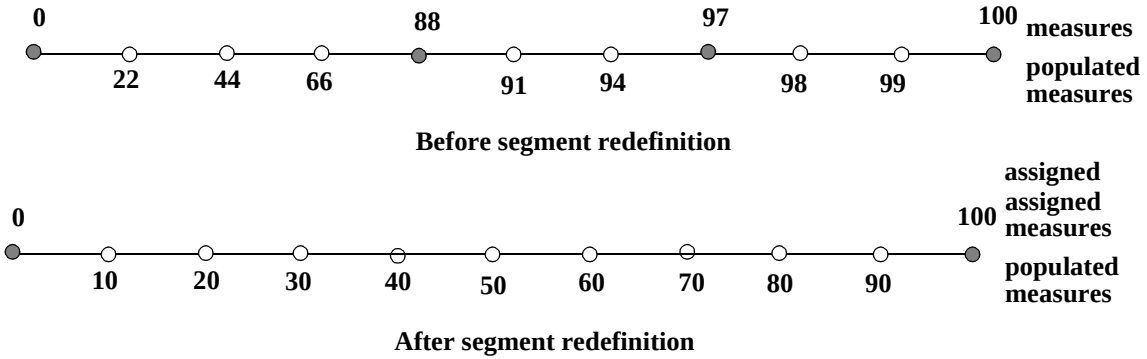
An LRS segment must be defined (or must already exist) before any LRS operations can proceed. That is, the start, end, and any other assigned measures must be present to derive the location from a specified measure. The measure information of intermediate shape points will automatically be populated if they are not assigned.

6.5.2 Redefining a Geometric Segment

You can redefine a geometric segment to replace the existing measures of all shape points between the start and end point with automatically calculated measures. Redefining a segment can be useful if errors have been made in one or more explicit measure assignments, and you want to start over with proportionally assigned measures.

Figure 6–9 shows the redefinition of a segment where the existing (before) assigned measure values are not proportional and reflect widely varying gaps.

Figure 6–9 Redefining a Geometric Segment

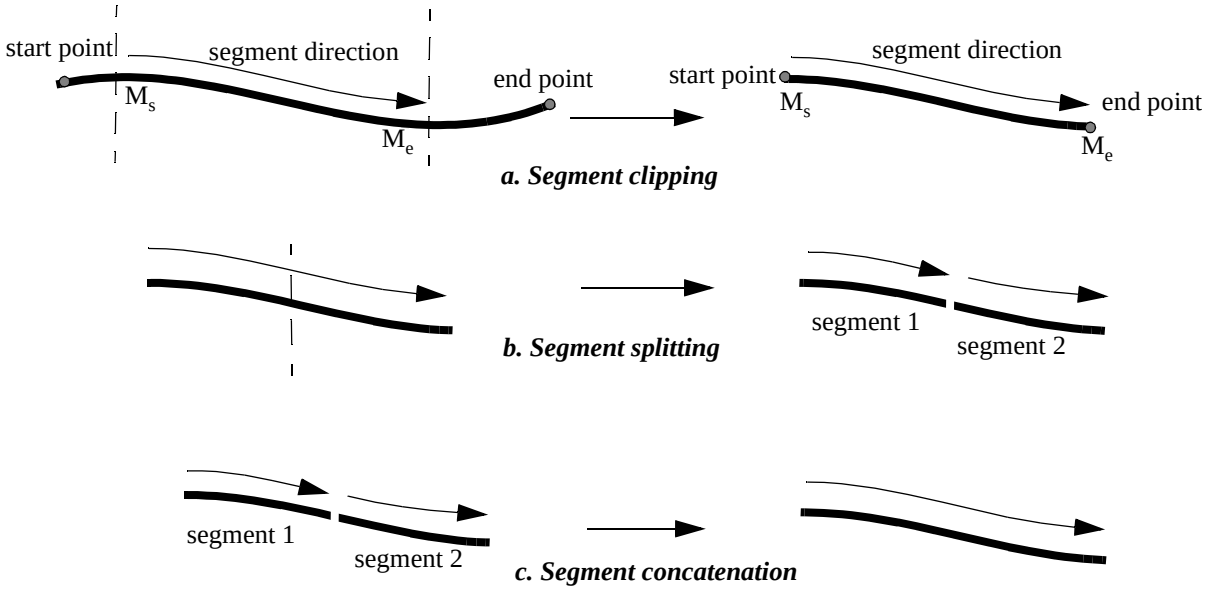


After the segment redefinition in Figure 6–9, the populated measures reflect proportional distances along the segment.

6.5.3 Clipping a Geometric Segment

You can clip a geometric segment to create a new geometric segment out of an existing geometric segment (Figure 6–10, part a).

Figure 6–10 Clipping, Splitting, and Concatenating Geometric Segments



6.5.4 Splitting a Geometric Segment

You can create two new geometric segments by splitting a geometric segment (Figure 6–10, part b).

Note: In Figure 6–10 and several that follow, small gaps between segments are used in illustrations of segment splitting and concatenation. Each gap simply reinforces the fact that two different segments are involved. However, the two segments (such as segment 1 and segment 2 in Figure 6–10, parts b and c) are actually connected. The tolerance (see Section 1.5.5) is considered in determining whether or not segments are connected.

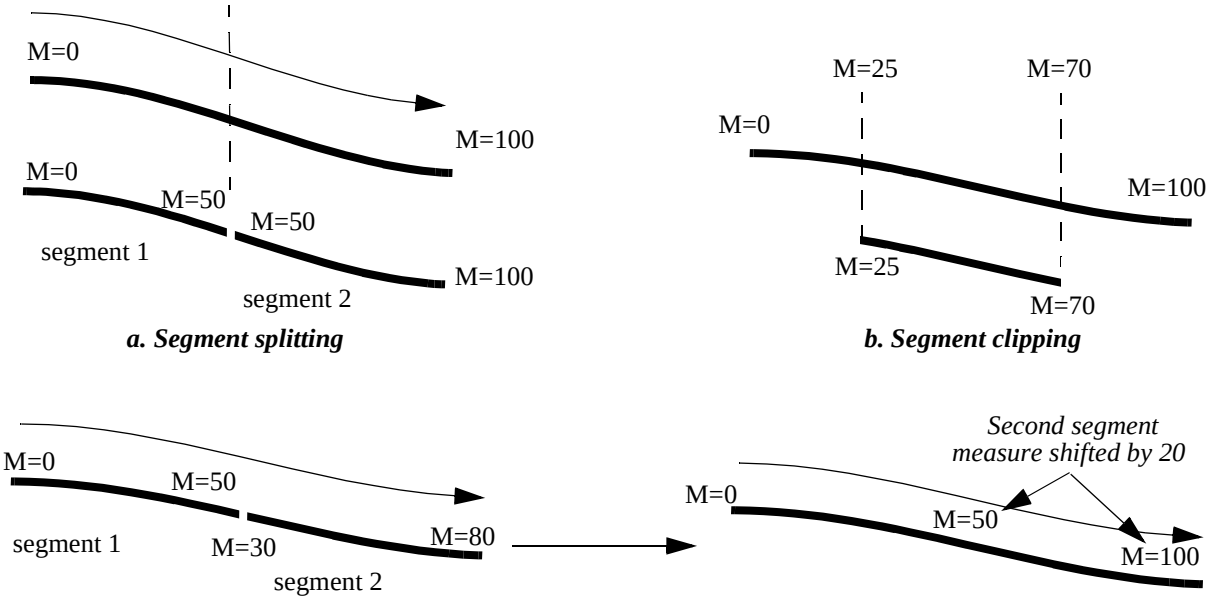
6.5.5 Concatenating Geometric Segments

You can create a new geometric segment by concatenating two geometric segments (Figure 6–10, part c). Note that the geometric segments do not need to be spatially connected, although they are connected in the illustration in Figure 6–10, part c. The

measures of the second geometric segment are shifted so that the end measure of the first segment is the same as the start measure of the second segment.

Measure assignments for the clipping, splitting, and concatenating operations in Figure 6–10 are shown in Figure 6–11. Measure information and segment direction are preserved in a consistent manner. The assignment is done automatically when the operations have completed.

Figure 6–11 Measure Assignment in Geometric Segment Operations



a. Segment splitting

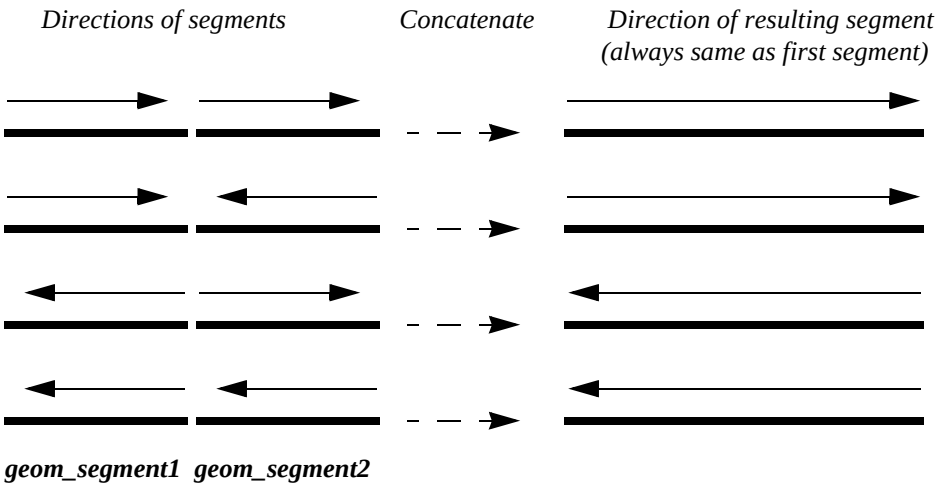
b. Segment clipping

Continuous measures for segment concatenation

c. Segment concatenation

The direction of the geometric segment resulting from concatenation is always the direction of the first segment (*geom_segment1* in the call to the SDO_LRS.CONCATENATE_GEOM_SEGMENTS function), as shown in Figure 6–12.

Figure 6–12 Segment Direction with Concatenation

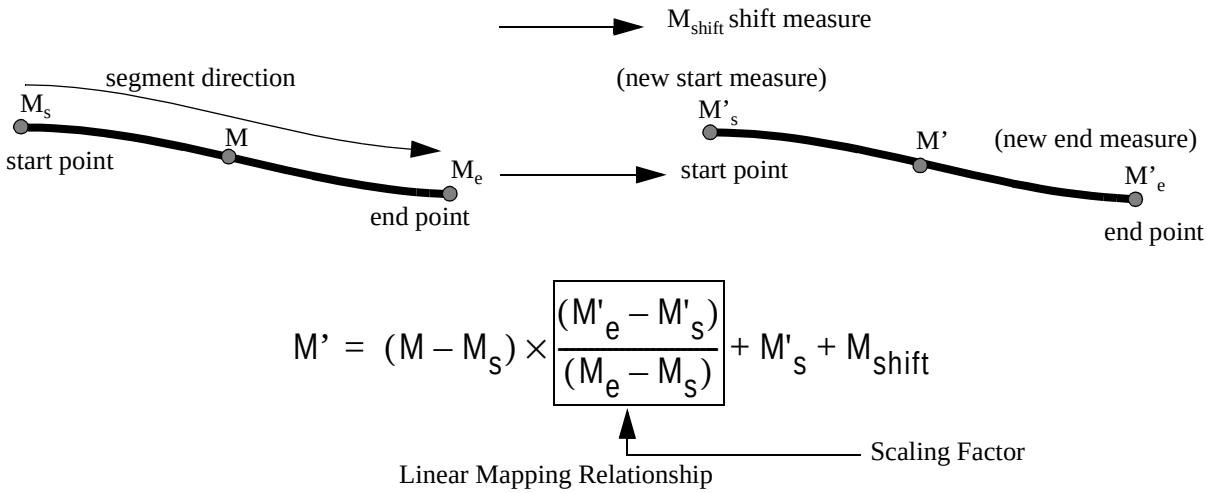


In addition to explicitly concatenating two connected segments using the SDO_LRS.CONCATENATE_GEOM_SEGMENTS function, you can perform aggregate concatenation: that is, you can concatenate all connected geometric segments in a column (layer) using the SDO_AGGR_LRS_CONCAT spatial aggregate function. (See the description and example of the SDO_AGGR_LRS_CONCAT spatial aggregate function in Chapter 13.)

6.5.6 Scaling a Geometric Segment

You can create a new geometric segment by performing a linear scaling operation on a geometric segment. Figure 6–13 shows the mapping relationship for geometric segment scaling.

Figure 6–13 Scaling a Geometric Segment



In general, scaling a geometric segment only involves rearranging measures of the newly created geometric segment. However, if the scaling factor is negative, the order of the shape points needs to be reversed so that measures will increase along the geometric segment's direction (which is defined by the order of the shape points).

A scale operation can perform any combination of the following operations:

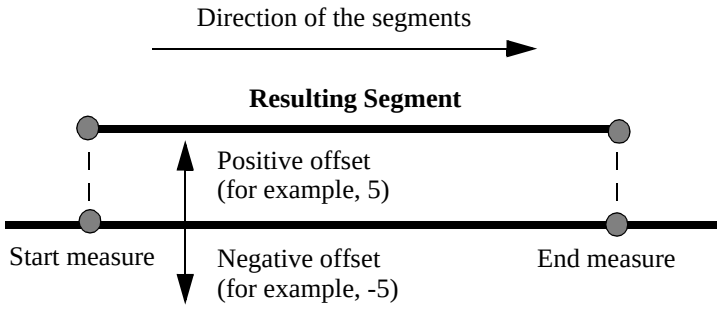
- Translating (shifting) measure information. (For example, add the same value to Ms and Me to get M's and M'e.)
- Reversing measure information. (Let M's = Me, M'e = Ms, and Mshift = 0.)
- Performing simple scaling of measure information. (Let Mshift = 0.)

For examples of these operations, see usage notes and examples for the SDO_LRS.SCALE_GEOM_SEGMENT function in Chapter 15.

6.5.7 Offsetting a Geometric Segment

You can create a new geometric segment by performing an offsetting operation on a geometric segment. Figure 6–14 shows the mapping relationship for geometric segment offsetting.

Figure 6–14 Offsetting a Geometric Segment



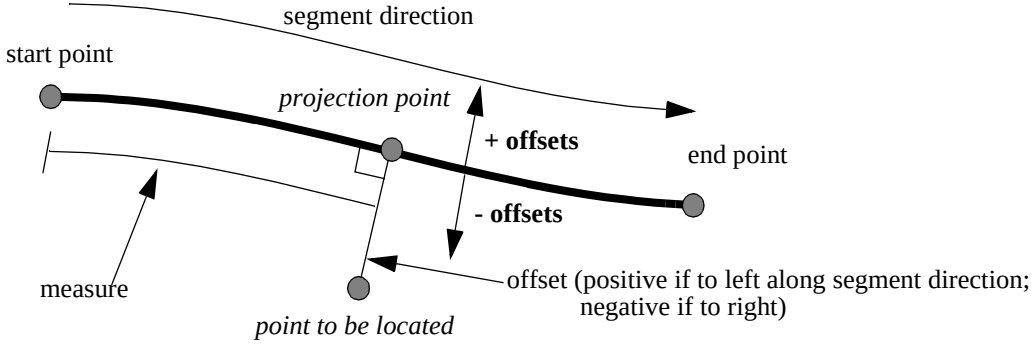
In the offsetting operation shown in Figure 6–14, the resulting geometric segment is offset by 5 units from the specified start and end measures of the original segment.

For more information, see usage notes and examples for the SDO_LRS.OFFSET_GEOM_SEGMENT function in Chapter 15.

6.5.8 Locating a Point on a Geometric Segment

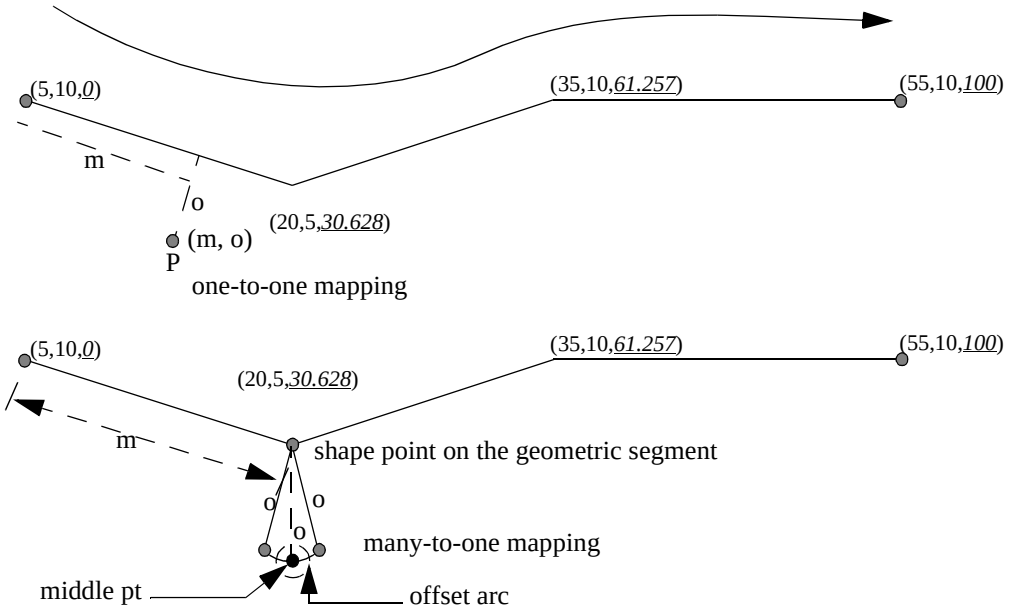
You can find the position of a point described by a measure and an offset on a geometric segment (see Figure 6–15).

Figure 6–15 Locating a Point Along a Segment with a Measure and an Offset



There is always a unique a location with a specific measure on a geometric segment. Ambiguity arises when offsets are given and the points described by the measures fall on shape points of the geometric segment (see Figure 6–16).

Figure 6–16 Ambiguity in Location Referencing with Offsets



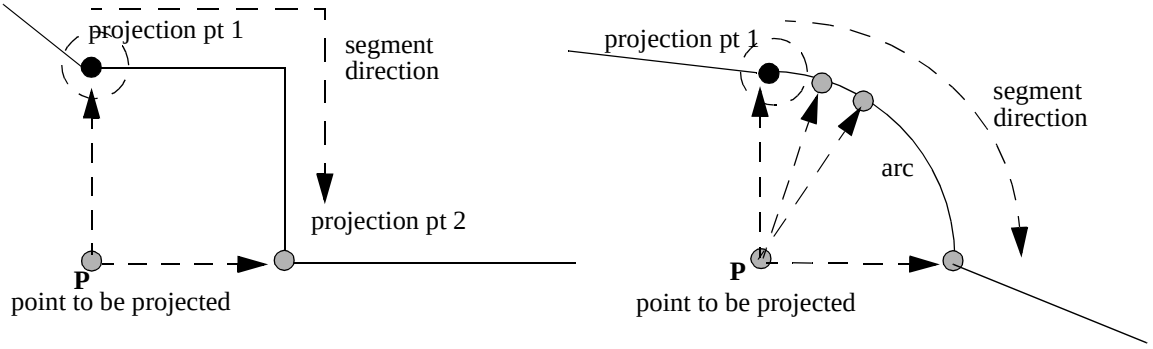
As shown in Figure 6–16, an offset arc of a shape point on a geometric segment is an arc on which all points have the same minimum distance to the shape point. As a result, all points on the offset arc are represented by the same (measure, offset) pair. To resolve this one-to-many mapping problem, the middle point on the offset arc is returned.

6.5.9 Projecting a Point onto a Geometric Segment

You can find the projection point of a point with respect to a geometric segment. The point to be projected can be on or off the segment. If the point is on the segment, the point and its projection point are the same.

Projection is a reverse operation of the point-locating operation shown in Figure 6–15. Similar to a point-locating operation, all points on the offset arc of a shape point will have the same projection point (that is, the shape point itself), measure, and offset (see Figure 6–16). If there are multiple projection points for a point, the first one from the start point is returned (projection pt 1 in both illustrations in Figure 6–17).

Figure 6–17 Multiple Projection Points



6.5.10 Converting Geometric Segments

You can convert geometric segments from standard line string format to Linear Referencing System format, and vice versa. The main use of conversion functions will probably occur if you have a large amount of existing line string data, in which case conversion is a convenient alternative to creating all of the LRS segments manually. However, if you need to convert LRS segments to standard line strings for certain applications, that capability is provided also.

Functions are provided to convert:

- Individual line strings

  For conversion from standard format to LRS format, a measure dimension (named *M* by default) is added, and measure information is provided for each point. For conversion from LRS format to standard format, the measure dimension and information are removed. In both cases, the dimensional information (DIMINFO) metadata in the USER_SDO_GEOM_METADATA view is not affected.

- Layers (all line strings in a column)

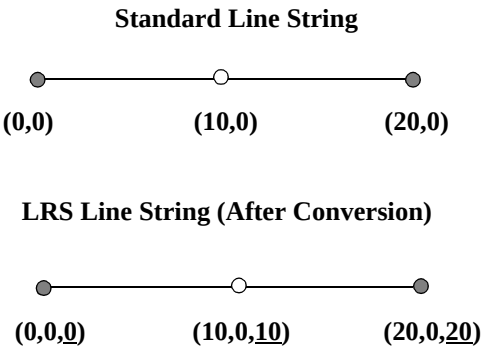
  For conversion from standard format to LRS format, a measure dimension (named *M* by default) is added, but no measure information is provided for each point. For conversion from LRS format to standard format, the measure dimension and information are removed. In both cases, the dimensional information (DIMINFO) metadata in the USER_SDO_GEOM_METADATA view is modified as needed.

- Dimensional information (DIMINFO)

The dimensional information (DIMINFO) metadata in the USER_SDO_GEOM_METADATA view is modified as needed. For example, converting a standard dimensional array with X and Y dimensions (SDO_DIM_ELEMENT) to an LRS dimensional array causes an M dimension (SDO_DIM_ELEMENT) to be added.

Figure 6–18 shows the addition of measure information when a standard line string is converted to an LRS line string (using the SDO_LRS.CONVERT_TO_LRS_GEOM function). The measure dimension values are underlined in Figure 6–18.

Figure 6–18 Conversion from Standard to LRS Line String



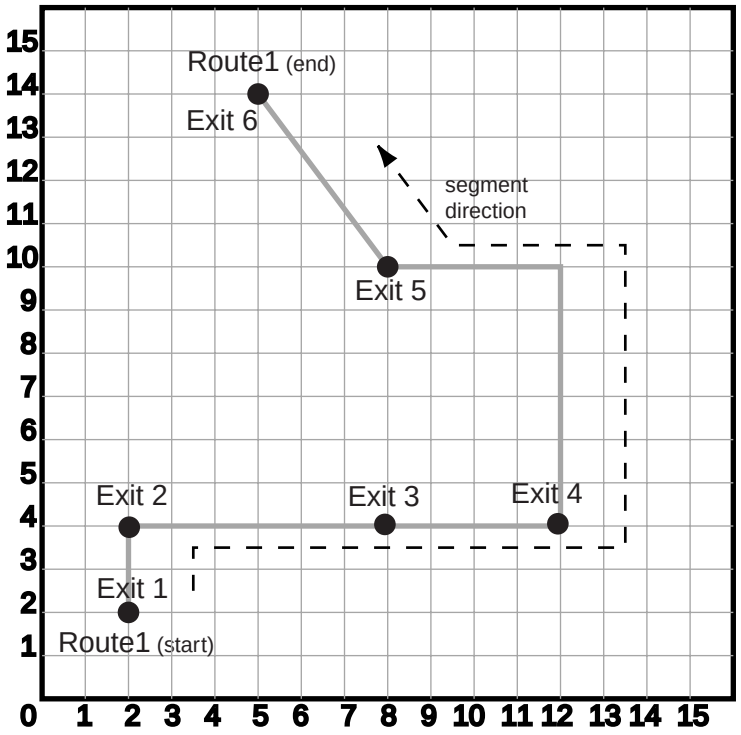
The conversion functions are listed in Table 15–3 in Chapter 15. See also the reference information in Chapter 15 about each conversion function.

6.6 Example of LRS Functions

This section presents a simplified example that uses LRS functions. It refers to concepts that are explained in this chapter and uses functions documented in Chapter 15.

This example uses the road that is illustrated in Figure 6–19.

Figure 6–19 Simplified LRS Example: Highway



In Figure 6–19, the highway (Route 1) starts at point 2,2 and ends at point 5,14, follows the path shown, and has six entrance-exit points (Exit 1 through Exit 6). For simplicity, each unit on the graph represents one unit of measure, and thus the measure from start to end is 27 (the segment from Exit 5 to Exit 6 being the hypotenuse of a 3-4-5 right triangle).

Each row in Table 6–1 lists an actual highway-related feature and the LRS feature that corresponds to it or that can be used to represent it.

Table 6–1 Highway Features and LRS Counterparts

Highway Feature	LRS Feature
Named route, road, or street	LRS segment, or linear feature (logical set of segments)
Mile or kilometer marker	Measure

Table 6–1 Highway Features and LRS Counterparts (Cont.)

Highway Feature	LRS Feature
Accident reporting and location tracking	SDO_LRS.LOCATE_PT function
Construction zone (portion of a road)	SDO_LRS.CLIP_GEOM_SEGMENT function
Road extension (adding at the beginning or end) or combination (designating or renaming two roads that meet as one road)	SDO_LRS.CONCATENATE_GEOM_SEGMENTS function
Road reconstruction or splitting (resulting in two named roads from one named road)	SDO_LRS.SPLIT_GEOM_SEGMENT function
Finding the closest point on the road to a point off the road (such as a building)	SDO_LRS.PROJECT_PT function
Guard rail or fence alongside a road.	SDO_LRS.OFFSET_GEOM_SEGMENT function

Example 6–3 does the following:

- Creates a table to hold the segment
- Inserts the definition of the highway into the table
- Inserts the necessary metadata into the USER_SDO_GEOM_METADATA view
- Uses PL/SQL and SQL statements to define the segment and perform operations on it

Example 6–4 includes the output of the SELECT statements in Example 6–3.

Example 6–3 Simplified Example: Highway

```
-- Create a table for routes (highways).
CREATE TABLE lrs_routes (
  route_id  NUMBER PRIMARY KEY,
  route_name  VARCHAR2(32),
  route_geometry  MDSYS.SDO_GEOMETRY);

-- Populate table with just one route for this example.
INSERT INTO lrs_routes VALUES(
  1,
  'Route1',
  MDSYS.SDO_GEOMETRY(
    3302,  -- line string, 3 dimensions: X,Y,M
    NULL,
```

```
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    MDSYS.SDO_ORDINATE_ARRAY(
      2,2,0,   -- Start point - Exit1; 0 is measure from start.
      2,4,2,   -- Exit2; 2 is measure from start.
      8,4,8,   -- Exit3; 8 is measure from start.
      12,4,12,  -- Exit4; 12 is measure from start.
      12,10,NULL,  -- Not an exit; measure automatically calculated and filled.
      8,10,22,  -- Exit5; 22 is measure from start.
      5,14,27)  -- End point (Exit6); 27 is measure from start.
  )
);

-- Update the Spatial metadata.
INSERT INTO USER_SDO_GEOM_METADATA
  VALUES (
  'lrs_routes',
  'route_geometry',
  MDSYS.SDO_DIM_ARRAY(   -- 20X20 grid
    MDSYS.SDO_DIM_ELEMENT('X', 0, 20, 0.005),
    MDSYS.SDO_DIM_ELEMENT('Y', 0, 20, 0.005),
    MDSYS.SDO_DIM_ELEMENT('M', 0, 20, 0.005) -- Measure dimension
     ),
  NULL   -- SRID
);

-- Create the spatial index.
-- Use SDO_INDX_DIMS parameter, to index only X, Y dimensions.
CREATE INDEX lrs_routes_idx ON lrs_routes(route_geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('SDO_INDX_DIMS=2');

-- Test the LRS procedures.
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
line_string MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;
result_geom_1 MDSYS.SDO_GEOMETRY;
result_geom_2 MDSYS.SDO_GEOMETRY;
result_geom_3 MDSYS.SDO_GEOMETRY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
```

```
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

-- Define the LRS segment for Route1. This will populate any null measures.
-- No need to specify start and end measures, because they're already defined
-- in the geometry.
SDO_LRS.DEFINE_GEOM_SEGMENT (geom_segment, dim_array);

SELECT a.route_geometry INTO line_string FROM lrs_routes a
  WHERE a.route_name = 'Route1';

-- Split Route1 into two segments.
SDO_LRS.SPLIT_GEOM_SEGMENT(line_string,dim_array,5,result_geom_1,result_geom_2);

-- Concatenate the segments that were just split.
result_geom_3 := SDO_LRS.CONCATENATE_GEOM_SEGMENTS(result_geom_1, dim_array,
result_geom_2, dim_array);

-- Update and insert geometries into table, to display later.
UPDATE lrs_routes a SET a.route_geometry = geom_segment
   WHERE a.route_id = 1;

INSERT INTO lrs_routes VALUES(
  11,
  'result_geom_1',
  result_geom_1
);
INSERT INTO lrs_routes VALUES(
  12,
  'result_geom_2',
  result_geom_2
);
INSERT INTO lrs_routes VALUES(
  13,
  'result_geom_3',
  result_geom_3
);

END;
/

-- First, display the data in the LRS table.
SELECT route_id, route_name, route_geometry FROM lrs_routes;
```

```
-- Are result_geom_1 and result_geom2 connected?
SELECT  SDO_LRS.CONNECTED_GEOM_SEGMENTS(a.route_geometry,
           b.route_geometry, 0.005)
  FROM lrs_routes a, lrs_routes b
  WHERE a.route_id = 11 AND b.route_id = 12;

-- Is the Route1 segment valid?
SELECT  SDO_LRS.VALID_GEOM_SEGMENT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- Is 50 a valid measure on Route1? (Should return FALSE; highest Route1 measure
is 27.)
SELECT  SDO_LRS.VALID_MEASURE(route_geometry, 50)
  FROM lrs_routes WHERE route_id = 1;

-- Is the Route1 segment defined?
SELECT  SDO_LRS.IS_GEOM_SEGMENT_DEFINED(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- How long is Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- What is the start measure of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- What is the end measure of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_END_MEASURE(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- What is the start point of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- What is the end point of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_END_PT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

-- Shift by 5 (for example, 5-mile segment added before original start)
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo, 0, 27, 5)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- "Convert" mile measures to kilometers (27 * 1.609 = 43.443)
```

```
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(route_geometry, 0, 43.443, 0)
  FROM lrs_routes WHERE route_id = 1;

-- Clip a piece of Route1.
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(route_geometry, 5, 10)
  FROM lrs_routes WHERE route_id = 1;

-- Point (9,3,NULL) is off the road; should return (9,4,9).
SELECT  SDO_LRS.PROJECT_PT(route_geometry,
  MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) )
  FROM lrs_routes WHERE route_id = 1;

-- Return the measure of the projected point.
SELECT  SDO_LRS.GET_MEASURE(
 SDO_LRS.PROJECT_PT(a.route_geometry, m.diminfo,
  MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) ),
 m.diminfo )
 FROM lrs_routes a, user_sdo_geom_metadata m
 WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Is point (9,3,NULL) a valid LRS point? (Should return TRUE.)
SELECT  SDO_LRS.VALID_LRS_PT(
  MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)),
  m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Locate the point on Route1 at measure 9, offset 0.
SELECT  SDO_LRS.LOCATE_PT(route_geometry, 9, 0)
  FROM lrs_routes WHERE route_id = 1;
```

Example 6–4 shows the output of the SELECT statements in Example 6–3.

Example 6–4 Simplified Example: Output of SELECT Statements

```
SQL> -- First, display the data in the LRS table.
SQL> SELECT route_id, route_name, route_geometry FROM lrs_routes;

  ROUTE_ID ROUTE_NAME
```

```
---------- --------------------------------
ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
         1 Route1
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))

        11 result_geom_1
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 5, 4, 5))

        12 result_geom_2

  ROUTE_ID ROUTE_NAME
---------- --------------------------------
ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 4, 5, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))

        13 result_geom_3
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 5, 4, 5, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27)
)

SQL>
SQL> -- Are result_geom_1 and result_geom2 connected?
SQL> SELECT  SDO_LRS.CONNECTED_GEOM_SEGMENTS(a.route_geometry,
  2  b.route_geometry, 0.005)
  3    FROM lrs_routes a, lrs_routes b
  4    WHERE a.route_id = 11 AND b.route_id = 12;

SDO_LRS.CONNECTED_GEOM_SEGMENTS(A.ROUTE_GEOMETRY,B.ROUTE_GEOMETRY,0.005)
--------------------------------------------------------------------------------
TRUE

SQL>
SQL> -- Is the Route1 segment valid?
SQL> SELECT  SDO_LRS.VALID_GEOM_SEGMENT(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.VALID_GEOM_SEGMENT(ROUTE_GEOMETRY)
--------------------------------------------------------------------------------
TRUE
```

```
SQL>
SQL> -- Is 50 a valid measure on Route1? (Should return FALSE; highest Route1
measure is 27.)
SQL> SELECT  SDO_LRS.VALID_MEASURE(route_geometry, 50)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.VALID_MEASURE(ROUTE_GEOMETRY,50)
--------------------------------------------------------------------------------
FALSE

SQL>
SQL> -- Is the Route1 segment defined?
SQL> SELECT  SDO_LRS.IS_GEOM_SEGMENT_DEFINED(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.IS_GEOM_SEGMENT_DEFINED(ROUTE_GEOMETRY)
--------------------------------------------------------------------------------
TRUE

SQL>
SQL> -- How long is Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_LENGTH(ROUTE_GEOMETRY)
-------------------------------------------
                                         27

SQL>
SQL> -- What is the start measure of Route1?
SQL> -- What is the start measure of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_START_MEASURE(ROUTE_GEOMETRY)
-------------------------------------------------
                                                0

SQL>
SQL> -- What is the end measure of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_END_MEASURE(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_END_MEASURE(ROUTE_GEOMETRY)
-----------------------------------------------
```

```
                                        27

SQL>
SQL> -- What is the start point of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_START_PT(ROUTE_GEOMETRY)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
2, 2, 0))

SQL>
SQL> -- What is the end point of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_END_PT(route_geometry)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_END_PT(ROUTE_GEOMETRY)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
5, 14, 27))

SQL>
SQL> -- Shift by 5 (for example, 5-mile segment added before original start)
SQL> SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo, 0, 27, 5)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
  4   AND a.route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,0,27,5)(SDO_GTYPE, SDO_SRI
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 5, 2, 4, 7, 8, 4, 13, 12, 4, 17, 12, 10, 23, 8, 10, 27, 5, 14, 32))

SQL>
SQL> -- "Convert" mile measures to kilometers (27 * 1.609 = 43.443)
SQL> SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(route_geometry, 0, 43.443, 0)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(ROUTE_GEOMETRY,0,43.443,0)(SDO_GTYPE, SDO_SRID, SDO_P
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 3.218, 8, 4, 12.872, 12, 4, 19.308, 12, 10, 28.962, 8, 10, 35.398
, 5, 14, 43.443))
```

```
SQL>
SQL> -- Clip a piece of Route1.
SQL> SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(route_geometry, 5, 10)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.CLIP_GEOM_SEGMENT(ROUTE_GEOMETRY,5,10)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 4, 5, 8, 4, 8, 10, 4, 10))

SQL>
SQL> -- Point (9,3,NULL) is off the road; should return (9,4,9).
SQL> SELECT  SDO_LRS.PROJECT_PT(route_geometry,
  2    MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
  3    MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
  4    MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) )
  5    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.PROJECT_PT(ROUTE_GEOMETRY,MDSYS.SDO_GEOMETRY(3301,NULL,NULL,MDSYS.SDO_EL
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
9, 4, 9))

SQL>
SQL> -- Return the measure of the projected point.
SQL> SELECT  SDO_LRS.GET_MEASURE(
  2   SDO_LRS.PROJECT_PT(a.route_geometry, m.diminfo,
  3    MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
  4    MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
  5    MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) ),
  6   m.diminfo )
  7   FROM lrs_routes a, user_sdo_geom_metadata m
  8   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
  9   AND a.route_id = 1;

SDO_LRS.GET_MEASURE(SDO_LRS.PROJECT_PT(A.ROUTE_GEOMETRY,M.DIMINFO,MDSYS.SDO_GEOM
--------------------------------------------------------------------------------
                                                                              9
SQL>
SQL> -- Is point (9,3,NULL) a valid LRS point? (Should return TRUE.)
SQL> SELECT  SDO_LRS.VALID_LRS_PT(
  2    MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
  3    MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
  4    MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)),
  5    m.diminfo)
```

```
  6    FROM lrs_routes a, user_sdo_geom_metadata m
  7    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
  8   AND a.route_id = 1;

SDO_LRS.VALID_LRS_PT(MDSYS.SDO_GEOMETRY(3301,NULL,NULL,MDSYS.SDO_ELEM_INFO_ARRAY
--------------------------------------------------------------------------------
TRUE

SQL>
SQL> -- Locate the point on Route1 at measure 9, offset 0.
SQL> SELECT  SDO_LRS.LOCATE_PT(route_geometry, 9, 0)
  2    FROM lrs_routes WHERE route_id = 1;

SDO_LRS.LOCATE_PT(ROUTE_GEOMETRY,9,0)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), S
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
9, 4, 9))
```

7

Generic Geocoding Interface

This chapter describes a generic interface to third-party geocoding software that lets users geocode their address information stored in database tables and obtain standardized addresses and corresponding location information as instances of predefined object types. This interface is part of the geocoding framework in the Oracle Spatial and Oracle *inter*Media Locator products.

A geocoding service is used for converting tables of address data into standardized address, location, and possibly other data. Given a geocoded address, one can then perform proximity or location queries using a spatial engine, such as Oracle Spatial or Oracle *inter*Media Locator, or demographic analysis using tools and data from Oracle's business partners.

Once data has been geocoded, users can perform location queries on this data. In addition, geocoded data can be used with other spatial data such as block group, postal code, and county code for association with demographic information. It is now possible for decision support, customer relationship management, supply chain analysis, and other applications to use spatial analyses as part of their information gathering and processing functions. Results of analyses or queries can be presented as maps, in addition to tabular formats, using third-party software integrated with Oracle *inter*Media Locator.

This chapter describes a set of interfaces and metadata schema that enables geocoding of an entire address table or a single row. It also describes the procedures for inserting or updating standardized address and spatial point data into another table (or the same table). The third-party geocoding service is assumed to have been installed on a local network and to be accessible through standard communication protocols, such as sockets or HTTP.

7.1 Locator Implementation: Benefits and Limitations

Oracle *inter*Media Locator contains a set of application programming interface (API) functions that allows the integration of Oracle Spatial with third-party geocoding products and Web-based geocoding services. A database user can issue a standard SQL call or construct PL/SQL routines to geocode an address, and retrieve the spatial and standardized address objects, both of which are defined as Oracle database object types. Users have the option of storing these in the database, or using the spatial objects in Locator functions for Euclidean within-distance queries.

The APIs offer great flexibility in extracting information from existing relational databases. Data conversion procedures are minimal. A geocode result also returns an additional set of information; there is no requirement to use all the information, and the application can decide which fields to extract and where to store them. However, to use the full range of features of Oracle Spatial or Oracle *inter*Media Locator, it is recommended that the Spatial object be stored as returned.

The existing Locator service is Web-based and requests are formatted in HTTP. Thus, each request in SQL must contain the URL of the Web site, proxy for the firewall (if any), and user account information on the service provider's Web site. An HTTP approach potentially limits the utility or practicality of the service when dealing with large tables or undertaking frequent updates to the base address information. In such situations, use a batch geocoding service made available within an intranet or local area network. The following sections describe the interface for a facility that can include the existing HTTP-based solution.

7.2 Generic Geocoding Client

A fast, scalable, highly available, and secure Java Virtual Machine (Java VM, or JVM) is integrated in the Oracle9*i* database server. The Java VM provides an ideal platform on which to deploy enterprise applications written in Java as Java Stored Procedures (JSPs), Enterprise Java Beans (EJBs), or Java Methods of Oracle9*i* object types.

Therefore, any client geocoder component written in Java can be embedded in the Oracle9*i* database as a JSP. This JSP interface can perform either one-record-at-a-time or batch geocoding. Java stored procedures are published using PL/SQL interfaces; thus, the generic geocoding interface can be compatible with existing Locator APIs.

The stored procedures have an interface, oracle.spatial.geocoder, that must be implemented by each vendor whose geocoder is integrated with Oracle Spatial and Oracle *inter*Media Locator. The procedures also require certain object types to be

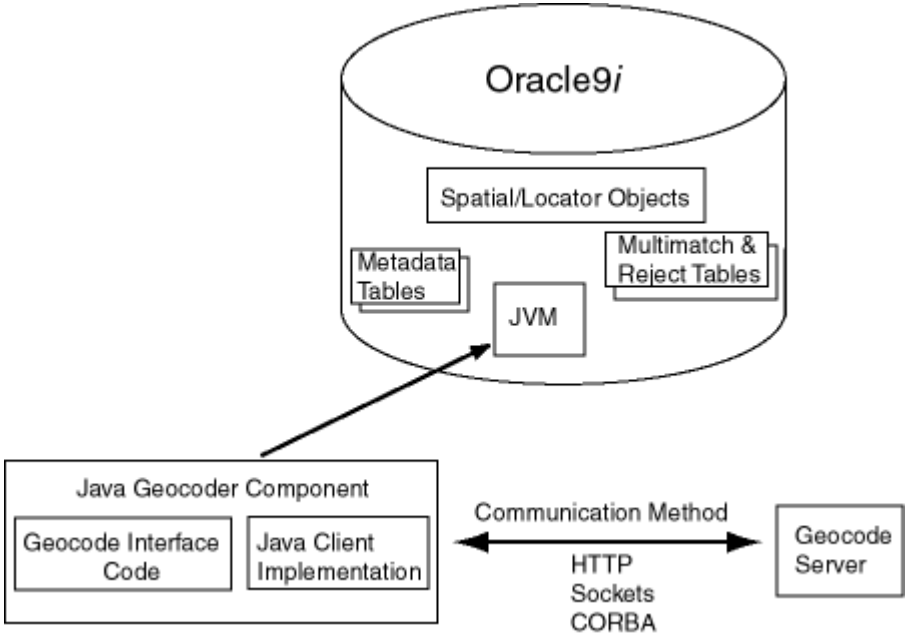
defined and metadata tables to be populated. The object types, metadata schema, and the geocoder interface are described in further detail in the following sections.

Although the database user MDSYS oversees all data types, operators, and schema objects for Oracle Spatial and Oracle *inter*Media Locator, the geocoding metadata must exist in each user's schema. Each user of the geocoder service must have tables that implement the metadata schema.

The third-party geocoding service must be installed on a local network and be accessible through standard communication protocols such as sockets, HTTP, or CORBA.

Figure 7–1 shows the Oracle geocoding framework.

Figure 7–1 Oracle Geocoding Framework



7.3 Geocoder Metadata

The metadata describes the properties of the geocoding server, the location and structure of the address data to be geocoded, and the nature and storage location of

the geocoding results. Other relevant information may include the name of the server machine, the port to which to connect, and so on. Together, these constitute the initialization parameters and are stored in metadata tables under the user's own schema. At client initialization, a data dictionary lookup is performed to locate the necessary metadata.

Batch geocoding lets the user simultaneously geocode many records from one table. Batch geocoding requires the following:

- Geocoding server setup, instructing the client where and how to connect to the geocoding service.
- Associating input fields and output fields with columns in the database tables. This is called the schema setup.
- Specifying how to handle geocoding situations such as rejects, multiple matches, or exceptions.

Thus, the metadata table consists of a task ID, geocoding information, and schema information. The task ID is a primary key that identifies the initialization parameters for a particular geocoding task. For example, geocoding a table of customers is one task, while geocoding a table of customer inquiries is a separate task.

The metadata is stored in a table named GEOCODE_TASK_METADATA, which is defined as follows:

```
Create table geocode_task_metadata (
  task_id  NUMBER,  -- primary key
  geocoder_info  MYSYS.GEOCODE_SERVER_PROPERTY_TYPE,
  schema_info  MDSYS.GEOCODE_SCHEMA_PROPERTY_TYPE
);
```

Note the following about the GEOCODE_TASK_METADATA table:

- The metadata is divided into a server object (described in Section 7.3.1) and a schema object (described in Section 7.3.2).
- Each object is identified by a unique *task_id* value.

7.3.1 Server Properties

The GEOCODER_INFO property column of the GEOCODE_TASK_METADATA table contains information describing the characteristics of the server, including machines, ports, and vendor-specific information.

The GEOCODE_SERVER_PROPERTY_TYPE type is defined as follows:

```
create type geocode_value_array as
  varray(1024) of varchar2(64)
/
create type geocode_server_property_type as object
(
  servers  geocode_value_array,
  protocol  varchar2(32),
  property_name  geocode_value_array,
  property_value  geocode_value_array,
  reject_level  integer,
  batch_size  integer
)
/
```

Note the following about the GEOCODE_SERVER_PROPERTY_TYPE definition:

- SERVERS is an array of character strings each in the form *Machine:Port* that uniquely identifies the geocoding service on the network. This also supports multiple services on the same network by providing an array of servers. Some geocoders, for example, can switch to secondary servers in the case of failures.
- PROTOCOL allows different transport mechanisms, such as HTTP or socket.
- Additional PROPERTY_NAME and PROPERTY_VALUE arrays allow customization for unique geocoder processing options. They are not intended to be used for name or password information, because a local geocoding service usually does not require this information.
- REJECT_LEVEL is a vendor-specific value that defines the criteria for rejecting a record. It is up to the implementation of the Java interface to interpret the value.
- BATCH_SIZE indicates how many records to send to the geocoder at one time.

7.3.2 Geocoding Input and Output Specification

The SCHEMA_INFO property column of the GEOCODE_TASK_METADATA table specifies the set of columns that makes up an address in the table to be geocoded, the table and columns into which the geocoded results are stored, and where rejected record data and multiple matches are stored.

The GEOCODE_SCHEMA_PROPERTY_TYPE type uses columns of type GEOCODE_TABLE_COLUMN_TYPE to describe the address fields in the input (table to be geocoded) and output (table containing geocoded results). The two types are defined as follows:

```
create type geocode_table_column_type as object
```

```
(
  firm  varchar2(32),
  street  varchar2(32),
  street2  varchar2(32),
  cty_subdivision  varchar2(32),
  city  varchar2(2332),
  country_subdivision  varchar2(32),  --state
  country  varchar2(32),
  postal_code  varchar2(32),
  postal_addon_code  varchar2(32),
  lastline  varchar2(32),
  col_name  geocode_value_array,
  col_value  geocode_value_array
)
/

create type geocode_schema_property_type as object
(
  language  varchar2(32),
  character_set  varchar2(32),
  in_table  varchar2(32),
  in_table_cols  geocode_table_column_type,
  out_table  varchar2(32),
  out_table_cols  geocode_table_column_type,
  out_sdo_geom  varchar2(32),
  out_geo_result  varchar2(32),
  in_primary_key  varchar2(32),
  out_foreign_key  varchar2(32),
  DML_option  varchar2(16),
  multi_match_table  varchar2(32),
  reject_table  varchar2(32),
  batch_commit  varchar2(5)
)
/
```

Note the following about the GEOCODE_TABLE_COLUMN_TYPE and GEOCODE_SCHEMA_PROPERTY_TYPE definitions:

- LANGUAGE and CHARACTER_SET are for internationalization.
- IN_TABLE identifies the name of the input address table (for example, CUSTOMERS).
- IN_TABLE_COLS identifies the standard set of fields for geocoding. The fields in the object are standard, and LASTLINE is redundant with the combination of CITY, STATE, POSTAL_CODE, and POSTAL_ADDON_CODE. Only one

(LASTLINE, or the combination of CITY, STATE, POSTAL_CODE, and POSTAL_ADDON_CODE) should be specified.

- OUT_TABLE and OUT_TABLE_COLS have the same meaning as IN_TABLE and IN_TABLE_COLS, except that these are the column names where the results are stored. Either a subset or all the OUT_TABLE_COLS fields can be null. OUT_TABLE_COLS and GEOCODE_RESULT contain similar information, that is, the standardized (corrected) address in case of successful geocoding. Users can choose to store the standardized address in two forms, expanded into a set of columns or as a single object.
- If the actual address definition differs from the fields in the GEOCODE_TABLE_COLUMN_TYPE definition, adjust the field mappings and insert null values as needed. For example, assume an input table CUSTOMERS defined as follows:

```
(custname  varchar2(32),
 company  varchar2(32),
 street  varchar2(64),
 city  varchar2(32),
 state  varchar2(32),
 country  varchar2(32),
 zip  varchar2(9))
```

  In the GEOCODE_SCHEMA_PROPERTY_TYPE column definition, the IN_TABLE_COLS attribute value would be specified as: GEOCODE_TABLE_COLUMN_TYPE('COMPANY', 'STREET', NULL, NULL, 'CITY', 'STATE', 'COUNTRY', 'ZIP', NULL, NULL, NULL, NULL).

  The COL_NAME and COL_VALUE information will be used for feature enhancement for individual geocoding services.

- OUT_SDO_GEOM and OUT_GEO_RESULT: SDO_GEOMETRY and GEOCODE_RESULT are the two database objects for storing a standard set of geocoded results, including standardized address and latitude/longitude information. If you are using Oracle Spatial, it is required that SDO_GEOMETRY objects be stored in the database. MDSYS.GEOCODE_RESULT exists in the current Locator implementation and is defined as follows:

```
Create type geocode_result as object  (
  matchcode  varchar2(16),
  firmname  varchar2(512),
  addrline  varchar2(512),
  addrline2  varchar2(512),
  city  varchar2(512),
```

```
  state  varchar2(512),
  zip  varchar2(5),
  zip4  varchar2(4),
  lastline  varchar2(512),
  county  varchar2(32),
  block  varchar2(32),
  loccode  varchar2(16),
  cart  varchar2(16),
  dpbc  varchar2(16),
  lotcode  vrchar2(16),
  lotnum  varchar2(16)
);
```

- IN_PRIMARY_KEY and OUT_FOREIGN_KEY designate a primary key and foreign key, respectively. Using a primary key and foreign key pair is a way to associate the input records to the output records, and is essential when the database stores the output results. Even if the input table and output table are the same, a primary key and foreign key pair (essentially the same column: for example, ID or ROWID) must be specified. There is no restriction on the data type, because no manipulation of the data is needed.
- DML_OPTION specifies whether to insert geocoded data into a new row in the result table (INSERT) or update existing rows in the table (UPDATE). If IN_TABLE is the same as OUT_TABLE, then DML_OPTION must be UPDATE, because adding new rows in an existing table is unnecessary. If IN_TABLE is different from OUT_TABLE and if UPDATE is specified, OUT_TABLE must have partial records available for primary and foreign key lookup. This permits the service to locate the exact row to update with the new objects.
- MULTI_MATCH_TABLE and REJECT_TABLE are table names where the primary key of the multiple matches and rejected records are stored. If these tables do not exist, they will be created automatically. Automatic creation is the preferred approach due to the fixed structure. The REJECT_TABLE table will be created with a primary key column type in the input table, a match code column, and an optional error message column. The MULTI_MATCH_TABLE table will contain a primary key, SDO_GEOMETRY, and GEO_RESULT. If these fields are null, no table will be created and no multiple matches will be returned.
- BATCH_COMMIT is a string containing TRUE or FALSE, indicating if a commit operation should be performed after each batch. If FALSE is specified, a large rollback segment will be needed for large address table geocoding.

7.3.2.1 Multiple Matches and Rejected Records

Tables can be specified to store multiple matches (MULTI_MATCH_TABLE) and rejected records (REJECT_TABLE) during batch geocoding. The primary key will be a user-specified field from the original table. Hence, any single column can be used. Currently, no composite primary keys are supported.

If a single address results in multiple matches, after the batch processing you can examine MULTI_MATCH_TABLE and select the correct entries for the original data rows. For example, you can create a table in the following format:

```
create table <user-defined multimatch table> (
  pk  <same data type as in input table>,
  location  mdsys.sdo_geometry,
  std_addr  mdsys.geocode_result
);
```

The match code in the geocode result object indicates the failure during geocoding. The rejection level is used in determining if a record has failed the geocoding. If a record has failed and REJECT_TABLE is defined, the primary key (specified by the user) is inserted into a rejection table. The interpretation of rejection level is left to the programmer. REJECT_TABLE can be defined in the following format:

```
create table <user-defined reject table> (
  pk  <same data type as in input table>,
  matchcode  varchar2(64),
  errcode  varchar2(128)
);
```

7.4 Metadata Helper Class

The geocoder metadata is comprehensive. To accelerate development and deployment, Oracle offers a sample class, oracle.spatial.geocoder.Metadata, to allow easy access (read and write) to these objects. Also, SELECT and INSERT SQL statements are constructed automatically for the caller. See the class implementation code for details.

7.5 Single-Record and Interactive Geocoding

Geocoding a row in a table is required when updating or inserting data in the address table. One way to maintain consistency between the base address table and the table of geocoded results is to use a trigger to call the geocoding function. The Java interface method geocode1( ) will take the primary key to perform the

geocoding task and insert or update the geocoded information into the specified table.

The GEOCODER_HTTP package functions are still supported for single-record geocoding. In addition, you are able to pass an address in as a parameter, and get back an array of matches. The Java interface takes a metadata structure (see the GEOCODE_SCHEMA_PROPERTY_TYPE definition in Section 7.3.2) and an address structure, and returns an array of this same address structure:

```
create type geocode_record_type as object
(
  firm  varchar2(40),
  street  varchar2(40),
  street2  varchar2(40),
  city_subdivision  varchar2(40),
  city  varchar2(40),
  country_subdivision  varchar2(40),
  country  varchar2(40),
  postal_code  varchar2(40),
  postal_addon_code  varchar2(40),
  lastline  varchar2(80),
  latitude  number,
  longitude  number
);
```

After performing geocoding, it will return an array (SQL collection type) of such structures as possible matches. With this method, no database table or schema is accessed. This method can enable interactive applications such as store locators.

7.6 Java Geocoder Service Interface

Each geocoder independent software vendor (ISV) must implement the following geocoder interface to integrate their products with Oracle Spatial and Oracle *inter*Media Locator.

The interface is defined as follows:

```
// Geocoder Interface
package oracle.spatial.geocoder;

public interface GeocoderInterface {
  public void geocode(int taskId)
    throws oracle.spatial.geocoder.GeocoderException, java.sql.SQLException;
  public void geocode1(int taskId, BigDecimal pkVal)
   throws oracle.spatial.geocoder.GeocoderException, java.sql.SQLException;
```

```
// … other geocode1 functions with different pkVal types

  public ARRAY interactive_geocode(STRUCT meta, STRUCT inAddr)
    throws oracle.spatial.geocoder.GeocoderException, java.sql.SQLException;
}

// Geocoder Exception Class
package oracle.spatial.geocoder;

public class GeocoderException extends java.lang.Exception {
  public GeocoderException() {}
  public GeocoderException(String mesg)
  {
    super(mesg);
  }
}
```

Further details, including some of the actual implementation, will be provided to developers.

7.7 Enabling Third-Party Geocoders

For customers to implement an Oracle solution with any vendor's Java client, they will have to download a copy of the Java client from the geocoder vendor's Web site, link the geocoder interface package with the vendor's code, and then upload the resulting JSP into the Oracle JVM. Once enabled, the Java client resides on the vendor's server and can provide the required services.

To load a client into the database, invoke the Oracle9*i* loadjava utility, and the Java geocoding method will be exposed as a SQL function call.

The vendor-specific geocoder interface implementation can be owned by any schema, such as MDSYS, a DBA account, or an account determined by the customer or vendor. The owner must grant the appropriate EXECUTE privileges to PUBLIC or some set of users of the service.

8

Extending Spatial Indexing Capabilities

This chapter shows how to create and use spatial indexes on objects other than a geometry column. In other chapters, the focus is on indexing and querying spatial data that is stored in a single column of type SDO_GEOMETRY. This chapter shows how to:

- Embed an SDO_GEOMETRY object in a user-defined object type, and index the geometry attribute of that type
- Create an use a function-based index where the function returns an SDO_ GEOMETRY object

The techniques in this chapter are intended for experienced and knowledgeable application developers. You should be familiar with the Spatial concepts and techniques described in other chapters. You should also be familiar with, or able to learn about, relevant Oracle database features, such as user-defined data types and functional indexing.

8.1 SDO_GEOMETRY Objects in User-Defined Type Definitions

The SDO_GEOMETRY type can be embedded in a user-defined data type definition. The procedure is very similar to that for using the SDO_GEOMETRY type for a spatial data column:

1. Create the user-defined data type.
2. Create a table with a column based on that data type.
3. Insert data into the table.
4. Update the USER_SDO_GEOM_METADATA view.
5. Create the spatial index on the geometry attribute.

6. Perform queries on the data.

For example, assume that you wanted to follow the cola markets scenario in the simplified example in Section 2.1, but wanted to incorporate the market name attribute and the geometry attribute in a single type. First, create the user-defined data type, as in the following example that creates an object type named MARKET_TYPE:

```
CREATE OR REPLACE TYPE market_type AS OBJECT
  (name VARCHAR2(32), shape MDSYS.SDO_GEOMETRY);
/
```

Create a table that includes a column based on the user-defined type. The following example creates a table named COLA_MARKETS_2 that will contain the same information as the COLA_MARKETS table used in the example in Section 2.1.

```
CREATE TABLE cola_markets_2 (
  mkt_id NUMBER PRIMARY KEY,
  market MARKET_TYPE);
```

Insert data into the table, using the object type name as a constructor. For example:

```
INSERT INTO cola_markets_2 VALUES(
  1,
  MARKET_TYPE('cola_a',
    MDSYS.SDO_GEOMETRY(
      2003,  -- 2-dimensional polygon
      NULL,
      NULL,
      MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3), -- one rectangle (1003 = exterior)
      MDSYS.SDO_ORDINATE_ARRAY(1,1, 5,7) -- only 2 points needed to
            -- define rectangle (lower left and upper right)
      )
  )
);
```

Update the USER_SDO_GEOM_METADATA view, using dot-notation to specify the column name and spatial attribute. The following example specifies MARKET.SHAPE as the COLUMN_NAME (explained in Section 2.4.2) in the metadata view.

```
INSERT INTO USER_SDO_GEOM_METADATA
  VALUES (
  'cola_markets_2',
  'market.shape',
  MDSYS.SDO_DIM_ARRAY(   -- 20X20 grid
```

```
    MDSYS.SDO_DIM_ELEMENT('X', 0, 20, 0.005),
    MDSYS.SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
     ),
  NULL   -- SRID
);
```

Create the spatial index, specifying the column name and spatial attribute using dot-notation. For example.

```
CREATE INDEX cola_spatial_idx_2
ON cola_markets_2(market.shape)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Perform queries on the data, using dot-notation to refer to attributes of the user-defined type. The following simple query returns information associated with the cola market named *cola_a*.

```
SELECT c.mkt_id, c.market.name, c.market.shape
  FROM cola_markets_2 c
  WHERE c.market.name = 'cola_a';
```

The following query returns information associated with all geometries that have any spatial interaction with a specified query window, namely, the rectangle with lower-left coordinates (4,6) and upper-right coordinates (8,8).

```
SELECT c.mkt_id, c.market.name, c.market.shape
  FROM cola_markets_2 c
  WHERE SDO_RELATE(c.market.shape,
            MDSYS.SDO_GEOMETRY(2003, NULL, NULL,
              MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
              MDSYS.SDO_ORDINATE_ARRAY(4,6, 8,8)),
            'mask=anyinteract querytype=window' = 'TRUE';
```

8.2 SDO_GEOMETRY Objects in Function-Based Indexes

A function-based spatial index facilitates queries that use locational information (of type SDO_GEOMETRY) returned by a function or expression. In this case, the spatial index is created based on the precomputed values returned by the function or expression.

If you are not already familiar with function-based indexes, see the following for detailed explanations of the their benefits, options, and requirements, as well as usage examples:

- *Oracle9i Application Developer's Guide - Fundamentals*

- *Oracle9i Database Administrator's Guide*

See especially the information in those documents about requirements and restrictions related to function-based indexes. For example, you must grant Spatial application users the QUERY REWRITE privilege, and you must have the initialization parameters COMPATIBLE set to 8.1.0.0.0 or higher, QUERY_REWRITE_ENABLED=TRUE, and QUERY_REWRITE_INTEGRITY=TRUSTED.

The procedure for using an SDO_GEOMETRY object in a function-based index is as follows:

1. Create the function that returns an SDO_GEOMETRY object.

   The function must be declared as DETERMINISTIC.

2. If the spatial data table does not already exist, create it, and insert data into the table.
3. Update the USER_SDO_GEOM_METADATA view.
4. Create the spatial index.

   For a function-based spatial index, the number of parameters must not exceed 32.

5. Perform queries on the data.

The rest of this section describes two example of using function-based indexes. In both examples, a function is created that returns an SDO_GEOMETRY object, and a spatial index is created on that function. In the first example, the input parameters to the function are a standard Oracle data type (NUMBER). In the second example, the input to the function is a user-defined object type.

8.2.1 Example: Function with Standard Types

In the following example, the input parameters to the function used for the function-based index are standard numeric values (longitude and latitude).

Assume that you wanted to create a function that returns the longitude and latitude of a point and to use that function in a spatial index. First, create the function, as in the following example that creates a function named GET_LONG_LAT_PT:

```
-- Create a function to return a point geometry (SDO_GTYPE = 2001) with
-- input of 2 numbers: longitude and latitude (SDO_SRID = 8307, for
-- "Longitude / Latitude (WGS 84)",  probably the most widely used
--  coordinate system, and the one used for GPS devices.
-- Specify DETERMINISTIC for the function.
```

```
create or replace function get_long_lat_pt(longitude in number,
                                           latitude in number)
return MDSYS.SDO_GEOMETRY deterministic is
begin
     return mdsys.sdo_geometry(2001, 8307,
               mdsys.sdo_point_type(longitude, latitude, NULL),NULL, NULL);
end;
/
```

If the spatial data table does not already exist, create the table and add data to it, as in the following example that creates a table named LONG_LAT_TABLE:

```
create table LONG_LAT_TABLE
(longitude number, latitude number, name varchar2(32));

insert into LONG_LAT_TABLE values (10,10, 'Place1');
insert into LONG_LAT_TABLE values (20,20, 'Place2');
insert into LONG_LAT_TABLE values (30,30, 'Place3');
```

Update the USER_SDO_GEOM_METADATA view, using dot-notation to specify the schema name and function name. The following example specifies SCOTT.GET_LONG_LAT_PT(LONGITUDE,LATITUDE) as the COLUMN_NAME (explained in Section 2.4.2) in the metadata view.

```
-- Set up the metadata entry for this table.
-- note that the column name sets up the function on top
-- of the two columns used in this function,
-- along with the owner of the function.
insert into user_sdo_geom_metadata values('LONG_LAT_TABLE',
 'scott.get_long_lat_pt(longitude,latitude)',
 mdsys.sdo_dim_array(
   mdsys.sdo_dim_element('Longitude', -180, 180, 0.005),
   mdsys.sdo_dim_element('Latitude', -90, 90, 0.005)), 8307);
```

Create the spatial index, specifying the function name with parameters. For example, creating an R-tree index:

```
create index LONG_LAT_TABLE_IDX on
   LONG_LAT_TABLE(get_long_lat_pt(longitude,latitude))
   indextype is mdsys.spatial_index;
```

Perform queries on the data. In the following example, the two queries accomplish the same thing; however, the first query does not use a user-defined function

(instead using a constructor to specify the point), whereas the second query uses the function to specify the point.

```
-- First query: call sdo_filter with an SDO_GEOMETRY constructor
select name from LONG_LAT_TABLE  a
   where sdo_filter(get_long_lat_pt(a.longitude,a.latitude),
      mdsys.sdo_geometry(2001, 8307,
        mdsys.sdo_point_type(10,10,NULL), NULL, NULL),
      'querytype=WINDOW')='TRUE';

-- Second query: call sdo_filter with the function that returns an sdo_geometry
select name from LONG_LAT_TABLE  a
   where sdo_filter(get_long_lat_pt(a.longitude,a.latitude),
     get_long_lat_pt(10,10),
     'querytype=WINDOW')='TRUE';
```

8.2.2 Example: Function with User-Defined Object Type

In the following example, the input parameter to the function used for the function-based index is an object of a user-defined type that includes the longitude and latitude.

Assume that you wanted to create a function that returns the longitude and latitude of a point and to create a spatial index on that function. First, create the user-defined data type, as in the following example that creates an object type named LONG_LAT and its member function GetGeometry:

```
create type long_lat as object (
   longitude number,
   latitude number,
member function GetGeometry(SELF in long_lat)
RETURN MDSYS.SDO_GEOMETRY DETERMINISTIC)
/

create or replace type body long_lat as
  member function GetGeometry(self in long_lat)
  return MDSYS.SDO_GEOMETRY is
    begin
       return mdsys.sdo_geometry(2001, 8307,
           mdsys.sdo_point_type(longitude, latitude, NULL), NULL,NULL);
    end;
end;
/
```

If the spatial data table does not already exist, create the table and add data to it, as in the following example that creates a table named TEST_LONG_LAT:

```
create table test_long_lat
   (location long_lat, name varchar2(32));

insert into test_long_lat values (long_lat(10,10), 'Place1');
insert into test_long_lat values (long_lat(20,20), 'Place2');
insert into test_long_lat values (long_lat(30,30), 'Place3');
```

Update the USER_SDO_GEOM_METADATA view, using dot-notation to specify the schema name, table name, and function name and parameter value. The following example specifies SCOTT.LONG_LAT.GetGeometry(LOCATION) as the COLUMN_NAME (explained in Section 2.4.2) in the metadata view.

```
insert into user_sdo_geom_metadata values('test_long_lat',
 'scott.long_lat.GetGeometry(location)',
 mdsys.sdo_dim_array(
   mdsys.sdo_dim_element('Longitude', -180, 180, 0.005),
   mdsys.sdo_dim_element('Latitude', -90, 90, 0.005)), 8307);
```

Create the spatial index, specifying the column name and function name using dot-notation. For example:

```
create index test_long_lat_idx on test_long_lat(location.GetGeometry())
  indextype is mdsys.spatial_index;
```

Perform queries on the data. The following performs a primary filter operation, asking for the names of geometries that are likely to interact spatially with point (10,10).

```
SELECT a.name FROM test_long_lat a
  WHERE SDO_FILTER(a.location.GetGeometry(),
            MDSYS.SDO_GEOMETRY(2001, 8307,
                MDSYS.SDO_POINT_TYPE(10,10,NULL), NULL, NULL),
            'querytype=window') = 'TRUE';
```

Part II

Reference Information

This document has two main parts:

- Part I provides conceptual and usage information about Oracle Spatial.
- Part II provides reference information about Oracle Spatial methods, operators, functions, and procedures.

Appendixes with supplementary information follow Part II.

Part II contains the following chapters with reference information:

9

SQL Statements for Indexing

This chapter describes the statements used when working with the spatial object data type. The statements are listed in Table 9–1.

Table 9–1 Spatial Index Creation and Usage Statements

Statement	Description
ALTER INDEX	Alters a spatial index on a column of type MDSYS.SDO_GEOMETRY.
ALTER INDEX REBUILD	Rebuilds a spatial index on a column of type MDSYS.SDO_GEOMETRY.
ALTER INDEX RENAME TO	Changes the name of a spatial index on a column of type MDSYS.SDO_GEOMETRY.
CREATE INDEX	Creates a spatial index on a column of type MDSYS.SDO_GEOMETRY.
DROP INDEX	Deletes a spatial index on a column of type MDSYS.SDO_GEOMETRY

This chapter focuses on using these SQL statements with spatial indexes. For complete reference information about any statement, see the *Oracle9i SQL Reference*.

ALTER INDEX

Purpose

Alters specific parameters for a spatial index or rebuilds a spatial index.

Syntax

ALTER INDEX [schema.]index PARAMETERS ('index_params [physical_storage_params]');

Keywords and Parameters

INDEX_PARAMS Allows you to change the characteristics of the spatial index, and the type (fixed or hybrid) of a quadtree index.

Keyword	Description
add_index	Specifies the name of the new index table to add. Data type is VARCHAR2.
delete_index	Specifies the name of the index table to delete. You can only delete index tables that were created with the ALTER INDEX add_index statement. The primary index table cannot be deleted with this parameter. To delete the primary index table, use the DROP INDEX statement. Data type is VARCHAR2.
sdo_commit_ interval	Specifies the number of underlying table rows that are processed between commit intervals for the index data. (Quadtree indexes only.) The default behavior commits the index data only after all rows in the underlying table have been processed. See the Usage Notes for further details. Data type is NUMBER.
sdo_indx_dims	Specifies the number of dimensions to be indexed. (R-tree indexes only.) For example, a value of 2 causes the first 2 dimensions to be indexed. Must be less than or equal to the number of actual dimensions (number of SDO_DIM_ELEMENT instances in the dimensional array that describes the geometry objects in the column). Data type is NUMBER. Default = number of actual dimensions.
sdo_level	Specifies the desired fixed-size tiling level. (Quadtree indexes only.) Data type is NUMBER.
sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object. (Quadtree indexes only.) Data type is NUMBER.

sdo_rtr_pctfree	Specifies the minimum percentage of slots in each index tree node to be left empty when the index is created. Slots that are left empty can be filled later when new data is inserted into the table. (R-tree indexes only.) The value can range from 0 to 50. The default value is best for most applications; however, a value of 0 is recommended if no updates will be performed to the geometry column. Data type is NUMBER. Default = 10.
PHYSICAL_ STORAGE_ PARAMS	Determines the storage parameters used for altering the spatial index data table. A spatial index data table is a standard Oracle table with a prescribed format. Not all physical storage parameters that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

Keyword	Description
tablespace	Specifies the tablespace in which the index data table is created. This parameter is the same as TABLESPACE in the STORAGE clause of a CREATE TABLE statement.
initial	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
next	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
minextents	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
maxextents	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
btree_initial	Is the same as INITIAL in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_next	Is the same as NEXT in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)

Prerequisites

- You must have EXECUTE privileges on the index type and its implementation type.
- The spatial index to be altered is not marked in-progress.

Usage Notes

This statement is used to change the parameters of an existing index. This is the only way you can add or build multiple indexes on the same column.

See the Usage Notes for the CREATE INDEX statement for usage information about many of the available parameters.

Examples

The following example adds a new index table named FIXED_INDEX$ to the index named QTREE.

```
ALTER INDEX qtree PARAMETERS ('add_index=fixed_index$
                              sdo_level=8
                              initial=100M
                              next=1M
                              pctincrease=0
                              btree_initial=5M
                              btree_next=1M
                              btree_pctincrease=0');
```

The following example modifies the tablespace and the SDO_LEVEL value for partition IP2 of the spatial index named BGI.

```
ALTER INDEX bgi MODIFY PARTITION ip2
   PAREMETERS ('tablespace=SYSTEM sdo_level=4');
```

Related Topics

- ALTER INDEX REBUILD
- ALTER INDEX RENAME TO
- CREATE INDEX
- ALTER TABLE (clauses for partition maintenance) in the *Oracle9i SQL Reference*

ALTER INDEX REBUILD

Syntax

```
ALTER INDEX [schema.]index REBUILD
  [PARAMETERS ('rebuild_params [physical_storage_params]' ) ] ;

ALTER INDEX [schema.]index REBUILD PARTITION partition
  [PARAMETERS ('rebuild_params [physical_storage_params]' ) ] ;
```

Purpose

Rebuilds a spatial index or a specified partition of a partitioned index.

Keywords and Parameters

REBUILD_PARAMS Specifies in a command string the index parameters to use in rebuilding the spatial index.

Keyword	Description
layer_gtype	Checks to ensure that all geometries are of a specified geometry type. The value must be from the Geometry Type column of Table 2–1 in Section 2.2.1. In addition, for a quadtree index specifying POINT, allows for optimized processing of point data. Data type is VARCHAR2.
rebuild_index	Specifies the name of the spatial index table to be rebuilt. Data type is VARCHAR2.
sdo_commit_interval	Specifies the number of underlying table rows that are processed between commit intervals for the index data. (Quadtree indexes only.) The default behavior commits the index data only after all rows in the underlying table have been processed. See the Usage Notes for further details. Data type is NUMBER.
sdo_indx_dims	Specifies the number of dimensions to be indexed. (R-tree indexes only.) For example, a value of 2 causes the first 2 dimensions to be indexed. Must be less than or equal to the number of actual dimensions (number of SDO_DIM_ELEMENT instances in the dimensional array that describes the geometry objects in the column). Data type is NUMBER. Default = number of actual dimensions.
sdo_level	Specifies the desired fixed-size tiling level. (Quadtree indexes only.) Data type is NUMBER.

sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object. (Quadtree indexes only.) Data type is NUMBER.
sdo_rtr_pctfree	Specifies the minimum percentage of slots in each index tree node to be left empty when the index is created. Slots that are left empty can be filled later when new data is inserted into the table. (R-tree indexes only.) The value can range from 0 to 50. Data type is NUMBER. Default = 10.
PHYSICAL_ STORAGE_ PARAMS	Determines the storage parameters used for rebuilding the spatial index data table. A spatial index data table is a regular Oracle table with a prescribed format. Not all physical storage parameters that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

Keyword	Description
tablespace	Specifies the tablespace in which the index data table is created. Same as TABLESPACE in the STORAGE clause of a CREATE TABLE statement.
initial	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
next	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
minextents	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
maxextents	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
btree_initial	Is the same as INITIAL in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_next	Is the same as NEXT in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)

Prerequisites

- You must have EXECUTE privileges on the index type and its implementation type.
- The spatial index to be altered is not marked in-progress.

Usage Notes

An ALTER INDEX REBUILD 'rebuild_params' statement rebuilds the index using supplied parameters. Spatial index creation involves creating and inserting index data, for each row in the underlying table column being spatially indexed, into a table with a prescribed format. The default, or normal, operation is that all rows in the underlying table are processed before the insertion of index data is committed. This requires adequate rollback segment space.

You may choose to commit index data after every *n* rows of the underlying table have been processed. This is done by specifying SDO_COMMIT_INTERVAL = *n*. The potential complication is that, if there is an error during index rebuild and if periodic commit operations have taken place, then the spatial index will be in an inconsistent state. The only recovery option is to use DROP INDEX (possibly with the FORCE option) and CREATE INDEX statements after ensuring that the various tablespaces are the required size and any other error conditions have been removed.

This statement does not use any previous parameters from the index creation. All parameters should be specified for the index you want to rebuild.

For more information about using the *layer_gtype* keyword to constrain data in a layer to a geometry type, see Section 4.1.4.

With a partitioned spatial index, you must use a separate ALTER INDEX REBUILD statement for each partition to be rebuilt.

See also the Usage Notes for the CREATE INDEX statement for usage information about many of the available parameters.

Examples

The following example rebuilds OLDINDEX with an SDO_LEVEL value of 12.

```
ALTER INDEX oldindex REBUILD PARAMETERS('sdo_level=12');
```

Related Topics

- CREATE INDEX
- DROP INDEX
- ALTER TABLE (clauses for partition maintenance) in the *Oracle9i SQL Reference*

ALTER INDEX RENAME TO

Syntax

ALTER INDEX [schema.]index RENAME TO <new_index_name>;

ALTER INDEX [schema.]index PARTITION partition RENAME TO <new_partition_name>;

Purpose

Alters the name of a spatial index or a partition of a spatial index.

Keywords and Parameters

new_index_name	Specifies the new name of the index.
new_partition_name	Specifies the new name of the partition.

Prerequisites

- You must have EXECUTE privileges on the index type and its implementation type.
- The spatial index to be altered is not marked in-progress.

Usage Notes

None.

Examples

The following example renames OLDINDEX to NEWINDEX.

```
ALTER INDEX oldindex RENAME TO newindex;
```

Related Topics

- CREATE INDEX
- DROP INDEX

CREATE INDEX

Syntax

```
CREATE INDEX [schema.]<index_name> ON [schema.]<tableName> (column)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  [PARAMETERS 'index_params  [physical_storage_params]']);
```

Purpose

Creates a spatial index on a column of type MDSYS.SDO_GEOMETRY.

Keywords and Parameters

INDEX_PARAMS Determine the type (R-tree or quadtree; and for quadtree, fixed or hybrid) and the characteristics of the spatial index.

Keyword	Description
geodetic	For a quadtree index, `'geodetic=FALSE'` allows geodetic data to be built on geodetic data, but with restrictions. (FALSE is the only acceptable value for this keyword.) Do not use this keyword with an R-tree index. See the Usage Notes for further details. Data type is VARCHAR2.
layer_gtype	Checks to ensure that all geometries are of a specified geometry type. The value must be from the Geometry Type column of Table 2–1 in Section 2.2.1. In addition, for a quadtree index specifying POINT, allows for optimized processing of point data. Data type is VARCHAR2.
sdo_commit_ interval	Specifies the number of underlying table rows that are processed between commit intervals for the index data. (Quadtree indexes only.) The default behavior commits the index data only after all rows in the underlying table have been processed. See the Usage Notes for further details. Data type is NUMBER.
sdo_indx_dims	Specifies the number of dimensions to be indexed. (R-tree indexes only.) For example, a value of 2 causes the first 2 dimensions to be indexed. Must be less than or equal to the number of actual dimensions (number of SDO_DIM_ELEMENT instances in the dimensional array that describes the geometry objects in the column). Data type is NUMBER. Default = number of actual dimensions.

sdo_level	Specifies the desired fixed-size tiling level. (Quadtree indexes only.) Data type is NUMBER.
sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object. (Quadtree indexes only.) Data type is NUMBER.
sdo_rtr_pctfree	Specifies the minimum percentage of slots in each index tree node to be left empty when the index is created. Slots that are left empty can be filled later when new data is inserted into the table. (R-tree indexes only.) The value can range from 0 to 50. Data type is NUMBER. Default = 10.
PHYSICAL_ STORAGE_ PARAMS	Determines the storage parameters used for creating the spatial index data table. A spatial index data table is a regular Oracle table with a prescribed format. Not all physical_storage_params that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

Keyword	Description
tablespace	Specifies the tablespace in which the index data table is created. Same as TABLESPACE in the STORAGE clause of a CREATE TABLE statement.
initial	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
next	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
minextents	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
maxextents	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
btree_initial	Is the same as INITIAL in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_next	Is the same as NEXT in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)
btree_pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index. (Quadtree indexes only.)

Prerequisites

- All the current SQL CREATE INDEX prerequisites apply.

- You must have EXECUTE privilege on the index type and its implementation type.
- The USER_SDO_GEOM_METADATA view must contain an entry with the dimensions and coordinate boundary information for the table column to be spatially indexed.

Usage Notes

For information about R-tree and quadtree indexes, see Section 1.7.

By default, an R-tree index is created if the *index_params* string does not contain the *sdo_level* keyword or if the *sdo_level* value is zero (0). If the *index_params* string contains the *sdo_level* keyword with a non-zero value, a quadtree index is created. Some keywords apply only to R-tree or quadtree indexes, as noted in the Keywords and Parameters section.

Before you create an R-tree index, be sure that the rollback segment size and the SORT_AREA_SIZE parameter value are adequate, as described in Section 4.1.1.

For a quadtree index, the *index_params* string must contain either *sdo_level* or both *sdo_level* and *sdo_numtiles*, and any values specified for these parameters must be valid.

With an R-tree index on linear referencing system (LRS) data, the *sdo_indx_dims* parameter must be used and must specify the number of dimensions minus one, so as not to index the measure dimension. For example, if the dimensions are X, Y, and M, specify *sdo_indx_dims=2* to index only the X and Y dimensions, and not the measure (M) dimension. (The LRS data model, including the measure dimension, is explained in Section 6.2.)

A partitioned spatial index can be created on a partitioned table. See Section 4.1.6 for more information about partitioned spatial indexes, including benefits and restrictions.

Other options available for regular indexes (such as ASC and DESC) are not applicable for spatial indexes.

Default values for quadtree indexing:

- *sdo_numtiles* must be supplied with a value greater than or equal to 1 to perform hybrid indexing. If this parameter is not supplied, indexing with fixed-size tiles is performed.
- *sdo_commit_interval* does not allow spatial data to be committed at intervals. Insertion of spatial index data is committed only at the end of the index creation

process. That is, it is committed after all rows in the underlying table have been processed.

The *sdo_level* value must be greater than zero.

If an *sdo_numtiles* value is specified, it might be overridden by the indexing algorithm.

Spatial index creation involves creating and inserting index data, for each row in the underlying table column being spatially indexed, into a table with a prescribed format. The default, or normal, operation is that all rows in the underlying table are processed before the insertion of index data is committed. This requires adequate rollback segment space.

You may choose to commit index data after every *n* rows of the underlying table have been processed. This is done by specifying SDO_COMMIT_INTERVAL = *n*. The potential complication is that, if there is an error during index rebuild and if periodic commit operations have taken place, then the spatial index will be in an inconsistent state. The only recovery option is to use DROP INDEX (possibly with the FORCE option) and CREATE INDEX statements after ensuring that the various tablespaces are the required size and any other error conditions have been removed.

Interpretation of *sdo_level* and *sdo_numtiles* value combinations (quadtree indexing) is shown in Table 9–2.

Table 9–2 SDO_LEVEL and SDO_NUMTILES Combinations

SDO_LEVEL	SDO_NUMTILES	Action
Not specified or 0	Not specified or 0	R-tree index.
>= 1	Not specified or 0	Fixed indexing (indexing with fixed-size tiles).
>= 1	>= 1	Hybrid indexing with fixed-size and variable-sized tiles. The SDO_LEVEL column defines the fixed tile size. The SDO_NUMTILES column defines the number of variable tiles to generate per geometry.
Not specified or 0	>= 1	Not supported (error).

If a tablespace name is provided in the parameters clause, the user (underlying table owner) must have appropriate privileges for that tablespace.

For more information about using the *layer_gtype* keyword to constrain data in a layer to a geometry type, see Section 4.1.4.

The *'geodetic=FALSE'* parameter allows you to bypass the restriction that a quadtree index cannot be used with geodetic data. However, using this parameter is not recommended, because much of the Oracle Spatial geodetic support will be disabled, and some Spatial operations that use the quadtree index with geodetic data will not work correctly or will return less accurate results. This parameter should only be used if you cannot yet reindex the data with an R-tree index and if the results using the quadtree index are acceptable.

If you are creating a function-based spatial index, the number of parameters must not exceed 32. For information about using function-based spatial indexes, see Section 8.2.

To determine if a CREATE INDEX statement for a spatial index has failed, check to see if the DOMIDX_OPSTATUS column in the USER_INDEXES view is set to FAILED. Note that this is different from the case of regular indexes, where you check to see if the STATUS column in the USER_INDEXES view is set to FAILED.

If the CREATE INDEX statement fails because of an invalid geometry, the ROWID of the failed geometry is returned in an error message along with the reason for the failure.

If the CREATE INDEX statement fails for any reason, then the DROP INDEX statement must be used to clean up the partially built index and associated metadata. If DROP INDEX does not work, add the FORCE parameter and try again.

Examples

The following example creates a spatial R-tree index named COLA_SPATIAL_IDX. (An R-tree index is created by default if no quadtree-specific parameters are specified.)

```
CREATE INDEX cola_spatial_idx ON cola_markets(shape)
   INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

The following example creates a spatial quadtree index named QTREE.

```
CREATE INDEX qtree ON POLY_4PT(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('sdo_level=6
    sdo_commit_interval=500 tablespace=system initial=10K
    next=10K pctincrease=10 minextents=10 maxextents=20');
```

The following example creates a spatial quadtree index named BG06075 with two partitions, named IP1 and IP2.

```
CREATE INDEX BGI ON BG06075(geometry)
```

```
INDEXTYPE IS MDSYS.SPATIAL_INDEX
 PARAMETERS ('tablespace=SYSTEM sdo_level=6') LOCAL
 (PARTITION IP1 PARAMETERS ('tablespace=SYSTEM sdo_level=6'),
  PARTITION IP2 PARAMETERS ('tablespace=TBS_3 sdo_level=6'));
```

Related Topics

- ALTER INDEX
- DROP INDEX

DROP INDEX

Syntax

DROP INDEX [schema.]index [FORCE];

Purpose

Deletes a spatial index.

Keywords and Parameters

FORCE	Causes the spatial index to be deleted from the system tables even if the index is marked in-progress or some other error condition occurs.

Prerequisites

You must have EXECUTE privileges on the index type and its implementation type.

Usage Notes

Use DROP INDEX indexname FORCE to clean up after a failure in the CREATE INDEX statement.

Examples

The following example deletes a spatial quadtree index named OLDINDEX and forces the deletion to be performed even if the index is marked in-process or an error occurs.

```
DROP INDEX oldindex FORCE;
```

Related Topics

- CREATE INDEX

10

SDO_GEOMETRY Object Type Methods

This chapter contains reference and usage information for the SDO_GEOMETRY object type methods.

The SDO_GEOMETRY object type is described in Section 2.2. The type methods are listed in Table 10–1.

Table 10–1 SDO_GEOMETRY Type Methods

Method	Description
GET_DIMS	Returns the number of dimensions of a geometry object.
GET_GTYPE	Returns the geometry type of a geometry object.
GET_LRS_DIM	Returns the measure dimension of an LRS geometry object.

GET_DIMS

Format

GET_DIMS( ) RETURN NUMBER;

Description

Returns the number of dimensions of a geometry object, as specified in its SDO_GTYPE value.

Parameters

None.

Usage Notes

The SDO_TYPE value is 4 digits in the format *dltt*, as described in Section 2.2.1. This method returns the *d* (dimensionality) value, that is, the number of dimensions.

Examples

The following example returns the number of dimensions of the *cola_d* geometry object. (The example uses the definitions and data from Section 2.1.)

```
SELECT c.mkt_id, c.shape.GET_DIMS()
  FROM cola_markets c WHERE c.name = 'cola_d';

    MKT_ID C.SHAPE.GET_DIMS()
---------- ------------------
         4                  2
```

GET_GTYPE

Format

GET_GTYPE( ) RETURN NUMBER;

Description

Returns the geometry type of a geometry object, as specified in its SDO_GTYPE value.

Parameters

None.

Usage Notes

The SDO_TYPE value is 4 digits in the format *dltt*, as described in Section 2.2.1. This method returns the *tt* value, that is, the geometry type.

Examples

The following example returns the geometry type of each geometry object in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1.)

```
SELECT c.mkt_id, c.shape.GET_GTYPE() FROM cola_markets c;

    MKT_ID C.SHAPE.GET_GTYPE()
---------- --------------------
         1                    3
         2                    3
         3                    3
         4                    3
```

GET_LRS_DIM

Format

GET_LRS_DIM( ) RETURN NUMBER;

Description

Returns the measure dimension of an LRS geometry object, as specified in its SDO_GTYPE value.

Parameters

None.

Usage Notes

The SDO_TYPE value is 4 digits in the format *dltt*, as described in Section 2.2.1. This method returns the *l* value.

The *l* value is meaningful only for LRS geometry objects, and must be 0, 3, or 4:

- 0 indicates that the geometry is a pre-release 9.0.1 LRS geometry with measure as the default (last) dimension, or that the geometry is a release 9.0.1 standard geometry.
- 3 indicates that the third dimension contains the measure information.
- 4 indicates that the fourth dimension contains the measure information.

Examples

The following example returns the measure dimension of the Route 1 geometry object. (This example uses the definitions from the example in Section 6.6.)

```
SELECT a.route_id, a.route_geometry.GET_LRS_DIM()
    FROM lrs_routes a WHERE  a.route_id = 1;

  ROUTE_ID A.ROUTE_GEOMETRY.GET_LRS_DIM()
---------- -------------------------------
         1                               3
```

11

Spatial Operators

This chapter describes the operators used when working with the spatial object data type. The operators are listed in Table 11–1.

Table 11–1 Spatial Usage Operators

Operator	Description
SDO_FILTER	Specifies which geometries may interact with a given geometry.
SDO_NN	Determines the nearest neighbor geometries to a geometry.
SDO_NN_DISTANCE	Returns the distance of an object returned by the SDO_NN operator.
SDO_RELATE	Determines whether or not two geometries interact in a specified way.
SDO_WITHIN_DISTANCE	Determines if two geometries are within a specified distance from one another.

SDO_FILTER

Format

SDO_FILTER(*geometry1, geometry2, params*);

Description

Uses the spatial index to identify either the set of spatial objects that are likely to interact spatially with a given object (such as an area of interest), or pairs of spatial objects that are likely to interact spatially. Objects interact spatially if they are not disjoint.

This operator performs only a primary filter operation. The secondary filtering operation, performed by the SDO_RELATE operator, can be used to determine with certainty if objects interact spatially.

Keywords and Parameters

geometry1 Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.

geometry2 Specifies either a geometry from a table or a transient instance of a geometry. (Specified using a bind variable or SDO_GEOMETRY constructor.) Data type is MDSYS.SDO_GEOMETRY.

PARAMS Determines the behavior of the operator. Data type is VARCHAR2.

Keyword	Description
querytype	Specifies valid query types: WINDOW or JOIN. This is a required parameter. WINDOW is recommended in almost all cases. WINDOW implies that a query is performed for every *geometry1* candidate geometry to be compared with *geometry2*. WINDOW can be used to compare a single geometry (*geometry2*) to all the geometries in a column (*geometry1*). JOIN is rarely used. Use JOIN when you want to compare all the geometries of a column to all the geometries of another column. JOIN implies that *geometry2* refers to a table column that must have a spatial index built on it. (See the Usage Notes for additional requirements.)
idxtab1	Specifies the name of the index table, if there are multiple spatial indexes, for *geometry1*.
idxtab2	Specifies the name of the index table, if there are multiple spatial indexes, for *geometry2*. Valid only if *querytype* is JOIN.

Returns

The expression SDO_FILTER(arg1, arg2, arg3) = 'TRUE' evaluates to TRUE for object pairs that are non-disjoint, and FALSE otherwise.

Usage Notes

The operator must always be used in a WHERE clause and the condition that includes the operator should be an expression of the form SDO_FILTER(arg1, arg2, arg3) = 'TRUE'.

If *querytype* is WINDOW, *geometry2* can come from a table or be a transient SDO_GEOMETRY object (such as a bind variable or SDO_GEOMETRY constructor).

- If the *geometry2* column is not spatially indexed, the operator indexes the query window in memory and performance is very good.
- If the *geometry2* column is spatially indexed with the same SDO_LEVEL value as the *geometry1* column, the operator reuses the existing index, and performance is very good or better.
- If the *geometry2* column is spatially indexed with a different SDO_LEVEL value than the *geometry1* column, the operator reindexes *geometry2* in the same way as if there were no index on the column originally, and then performance is very good.
- If two or more geometries from *geometry2* are passed to the operator, the ORDERED optimizer hint must be specified, and the table in *geometry2* must be specified first in the FROM clause.

If *querytype* is JOIN:

- *geometry2* must be a column in a table.
- For best performance, both *geometry1* and *geometry2* should have the same type of index (R-tree or quadtree); and if the geometries have quadtree indexes, the indexes should have the same *sdo_level* value. If the geometries do not have the same index type (and for quadtree indexes the same *sdo_level* value), *geometry2* is reindexed to be indexed as *geometry1* (with the considerations listed for *querytype* = WINDOW), and performance is less efficient.
- An exception is raised if *geometry1* and *geometry2* are based on different coordinate systems.

The *layer_gtype* keyword for *PARAMS* has been deprecated, and it is ignored if specified. The operator automatically optimizes its behavior based on the SDO_GTYPE value (explained in Section 2.2.1) of the geometries.

Examples

The following example selects the GID values from the POLYGONS table where the GEOMETRY column objects are likely to interact spatially with the GEOMETRY column object in the QUERY_POLYS table that has a GID value of 1.

```
SELECT A.gid
  FROM Polygons A, query_polys B
  WHERE B.gid = 1
  AND SDO_FILTER(A.Geometry, B.Geometry, 'querytype = WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is likely to interact spatially with the geometry stored in the *aGeom* variable.

```
Select A.Gid
  FROM Polygons A
  WHERE SDO_FILTER(A.Geometry, :aGeom, 'querytype=WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is likely to interact spatially with the specified rectangle having the lower-left coordinates (x1,y1) and the upper-right coordinates (x2, y2).

```
Select A.Gid
  FROM Polygons A
  WHERE SDO_FILTER(A.Geometry, mdsys.sdo_geometry(2003,NULL,NULL,
                                  mdsys.sdo_elem_info_array(1,1003,3),
                                  mdsys.sdo_ordinate_array(x1,y1,x2,y2)),
                    'querytype=WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is likely to interact spatially with any GEOMETRY column object in the QUERY_POLYS table. In this example, the ORDERED optimizer hint is used and QUERY_POLYS (*geometry2*) table is specified first in the FROM clause, because multiple geometries from *geometry2* are involved (see the Usage Notes)

```
SELECT /*+ ORDERED */
  A.gid
  FROM query_polys B, polygons A
  WHERE SDO_FILTER(A.Geometry, B.Geometry, 'querytype = WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is likely to interact spatially with any GEOMETRY column object in the QUERY_POLYS table. In this example, the QUERY_ POLYS.GEOMETRY column must be spatially indexed.

```
SELECT A.gid
  FROM Polygons A, query_polys B
  WHERE SDO_FILTER(A.Geometry, B.Geometry, 'querytype = JOIN') = 'TRUE';
```

Related Topics

- SDO_RELATE

SDO_NN

Format

SDO_NN(geometry1, geometry2, param [, number]);

Description

Uses the spatial index to identify the nearest neighbors for a geometry.

Keywords and Parameters

geometry1 Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.

geometry2 Specifies either a geometry from a table or a transient instance of a geometry. The nearest neighbor or neighbors to *geometry2* will be returned from *geometry1*. (*geometry2* is specified using a bind variable or SDO_GEOMETRY constructor.) Data type is MDSYS.SDO_GEOMETRY.

param Determines the behavior of the operator. The available keywords are listed in Table 11–2. Data type is VARCHAR2.

number If the SDO_NN_DISTANCE ancillary operator is included in the call to SDO_NN, specifies the same number used in the call to SDO_NN_DISTANCE. Data type is NUMBER.

Table 11–2 lists the keywords for the *param* parameter.

Table 11–2 Keywords for SDO_NN Parameter

Keyword	Description
sdo_batch_size	Specifies the number of rows to be evaluated at a time when the SDO_NN expression may need to be evaluated multiple times in order to return the desired number of results that satisfy the WHERE clause. Available only when an R-tree index is used. See the Usage Notes for more information. Data type is NUMBER. For example: 'sdo_batch_size=10'

Table 11–2 Keywords for SDO_NN Parameter (Cont.)

Keyword	Description
sdo_num_res	If *sdo_batch_size* is not specified, specifies the number of results (nearest neighbors) to be returned. If *sdo_batch_size* is specified, this keyword is ignored; instead, use the ROWNUM pseudocolumn to limit the number of results. See the Usage Notes and Examples for more information. Data type is NUMBER. Default = 1. For example: 'sdo_num_res=5'
unit	If the SDO_NN_DISTANCE ancillary operator is included in the call to SDO_NN, specifies the unit of measurement: a quoted string with *unit*= and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table. See Section 2.6 for more information about unit of measurement specification. Data type is NUMBER. Default = unit of measurement associated with the data. For projected data, the default is meters. For example: 'unit=KM'

Returns

This operator returns the *sdo_num_res* number of objects from *geometry1* that are nearest to *geometry2* in the query. In determining how near two geometry objects are, the shortest possible distance between any two points on the surface of each object is used.

Usage Notes

The operator must always be used in a WHERE clause, and the condition that includes the operator should be an expression of the form SDO_NN(arg1, arg2, '<some_parameter>') = 'TRUE'.

The operator can be used in two ways:

- If all geometries in the layer are candidates, use the *sdo_num_res* keyword to specify the number of geometries returned.
- If any geometries in the table might be nearer than the geometries specified in the WHERE clause, use the *sdo_batch_size* keyword and use the WHERE clause (including the ROWNUM pseudocolumn) to limit the number of geometries returned.

Specify the *sdo_batch_size* keyword if any geometries in the table might be nearer than the geometries specified in the WHERE clause. For example, assume that a RESTAURANTS table contained different types of restaurants, and you wanted to

find the two nearest Italian restaurants to your hotel. The query might look like the following:

```
SELECT r.name FROM restaurants r WHERE
   SDO_NN(r.geometry, :my_hotel, 'sdo_batch_size=10') = 'TRUE'
   AND r.cuisine = 'Italian' AND ROWNUM <=2;
```

If the *sdo_batch_size* keyword is not specified in this example, only the two nearest restaurants are returned, regardless of their CUISINE value; and if the CUISINE value of these two rows is not *Italian*, the query may return no rows. The `ROWNUM <=2` clause is necessary to limit the number of results returned to no more than 2 where CUISINE is *Italian*.

The *sdo_batch_size* keyword can be used only when SDO_NN will be using an R-tree index to perform the operation. This keyword cannot be used with a quadtree index.

The *sdo_batch_size* value can affect the performance of nearest neighbor queries. A good general guideline is to specify the number of candidate rows likely to satisfy the WHERE clause. Using the preceding example of a query for Italian restaurants, if approximately 20 percent of the restaurants nearest to the hotel are Italian and if you want 2 restaurants, an *sdo_batch_size* value of 10 will probably result in the best performance. On the other hand, if only approximately 5 percent of the restaurants nearest to the hotel are Italian and if you want 2 restaurants, an *sdo_batch_size* value of 40 would be better.

If the *sdo_batch_size* keyword is specified, any *sdo_num_res* value is ignored. Do not specify both keywords.

Specify the *number* parameter only if you are using the SDO_NN_DISTANCE ancillary operator in the call to SDO_NN. See the information about SDO_NN_DISTANCE.

If two or more objects from *geometry1* are an equal distance from *geometry2*, any of the objects can be returned on any call to the function. For example, if *item_a*, *item_b*, and *item_c* are nearest to and equally distant from *geometry2*, and if SDO_NUM_RES=2, two of those three objects are returned, but they can be any two of the three.

If the SDO_NN operator uses a partitioned spatial index (see Section 4.1.6), the requested number of geometries is returned for *each* partition that contains candidate rows based on the query criteria. For example, if you request the 5 nearest restaurants to a point and the spatial index has 4 partitions, the operator returns up to 20 (5*4) geometries. In this case, you must use the ROWNUM pseudocolumn (here, `WHERE ROWNUM <=5`) to return the 5 nearest restaurants.

An exception is raised if *geometry1* and *geometry2* are based on different coordinate systems.

SDO_NN is not supported for spatial joins.

In some situations the SDO_NN operator will not use the spatial index unless an optimizer hint forces the index to be used. This can occur when a query involves a join; and if the optimizer hint is not used in such situations, an internal error occurs. To prevent such errors, you should always specify an optimizer hint to use the spatial index with the SDO_NN operator, regardless of how simple or complex the query is. For example, the following excerpt from a query specifies to use the COLA_SPATIAL_IDX index that is defined on the COLA_MARKETS table:

```
SELECT /*+ INDEX(cola_markets cola_spatial_idx) */
  c.mkt_id, c.name, ... FROM cola_markets c, ...;
```

For detailed information about using optimizer hints, see *Oracle9i Database Performance Guide and Reference*.

Examples

The following example finds the two objects from the SHAPE column in the COLA_MARKETS table that are nearest to a specified point (10,7). (The example uses the definitions and data from Section 2.1.)

```
SELECT /*+ INDEX(cola_markets cola_spatial_idx) */
 c.mkt_id, c.name  FROM cola_markets c  WHERE SDO_NN(c.shape,
   mdsys.sdo_geometry(2001, NULL, mdsys.sdo_point_type(10,7,NULL), NULL,
   NULL),  'sdo_num_res=2') = 'TRUE';

    MKT_ID NAME
---------- --------------------------------
         2 cola_b
         4 cola_d
```

The following example uses the *sdo_batch_size* keyword to find the two objects (ROWNUM <=2), with a NAME value less than 'cola_d', from the SHAPE column in the COLA_MARKETS table that are nearest to a specified point (10,7). The value of 3 for *sdo_batch_size* represents a best guess at the number of nearest geometries that need to be evaluated before the WHERE clause condition is satisfied. (The example uses the definitions and data from Section 2.1.)

```
SELECT /*+ INDEX(cola_markets cola_spatial_idx) */ c.mkt_id, c.name
   FROM cola_markets c
   WHERE SDO_NN(c.shape,  mdsys.sdo_geometry(2001, NULL,
```

```
      mdsys.sdo_point_type(10,7,NULL), NULL,  NULL),
      'sdo_batch_size=3') = 'TRUE'
   AND c.name < 'cola_d' AND ROWNUM <= 2;

    MKT_ID NAME
---------- --------------------------------
         2 cola_b
         3 cola_c
```

Related Topics

- SDO_NN_DISTANCE

SDO_NN_DISTANCE

Format

SDO_NN_DISTANCE(number);

Description

Returns the distance of an object returned by the SDO_NN operator. Valid only within a call to the SDO_NN operator.

Keywords and Parameters

number Specifies a number that must be the same as the last parameter passed to the SDO_NN operator.
Data type is NUMBER.

Returns

This operator returns the distance of an object returned by the SDO_NN operator. In determining how near two geometry objects are, the shortest possible distance between any two points on the surface of each object is used.

Usage Notes

SDO_NN_DISTANCE is an ancillary operator to the SDO_NN operator. It returns the distance between the specified geometry and a nearest neighbor object. This distance is passed as ancillary data to the SDO_NN operator. (For an explanation of how operators can use ancillary data, see the section on ancillary data in the chapter on domain indexes in the *Oracle9i Data Cartridge Developer's Guide*.)

You can choose any arbitrary number for the *number* parameter. The only requirement is that it must match the last parameter in the call to the SDO_NN operator.

Use a bind variable to store and operate on the distance value.

Examples

The following example finds the two objects from the SHAPE column in the COLA_MARKETS table that are nearest to a specified point (10,7), and it finds the distance

between each object and the point. (The example uses the definitions and data from Section 2.1.)

```
SELECT   /*+ INDEX(cola_markets cola_spatial_idx) */
   c.mkt_id, c.name, mdsys.SDO_NN_DISTANCE(1) dist
   FROM cola_markets c
   WHERE SDO_NN(c.shape,  mdsys.sdo_geometry(2001, NULL,
      mdsys.sdo_point_type(10,7,NULL), NULL,  NULL),
      'sdo_num_res=2', 1) = 'TRUE' ORDER BY dist;

    MKT_ID NAME                                DIST
---------- ------------------------------ ----------
         4 cola_d                         .828427125
         2 cola_b                         2.23606798
```

Note the following about this example:

- 1 is used as the *number* parameter for SDO_NN_DISTANCE, and 1 is also specified as the last parameter to SDO_NN (after `'sdo_num_res=2'`).
- The column alias *dist* holds the distance between the object and the point. (For geodetic data, the distance unit is meters; for non-geodetic data, the distance unit is the unit associated with the data.)

Related Topics

- SDO_NN

SDO_RELATE

Format

SDO_RELATE(geometry1, geometry2, params);

Description

Uses the spatial index to identify either the spatial objects that have a particular spatial interaction with a given object such as an area of interest, or pairs of spatial objects that have a particular spatial interaction.

This operator performs both primary and secondary filter operations.

Keywords and Parameters

geometry1 Specifies a geometry column in a table. The column must be spatially indexed.
Data type is MDSYS.SDO_GEOMETRY.

geometry2 Specifies either a geometry from a table or a transient instance of a geometry. (Specified using a bind variable or SDO_GEOMETRY constructor.)
Data type is MDSYS.SDO_GEOMETRY.

PARAMS Determines the behavior of the operator.
Data type is VARCHAR2.

Keyword	Description
mask	Specifies the topological relation of interest. This is a required parameter. Valid values are one or more of the following in the 9-intersection pattern: TOUCH, OVERLAPBDYDISJOINT, OVERLAPBDYINTERSECT, EQUAL, INSIDE, COVEREDBY, CONTAINS, COVERS, ANYINTERACT, ON. Multiple masks are combined with the logical Boolean operator OR, for example, 'mask=inside+touch'; however, see the Usage Notes for an alternative syntax using UNION ALL that may result in better performance. See Section 1.8 for an explanation of the 9-intersection relationship pattern.

querytype	Valid query types are: WINDOW or JOIN. This is a required parameter if *geometry2* is from another table, but it is not a required parameter if *geometry2* is a literal or a host variable. WINDOW is recommended in almost all cases. WINDOW implies that a query is performed for every *geometry1* candidate geometry to be compared with *geometry2*. WINDOW can be used to compare a single geometry (*geometry2*) to all the geometries in a column (*geometry1*). JOIN is rarely used. Use JOIN when you want to compare all the geometries of a column to all the geometries of another column. JOIN implies that *geometry2* refers to a table column that must have a spatial index built on it. (See the Usage Notes for additional requirements.)
idxtab1	Specifies the name of the index table, if there are multiple spatial indexes, for *geometry1*.
idxtab2	Specifies the name of the index table, if there are multiple spatial indexes, for *geometry2*. Only valid for 'querytype = JOIN'.

Returns

The expression SDO_RELATE(geometry1,geometry2, 'mask = <some_mask_val> querytype = <some_querytype>') = 'TRUE' evaluates to TRUE for object pairs that have the topological relationship specified by <some_mask_val>, and FALSE otherwise.

Usage Notes

The operator must always be used in a WHERE clause, and the condition that includes the operator should be an expression of the form SDO_RELATE(arg1, arg2, 'mask = <some_mask_val> querytype = <some_querytype>') = 'TRUE'.

If *querytype* is WINDOW, *geometry2* can come from a table or be a transient SDO_GEOMETRY object (such as a bind variable or SDO_GEOMETRY constructor).

- If the *geometry2* column is not spatially indexed, the operator indexes the query window in memory and performance is very good.
- If the *geometry2* column is spatially indexed with the same SDO_LEVEL value as the *geometry1* column, the operator reuses the existing index, and performance is very good or better.
- If the *geometry2* column is spatially indexed with a different SDO_LEVEL value than the *geometry1* column, the operator reindexes *geometry2* in the same way as if there were no index on the column originally, and then performance is very good.

- If two or more geometries from *geometry2* are passed to the operator, the ORDERED optimizer hint must be specified, and the table in *geometry2* must be specified first in the FROM clause.

If *querytype* is JOIN:

- *geometry2* must be a column in a table.
- For best performance, both *geometry1* and *geometry2* should have the same type of index (R-tree or quadtree); and if the geometries have quadtree indexes, the indexes should have the same *sdo_level* value. If the geometries do not have the same index type (and for quadtree indexes the same *sdo_level* value), *geometry2* is reindexed to be indexed as *geometry1* (with the considerations listed for *querytype* = WINDOW), and performance is less efficient.
- An exception is raised if *geometry1* and *geometry2* are based on different coordinate systems.

The *layer_gtype* keyword for *PARAMS* has been deprecated, and it is ignored if specified. The operator automatically optimizes its behavior based on the SDO_GTYPE value (explained in Section 2.2.1) of the geometries.

Unlike with the SDO_GEOM.RELATE function, DISJOINT and DETERMINE masks are not allowed in the relationship mask with the SDO_RELATE operator. This is because SDO_RELATE uses the spatial index to find candidates that may interact, and the information to satisfy DISJOINT or DETERMINE is not present in the index.

Although multiple masks can be combined using the logical Boolean operator OR, for example, 'mask=inside+coveredby', better performance may result if the spatial query specifies each mask individually and uses the UNION ALL syntax to combine the results. This is due to internal optimizations that Spatial can apply under certain conditions when masks are specified singly rather than grouped within the same SDO_RELATE operator call. For example, the following query using the logical Boolean operator OR to group multiple masks:

```
SELECT a.gid
  FROM polygons a, query_polys B
  WHERE B.gid = 1
  AND SDO_RELATE(A.Geometry, B.Geometry,
                  'mask=inside+coveredby querytype=WINDOW') = 'TRUE';
```

may result in better performance if it is expressed thus, using UNION ALL to combine results of multiple SDO_RELATE operator calls, each with a single mask:

```
SELECT a.gid
```

```
      FROM polygons a, query_polys B
      WHERE B.gid = 1
      AND SDO_RELATE(A.Geometry, B.Geometry,
                  'mask=inside querytype=WINDOW') = 'TRUE'
UNION ALL
SELECT a.gid
      FROM polygons a, query_polys B
      WHERE B.gid = 1
      AND SDO_RELATE(A.Geometry, B.Geometry,
                  'mask=coveredby querytype=WINDOW') = 'TRUE';
```

Examples

The following examples are similar to those for the SDO_FILTER operator; however, they identify a specific type of interaction (using the *mask* parameter), and they determine with certainty (not mere likelihood) if the spatial interaction occurs.

The following example selects the GID values from the POLYGONS table where the GEOMETRY column objects have any spatial interaction with the GEOMETRY column object in the QUERY_POLYS table that has a GID value of 1.

```
SELECT A.gid
  FROM Polygons A, query_polys B
  WHERE B.gid = 1
  AND SDO_RELATE(A.Geometry, B.Geometry,
                  'mask=ANYINTERACT querytype=WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where a GEOMETRY column object has any spatial interaction with the geometry stored in the *aGeom* variable.

```
Select A.Gid
  FROM Polygons A
  WHERE SDO_RELATE(A.Geometry, :aGeom, 'mask=ANYINTERACT querytype=WINDOW')
   = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where a GEOMETRY column object has any spatial interaction with the specified rectangle having the lower-left coordinates (x1,y1) and the upper-right coordinates (x2, y2).

```
Select A.Gid
  FROM Polygons A
  WHERE SDO_RELATE(A.Geometry, mdsys.sdo_geometry(2003,NULL,NULL,
                                  mdsys.sdo_elem_info_array(1,1003,3),
                                  mdsys.sdo_ordinate_array(x1,y1,x2,y2)),
                    'mask=ANYINTERACT querytype=WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object has any spatial interaction with any GEOMETRY column object in the QUERY_POLYS table. In this example, the ORDERED optimizer hint is used and QUERY_POLYS (*geometry2*) table is specified first in the FROM clause, because multiple geometries from *geometry2* are involved (see the Usage Notes).

```
SELECT /*+ ORDERED */
  A.gid
  FROM query_polys B, polygons A
  WHERE SDO_RELATE(A.Geometry, B.Geometry, 'querytype = WINDOW') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where a GEOMETRY column object has any spatial interaction with any GEOMETRY column object in the QUERY_POLYS table. In this example, the QUERY_POLYS.GEOMETRY column must be spatially indexed.

```
SELECT A.gid
  FROM Polygons A, query_polys B
  WHERE SDO_RELATE(A.Geometry, B.Geometry,
                   'mask=ANYINTERACT querytype=JOIN') = 'TRUE';
```

Related Topics

- SDO_FILTER
- SDO_WITHIN_DISTANCE
- SDO_GEOM.RELATE function

SDO_WITHIN_DISTANCE

Format

SDO_WITHIN_DISTANCE(geometry1, aGeom, params);

Description

Uses the spatial index to identify the set of spatial objects that are within some specified distance of a given object (such as an area of interest or point of interest).

Keywords and Parameters

geometry1 Specifies a geometry column in a table. The column has the set of geometry objects that will be operated on to determine if they are within the specified distance of the given object (*aGeom*). The column must be spatially indexed.
Data type is MDSYS.SDO_GEOMETRY.

aGeom Specifies the object to be checked for distance against the geometry objects in *geometry1*. Specify either a geometry from a table (using a bind variable) or a transient instance of a geometry (using the SDO_GEOMETRY constructor).
Data type is MDSYS.SDO_GEOMETRY.

PARAMS Determines the behavior of the operator.
Data type is VARCHAR2.

Keyword	Description
distance	Specifies the distance value. If a coordinate system is associated with the geometry, the distance unit is assumed to be the unit associated with the coordinate system. This is a required parameter. Data type is NUMBER.
idxtab1	Specifies the name of the index table if there are multiple spatial index tables for *geometry1*.

querytype	Set 'querytype=FILTER' to perform only a primary filter operation. If *querytype* is not specified, both primary and secondary filter operations are performed (default). Data type is VARCHAR2.
unit	Specifies the unit of measurement: a quoted string with *unit*= and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, 'unit=KM'). See Section 2.6 for more information about unit of measurement specification. Data type is NUMBER. Default = unit of measurement associated with the data. For projected data, the default is meters.

Returns

The expression SDO_WITHIN_DISTANCE(arg1, arg2, arg3) = 'TRUE' evaluates to TRUE for object pairs that are within the specified distance, and FALSE otherwise.

Usage Notes

Distance between two extended objects (nonpoint objects such as lines and polygons) is defined as the minimum distance between these two objects. The distance between two adjacent polygons is zero.

If this operator is used with geodetic data, the data must be indexed with an R-tree spatial index.

The operator must always be used in a WHERE clause and the condition that includes the operator should be an expression of the form:

```
SDO_WITHIN_DISTANCE(arg1, arg2, 'distance = <some_dist_val>') = 'TRUE'
```

The geometry column must have a spatial index built on it. If the data is geodetic, the spatial index must be an R-tree index.

The *layer_gtype* keyword for *PARAMS* has been deprecated, and it is ignored if specified. The operator automatically optimizes its behavior based on the SDO_GTYPE value (explained in Section 2.2.1) of the geometries.

SDO_WITHIN_DISTANCE is not supported for spatial joins. See Section 4.2.2.3 for a discussion on how to perform a spatial join within-distance operation.

Examples

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is within 10 distance units of the geometry stored in the *aGeom* variable.

```
SELECT A.GID
  FROM POLYGONS A
  WHERE
    SDO_WITHIN_DISTANCE(A.Geometry, :aGeom, 'distance = 10') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GEOMETRY column object is within 10 distance units of the specified rectangle having the lower-left coordinates (x1,y1) and the upper-right coordinates (x2, y2).

```
SELECT A.GID
  FROM POLYGONS A
  WHERE
    SDO_WITHIN_DISTANCE(A.Geometry, mdsys.sdo_geometry(2003,NULL,NULL,
                              mdsys.sdo_elem_info_array(1,1003,3),
                              mdsys.sdo_ordinate_array(x1,y1,x2,y2)),
                    'distance = 10') = 'TRUE';
```

The following example selects the GID values from the POLYGONS table where the GID value in the QUERY_POINTS table is 1 and a POLYGONS.GEOMETRY object is within 10 distance units of the QUERY_POINTS.GEOMETRY object.

```
SELECT A.GID
  FROM POLYGONS A, Query_Points B
  WHERE B.GID = 1 AND
    SDO_WITHIN_DISTANCE(A.Geometry, B.Geometry, 'distance = 10') = 'TRUE';
```

Related Topics

- SDO_FILTER
- SDO_RELATE

12

Geometry Functions

This chapter contains descriptions of the geometry functions, which can be grouped into the following categories:

- Relationship (True / False) between two objects: RELATE, WITHIN_DISTANCE
- Validation: VALIDATE_GEOMETRY, VALIDATE_LAYER
- Single-object operations: SDO_ARC_DENSIFY, SDO_AREA, SDO_BUFFER, SDO_CENTROID, SDO_CONVEXHULL, SDO_LENGTH, SDO_MBR, SDO_POINTONSURFACE
- Two-object operations: SDO_DISTANCE, SDO_DIFFERENCE, SDO_INTERSECTION, SDO_UNION, SDO_XOR

The geometry functions are listed Table 12–1, and some usage information follows the table.

Table 12–1 Geometry Functions

Function	Description
SDO_GEOM.RELATE	Determines how two objects interact.
SDO_GEOM.SDO_ARC_DENSIFY	Changes each circular arc into an approximation consisting of straight lines, and each circle into a polygon consisting of a series of straight lines that approximate the circle.
SDO_GEOM.SDO_AREA	Computes the area of a two-dimensional polygon.
SDO_GEOM.SDO_BUFFER	Generates a buffer polygon around a geometry.
SDO_GEOM.SDO_CENTROID	Returns the centroid of a polygon.

Table 12–1 Geometry Functions (Cont.)

Function	Description
SDO_GEOM.SDO_CONVEXHULL	Returns a polygon-type object that represents the convex hull of a geometry object.
SDO_GEOM.SDO_DIFFERENCE	Returns a geometry object that is the topological difference (MINUS operation) of two geometry objects.
SDO_GEOM.SDO_DISTANCE	Computes the distance between two geometry objects.
SDO_GEOM.SDO_INTERSECTION	Returns a geometry object that is the topological intersection (AND operation) of two geometry objects.
SDO_GEOM.SDO_LENGTH	Computes the length or perimeter of a geometry.
SDO_GEOM.SDO_MAX_MBR_ ORDINATE	Returns the maximum value for the specified ordinate of the minimum bounding rectangle of a geometry object.
SDO_GEOM.SDO_MBR	Returns the minimum bounding rectangle of a geometry.
SDO_GEOM.SDO_MIN_MBR_ ORDINATE	Returns the minimum value for the specified ordinate of the minimum bounding rectangle of a geometry object.
SDO_GEOM.SDO_POINTONSURFACE	Returns a point that is guaranteed to be on the surface of a polygon.
SDO_GEOM.SDO_UNION	Returns a geometry object that is the topological union (OR operation) of two geometry objects.
SDO_GEOM.SDO_XOR	Returns a geometry object that is the topological symmetric difference (XOR operation) of two geometry objects.
SDO_GEOM.VALIDATE_GEOMETRY	Determines if a geometry is valid.
SDO_GEOM.VALIDATE_LAYER	Determines if all the geometries stored in a column are valid.
SDO_GEOM.WITHIN_DISTANCE	Determines if two geometries are within a specified Euclidean distance from one another.

Note: The SDO_POLY_xxx functions were deprecated at release 8.1.6 and have been removed from this guide. You should use instead the corresponding generic (not restricted to polygons) SDO_xxx functions: SDO_GEOM.SDO_DIFFERENCE, SDO_GEOM.SDO_INTERSECTION, SDO_GEOM.SDO_UNION, and SDO_GEOM.SDO_XOR.

The following usage information applies to the geometry functions. (See also the Usage Notes under the reference information for each function.)

- Certain combinations of input parameters and operations can return a null value, that is, an empty geometry. For example, requesting the intersection of two disjoint geometry objects returns a null value.
- A null value (empty geometry) as an input parameter to a geometry function (for example, SDO_GEOM.RELATE) produces an error.
- Certain operations can return a geometry of a different type than one or both input geometries. For example, the intersection of a line and an overlapping polygon returns a line; the intersection of two lines returns a point; and the intersection of two tangent polygons returns a line.

SDO_GEOM.RELATE

Format

```
SDO_GEOM.RELATE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    mask   IN VARCHAR2,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN VARCHAR2;
```

or

```
SDO_GEOM.RELATE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    mask   IN VARCHAR2,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN VARCHAR2;
```

Description

Examines two geometry objects to determine their spatial relationship.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

mask
Specifies a list of relationships to check. See the list of keywords in the Usage Notes.

geom2
Geometry object.

dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

The MDSYS.SDO_GEOM.RELATE function can return the following types of answers:

- If you pass a *mask* listing one or more relationships, the function returns the name of the relationship if it is true for the pair of geometries. If all the relationships are false, the procedure returns FALSE.
- If you pass the DETERMINE keyword in *mask*, the function returns the one relationship keyword that best matches the geometries. DETERMINE can only be used when SDO_GEOM.RELATE is in the SELECT clause of the SQL statement.
- If you pass the ANYINTERACT keyword in *mask*, the function returns TRUE if the two geometries are not disjoint.

The following *mask* relationships can be tested:

- ANYINTERACT: Returns TRUE if the objects are not disjoint.
- CONTAINS: Returns CONTAINS if the second object is entirely within the first object and the object boundaries do not touch; otherwise, returns FALSE.
- COVEREDBY: Returns COVEREDBY 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 COVERS if the second object is entirely within the first object and the boundaries touch in one or more places; otherwise, returns FALSE.
- DISJOINT: Returns DISJOINT if the objects have no common boundary or interior points; otherwise, returns FALSE.
- EQUAL: Returns EQUAL if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns FALSE.

- INSIDE: Returns INSIDE if the first object is entirely within the second object and the object boundaries do not touch; otherwise, returns FALSE.
- OVERLAPBDYDISJOINT: Returns OVERLAPBDYDISJOINT if the objects overlap, but their boundaries do not interact; otherwise, returns FALSE.
- OVERLAPBDYINTERSECT: Returns OVERLAPBDYINTERSECT if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns FALSE.
- TOUCH: Returns TOUCH if the two objects share a common boundary point, but no interior points; otherwise, returns FALSE.

Values for *mask* can be combined using the logical Boolean operator OR. For example, 'INSIDE + TOUCH' returns 'INSIDE + TOUCH' or 'FALSE' depending on the outcome of the test.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

Examples

The following example checks if there is any spatial interaction between geometry objects *cola_b* and *cola_d*. (The example uses the definitions and data from Section 2.1.)

```
SELECT SDO_GEOM.RELATE(c_b.shape, 'anyinteract', c_d.shape, 0.005)
  FROM cola_markets c_b, cola_markets c_d
  WHERE c_b.name = 'cola_b' AND c_d.name = 'cola_d';

SDO_GEOM.RELATE(C_B.SHAPE,'ANYINTERACT',C_D.SHAPE,0.005)
--------------------------------------------------------------------------------
FALSE
```

Related Topics

None.

SDO_GEOM.SDO_ARC_DENSIFY

Format

```
SDO_GEOM.SDO_ARC_DENSIFY(
    geom    IN MDSYS.SDO_GEOMETRY,
    dim     IN MDSYS.SDO_DIM_ARRAY
    params  IN VARCHAR2
    ) MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_ARC_DENSIFY(
    geom    IN MDSYS.SDO_GEOMETRY,
    tol     IN NUMBER
    params  IN VARCHAR2
    ) MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry in which each circular arc in the input geometry is changed into an approximation of the circular arc consisting of straight lines, and each circle is changed into a polygon consisting of a series of straight lines that approximate the circle.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

params
A quoted string containing an arc tolerance value and optionally a unit value. See the Usage Notes for an explanation of the format and meaning.

Usage Notes

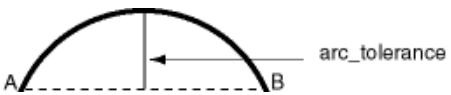
This function is especially useful when operations involve geodetic coordinate systems. Geodetic coordinate system support is provided only for geometries that consist of points or geodesics (lines on the sphere). If you have geometries containing circles or circular arcs, you can transform them to a projected coordinate system, use this function to densify them into regular polygons, and perform Spatial operations on the resulting geometries. You can then transform the geometries to any projected or geodetic coordinate system.

The *params* parameter is a quoted string that must contain the *arc_tolerance* keyword and may contain the *unit* keyword. For example:

```
'arc_tolerance=0.05 unit=km'
```

The *arc_tolerance* keyword specifies, for each arc in the geometry, the maximum length of the perpendicular line between the surface of the arc and the straight line between the start and end points of the arc. Figure 12–1 shows a line whose length is the *arc_tolerance* value for the arc between points A and B.

Figure 12–1 Arc Tolerance



The *arc_tolerance* keyword value must be greater than or equal to the tolerance value associated with the geometry. As you increase the *arc_tolerance* keyword value, the resulting polygon has fewer sides and a smaller area; as you decrease the *arc_tolerance* keyword value, the resulting polygon has more sides and a larger area (but never larger than the original geometry).

If the *unit* keyword is specified, the value must be an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, 'unit=KM'). If the *unit* keyword is not specified, the unit of measurement associated with the geometry is used. See Section 2.6 for more information about unit of measurement specification.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

Examples

The following example returns the geometry that results from the arc densification of *cola_d*, which is a circle. (The example uses the definitions and data from Section 2.1.)

```
-- Arc densification of the circle cola_d
SELECT c.name, SDO_GEOM.SDO_ARC_DENSIFY(c.shape, m.diminfo,
                                        'arc_tolerance=0.05')
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_d';

NAME
--------------------------------
SDO_GEOM.SDO_ARC_DENSIFY(C.SHAPE,M.DIMINFO,'ARC_TOLERANCE=0.05')(SDO_GTYPE, SDO_
--------------------------------------------------------------------------------
cola_d
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(8, 7, 8.76536686, 7.15224093, 9.41421356, 7.58578644, 9.84775907, 8.23463314,
 10, 9, 9.84775907, 9.76536686, 9.41421356, 10.4142136, 8.76536686, 10.8477591,
8, 11, 7.23463314, 10.8477591, 6.58578644, 10.4142136, 6.15224093, 9.76536686, 6
, 9, 6.15224093, 8.23463314, 6.58578644, 7.58578644, 7.23463314, 7.15224093, 8,
7))
```

Related Topics

- Section 5.2.3, "Other Considerations and Requirements with Geodetic Data"

SDO_GEOM.SDO_AREA

Format

```
SDO_GEOM.SDO_AREA(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    [, unit  IN VARCHAR2]
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_AREA(
    geom  IN MDSYS.SDO_GEOMETRY,
    tol   IN NUMBER
    [, unit  IN VARCHAR2]
    ) RETURN NUMBER;
```

Description

Returns the area of a two-dimensional polygon.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

unit
Unit of measurement: a quoted string with *unit*= and an SDO_UNIT value from the MDSYS.SDO_AREA_UNITS table (for example, 'unit=SQ_KM'). See Section 2.6 for more information about unit of measurement specification.

If this parameter is not specified, the unit of measurement associated with the data is assumed. For geodetic data, the default unit of measurement is square meters.

tol
Tolerance value (see Section 1.5.5).

Usage Notes

This function works with any polygon, including polygons with holes.

This function does not support the *units* parameter that is included with the LOCATOR_WITHIN_DISTANCE operator of *inter*Media Locator, which is a component of the Oracle *inter*Media product.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

Examples

The following example returns the areas of geometry objects stored in the COLA_MARKETS table. The first statement returns the areas all objects; the second returns just the area of *cola_a*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the areas of all cola markets.
SELECT name, SDO_GEOM.SDO_AREA(shape, 0.005) FROM cola_markets;

NAME                           SDO_GEOM.SDO_AREA(SHAPE,0.005)
------------------------------ ------------------------------
cola_a                                                     24
cola_b                                                   16.5
cola_c                                                      5
cola_d                                               12.5663706

-- Return the area of just cola_a.
SELECT c.name, SDO_GEOM.SDO_AREA(c.shape, 0.005) FROM cola_markets c
   WHERE c.name = 'cola_a';

NAME                           SDO_GEOM.SDO_AREA(C.SHAPE,0.005)
------------------------------ --------------------------------
cola_a                                                       24
```

Related Topics

None.

SDO_GEOM.SDO_BUFFER

Format

```
SDO_GEOM.SDO_BUFFER(
    geom       IN MDSYS.SDO_GEOMETRY,
    dim        IN MDSYS.SDO_DIM_ARRAY,
    dist       IN NUMBER
    [, params  IN VARCHAR2]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_BUFFER(
    geom       IN MDSYS.SDO_GEOMETRY,
    dist       IN NUMBER,
    tol        IN NUMBER
    [, params  IN VARCHAR2]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Generates a buffer polygon around a geometry object.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

dist
Euclidean distance value. Must be greater than the tolerance value, as specified in the dimensional array (*dim* parameter) or in the *tol* parameter.

tol
Tolerance value (see Section 1.5.5).

params
A quoted string with one or both of the following keywords:

- *unit* and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table. See Section 2.6 for more information about unit of measurement specification.
- *arc_tolerance* and an arc tolerance value. See the Usage Notes for the SDO_GEOM.SDO_ARC_DENSIFY function in this chapter for more information about the *arc_tolerance* keyword.

For example: 'unit=km arc_tolerance=0.05'

If the input geometry is geodetic data, this parameter is required, and *arc_tolerance* must be specified. If the input geometry is Cartesian or projected data, *arc_tolerance* has no effect and should not be specified.

If this parameter is not specified for a Cartesian or projected geometry, or if the *arc_tolerance* keyword is specified for a geodetic geometry but the *unit* keyword is not specified, the unit of measurement associated with the data is assumed.

Usage Notes

This function returns a geometry object representing the buffer polygon.

This function creates a rounded buffer around a point, line, or polygon. The buffer within a void is also rounded, and is the same distance from the inner boundary as the outer buffer is from the outer boundary. See Figure 1–11 for an illustration.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

With geodetic data, this function is supported by approximations, as explained in Section 5.7.2.

This function does not support the *units* parameter that is included with the LOCATOR_WITHIN_DISTANCE operator of *inter*Media Locator, which is a component of the Oracle *inter*Media product.

Examples

The following example returns a polygon representing a buffer of 1 around *cola_a*. Note the "rounded" corners (for example, at .292893219,.292893219) in the returned polygon. (The example uses the non-geodetic definitions and data from Section 2.1.)

```
-- Generate a buffer of 1 unit around a geometry.
SELECT c.name, SDO_GEOM.SDO_BUFFER(c.shape, m.diminfo, 1)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS'  AND m.column_name = 'SHAPE'
  AND c.name = 'cola_a';

NAME
------------------------------
SDO_GEOM.SDO_BUFFER(C.SHAPE,M.DIMINFO,1)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z)
--------------------------------------------------------------------------------
cola_a
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1005, 8, 1, 2, 2, 5, 2, 1,
 7, 2, 2, 11, 2, 1, 13, 2, 2, 17, 2, 1, 19, 2, 2, 23, 2, 1), SDO_ORDINATE_ARRAY(
0, 1, .292893219, .292893219, 1, 0, 5, 0, 5.70710678, .292893219, 6, 1, 6, 7, 5.
70710678, 7.70710678, 5, 8, 1, 8, .292893219, 7.70710678, 0, 7, 0, 1))
```

The following example returns a polygon representing a buffer of 1 around *cola_a* using the geodetic definitions and data from Section 5.8.

```
-- Generate a buffer of 1 kilometer around a geometry.
SELECT c.name, SDO_GEOM.SDO_BUFFER(c.shape, m.diminfo, 1,
                                   'unit=km arc_tolerance=0.05')
 FROM cola_markets c, user_sdo_geom_metadata m
 WHERE m.table_name = 'COLA_MARKETS'
 AND m.column_name = 'SHAPE' AND c.name = 'cola_a';

NAME
------------------------------
SDO_GEOM.SDO_BUFFER(C.SHAPE,M.DIMINFO,1,'UNIT=KMARC_TOLERANCE=0.05')(SDO_GTYPE,
--------------------------------------------------------------------------------
cola_a
SDO_GEOMETRY(2003, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(.991023822, 1.00002073, .992223711, .995486419, .99551726, .99217077, 1.00001
929, .990964898, 4.99998067, .990964929, 5.00448268, .9921708, 5.00777624, .9954
86449, 5.00897618, 1.00002076, 5.00904194, 6.99997941, 5.00784065, 7.00450033, 5
.00454112, 7.00781357, 5.00002479, 7.009034, .999975166, 7.00903403, .995458814,
 7.00781359, .992159303, 7.00450036, .990958058, 6.99997944, .991023822, 1.00002
073))
```

Related Topics

- SDO_TUNE.EXTENT_OF
- SDO_GEOM.SDO_UNION
- SDO_GEOM.SDO_INTERSECTION

- SDO_GEOM.SDO_UNION
- SDO_GEOM.SDO_XOR

SDO_GEOM.SDO_CENTROID

Format

```
SDO_GEOM.SDO_CENTROID(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_CENTROID(
    geom1  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a point geometry that is the centroid of a polygon, multipolygon, point, or point cluster. (The centroid is also known as the "center of gravity.")

For an input geometry consisting of multiple objects, the result is weighted by the area of each polygon in the geometry objects. If the geometry objects are a mixture of polygons and points, the points are not used in the calculation of the centroid. If the geometry objects are all points, the points have equal weight.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

The function returns a null value if *geom* is not a polygon, multipolygon, point, or point cluster.

If *geom1* is a point, the function returns the point (the input geometry).

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

With geodetic data, this function is supported by approximations, as explained in Section 5.7.2.

Examples

The following example returns a geometry object that is the centroid of *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the centroid of a geometry.
SELECT c.name, SDO_GEOM.SDO_CENTROID(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

NAME
--------------------------------
SDO_GEOM.SDO_CENTROID(C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z)
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
4.73333333, 3.93333333))
```

Related Topics

None.

SDO_GEOM.SDO_CONVEXHULL

Format

```
SDO_GEOM.SDO_CONVEXHULL(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_CONVEXHULL(
    geom1  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a polygon-type object that represents the convex hull of a geometry object.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

The **convex hull** is a simple convex polygon that completely encloses the geometry object. Spatial uses as few straight-line sides as possible to create the smallest polygon that completely encloses the specified object. A convex hull is a convenient way to get an approximation of a complex geometry object.

If the geometry (*geom1*) contains any arc elements, the function calculates the minimum bounding rectangle (MBR) for each arc element and uses these MBRs in calculating the convex hull of the geometry. If the geometry object (*geom1*) is a circle, the function returns a square that minimally encloses the circle.

The function returns the original (input) geometry if *geom* is of point type, has fewer than three points or vertices, or consists of multiple points all in a straight line.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

With geodetic data, this function is supported by approximations, as explained in Section 5.7.2.

Examples

The following example returns a geometry object that is the convex hull of *cola_c*. (The example uses the definitions and data from Section 2.1. This specific example, however, does not produce useful output -- the returned polygon is identical to the input polygon -- because the input polygon is already a simple convex polygon.)

```
-- Return the convex hull of a polygon.
SELECT c.name, SDO_GEOM.SDO_CONVEXHULL(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

NAME
--------------------------------
SDO_GEOM.SDO_CONVEXHULL(C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y,
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(6, 3, 6, 5, 4, 5, 3, 3, 6, 3))
```

Related Topics

None.

SDO_GEOM.SDO_DIFFERENCE

Format

```
SDO_GEOM.SDO_DIFFERENCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_DIFFERENCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the topological difference (*MINUS* operation) of two geometry objects.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

geom2
Geometry object.

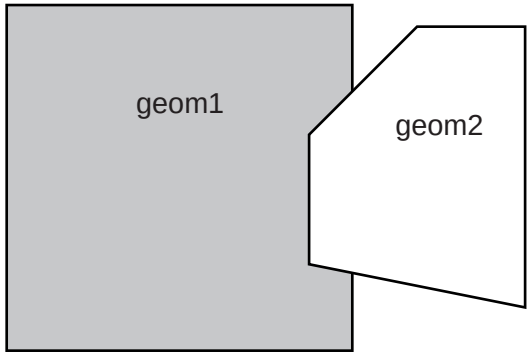
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

In Figure 12–2, the shaded area represents the polygon returned when SDO_DIFFERENCE is used with a square (*geom1*) and another polygon (*geom2*).

Figure 12–2 SDO_GEOM.SDO_DIFFERENCE



If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

You should use this function instead of the deprecated function SDO_GEOM.SDO_POLY_DIFFERENCE.

Examples

The following example returns a geometry object that is the topological difference (MINUS operation) of *cola_a* and *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the topological difference of two geometries.
SELECT SDO_GEOM.SDO_DIFFERENCE(c_a.shape, m.diminfo, c_c.shape, m.diminfo)
  FROM cola_markets c_a, cola_markets c_c, user_sdo_geom_metadata m
```

```
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c_a.name = 'cola_a' AND c_c.name = 'cola_c';

SDO_GEOM.SDO_DIFFERENCE(C_A.SHAPE,M.DIMINFO,C_C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(1, 7, 1, 1, 5, 1, 5, 3, 3, 3, 4, 5, 5, 5, 5, 7, 1, 7)
```

Note that in the returned polygon, the SDO_ORDINATE_ARRAY starts and ends at the same point (1, 7).

Related Topics

- SDO_GEOM.SDO_INTERSECTION
- SDO_GEOM.SDO_UNION
- SDO_GEOM.SDO_XOR

SDO_GEOM.SDO_DISTANCE

Format

```
SDO_GEOM.SDO_DISTANCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    [, unit  IN VARCHAR2]
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_DISTANCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    [, unit   IN VARCHAR2]
    ) RETURN NUMBER;
```

Description

Computes the distance between two geometry objects. The distance between two geometry objects is the distance between the closest pair of points or segments of the two objects.

Parameters

geom1
Geometry object whose distance from *geom2* is to be computed.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

geom2
Geometry object whose distance from *geom1* is to be computed.

dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

unit
Unit of measurement: a quoted string with *unit*= and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, 'unit=KM'). See Section 2.6 for more information about unit of measurement specification.

If this parameter is not specified, the unit of measurement associated with the data is assumed.

tol
Tolerance value (see Section 1.5.5).

Usage Notes

This function does not support the *units* parameter that is included with the LOCATOR_WITHIN_DISTANCE operator of *inter*Media Locator, which is a component of the Oracle *inter*Media product.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

Examples

The following example returns the shortest distance between *cola_b* and *cola_d*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the distance between two geometries.
SELECT SDO_GEOM.SDO_DISTANCE(c_b.shape, c_d.shape, 0.005)
   FROM cola_markets c_b, cola_markets c_d
   WHERE c_b.name = 'cola_b' AND c_d.name = 'cola_d';

SDO_GEOM.SDO_DISTANCE(C_B.SHAPE,C_D.SHAPE,0.005)
------------------------------------------------
                                     .846049894
```

Related Topics

- SDO_GEOM.WITHIN_DISTANCE

SDO_GEOM.SDO_INTERSECTION

Format

```
SDO_GEOM.SDO_INTERSECTION(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_INTERSECTION(
    geom1  IN MDSYS.SDO_GEOMETRY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the topological intersection (*AND* operation) of two geometry objects.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

geom2
Geometry object.

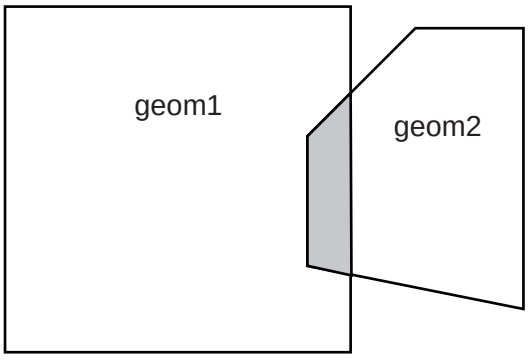
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

In Figure 12–3, the shaded area represents the polygon returned when SDO_INTERSECTION is used with a square (*geom1*) and another polygon (*geom2*).

Figure 12–3 SDO_GEOM.SDO_INTERSECTION



If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

You should use this function instead of the deprecated function SDO_GEOM.SDO_POLY_INTERSECTION.

Examples

The following example returns a geometry object that is the topological intersection (AND operation) of *cola_a* and *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the topological intersection of two geometries.
SELECT SDO_GEOM.SDO_INTERSECTION(c_a.shape, c_c.shape, 0.005)
   FROM cola_markets c_a, cola_markets c_c
```

```
   WHERE c_a.name = 'cola_a' AND c_c.name = 'cola_c';

SDO_GEOM.SDO_INTERSECTION(C_A.SHAPE,C_C.SHAPE,0.005)(SDO_GTYPE, SDO_SRID, SDO_PO
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(4, 5, 3, 3, 5, 3, 5, 5, 4, 5))
```

Note that in the returned polygon, the SDO_ORDINATE_ARRAY starts and ends at the same point (4, 5).

Related Topics

- SDO_GEOM.SDO_DIFFERENCE
- SDO_GEOM.SDO_UNION
- SDO_GEOM.SDO_XOR

SDO_GEOM.SDO_LENGTH

Format

```
SDO_GEOM.SDO_LENGTH(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    [, unit  IN VARCHAR2]
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_LENGTH(
    geom  IN MDSYS.SDO_GEOMETRY,
    tol   IN NUMBER
    [, unit  IN VARCHAR2]
    ) RETURN NUMBER;
```

Description

Returns the length or perimeter of a geometry object.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

unit
Unit of measurement: a quoted string with *unit*= and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table (for example, 'unit=KM'). See Section 2.6 for more information about unit of measurement specification.

If this parameter is not specified, the unit of measurement associated with the data is assumed. For geodetic data, the default unit of measurement is meters.

tol
Tolerance value (see Section 1.5.5).

Usage Notes

If the input polygon contains one or more holes, this function calculates the perimeters of the exterior boundary and all holes. It returns the sum of all the perimeters.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

This function does not support the *units* parameter that is included with the LOCATOR_WITHIN_DISTANCE operator of *inter*Media Locator, which is a component of the Oracle *inter*Media product.

Examples

The following example returns the perimeters of geometry objects stored in the COLA_MARKETS table. The first statement returns the perimeters of all objects; the second returns just the perimeter of *cola_a*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the perimeters of all cola markets.
SELECT c.name, SDO_GEOM.SDO_LENGTH(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE';

NAME                           SDO_GEOM.SDO_LENGTH(C.SHAPE,M.DIMINFO)
------------------------------ --------------------------------------
cola_a                                                             20
cola_b                                                      17.1622777
cola_c                                                      9.23606798
cola_d                                                      12.5663706

-- Return the perimeter of just cola_a.
SELECT c.name, SDO_GEOM.SDO_LENGTH(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_a';

NAME                           SDO_GEOM.SDO_LENGTH(C.SHAPE,M.DIMINFO)
------------------------------ --------------------------------------
cola_a                                                             20
```

Related Topics

None.

SDO_GEOM.SDO_MAX_MBR_ORDINATE

Format

```
SDO_GEOM.SDO_MAX_MBR_ORDINATE(
    geom          IN MDSYS.SDO_GEOMETRY,
    ordinate_pos  IN NUMBER
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_MAX_MBR_ORDINATE(
    geom          IN MDSYS.SDO_GEOMETRY,
    dim           IN MDSYS.SDO_DIM_ARRAY,
    ordinate_pos  IN NUMBER
    ) RETURN NUMBER;
```

Description

Returns the maximum value for the specified ordinate of the minimum bounding rectangle of a geometry object.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

ordinate_pos
Position of the ordinate in the definition of the geometry object: 1 for the first ordinate, 2 for the second ordinate, and so on. For example, if *geom* has X, Y ordinates, 1 identifies the X ordinate and 2 identifies the Y ordinate.

Usage Notes

This function is not supported with geodetic data.

Examples

The following example returns the maximum X (first) ordinate value of the minimum bounding rectangle of the *cola_d* geometry in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1. The minimum bounding rectangle of *cola_d* is returned in the example for the SDO_GEOM.SDO_MBR function.)

```
SELECT SDO_GEOM.SDO_MAX_MBR_ORDINATE(c.shape, m.diminfo, 1)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_d';

SDO_GEOM.SDO_MAX_MBR_ORDINATE(C.SHAPE,M.DIMINFO,1)
--------------------------------------------------
                                                10
```

Related Topics

- SDO_GEOM.SDO_MBR
- SDO_GEOM.SDO_MIN_MBR_ORDINATE

SDO_GEOM.SDO_MBR

Format

```
SDO_GEOM.SDO_MBR(
    geom   IN MDSYS.SDO_GEOMETRY
    [, dim   IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the minimum bounding rectangle of a geometry object, that is, a single rectangle that minimally encloses the geometry.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

Usage Notes

This function is not supported with geodetic data.

Examples

The following example returns the minimum bounding rectangle of the *cola_d* geometry in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1. Because *cola_d* is a circle, the minimum bounding rectangle in this case is a square.)

```
-- Return the minimum bounding rectangle of cola_d (a circle).
SELECT SDO_GEOM.SDO_MBR(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_d';

SDO_GEOM.SDO_MBR(C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO
```

```
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3), SDO_ORDINATE_ARR
AY(6, 7, 10, 11))
```

Related Topics

- SDO_GEOM.SDO_MAX_MBR_ORDINATE
- SDO_GEOM.SDO_MIN_MBR_ORDINATE

SDO_GEOM.SDO_MIN_MBR_ORDINATE

Format

```
SDO_GEOM.SDO_MIN_MBR_ORDINATE(
    geom          IN MDSYS.SDO_GEOMETRY,
    ordinate_pos  IN NUMBER
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_MIN_MBR_ORDINATE(
    geom          IN MDSYS.SDO_GEOMETRY,
    dim           IN MDSYS.SDO_DIM_ARRAY,
    ordinate_pos  IN NUMBER
    ) RETURN NUMBER;
```

Description

Returns the minimum value for the specified ordinate of the minimum bounding rectangle of a geometry object.

Parameters

geom
Geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

ordinate_pos
Position of the ordinate in the definition of the geometry object: 1 for the first ordinate, 2 for the second ordinate, and so on. For example, if *geom* has X, Y ordinates, 1 identifies the X ordinate and 2 identifies the Y ordinate.

Usage Notes

This function is not supported with geodetic data.

Examples

The following example returns the minimum X (first) ordinate value of the minimum bounding rectangle of the *cola_d* geometry in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1. The minimum bounding rectangle of *cola_d* is returned in the example for the SDO_GEOM.SDO_MBR function.)

```
SELECT SDO_GEOM.SDO_MIN_MBR_ORDINATE(c.shape, m.diminfo, 1)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_d';

SDO_GEOM.SDO_MIN_MBR_ORDINATE(C.SHAPE,M.DIMINFO,1)
--------------------------------------------------
                                                 6
```

Related Topics

- SDO_GEOM.SDO_MAX_MBR_ORDINATE
- SDO_GEOM.SDO_MBR

SDO_GEOM.SDO_POINTONSURFACE

Format

```
SDO_GEOM.SDO_POINTONSURFACE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_POINTONSURFACE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a point that is guaranteed to be on the surface of a polygon geometry object.

Parameters

geom1
Polygon geometry object.

dim1
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

This function returns a point geometry object representing a point that is guaranteed to be on the surface of *geom1*.

The returned point can be any point on the surface. You should not make any assumptions about where on the surface the returned point is, or about whether the point is the same or different when the function is called multiple times with the same input parameter values.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

Examples

The following example returns a geometry object that is a point on the surface of *cola_a*. (The example uses the definitions and data from Section 2.1.)

```
-- Return a point on the surface of a geometry.
SELECT SDO_GEOM.SDO_POINTONSURFACE(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_a';

SDO_GEOM.SDO_POINTONSURFACE(C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
1, 1))
```

Related Topics

None.

SDO_GEOM.SDO_UNION

Format

```
SDO_GEOM.SDO_UNION(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_UNION(
    geom1  IN MDSYS.SDO_GEOMETRY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the topological union (*OR* operation) of two geometry objects.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

geom2
Geometry object.

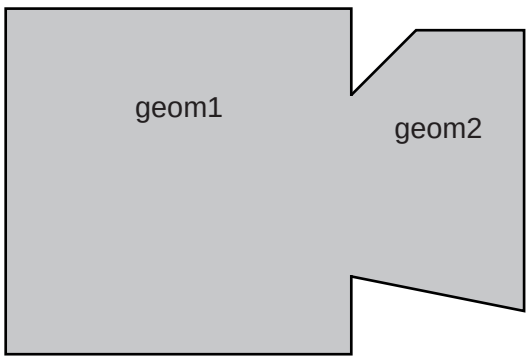
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

In Figure 12–4, the shaded area represents the polygon returned when SDO_UNION is used with a square (*geom1*) and another polygon (*geom2*).

Figure 12–4 SDO_GEOM.SDO_UNION



If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

You should use this function instead of the deprecated function SDO_GEOM.SDO_POLY_UNION.

Examples

The following example returns a geometry object that is the topological union (OR operation) of *cola_a* and *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the topological intersection of two geometries.
SELECT SDO_GEOM.SDO_UNION(c_a.shape, m.diminfo, c_c.shape, m.diminfo)
  FROM cola_markets c_a, cola_markets c_c, user_sdo_geom_metadata m
```

```
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c_a.name = 'cola_a' AND c_c.name = 'cola_c';

SDO_GEOM.SDO_UNION(C_A.SHAPE,M.DIMINFO,C_C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID,
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(5, 5, 5, 7, 1, 7, 1, 1, 5, 1, 5, 3, 6, 3, 6, 5, 5, 5))
```

Note that in the returned polygon, the SDO_ORDINATE_ARRAY starts and ends at the same point (5, 5).

Related Topics

- SDO_GEOM.SDO_DIFFERENCE
- SDO_GEOM.SDO_INTERSECTION
- SDO_GEOM.SDO_XOR

SDO_GEOM.SDO_XOR

Format

```
SDO_GEOM.SDO_XOR(
    geom1  IN MDSYS.SDO_XOR,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_XOR(
    geom1  IN MDSYS.SDO_GEOMETRY,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the topological symmetric difference (*XOR* operation) of two geometry objects.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

geom2
Geometry object.

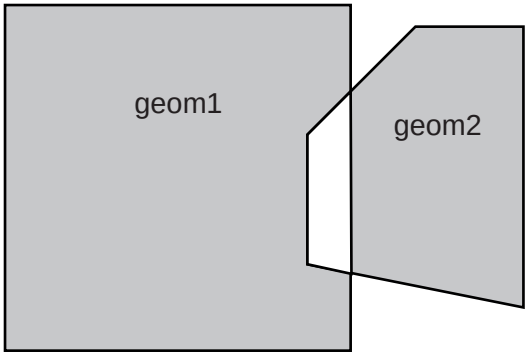
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

In Figure 12–5, the shaded area represents the polygon returned when SDO_XOR is used with a square (*geom1*) and another polygon (*geom2*).

Figure 12–5 SDO_GEOM.SDO_XOR



If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

You should use this function instead of the deprecated function SDO_GEOM.SDO_POLY_XOR.

Examples

The following example returns a geometry object that is the topological symmetric difference (XOR operation) of *cola_a* and *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Return the topological symmetric difference of two geometries.
SELECT SDO_GEOM.SDO_XOR(c_a.shape, m.diminfo, c_c.shape, m.diminfo)
```

```
  FROM cola_markets c_a, cola_markets c_c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c_a.name = 'cola_a' AND c_c.name = 'cola_c';

SDO_GEOM.SDO_XOR(C_A.SHAPE,M.DIMINFO,C_C.SHAPE,M.DIMINFO)(SDO_GTYPE, SDO_SRID, S
--------------------------------------------------------------------------------
SDO_GEOMETRY(2007, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1, 19, 1003, 1), SDO
_ORDINATE_ARRAY(1, 7, 1, 1, 5, 1, 5, 3, 3, 3, 4, 5, 5, 5, 5, 7, 1, 7, 5, 5, 5, 3
, 6, 3, 6, 5, 5, 5))
```

Note that in the returned polygon is a multipolygon (SDO_GTYPE = 2007), and the SDO_ORDINATE_ARRAY describes two polygons: one starting and ending at (1, 7) and the other starting and ending at (5, 5).

Related Topics

- SDO_GEOM.SDO_DIFFERENCE
- SDO_GEOM.SDO_INTERSECTION
- SDO_GEOM.SDO_UNION

SDO_GEOM.VALIDATE_GEOMETRY

Format

```
SDO_GEOM.VALIDATE_GEOMETRY(
    theGeometry  IN MDSYS.SDO_GEOMETRY,
    theDimInfo     IN MDSYS.SDO_DIM_ARRAY
    ) RETURN VARCHAR2;
```

or

```
SDO_GEOM.VALIDATE_GEOMETRY(
    theGeometry  IN MDSYS.SDO_GEOMETRY,
    tolerance       IN NUMBER
    ) RETURN VARCHAR2;
```

Description

Performs a consistency check for valid geometry types. The function checks the representation of the geometry from the tables against the element definitions.

Parameters

theGeometry
Geometry object.

theDimInfo
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

If the geometry is valid, this function returns TRUE.

If the geometry is not valid, this function returns one of the following:

- An Oracle error message number based on the specific reason the geometry is invalid
- FALSE if the geometry fails for some other reason

This function checks for type consistency and geometry consistency.

For type consistency, the function checks for the following:

- The SDO_GTYPE is valid.
- The SDO_ETYPE values are consistent with the SDO_GTYPE value. For example, if the SDO_GTYPE is 2003, there should be at least one element of type POLYGON in the geometry.
- The SDO_ELEM_INFO_ARRAY has valid triplet values.

For geometry consistency, the function checks for the following, as appropriate for the specific geometry type:

- Polygons have at least four points, which includes the point that closes the polygon. (The last point is the same as the first.)
- Polygons are not self-crossing.
- No two vertices on a line or polygon are the same.
- Polygons are oriented correctly. (Exterior ring boundaries must be oriented counterclockwise, and interior ring boundaries must be oriented clockwise.)
- An interior polygon ring touches the exterior polygon ring at no more than one point.
- If two or more interior polygon rings are in an exterior polygon ring, the interior polygon rings touch at no more than one point.
- Line strings have at least two points.
- 1-digit and 4-digit SDO_ETYPE values are not mixed (that is, both used) in defining polygon ring elements.
- Points on an arc are not colinear (that is, are not on a straight line) and are not the same point.
- Geometries are within the specified bounds of the applicable DIMINFO column value (from the USER_SDO_GEOM_METADATA view).
- LRS geometries (see Chapter 6) have 3 or 4 dimensions and a valid measure dimension position (3 or 4, depending on the number of dimensions).
- Geometries are within the extent of the coordinate system.

In checking for geometry consistency, the function considers the geometry's tolerance value in determining if lines touch or if points are the same.

If the function format with *tolerance* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

You can use this function in a PL/SQL procedure as an alternative to using the SDO_GEOM.VALIDATE_LAYER procedure. See the Usage Notes for SDO_GEOM.VALIDATE_LAYER for more information.

Examples

The following example validates the geometry of *cola_c*. (The example uses the definitions and data from Section 2.1.)

```
-- Is a geometry valid?
SELECT c.name, SDO_GEOM.VALIDATE_GEOMETRY(c.shape, 0.005)
   FROM cola_markets c WHERE c.name = 'cola_c';

NAME
------------------------------
SDO_GEOM.VALIDATE_GEOMETRY(C.SHAPE,0.005)
--------------------------------------------------------------------------------
cola_c
TRUE
```

Related Topics

- SDO_GEOM.VALIDATE_LAYER

SDO_GEOM.VALIDATE_LAYER

Format

```
SDO_GEOM.VALIDATE_LAYER(
    geom_table      IN VARCHAR2,
    geom_column  IN VARCHAR2,
    pkey_column     IN VARCHAR2,
    result_table      IN VARCHAR2
    [, commit_interval  IN NUMBER]);
```

Description

Examines a geometry column to determine if the stored geometries follow the defined rules for geometry objects.

Parameters

geom_table
Geometry table.

geom_column
Geometry object column to be examined.

pkey_column
The primary key column. This must be a single numeric (NUMBER data type) column.

result_table
Result table to hold the validation results. A row is added to *result_table* for each invalid geometry. If there are no invalid geometries, one or more (depending on the *commit_interval* value) rows with a result of DONE are added.

commit_interval
Number of geometries to validate before Spatial performs an internal commit operation and writes a row with a result of DONE to *result_table* (if no rows for invalid geometries have been written since the last commit operation). If *commit_*

interval is not specified, no internal commit operations are performed during the validation.

The *commit_interval* option is helpful if you want to look at the contents of *result_table* while the validation is in progress. If the primary key is indexed, you can look at the last PKEY_COLUMN value to see approximately how much of the validation is completed.

Usage Notes

This procedure loads the result table with validation results. (VALIDATE_LAYER is a procedure, not a function.)

An empty result table (*result_table* parameter) must be created before calling this procedure. The format of the result table is: (pkey_column NUMBER, result VARCHAR2(10)). If *result_table* is not empty, you should truncate the table before calling the procedure; otherwise, the procedure appends rows to the existing data in the table.

The result table contains one row for each invalid geometry. A row is not written if a geometry is valid, except as follows:

- If *commit_interval* is not specified (or if the *commit_interval* value is greater than the number of geometries in the layer) and no invalid geometries are found, a single row with a RESULT value of DONE is written.
- If *commit_interval* is specified and if no invalid geometries are found between an internal commit and the previous internal commit (or start of validation for the first internal commit), a single row with the primary key of the last geometry validated and a RESULT value of DONE is written. (If there have been no invalid geometries since the last internal commit operation, this row replaces the previous row that had a result of DONE.)

In each row for an invalid geometry, the PKEY_COLUMN column contains the primary key value of the row containing the invalid geometry, and the RESULT column contains an Oracle error message number. You can then look up this error message to determine the cause of the failure.

This procedure performs the following checks on each geometry in the layer (*geom_column*):

- All the type consistency and geometry consistency checks that are performed by the SDO_GEOM.VALIDATE_GEOMETRY function (see Usage Notes for that function).

- If 4-digit SDO_GTYPE values are used, the geometry's SDO_GTYPE specifies the same dimensionality as specified in the applicable DIMINFO column value (from the USER_SDO_GEOM_METADATA view).
- The geometry's SRID value (coordinate system) is the same as the one specified in the applicable DIMINFO column value (from the USER_SDO_GEOM_METADATA view).

As an alternative to using the VALIDATE_LAYER procedure, you can use a PL/SQL procedure that calls the SDO_GEOM.VALIDATE_GEOMETRY function for each input row and inserts rows in a result table for errors. With this approach, you have more options in defining the result table; for example, the key field can be a rowid or anything else that you choose.

Examples

The following example validates the geometry objects stored in the SHAPE column of the COLA_MARKETS table. The example includes the creation of the result table. (The example uses the definitions and data from Section 2.1.) In this case, a row with a RESULT value of DONE is written to the result table, because all the geometries are valid.

```
-- Is a layer valid? (First, create the result table.)
CREATE TABLE val_results (mkt_id number, result varchar2(10));
EXECUTE SDO_GEOM.VALIDATE_LAYER('COLA_MARKETS','SHAPE','MKT_ID','VAL_RESULTS');

PL/SQL procedure successfully completed.

SELECT * from val_results;

    MKT_ID RESULT
---------- ----------
           DONE
```

Related Topics

- SDO_GEOM.VALIDATE_GEOMETRY

SDO_GEOM.WITHIN_DISTANCE

Format

```
SDO_GEOM.WITHIN_DISTANCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dim1   IN MDSYS.SDO_DIM_ARRAY,
    dist   IN NUMBER,
    geom2  IN MDSYS.SDO_GEOMETRY,
    dim2   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN VARCHAR2;
```

or

```
SDO_GEOM.WITHIN_DISTANCE(
    geom1  IN MDSYS.SDO_GEOMETRY,
    dist   IN NUMBER,
    geom2  IN MDSYS.SDO_GEOMETRY,
    tol    IN NUMBER
    ) RETURN VARCHAR2;
```

Description

Determines if two spatial objects are within some specified Euclidean distance from each other.

Parameters

geom1
Geometry object.

dim1
Dimensional information array corresponding to *geom1*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

dist
Euclidean distance value.

geom2
Geometry object.

dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tol
Tolerance value (see Section 1.5.5).

Usage Notes

This function returns TRUE for object pairs that are within the specified distance, and FALSE otherwise.

The distance between two extended objects (for example, nonpoint objects such as lines and polygons) is defined as the minimum distance between these two objects. Thus the distance between two adjacent polygons is zero.

If the function format with *tol* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

An exception is raised if *geom1* and *geom2* are based on different coordinate systems.

Examples

The following example checks if *cola_b* and *cola_d* are within 1 unit apart at the shortest distance between them. (The example uses the definitions and data from Section 2.1.)

```
-- Are two geometries within 1 unit of distance apart?
SELECT SDO_GEOM.WITHIN_DISTANCE(c_b.shape, m.diminfo, 1,
     c_d.shape, m.diminfo)
  FROM cola_markets c_b, cola_markets c_d, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c_b.name = 'cola_b' AND c_d.name = 'cola_d';

SDO_GEOM.WITHIN_DISTANCE(C_B.SHAPE,M.DIMINFO,1,C_D.SHAPE,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE
```

Related Topics

- SDO_GEOM.SDO_DISTANCE

13

Spatial Aggregate Functions

This chapter contains reference and usage information for the spatial aggregate functions, which are listed in Table 13–1.

Table 13–1 Spatial Aggregate Functions

Method	Description
SDO_AGGR_CENTROID	Returns a geometry object that is the centroid ("center of gravity") of the specified geometry objects.
SDO_AGGR_CONVEXHULL	Returns a geometry object that is the convex hull of the specified geometry objects.
SDO_AGGR_LRS_CONCAT	Returns an LRS geometry object that concatenates specified LRS geometry objects.
SDO_AGGR_MBR	Returns the minimum bounding rectangle of the specified geometry objects
SDO_AGGR_UNION	Returns a geometry object that is the topological union (*OR* operation) of the specified geometry objects.

See the usage information about spatial aggregate functions in Section 1.9.

Most of these aggregate functions accept a parameter of type MDSYS.SDOAGGRTYPE, which is described in Section 1.9.1.

SDO_AGGR_CENTROID

Format

```
SDO_AGGR_CENTROID(
    AggregateGeometry  MDSYS.SDOAGGRTYPE
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the centroid ("center of gravity") of the specified geometry objects.

Parameters

AggregateGeometry
An object of type MDSYS.SDOAGGRTYPE (see Section 1.9.1) that specifies the geometry column and dimensional array.

Usage Notes

The behavior of the function depends on whether the geometry objects are all polygons, all points, or a mixture of polygons and points:

- If the geometry objects are all polygons, the centroid of all the objects is returned.
- If the geometry objects are all points, the centroid of all the objects is returned.
- If the geometry objects are a mixture of polygons and points (specifically, if they include at least one polygon and at least one point), any points are ignored, and the centroid of all the polygons is returned.

The result is weighted by the area of each polygon in the geometry objects. If the geometry objects are a mixture of polygons and points, the points are not used in the calculation of the centroid. If the geometry objects are all points, the points have equal weight.

See also the information about the SDO_GEOM.SDO_CENTROID function in Chapter 13.

Examples

The following example returns the centroid of the geometry objects in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1.)

```
SELECT SDO_AGGR_CENTROID(MDSYS.SDOAGGRTYPE(shape, 0.005))
  FROM cola_markets;

SDO_AGGR_CENTROID(MDSYS.SDOAGGRTYPE(SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POINT
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(5.21295938, 5.00744233, NULL), NULL, NUL
L)
```

SDO_AGGR_CONVEXHULL

Format

```
SDO_AGGR_CONVEXHULL(
    AggregateGeometry  MDSYS.SDOAGGRTYPE
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the convex hull of the specified geometry objects.

Parameters

AggregateGeometry
An object of type MDSYS.SDOAGGRTYPE (see Section 1.9.1) that specifies the geometry column and dimensional array.

Usage Notes

See also the information about the SDO_GEOM.SDO_CONVEXHULL in Chapter 13.

Examples

The following example returns the convex hull of the geometry objects in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1.)

```
SELECT SDO_AGGR_CONVEXHULL(MDSYS.SDOAGGRTYPE(shape, 0.005))
  FROM cola_markets;

SDO_AGGR_CONVEXHULL(MDSYS.SDOAGGRTYPE(SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POI
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(8, 1, 10, 7, 10, 11, 8, 11, 6, 11, 1, 7, 1, 1, 8, 1))
```

SDO_AGGR_LRS_CONCAT

Format

```
SDO_AGGR_LRS_CONCAT(
    AggregateGeometry  MDSYS.SDOAGGRTYPE
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns an LRS geometry that concatenates specified LRS geometries.

Parameters

AggregateGeometry
An object of type MDSYS.SDOAGGRTYPE (see Section 1.9.1) that specifies the geometry column and dimensional array.

Usage Notes

This function performs an aggregate concatenation of any number of LRS geometries. If you want to control the order in which the geometries are concatenated, you must use a subquery with the NO_MERGE optimizer hint and the ORDER BY clause. (See the examples.)

The direction of the resulting segment is the same as the direction of the first geometry in the concatenation.

A *3D* format of this function (SDO_AGGR_LRS_CONCAT_3D) is available. For information about *3D* formats of LRS functions, see Section 6.4.)

For information about the Spatial linear referencing system, see Chapter 6.

Examples

The following example adds an LRS geometry to the LRS_ROUTES table, and then performs two queries that concatenate the LRS geometries in the table. The first query does not control the order of concatenation, and the second query controls the order of concatenation. Notice the difference in direction of the two segments: the segment resulting from the second query has decreasing measure values

because the first segment in the concatenation (*Route0*) has decreasing measure values. (This example uses the definitions from the example in Section 6.6.)

```
-- Add a segment with route_id less than 1 (here, zero).
INSERT INTO lrs_routes VALUES(
  0,
  'Route0',
  MDSYS.SDO_GEOMETRY(
    3302,  -- line string, 3 dimensions (X,Y,M), 3rd is linear referencing
dimension
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,2,1), -- one line string, straight segments
    MDSYS.SDO_ORDINATE_ARRAY(
      5,14,5,   -- Starting point - 5 is measure from start.
      10,14,0)  -- Ending point - 0 measure (decreasing measure)
  )
);

1 row created.

-- Concatenate all routes (no ordering specified).
SELECT SDO_AGGR_LRS_CONCAT(MDSYS.SDOAGGRTYPE(route_geometry, 0.005))
    FROM lrs_routes;

SDO_AGGR_LRS_CONCAT(MDSYS.SDOAGGRTYPE(ROUTE_GEOMETRY,0.005))(SDO_GTYPE, SDO_SRID
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27, 10, 14,
32))

-- Aggregate concatenation using subquery for ordering.
SELECT
SDO_AGGR_LRS_CONCAT(MDSYS.SDOAGGRTYPE(route_geometry, 0.005))
FROM (
             SELECT /*+ NO_MERGE */ route_geometry
             FROM lrs_routes
             ORDER BY route_id);

SDO_AGGR_LRS_CONCAT(MDSYS.SDOAGGRTYPE(ROUTE_GEOMETRY,0.005))(SDO_GTYPE, SDO_SRID
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 32, 2, 4, 30, 8, 4, 24, 12, 4, 20, 12, 10, 14, 8, 10, 10, 5, 14, 5, 10, 14
, 0))
```

SDO_AGGR_MBR

Format

```
SDO_AGGR_MBR(
    geom  MDSYS.SDO_GEOMETRY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the minimum bounding rectangle (MBR) of the specified geometries, that is, a single rectangle that minimally encloses the geometries.

Parameters

geom
Geometry objects.

Usage Notes

Use this function instead of the deprecated SDO_TUNE.EXTENT_OF function to return the MBR of geometries. The SDO_TUNE.EXTENT_OF function is limited to 2-dimensional geometries, whereas this function is not.

This function is not supported with geodetic data.

Examples

The following example returns the minimum bounding rectangle of the geometry objects in the COLA_MARKETS table. (The example uses the definitions and data from Section 2.1.)

```
SELECT SDO_AGGR_MBR(shape) FROM cola_markets;

SDO_AGGR_MBR(C.SHAPE)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SD
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3), SDO_ORDINATE_ARR
AY(1, 1, 10, 11))
```

SDO_AGGR_UNION

Format

```
SDO_AGGR_UNION(
    AggregateGeometry  MDSYS.SDOAGGRTYPE
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a geometry object that is the topological union (*OR* operation) of the specified geometry objects.

Parameters

AggregateGeometry
An object of type MDSYS.SDOAGGRTYPE (see Section 1.9.1) that specifies the geometry column and dimensional array.

Usage Notes

See also the information about the SDO_GEOM.SDO_UNION function in Chapter 13.

Examples

The following example returns the union of the first three geometry objects in the COLA_MARKETS table (that is, all except *cola_d*). (The example uses the definitions and data from Section 2.1.)

```
SELECT SDO_AGGR_UNION(
  MDSYS.SDOAGGRTYPE(c.shape, 0.005))
  FROM cola_markets c
  WHERE c.name < 'cola_d';

SDO_AGGR_UNION(MDSYS.SDOAGGRTYPE(C.SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POINT(
--------------------------------------------------------------------------------
SDO_GEOMETRY(2007, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 2, 11, 1003, 1), SDO
_ORDINATE_ARRAY(8, 11, 6, 9, 8, 7, 10, 9, 8, 11, 1, 7, 1, 1, 5, 1, 8, 1, 8, 6, 5
, 7, 1, 7))
```

14

Coordinate System Transformation Functions

The MDSYS.SDO_CS package contains functions and procedures for working with coordinate systems. You can perform explicit coordinate transformations on a single geometry or an entire layer of geometries (that is, all geometries in a specified column in a table).

To use the functions and procedures in this chapter, you must understand the conceptual information about coordinate systems in Section 1.5.4 and Chapter 5.

Table 14–1 lists the coordinate systems functions and procedures.

Table 14–1 Functions and Procedures for Coordinate Systems

Function	Description
SDO_CS.TRANSFORM	Transforms a geometry representation using a coordinate system (specified by SRID or name).
SDO_CS.TRANSFORM_LAYER	Transforms an entire layer of geometries (that is, all geometries in a specified column in a table).
SDO_CS.VIEWPORT_TRANSFORM	Transforms an optimized rectangle into a valid geodetic polygon for use with Spatial operators and functions.

The rest of this chapter provides reference information on the functions and procedures, listed in alphabetical order.

SDO_CS.TRANSFORM

Format

```
SDO_CS.TRANSFORM(
    geom    IN MDSYS.SDO_GEOMETRY,
    to_srid  IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
or
SDO_CS.TRANSFORM(
    geom    IN MDSYS.SDO_GEOMETRY,
    dim      IN MDSYS.SDO_DIM_ARRAY,
    to_srid  IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
or
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    to_srname  IN VARCHAR2
    ) RETURN MDSYS.SDO_GEOMETRY;
or
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    dim          IN MDSYS.SDO_DIM_ARRAY,
    to_srname  IN VARCHAR2
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Transforms a geometry representation using a coordinate system (specified by SRID or name).

Parameters

geom
Geometry whose representation is to be transformed using another coordinate system. The input geometry must have a valid non-null SRID, that is, a value in the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1).

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

to_srid
The SRID of the coordinate system to be used for the transformation. It must be a value in the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1).

to_srname
The name of the coordinate system to be used for the transformation. It must be a value (specified exactly) in the CS_NAME column of the MDSYS.CS_SRS table (described in Section 5.4.1).

Usage Notes

Transformation can be done only between two different georeferenced coordinate systems or between two different local coordinate systems.

An exception is raised if *geom*, *to_srid*, or *to_srname* is invalid. For *geom* to be valid for this function, its definition must include an SRID value matching a value in the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1).

Examples

The following example transforms the *cola_c* geometry to a representation that uses SRID value 8199. (This example uses the definitions from the example in Section 5.8.)

```
-- Return the transformation of cola_c using to_srid 8199
-- ('Longitude / Latitude (Arc 1950)')
SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 8199)
  FROM cola_markets_cs c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

NAME
--------------------------------
```

```
SDO_CS.TRANSFORM(C.SHAPE,M.DIMINFO,8199)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z)
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074114, 3.00291482, 6.00067068, 3.00291287, 6.0006723, 5.00307625, 4.0007
1961, 5.00307838, 3.00074114, 3.00291482))

-- Same as preceding, but using to_srname parameter.
SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo,
      'Longitude / Latitude (Arc 1950)')
  FROM cola_markets_cs c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS_CS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

NAME
--------------------------------
SDO_CS.TRANSFORM(C.SHAPE,M.DIMINFO,'LONGITUDE/LATITUDE(ARC1950)')(SDO_GTYPE, SDO
--------------------------------------------------------------------------------
cola_c
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074114, 3.00291482, 6.00067068, 3.00291287, 6.0006723, 5.00307625, 4.0007
1961, 5.00307838, 3.00074114, 3.00291482))
```

SDO_CS.TRANSFORM_LAYER

Format

```
SDO_CS.TRANSFORM_LAYER(
    table_in     IN VARCHAR2,
    column_in    IN VARCHAR2,
    table_out    IN VARCHAR2,
    to_srid      IN NUMBER);
```

Description

Transforms an entire layer of geometries (that is, all geometries in a specified column in a table).

Parameters

table_in
Table containing the layer (*column_in*) whose geometries are to be transformed.

column_in
Column in *table_in* that contains the geometries to be transformed.

table_out
Table that will be created and that will contain the results of the transformation. See the Usage Notes for information about the format of this table.

to_srid
The SRID of the coordinate system to be used for the transformation. *to_srid* must be a value in the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1).

Usage Notes

Transformation can be done only between two different georeferenced coordinate systems or between two different local coordinate systems.

An exception is raised if any of the following occurs:

- *table_in* does not exist, or *column_in* does not exist in the table.

- The geometries in *column_in* have a null or invalid SDO_SRID value.
- *table_out* already exists.
- *to_srid* is invalid.

The *table_out* table is created by the procedure and is filled with one row for each transformed geometry. This table has the columns shown in Table 14–2.

Table 14–2 Table to Hold Transformed Layer

Column Name	Data Type	Description
SDO_ROWID	ROWID	Oracle ROWID (row address identifier). For more information about the ROWID data type, see the *Oracle9i SQL Reference*.
GEOMETRY	MDSYS.SDO_GEOMETRY	Geometry object with coordinate values in the specified (*to_srid* parameter) coordinate system.

Examples

The following example transforms the geometries in the *shape* column in the COLA_MARKETS_CS table to a representation that uses SRID value 8199. The transformed geometries are stored in the newly created table named COLA_MARKETS_CS_8199. (This example uses the definitions from the example in Section 5.8.)

```
-- Transform the entire SHAPE layer and put results in the table
-- named cola_markets_cs_8199, which the procedure will create.
EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS_CS','SHAPE','COLA_MARKETS_CS_8199',8199);
```

Example 5–2 in Section 5.8 includes a display of the geometry object coordinates in both tables (COLA_MARKETS_CS and COLA_MARKETS_CS_8199).

SDO_CS.VIEWPORT_TRANSFORM

Format

```
SDO_CS.VIEWPORT_TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    to_srid     IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_CS.TRANSFORM(
    geom         IN MDSYS.SDO_GEOMETRY,
    to_srname    IN VARCHAR2
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Transforms an optimized rectangle into a valid geodetic polygon for use with Spatial operators and functions.

Parameters

geom
Geometry whose representation is to be transformed from an optimized rectangle to a valid geodetic polygon. The input geometry must have an SRID value of 0 (zero), as explained in the Usage Notes.

to_srid
The SRID of the coordinate system to be used for the transformation. *to_srid* must be a value in the SRID column of the MDSYS.CS_SRS table (described in Section 5.4.1).

to_srname
The name of the coordinate system to be used for the transformation. *to_srname* must be a value (specified exactly) in the CS_NAME column of the MDSYS.CS_SRS table (described in Section 5.4.1).

Usage Notes

The geometry passed in must be an optimized rectangle. Visualiser applications that work on geodetic data usually treat the longitude / latitude space as a regular Cartesian coordinate system. Fetching the data corresponding to a viewport is usually done with the help of an SDO_FILTER or SDO_GEOM.RELATE operation where the viewport (with an optimized rectangle representation) is sent as the window query. With the current restriction of not allowing this optimized rectangle type in geodetic space, this type of viewport queries cannot be sent to the database.

The VIEWPORT_TRANSFORM function provides a workaround. The viewport rectangles should be constructed with the SRID value as 0 and input to the function to generate a corresponding valid geodetic polygon. This geodetic polygon can then be used in the SDO_FILTER or SDO_GEOM.RELATE call as the window object.

Note that an SRID value of 0 should only be specified when calling the VIEWPORT_TRANSFORM function. It is not valid in any other context in Spatial.

This function should be used only when the display space is equi-rectangular (a rectangle), and the data displayed is geodetic.

Examples

The following example specifies the viewport as the whole Earth represented by an optimized rectangle. It returns the names of all four cola markets. (This example uses the definitions from the example in Section 5.8.)

```
SELECT c.name FROM cola_markets_cs c WHERE
   SDO_FILTER(c.shape, SDO_CS.VIEWPORT_TRANSFORM(
       MDSYS.SDO_GEOMETRY(
           2003,
           0,    -- SRID = 0 (special case)
           NULL,
           MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
           MDSYS.SDO_ORDINATE_ARRAY(-180,-90,180,90)),
       8307), 'querytype=window') = 'TRUE';

NAME
--------------------------------
cola_a
cola_c
cola_b
cola_d
```

If the optimizer does not generate an optimal plan and performance is not as you expect, you can try the following alternative version of the query.

```
SELECT c.name FROM cola_markets_cs c,
   (SELECT
   SDO_CS.VIEWPORT_TRANSFORM(
      MDSYS.SDO_GEOMETRY(2003, 0, NULL,
      MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
      MDSYS.SDO_ORDINATE_ARRAY(-180,-90,180,90)), 8307)
   window_geom FROM DUAL)
WHERE SDO_FILTER(c.shape, window_geom, 'querytype=window') = 'TRUE';

NAME
--------------------------------
cola_a
cola_c
cola_b
cola_d
```

15

Linear Referencing Functions

The MDSYS.SDO_LRS package contains functions that create, modify, query, and convert linear referencing elements. These functions do not change the state of the database.

Note: Most Oracle LRS interfaces are functions. Any that are procedures, such as DEFINE_GEOM_SEGMENT, are identified as such. (Functions return a value; procedures do not return a value.)

The word *functions* is often used to refer to LRS interfaces (both functions and procedures) collectively.

To use the functions in this chapter, you must understand the linear referencing system (LRS) concepts and techniques described in Chapter 6.

Table 15–1 lists functions related to creating and editing geometric segments.

Table 15–1 Functions for Creating and Editing Geometric Segments

Function	Description
SDO_LRS.DEFINE_GEOM_SEGMENT (procedure)	Defines a geometric segment.
SDO_LRS.REDEFINE_GEOM_SEGMENT (procedure)	Populates the measures of all shape points of a geometric segment based on the start and end measures, overriding any previously assigned measures between the start point and end point.
SDO_LRS.CLIP_GEOM_SEGMENT	Clips a geometric segment (synonym of SDO_LRS.DYNAMIC_SEGMENT).
SDO_LRS.DYNAMIC_SEGMENT	Clips a geometric segment (synonym of SDO_LRS.CLIP_GEOM_SEGMENT).

Table 15–1 Functions for Creating and Editing Geometric Segments (Cont.)

Function	Description
SDO_LRS.CONCATENATE_GEOM_SEGMENTS	Concatenates two geometric segments into one segment.
SDO_LRS.OFFSET_GEOM_SEGMENT	Returns the geometric segment at a specified offset from a geometric segment.
SDO_LRS.SCALE_GEOM_SEGMENT	Scales a geometric segment.
SDO_LRS.SPLIT_GEOM_SEGMENT (procedure)	Splits a geometric segment into two segments.
SDO_LRS.REVERSE_MEASURE	Returns a new geometric segment by reversing the original geometric segment.
SDO_LRS.TRANSLATE_MEASURE	Returns a new geometric segment by translating the original geometric segment (that is, shifting the start and end measures by a specified value).
SDO_LRS.REVERSE_GEOMETRY	Returns a new geometric segment by reversing the measure values and the direction of the original geometric segment.

Table 15–2 lists functions related to querying geometric segments.

Table 15–2 Functions for Querying Geometric Segments

Function	Description
SDO_LRS.VALID_GEOM_SEGMENT	Checks if a geometric segment is valid.
SDO_LRS.VALID_LRS_PT	Checks if an LRS point is valid.
SDO_LRS.VALID_MEASURE	Checks if a measure falls within the measure range of a geometric segment.
SDO_LRS.CONNECTED_GEOM_SEGMENTS	Checks if two geometric segments are connected.
SDO_LRS.GEOM_SEGMENT_LENGTH	Returns the length of a geometric segment.
SDO_LRS.GEOM_SEGMENT_START_PT	Returns the start point of a geometric segment.
SDO_LRS.GEOM_SEGMENT_END_PT	Returns the end point of a geometric segment.
SDO_LRS.GEOM_SEGMENT_START_MEASURE	Returns the start measure of a geometric segment.

Table 15–2 Functions for Querying Geometric Segments (Cont.)

Function	Description
SDO_LRS.GEOM_SEGMENT_END_MEASURE	Returns the end measure of a geometric segment.
SDO_LRS.GET_MEASURE	Returns the measure of an LRS point.
SDO_LRS.MEASURE_RANGE	Returns the measure range of a geometric segment, that is, the difference between the start measure and end measure.
SDO_LRS.MEASURE_TO_PERCENTAGE	Returns the percentage (0 to 100) that a specified measure is of the measure range of a geometric segment.
SDO_LRS.PERCENTAGE_TO_MEASURE	Returns the measure value of a specified percentage (0 to 100) of the measure range of a geometric segment.
SDO_LRS.LOCATE_PT	Finds the location of a point described by a measure and an offset on a geometric segment.
SDO_LRS.PROJECT_PT	Returns the projection point of a point on a geometric segment.
SDO_LRS.FIND_LRS_DIM_POS	Returns the position of the measure dimension within the SDO_DIM_ARRAY structure for a specified SDO_GEOMETRY column.
SDO_LRS.FIND_MEASURE	Returns the measure of the closest point on a segment to a specified projection point.

Table 15–3 lists functions related to converting geometric segments.

Table 15–3 Functions for Converting Geometric Segments

Function	Description
SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY	Converts a standard dimensional array to a Linear Referencing System dimensional array by creating a measure dimension.
SDO_LRS.CONVERT_TO_LRS_GEOM	Converts a standard SDO_GEOMETRY line string to a Linear Referencing System geometric segment by adding measure information.

Table 15–3 Functions for Converting Geometric Segments (Cont.)

Function	Description
SDO_LRS.CONVERT_TO_LRS_LAYER	Converts all geometry objects in a column of type SDO_GEOMETRY from standard line string geometries without measure information to Linear Referencing System geometric segments with measure information, and updates the metadata.
SDO_LRS.CONVERT_TO_STD_DIM_ARRAY	Converts a Linear Referencing System dimensional array to a standard dimensional array by removing the measure dimension.
SDO_LRS.CONVERT_TO_STD_GEOM	Converts a Linear Referencing System geometric segment to a standard SDO_GEOMETRY line string by removing measure information.
SDO_LRS.CONVERT_TO_STD_LAYER	Converts all geometry objects in a column of type SDO_GEOMETRY from Linear Referencing System geometric segments with measure information to standard line string geometries without measure information, and updates the metadata.

For more information about conversion functions, see Section 6.5.10.

The rest of this chapter provides reference information on the functions, listed in alphabetical order.

SDO_LRS.CLIP_GEOM_SEGMENT

Format

```
SDO_LRS.CLIP_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    start_measure    IN NUMBER,
    end_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.CLIP_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array        IN MDSYS.SDO_DIM_ARRAY,
    start_measure    IN NUMBER,
    end_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the geometry object resulting from a clip operation on a geometric segment.

> **Note:** SDO_LRS.CLIP_GEOM_SEGMENT and SDO_LRS.DYNAMIC_SEGMENT are synonyms: both functions have the same parameters, behavior, and return value.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Start measure of the geometric segment.

end_measure
End measure of the geometric segment.

Usage Notes

An exception is raised if *geom_segment*, *start_measure*, or *end_measure* is invalid.

start_measure and *end_measure* can be any points on the geometric segment. They do not have to be in any specific order. For example, *start_measure* and *end_measure* can be 5 and 10, respectively, or 10 and 5, respectively.

The direction and measures of the resulting geometric segment are preserved (that is, they reflect the original segment).

The *_3D* format of this function (SDO_LRS.CLIP_GEOM_SEGMENT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about clipping geometric segments, see Section 6.5.3.

Examples

The following example clips the geometric segment representing Route 1, returning the segment from measures 5 through 10. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(route_geometry, 5, 10)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.CLIP_GEOM_SEGMENT(ROUTE_GEOMETRY,5,10)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 4, 5, 8, 4, 8, 10, 4, 10))
```

SDO_LRS.CONCATENATE_GEOM_SEGMENTS

Format

```
SDO_LRS.CONCATENATE_GEOM_SEGMENTS(
    geom_segment_1  IN MDSYS.SDO_GEOMETRY,
    geom_segment_2  IN MDSYS.SDO_GEOMETRY,
    tolerance       IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.CONCATENATE_GEOM_SEGMENTS(
    geom_segment_1  IN MDSYS.SDO_GEOMETRY,
    dim_array_1     IN MDSYS.SDO_DIM_ARRAY,
    geom_segment_2  IN MDSYS.SDO_GEOMETRY,
    dim_array_2     IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the geometry object resulting from the concatenation of two geometric segments.

Parameters

geom_segment_1
First geometric segment to be concatenated.

dim_array_1
Dimensional information array corresponding to *geom_segment_1*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

geom_segment_2
Second geometric segment to be concatenated.

dim_array_2
Dimensional information array corresponding to *geom_segment_2*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

tolerance
Tolerance value (see Section 1.5.5).

Usage Notes

An exception is raised if *geom_segment_1* or *geom_segment_2* has an invalid geometry type or dimensionality, or if *geom_segment_1* and *geom_segment_2* are based on different coordinate systems.

The direction of the first geometric segment is preserved, and all measures of the second segment are shifted so that its start measure is the same as the end measure of the first segment.

The *_3D* format of this function (SDO_LRS.CONCATENATE_GEOM_SEGMENTS_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about concatenating geometric segments, see Section 6.5.5

Examples

The following example defines the geometric segment, splits it into two segments, then concatenates those segments. (This example uses the definitions from the example in Section 6.6. The definitions of *result_geom_1*, *result_geom_2*, and *result_geom_3* are displayed in Example 6–4.)

```
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
line_string MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;
result_geom_1 MDSYS.SDO_GEOMETRY;
result_geom_2 MDSYS.SDO_GEOMETRY;
result_geom_3 MDSYS.SDO_GEOMETRY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';
```

```
-- Define the LRS segment for Route1.
SDO_LRS.DEFINE_GEOM_SEGMENT (geom_segment,
  dim_array,
  0,    -- Zero starting measure: LRS segment starts at start of route.
  27);  -- End of LRS segment is at measure 27.

SELECT a.route_geometry INTO line_string FROM lrs_routes a
  WHERE a.route_name = 'Route1';

-- Split Route1 into two segments.
SDO_LRS.SPLIT_GEOM_SEGMENT(line_string,dim_array,5,result_geom_1,result_geom_2);

-- Concatenate the segments that were just split.
result_geom_3 := SDO_LRS.CONCATENATE_GEOM_SEGMENTS(result_geom_1, dim_array,
result_geom_2, dim_array);

-- Insert geometries into table, to display later.
INSERT INTO lrs_routes VALUES(
  11,
  'result_geom_1',
  result_geom_1
);
INSERT INTO lrs_routes VALUES(
  12,
  'result_geom_2',
  result_geom_2
);
INSERT INTO lrs_routes VALUES(
  13,
  'result_geom_3',
  result_geom_3
);

END;
/
```

SDO_LRS.CONNECTED_GEOM_SEGMENTS

Format

```
SDO_LRS.CONNECTED_GEOM_SEGMENTS(
    geom_segment_1  IN MDSYS.SDO_GEOMETRY,
    geom_segment_2  IN MDSYS.SDO_GEOMETRY,
    tolerance       IN NUMBER
    ) RETURN VARCHAR2;
```

or

```
SDO_LRS.CONNECTED_GEOM_SEGMENTS(
    geom_segment_1  IN MDSYS.SDO_GEOMETRY,
    dim_array_1     IN MDSYS.SDO_DIM_ARRAY,
    geom_segment_2  IN MDSYS.SDO_GEOMETRY,
    dim_array_2     IN MDSYS.SDO_DIM_ARRAY
    ) RETURN VARCHAR2;
```

Description

Checks if two geometric segments are spatially connected.

Parameters

geom_segment_1
First of two geometric segments to be checked.

dim_array_1
Dimensional information array corresponding to *geom_segment_1*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

geom_segment_2
Second of two geometric segments to be checked.

dim_array_2
Dimensional information array corresponding to *geom_segment_2*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

tolerance
Tolerance value (see Section 1.5.5).

Usage Notes

This function returns TRUE if the geometric segments are spatially connected and FALSE if the geometric segments are not spatially connected.

An exception is raised if *geom_segment_1* or *geom_segment_2* has an invalid geometry type or dimensionality, or if *geom_segment_1* and *geom_segment_2* are based on different coordinate systems.

The *_3D* format of this function (SDO_LRS.CONNECTED_GEOM_SEGMENTS_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example checks if two geometric segments (results of a previous split operation) are spatially connected.

```
-- Are result_geom_1 and result_geom2 connected?
SELECT  SDO_LRS.CONNECTED_GEOM_SEGMENTS(a.route_geometry,
           b.route_geometry, 0.005)
  FROM lrs_routes a, lrs_routes b
  WHERE a.route_id = 11 AND b.route_id = 12;

SDO_LRS.CONNECTED_GEOM_SEGMENTS(A.ROUTE_GEOMETRY,B.ROUTE_GEOMETRY,0.005)
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY

Format

```
SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY(
    dim_array       IN MDSYS.SDO_DIM_ARRAY
    [, lower_bound  IN NUMBER,
    upper_bound     IN NUMBER,
    tolerance       IN NUMBER]
    ) RETURN MDSYS.SDO_DIM_ARRAY;
    or
SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY(
    dim_array       IN MDSYS.SDO_DIM_ARRAY,
    dim_name        IN VARCHAR2
    [, lower_bound  IN NUMBER,
    upper_bound     IN NUMBER,
    tolerance       IN NUMBER]
    ) RETURN MDSYS.SDO_DIM_ARRAY;
    or
SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY(
    dim_array       IN MDSYS.SDO_DIM_ARRAY,
    dim_name        IN VARCHAR2,
    dim_pos         IN INTEGER
    [, lower_bound  IN NUMBER,
    upper_bound     IN NUMBER,
    tolerance       IN NUMBER]
    ) RETURN MDSYS.SDO_DIM_ARRAY;
```

Description

Converts a standard dimensional array to a Linear Referencing System dimensional array by creating a measure dimension.

Parameters

dim_array
Dimensional information array corresponding to the layer (column of geometries) to be converted, usually selected from one of the xxx_SDO_GEOM_METADATA views.

dim_name
Name of the measure dimension (M, if not otherwise specified).

dim_pos
Position of the measure dimension (the last SDO_DIM_ELEMENT object position in the SDO_DIM_ARRAY, if not otherwise specified).

lower_bound
Lower bound (SDO_LB value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

upper_bound
Upper bound (SDO_UB value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

tolerance
Tolerance value (see Section 1.5.5).

Usage Notes

This function converts a standard dimensional array to a Linear Referencing System dimensional array by creating a measure dimension. Specifically, it adds an SDO_DIM_ELEMENT object at the end of the current SDO_DIM_ELEMENT objects in the SDO_DIM_ARRAY for the diminfo (unless another *dim_pos* is specified), and sets the SDO_DIMNAME value in this added SDO_DIM_ELEMENT to M (unless another *dim_name* is specified). It sets the other values in the added SDO_DIM_ELEMENT according to the values if the *upper_bound*, *lower_bound*, and *tolerance* parameter values.

If *dim_array* already contains dimensional information, the *dim_array* is returned.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the dimensional array for the LRS_ROUTES table to Linear Referencing System format. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY(m.diminfo)
   FROM user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY(M.DIMINFO)(SDO_DIMNAME, SDO_LB, SDO_UB, SDO_TOL
--------------------------------------------------------------------------------
SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X', 0, 20, .005), SDO_DIM_ELEMENT('Y', 0, 20, .00
5), SDO_DIM_ELEMENT('M', 0, 20, .005))
```

SDO_LRS.CONVERT_TO_LRS_GEOM

Format

```
SDO_LRS.CONVERT_TO_LRS_GEOM(
    standard_geom   IN MDSYS.SDO_GEOMETRY
    [, start_measure  IN NUMBER,
    end_measure     IN NUMBER]
    ) RETURN MDSYS.SDO_GEOMETRY;
    or
SDO_LRS.CONVERT_TO_LRS_GEOM(
    standard_geom   IN MDSYS.SDO_GEOMETRY,
    dim_array       IN MDSYS.SDO_DIM_ARRAY
    [, start_measure  IN NUMBER,
    end_measure     IN NUMBER]
    ) RETURN MDSYS.SDO_GEOMETRY;
    or
SDO_LRS.CONVERT_TO_LRS_GEOM(
    standard_geom   IN MDSYS.SDO_GEOMETRY,
    m_pos            IN INTEGER
    [, start_measure  IN NUMBER,
    end_measure     IN NUMBER]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Converts a standard SDO_GEOMETRY line string to a Linear Referencing System geometric segment by adding measure information.

Parameters

standard_geom
Line string geometry that does not contain measure information.

dim_array
Dimensional information array corresponding to *standard_geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

m_pos
Position of the measure dimension. If specified, must be 3 or 4. By default, the measure dimension is the last dimension in the SDO_DIM_ARRAY.

start_measure
Distance measured from the start point of a geometric segment to the start point of the linear feature. The default is 0.

end_measure
Distance measured from the end point of a geometric segment to the start point of the linear feature. The default is the cartographic length (for example, 75 if the cartographic length is 75 miles and the unit of measure is miles).

Usage Notes

This function returns a Linear Referencing System geometric segment with measure information, with measure information provided for all shape points.

An exception is raised if *standard_geom* has an invalid geometry type or dimensionality, if *m_pos* is less than 3 or greater than 4, or if *start_measure* or *end_measure* is out of range.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_LRS_GEOM_3D) is available; however, the *m_pos* parameter is not available for SDO_LRS.CONVERT_TO_LRS_GEOM_3D. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the geometric segment representing Route 1 to Linear Referencing System format. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.CONVERT_TO_LRS_GEOM(a.route_geometry, m.diminfo)
```

```
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.CONVERT_TO_LRS_GEOM(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, NULL, 8, 10, 22, 5, 14, 27))
```

SDO_LRS.CONVERT_TO_LRS_LAYER

Format

```
SDO_LRS.CONVERT_TO_LRS_LAYER(
    table_name      IN VARCHAR2,
    column_name     IN VARCHAR2
    [, lower_bound  IN NUMBER,
    upper_bound     IN NUMBER,
    tolerance       IN NUMBER]
    ) RETURN VARCHAR2;
```

or

```
SDO_LRS.CONVERT_TO_LRS_LAYER(
    table_name      IN VARCHAR2,
    column_name     IN VARCHAR2,
    dim_name        IN VARCHAR2,
    dim_pos         IN INTEGER
    [, lower_bound  IN NUMBER,
    upper_bound     IN NUMBER,
    tolerance       IN NUMBER]
    ) RETURN VARCHAR2;
```

Description

Converts all geometry objects in a column of type SDO_GEOMETRY (that is, converts a layer) from standard line string geometries without measure information to Linear Referencing System geometric segments with measure information, and updates the metadata in the USER_SDO_GEOM_METADATA view.

Parameters

table_name
Table containing the column with the SDO_GEOMETRY objects.

column_name
Column in *table_name* containing the SDO_GEOMETRY objects.

dim_name
Name of the measure dimension. If this parameter is null, M is assumed.

dim_pos
Position of the measure dimension within the SDO_DIM_ARRAY structure for the specified SDO_GEOMETRY column. If this parameter is null, the number corresponding to the last position is assumed.

lower_bound
Lower bound (SDO_LB value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

upper_bound
Upper bound (SDO_UB value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

tolerance
Tolerance value (see Section 1.5.5).

Usage Notes

This function returns TRUE if the conversion was successful or if the layer already contains measure information, and the function returns an exception if the conversion was not successful.

An exception is raised if the existing dimensional information for the table is invalid.

The measure values are assigned based on a start measure of zero and an end measure of the cartographic length.

If a spatial index already exists on *column_name*, you must delete (drop) the index before converting the layer and create a new index after converting the layer. For information about deleting and creating indexes, see the DROP INDEX and CREATE INDEX statements in Chapter 9.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_LRS_LAYER_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the geometric segments in the ROUTE_GEOMETRY column of the LRS_ROUTES table to Linear Referencing System format. (This example uses the definitions from the example in Section 6.6.) The SELECT statement shows that dimensional information has been added (that is, SDO_DIM_ELEMENT('M', NULL, NULL, NULL) included in the definition).

```
BEGIN
  IF (SDO_LRS.CONVERT_TO_LRS_LAYER('LRS_ROUTES', 'ROUTE_GEOMETRY') =  'TRUE')
     THEN
       DBMS_OUTPUT.PUT_LINE('Conversion from STD_LAYER to LRS_LAYER succeeded');
     ELSE
       DBMS_OUTPUT.PUT_LINE('Conversion from STD_LAYER to LRS_LAYER failed');
  END IF;
END;
.
/
Conversion from STD_LAYER to LRS_LAYER succeeded

PL/SQL procedure successfully completed.

SQL> SELECT diminfo FROM user_sdo_geom_metadata WHERE table_name = 'LRS_ROUTES'
AND column_name = 'ROUTE_GEOMETRY';

DIMINFO(SDO_DIMNAME, SDO_LB, SDO_UB, SDO_TOLERANCE)
--------------------------------------------------------------------------------
SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X', 0, 20, .005), SDO_DIM_ELEMENT('Y', 0, 20, .00
5), SDO_DIM_ELEMENT('M', NULL, NULL, NULL))
```

SDO_LRS.CONVERT_TO_STD_DIM_ARRAY

Format

```
SDO_LRS.CONVERT_TO_STD_DIM_ARRAY(
    dim_array  IN MDSYS.SDO_DIM_ARRAY
    [, m_pos     IN INTEGER]
    ) RETURN MDSYS.SDO_DIM_ARRAY;
```

Description

Converts a Linear Referencing System dimensional array to a standard dimensional array by removing the measure dimension.

Parameters

dim_array
Dimensional information array corresponding to the layer (column of geometries) to be converted, usually selected from one of the xxx_SDO_GEOM_METADATA views.

m_pos
Position of the measure dimension. If specified, must be 3 or 4. By default, the measure dimension is the last dimension in the SDO_DIM_ARRAY.

Usage Notes

This function converts a Linear Referencing System dimensional array to a standard dimensional array by removing the measure dimension. Specifically, it removes the SDO_DIM_ELEMENT object at the end of the current SDO_DIM_ELEMENT objects in the SDO_DIM_ARRAY for the *diminfo*.

An exception is raised if *m_pos* is invalid (less than 3 or greater than 4).

If *dim_array* is already a standard dimensional array (that is, does not contain dimensional information), the *dim_array* is returned.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_STD_DIM_ARRAY_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the dimensional array for the LRS_ROUTES table to standard format. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.CONVERT_TO_STD_DIM_ARRAY(m.diminfo)
   FROM user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

SDO_LRS.CONVERT_TO_STD_DIM_ARRAY(M.DIMINFO)(SDO_DIMNAME, SDO_LB, SDO_UB, SDO_TOL
--------------------------------------------------------------------------------
SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X', 0, 20, .005), SDO_DIM_ELEMENT('Y', 0, 20, .00
5))
```

SDO_LRS.CONVERT_TO_STD_GEOM

Format

```
SDO_LRS.CONVERT_TO_STD_GEOM(
    lrs _geom     IN MDSYS.SDO_GEOMETRY
    [, dim_array  IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Converts a Linear Referencing System geometric segment to a standard SDO_GEOMETRY line string by removing measure information.

Parameters

lrs_geom
Linear Referencing System geometry that contains measure information.

dim_array
Dimensional information array corresponding to *lrs_geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns an SDO_GEOMETRY object in which all measure information is removed.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_STD_GEOM_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the geometric segment representing Route 1 to standard format. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.CONVERT_TO_STD_GEOM(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
```

```
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.CONVERT_TO_STD_GEOM(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(2002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 2, 4, 8, 4, 12, 4, 12, 10, 8, 10, 5, 14))
```

SDO_LRS.CONVERT_TO_STD_LAYER

Format

```
SDO_LRS.CONVERT_TO_STD_LAYER(
    table_name      IN VARCHAR2,
    column_name  IN VARCHAR2
    ) RETURN VARCHAR2;
```

Description

Converts all geometry objects in a column of type SDO_GEOMETRY (that is, converts a layer) from Linear Referencing System geometric segments with measure information to standard line string geometries without measure information, and updates the metadata in the USER_SDO_GEOM_METADATA view.

Parameters

table_name
Table containing the column with the SDO_GEOMETRY objects.

column_name
Column in *table_name* containing the SDO_GEOMETRY objects.

Usage Notes

This function returns TRUE if the conversion was successful or if the layer already is a standard layer (that is, contains geometries without measure information), and the function returns an exception if the conversion was not successful.

An exception is raised if the conversion failed.

If a spatial index already exists on *column_name*, you must delete (drop) the index before converting the layer and create a new index after converting the layer. For information about deleting and creating indexes, see the DROP INDEX and CREATE INDEX statements in Chapter 9.

The *_3D* format of this function (SDO_LRS.CONVERT_TO_STD_LAYER_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about conversion functions, see Section 6.5.10.

Examples

The following example converts the geometric segments in the ROUTE_GEOMETRY column of the LRS_ROUTES table to standard format. (This example uses the definitions from the example in Section 6.6.) The SELECT statement shows that dimensional information has been removed (that is, no SDO_DIM_ELEMENT('M', NULL, NULL, NULL) included in the definition).

```
BEGIN
  IF (SDO_LRS.CONVERT_TO_STD_LAYER('LRS_ROUTES', 'ROUTE_GEOMETRY') =  'TRUE')
     THEN
       DBMS_OUTPUT.PUT_LINE('Conversion from LRS_LAYER to STD_LAYER succeeded');
     ELSE
       DBMS_OUTPUT.PUT_LINE('Conversion from LRS_LAYER to STD_LAYER failed');
  END IF;
END;
.
/
Conversion from LRS_LAYER to STD_LAYER succeeded

PL/SQL procedure successfully completed.

SELECT diminfo FROM user_sdo_geom_metadata
   WHERE table_name = 'LRS_ROUTES' AND column_name = 'ROUTE_GEOMETRY';

DIMINFO(SDO_DIMNAME, SDO_LB, SDO_UB, SDO_TOLERANCE)
--------------------------------------------------------------------------------
SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X', 0, 20, .005), SDO_DIM_ELEMENT('Y', 0, 20, .00
5))
```

SDO_LRS.DEFINE_GEOM_SEGMENT

Format

```
SDO_LRS.DEFINE_GEOM_SEGMENT(
    geom_segment   IN OUT MDSYS.SDO_GEOMETRY
    [, start_measure  IN NUMBER,
    end_measure     IN NUMBER]);
```

or

```
SDO_LRS.DEFINE_GEOM_SEGMENT(
    geom_segment   IN OUT MDSYS.SDO_GEOMETRY,
    dim_array         IN MDSYS.SDO_DIM_ARRAY
    [, start_measure  IN NUMBER,
    end_measure     IN NUMBER]);
```

Description

Defines a geometric segment by assigning start and end measures to a geometric segment, and assigns values to any null measures. (This is a procedure, not a function.)

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Distance measured from the start point of a geometric segment to the start point of the linear feature. The default is the existing value (if any) in the measure dimension; otherwise, the default is 0.

end_measure
Distance measured from the end point of a geometric segment to the start point of the linear feature. The default is the existing value (if any) in the measure dimension; otherwise, the default is the cartographic length of the segment.

Usage Notes

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality, or if *start_measure* or *end_measure* is out of range.

All unassigned measures of the geometric segment will be populated automatically.

To store the resulting geometric segment (*geom_segment*) in the database, you must execute and UPDATE or INSERT statement, as appropriate.

The *_3D* format of this procedure (SDO_LRS.DEFINE_GEOM_SEGMENT_3D) is available. For information about *_3D* formats of LRS functions and procedures, see Section 6.4.

For more information about defining a geometric segment, see Section 6.5.1

Examples

The following example defines the geometric segment, splits it into two segments, then concatenates those segments. (This example uses the definitions from the example in Section 6.6. The definitions of *result_geom_1*, *result_geom_2*, and *result_geom_3* are displayed in Example 6–4.)

```
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
line_string MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;
result_geom_1 MDSYS.SDO_GEOMETRY;
result_geom_2 MDSYS.SDO_GEOMETRY;
result_geom_3 MDSYS.SDO_GEOMETRY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

-- Define the LRS segment for Route1. This will populate any null measures.
SDO_LRS.DEFINE_GEOM_SEGMENT (geom_segment,
```

```
  dim_array,
  0,    -- Zero starting measure: LRS segment starts at start of route.
  27);  -- End of LRS segment is at measure 27.

SELECT a.route_geometry INTO line_string FROM lrs_routes a
  WHERE a.route_name = 'Route1';

-- Split Route1 into two segments.
SDO_LRS.SPLIT_GEOM_SEGMENT(line_string,dim_array,5,result_geom_1,result_geom_2);

-- Concatenate the segments that were just split.
result_geom_3 := SDO_LRS.CONCATENATE_GEOM_SEGMENTS(result_geom_1, dim_array,
result_geom_2, dim_array);

-- Update and insert geometries into table, to display later.
UPDATE lrs_routes a SET a.route_geometry = geom_segment
   WHERE a.route_id = 1;

INSERT INTO lrs_routes VALUES(
  11,
  'result_geom_1',
  result_geom_1
);
INSERT INTO lrs_routes VALUES(
  12,
  'result_geom_2',
  result_geom_2
);
INSERT INTO lrs_routes VALUES(
  13,
  'result_geom_3',
  result_geom_3
);

END;
/
```

SDO_LRS.DYNAMIC_SEGMENT

Format

```
SDO_LRS.DYNAMIC_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    start_measure    IN NUMBER,
    end_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.DYNAMIC_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array          IN MDSYS.SDO_DIM_ARRAY,
    start_measure    IN NUMBER,
    end_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the geometry object resulting from a clip operation on a geometric segment.

Note: SDO_LRS.CLIP_GEOM_SEGMENT and SDO_LRS.DYNAMIC_SEGMENT are synonyms: both functions have the same parameters, behavior, and return value.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Start measure of the geometric segment.

end_measure
End measure of the geometric segment.

Usage Notes

An exception is raised if *geom_segment*, *start_measure*, or *end_measure* is invalid.

The direction and measures of the resulting geometric segment are preserved.

For more information about clipping a geometric segment, see Section 6.5.3

Examples

The following example clips the geometric segment representing Route 1, returning the segment from measures 5 through 10. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.DYNAMIC_SEGMENT(route_geometry, 5, 10)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.DYNAMIC_SEGMENT(ROUTE_GEOMETRY,5,10)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 4, 5, 8, 4, 8, 10, 4, 10))
```

SDO_LRS.FIND_LRS_DIM_POS

Format

```
SDO_LRS.FIND_LRS_DIM_POS(
    table_name      IN VARCHAR2,
    column_name  IN VARCHAR2
    ) RETURN INTEGER;
```

Description

Returns the position of the measure dimension within the SDO_DIM_ARRAY structure for a specified SDO_GEOMETRY column.

Parameters

table_name
Table containing the column with the SDO_GEOMETRY objects.

column_name
Column in *table_name* containing the SDO_GEOMETRY objects.

Usage Notes

None.

Examples

The following example returns the position of the measure dimension within the SDO_DIM_ARRAY structure for geometries in the ROUTE_GEOMETRY column of the LRS_ROUTES table. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.FIND_LRS_DIM_POS('LRS_ROUTES', 'ROUTE_GEOMETRY') FROM DUAL;

SDO_LRS.FIND_LRS_DIM_POS('LRS_ROUTES','ROUTE_GEOMETRY')
------------------------------------------------------
                                                     3
```

SDO_LRS.FIND_MEASURE

Format

```
SDO_LRS.FIND_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    point         IN MDSYS.SDO_GEOMETRY
    ) RETURN NUMBER;
```

or

```
SDO_LRS.FIND_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    point         IN MDSYS.SDO_GEOMETRY
    ) RETURN NUMBER;
```

Description

Returns the measure of the closest point on a segment to a specified projection point.

Parameters

geom_segment
Cartographic representation of a linear feature. This function returns the measure of the point on this segment that is closest to the projection point.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

point
Projection point. This function returns the measure of the point on *geom_segment* that is closest to the projection point.

Usage Notes

This function returns the measure of the point on *geom_segment* that is closest to the projection point. For example, if the projection point represents a shopping mall, the function could be used to find how far from the start of the highway is the point on the highway that is closest to the shopping mall.

An exception is raised if *geom_segment_1* or *geom_segment_2* has an invalid geometry type or dimensionality, or if *geom_segment* and *point* are based on different coordinate systems.

The *_3D* format of this function (SDO_LRS.FIND_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example finds the measure for the point on the geometric segment representing Route 1 that is closest to the point (10, 7). (This example uses the definitions from the example in Section 6.6.)

```
-- Find measure for point on segment closest to 10,7
-- Should return 15 (for point 12,7)
SELECT  SDO_LRS.FIND_MEASURE(a.route_geometry, m.diminfo,
  MDSYS.SDO_GEOMETRY(3001, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(10, 7, NULL)) )
 FROM lrs_routes a, user_sdo_geom_metadata m
 WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
   AND a.route_id = 1;

SDO_LRS.FIND_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO,MDSYS.SDO_GEOMETRY(3001,NULL,NUL
--------------------------------------------------------------------------------
                                                                             15
```

SDO_LRS.GEOM_SEGMENT_END_MEASURE

Format

```
SDO_LRS.GEOM_SEGMENT_END_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN NUMBER;
```

Description

Returns the end measure of a geometric segment.

Parameters

geom_segment
Geometric segment whose end measure is to be returned.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the end measure of *geom_segment*.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.GEOM_SEGMENT_END_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the end measure of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_END_MEASURE(route_geometry)
  FROM lrs_routes WHERE route_id = 1;
```

```
SDO_LRS.GEOM_SEGMENT_END_MEASURE(ROUTE_GEOMETRY)
-------------------------------------------------
                                               27
```

SDO_LRS.GEOM_SEGMENT_END_PT

Format

```
SDO_LRS.GEOM_SEGMENT_END_PT(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the end point of a geometric segment.

Parameters

geom_segment
Geometric segment whose end point is to be returned.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the end point of *geom_segment*.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.GEOM_SEGMENT_END_PT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the end point of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_END_PT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_END_PT(ROUTE_GEOMETRY)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y,
```

```
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
5, 14, 27))
```

SDO_LRS.GEOM_SEGMENT_LENGTH

Format

```
SDO_LRS.GEOM_SEGMENT_LENGTH(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN NUMBER;
```

Description

Returns the length of a geometric segment.

Parameters

geom_segment
Geometric segment whose length is to be calculated.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the length of *geom_segment*. The length is the geometric length, which is not the same as the total of the measure unit values. To determine how long a segment is in terms of measure units, subtract the result of an SDO_LRS.GEOM_SEGMENT_START_MEASURE operation from the result of an SDO_LRS.GEOM_SEGMENT_END_MEASURE operation.

SDO_LRS.GEOM_SEGMENT_LENGTH is an alias of the SDO_GEOM.SDO_LENGTH Spatial function.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.GEOM_SEGMENT_LENGTH_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the length of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.GEOM_SEGMENT_LENGTH(ROUTE_GEOMETRY)
-------------------------------------------
                                         27
```

SDO_LRS.GEOM_SEGMENT_START_MEASURE

Format

```
SDO_LRS.GEOM_SEGMENT_START_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN NUMBER;
```

Description

Returns the start measure of a geometric segment.

Parameters

geom_segment
Geometric segment whose start measure is to be returned.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the start measure of *geom_segment*.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.GEOM_SEGMENT_START_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the start measure of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(route_geometry)
  FROM lrs_routes WHERE route_id = 1;
```

```
SDO_LRS.GEOM_SEGMENT_START_MEASURE(ROUTE_GEOMETRY)
--------------------------------------------------
                                                 0
```

SDO_LRS.GEOM_SEGMENT_START_PT

Format

```
SDO_LRS.GEOM_SEGMENT_START_PT(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the start point of a geometric segment.

Parameters

geom_segment
Geometric segment whose start point is to be returned.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the start point of *geom_segment*.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.GEOM_SEGMENT_START_PT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the start point of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;
```

```
SDO_LRS.GEOM_SEGMENT_START_PT(ROUTE_GEOMETRY)(SDO_GTYPE, SDO_SRID, SDO_POINT(X,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
2, 2, 0))
```

SDO_LRS.GET_MEASURE

Format

```
SDO_LRS.GET_MEASURE(
    point          IN MDSYS.SDO_GEOMETRY
    [, dim_array  IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN NUMBER;
```

Description

Returns the measure of an LRS point.

Parameters

point
Point whose measure is to be returned.

dim_array
Dimensional information array corresponding to *point*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the measure of an LRS point.

If *point* is not valid, an "invalid LRS point" exception is raised.

Contrast this function with SDO_LRS.PROJECT_PT, which accepts as input a point that is not necessarily on the geometric segment, but which returns a point that is on the geometric segment, as opposed to a measure value. As the following example shows, the SDO_LRS.GET_MEASURE function can be used to return the measure of the projected point returned by SDO_LRS.PROJECT_PT.

The *_3D* format of this function (SDO_LRS.GET_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the measure of a projected point. In this case, the point resulting from the projection is 9 units from the start of the segment.

```
SQL> SELECT SDO_LRS.GET_MEASURE(
   SDO_LRS.PROJECT_PT(a.route_geometry, m.diminfo,
    MDSYS.SDO_GEOMETRY(3001, NULL, NULL,
       MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
       MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) ),
   m.diminfo )
   FROM lrs_routes a, user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
       AND a.route_id = 1;

SDO_LRS.GET_MEASURE(SDO_LRS.PROJECT_PT(A.ROUTE_GEOMETRY,M.DIMINFO,MDSYS.SDO_GEOM
--------------------------------------------------------------------------------
                                                                               9
```

SDO_LRS.IS_GEOM_SEGMENT_DEFINED

Format

```
SDO_LRS.IS_GEOM_SEGMENT_DEFINED(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if an LRS segment is defined correctly.

Parameters

geom_segment
Geometric segment to be checked.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if *geom_segment* is defined correctly and FALSE if *geom_segment* is not defined correctly.

The start and end measures of *geom_segment* must be defined (cannot be null), and any measures assigned must be in an ascending or descending order along the segment direction.

The *_3D* format of this function (SDO_LRS.IS_GEOM_SEGMENT_DEFINED_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

See also the SDO_LRS.VALID_GEOM_SEGMENT function.

Examples

The following example checks if the geometric segment representing Route 1 is defined. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.IS_GEOM_SEGMENT_DEFINED(route_geometry)
```

```
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.IS_GEOM_SEGMENT_DEFINED(ROUTE_GEOMETRY)
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.IS_MEASURE_DECREASING

Format

```
SDO_LRS.IS_MEASURE_DECREASING(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if the measure values along an LRS segment are decreasing (that is, descending in numerical value).

Parameters

geom_segment
Geometric segment to be checked.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if the measure values along an LRS segment are decreasing and FALSE if the measure values along an LRS segment are not decreasing.

The start and end measures of *geom_segment* must be defined (cannot be null).

The *_3D* format of this function (SDO_LRS.IS_MEASURE_DECREASING_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

See also the SDO_LRS.IS_MEASURE_INCREASING function.

Examples

The following example checks if the measure values along the geometric segment representing Route 1 are decreasing. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.IS_MEASURE_DECREASING(a.route_geometry, m.diminfo)
   FROM lrs_routes a, user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
   AND a.route_id = 1;

SDO_LRS.IS_MEASURE_DECREASING(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
FALSE
```

SDO_LRS.IS_MEASURE_INCREASING

Format

```
SDO_LRS.IS_MEASURE_INCREASING(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if the measure values along an LRS segment are increasing (that is, ascending in numerical value).

Parameters

geom_segment
Geometric segment to be checked.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if the measure values along an LRS segment are increasing and FALSE if the measure values along an LRS segment are not increasing.

The start and end measures of *geom_segment* must be defined (cannot be null).

The *_3D* format of this function (SDO_LRS.IS_MEASURE_INCREASING_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

See also the SDO_LRS.IS_MEASURE_DECREASING function.

Examples

The following example checks if the measure values along the geometric segment representing Route 1 are increasing. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.IS_MEASURE_INCREASING(a.route_geometry, m.diminfo)
   FROM lrs_routes a, user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
   AND a.route_id = 1;

SDO_LRS.IS_MEASURE_INCREASING(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.LOCATE_PT

Format

```
SDO_LRS.LOCATE_PT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    measure       IN NUMBER
    [, offset      IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.LOCATE_PT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    measure       IN NUMBER
    [, offset      IN NUMBER]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the point located at a specified distance from the start of a geometric segment.

Parameters

geom_segment
Geometric segment to be checked to see if it falls within the measure range of *measure*.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

measure
Distance to measure from the start point of *geom_segment*.

offset
Distance to measure perpendicularly from the point that is located at *measure* units from the start point of *geom_segment*. The default is 0 (that is, the point is on *geom_segment*).

Usage Notes

This function returns the referenced point.

The unit of measurement for *offset* is the same as for the coordinate system associated with *geom_segment*. For geodetic data, the default unit of measurement is meters.

With geodetic data using the WGS 84 coordinate system, this function can be used to return the longitude and latitude coordinates of any point on or offset from the segment.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality, or if the location is out of range.

The *_3D* format of this function (SDO_LRS.LOCATE_PT_3D) is available; however, the *offset* parameter is not available for SDO_LRS.LOCATE_PT_3D. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about locating a point on a geometric segment, see Section 6.5.8.

Examples

The following example returns the point at measure 9 and on (that is, offset 0) the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.LOCATE_PT(route_geometry, 9, 0)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.LOCATE_PT(ROUTE_GEOMETRY,9,0)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), S
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
9, 4, 9))
```

SDO_LRS.MEASURE_RANGE

Format

```
SDO_LRS.MEASURE_RANGE(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN NUMBER;
```

Description

Returns the measure range of a geometric segment, that is, the difference between the start measure and end measure.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function subtracts the start measure of *geom_segment* from the end measure of *geom_segment*.

The *_3D* format of this function (SDO_LRS.MEASURE_RANGE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example returns the measure range of the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.MEASURE_RANGE(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.MEASURE_RANGE(ROUTE_GEOMETRY)
```

```
-----------------------------------
                                 27
```

SDO_LRS.MEASURE_TO_PERCENTAGE

Format

```
SDO_LRS.MEASURE_TO_PERCENTAGE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    measure       IN NUMBER
    ) RETURN NUMBER;
```

or

```
SDO_LRS.MEASURE_TO_PERCENTAGE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    measure       IN NUMBER
    ) RETURN NUMBER;
```

Description

Returns the percentage (0 to 100) that a specified measure is of the measure range of a geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

measure
Measure value. This function returns the percentage that this measure value is of the measure range.

Usage Notes

This function returns a number (0 to 100) that is the percentage of the measure range that the specified measure represents. (The measure range is the end measure minus the start measure.) For example, if the measure range of *geom_segment* is 50 and *measure* is 20, the function returns 40 (because 20 / 50 = 40%).

This function performs the reverse of the SDO_LRS.PERCENTAGE_TO_MEASURE function, which returns the measure that corresponds to a percentage value.

An exception is raised if *geom_segment* or *measure* is invalid.

Examples

The following example returns the percentage that 5 is of the measure range of geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.) The measure range of this segment is 27, and 5 is approximately 18.5 percent of 27.

```
SELECT SDO_LRS.MEASURE_TO_PERCENTAGE(a.route_geometry, m.diminfo, 5)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.MEASURE_TO_PERCENTAGE(A.ROUTE_GEOMETRY,M.DIMINFO,5)
-----------------------------------------------------------
                                                 18.5185185
```

SDO_LRS.OFFSET_GEOM_SEGMENT

Format

```
SDO_LRS.OFFSET_GEOM_SEGMENT(
    geom_segment   IN MDSYS.SDO_GEOMETRY,
    start_measure  IN NUMBER,
    end_measure    IN NUMBER,
    offset         IN NUMBER
    [, tolerance   IN NUMBER]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.OFFSET_GEOM_SEGMENT(
    geom_segment   IN MDSYS.SDO_GEOMETRY,
    start_measure  IN NUMBER,
    end_measure    IN NUMBER,
    offset         IN NUMBER,
    tolerance      IN NUMBER
    [, unit        IN VARCHAR2]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.OFFSET_GEOM_SEGMENT(
    geom_segment   IN MDSYS.SDO_GEOMETRY,
    dim_array      IN MDSYS.SDO_DIM_ARRAY,
    start_measure  IN NUMBER,
    end_measure    IN NUMBER,
    offset         IN NUMBER
    [, unit        IN VARCHAR2]
```

```
) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the geometric segment at a specified offset from a geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Start measure of *geom_segment* at which to start the offset operation.

end_measure
End measure of *geom_segment* at which to start the offset operation.

offset
Distance to measure perpendicularly from the points along *geom_segment*. Positive offset values are to the left of *geom_segment*; negative offset values are to the right of *geom_segment*.

tolerance
Tolerance value (see Section 1.5.5).

unit
Unit of measurement specification: a quoted string with one or both of the following keywords:

- *unit* and an SDO_UNIT value from the MDSYS.SDO_DIST_UNITS table. See Section 2.6 for more information about unit of measurement specification.
- *arc_tolerance* and an arc tolerance value. See the Usage Notes for the SDO_GEOM.SDO_ARC_DENSIFY function in Chapter 12 for more information about the *arc_tolerance* keyword.

For example: 'unit=km arc_tolerance=0.05'

If the input geometry is geodetic data, this parameter is required, and *arc_tolerance* must be specified. If the input geometry is Cartesian or projected data, *arc_tolerance* has no effect and should not be specified.

If this parameter is not specified for a Cartesian or projected geometry, or if the *arc_tolerance* keyword is specified for a geodetic geometry but the *unit* keyword is not specified, the unit of measurement associated with the data is assumed.

Usage Notes

start_measure and *end_measure* can be any points on the geometric segment. They do not have to be in any specific order. For example, *start_measure* and *end_measure* can be 5 and 10, respectively, or 10 and 5, respectively.

The direction and measures of the resulting geometric segment are preserved (that is, they reflect the original segment).

The geometry type of *geom_segment* must be line or multiline. For example, it cannot be a polygon.

An exception is raised if *geom_segment*, *start_measure*, or *end_measure* is invalid.

Examples

The following example returns the geometric segment 2 distance units to the left (positive offset 2) of the segment from measures 5 through 10 of Route 1. (This example uses the definitions from the example in Section 6.6.)

```
-- Create a segment offset 2 to the left from measures 5 through 10.
-- First, display the original segment; then, offset.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))

SELECT  SDO_LRS.OFFSET_GEOM_SEGMENT(a.route_geometry, m.diminfo, 5, 10, 2)
    FROM lrs_routes a, user_sdo_geom_metadata m
    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
      AND a.route_id = 1;

SDO_LRS.OFFSET_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,5,10,2)(SDO_GTYPE, SDO_SR
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 6, 5, 10, 6, 10))
```

Note in SDO_ORDINATE_ARRAY of the returned segment that the Y values (6) are 2 greater than the Y values (4) of the relevant part of the original segment.

SDO_LRS.PERCENTAGE_TO_MEASURE

Format

```
SDO_LRS.PERCENTAGE_TO_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    percentage    IN NUMBER
    ) RETURN NUMBER;
```

or

```
SDO_LRS.PERCENTAGE_TO_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    percentage    IN NUMBER
    ) RETURN NUMBER;
```

Description

Returns the measure value of a specified percentage (0 to 100) of the measure range of a geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

percentage
Percentage value. Must be from 0 to 100. This function returns the measure value corresponding to this percentage of the measure range.

Usage Notes

This function returns the measure value corresponding to this percentage of the measure range. (The measure range is the end measure minus the start measure.) For example, if the measure range of *geom_segment* is 50 and *percentage* is 40, the function returns 20 (because 40% of 50 = 20).

This function performs the reverse of the SDO_LRS.MEASURE_TO_PERCENTAGE function, which returns the percentage value that corresponds to a measure.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality, or if *percentage* is less than 0 or greater than 100.

Examples

The following example returns the measure that is 50 percent of the measure range of geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.) The measure range of this segment is 27, and 50 percent of 17 is 13.5.

```
SELECT SDO_LRS.PERCENTAGE_TO_MEASURE(a.route_geometry, m.diminfo, 50)
  FROM lrs_routes a, user_sdo_geom_metadata m
  HERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.PERCENTAGE_TO_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO,50)
------------------------------------------------------------
                                                        13.5
```

SDO_LRS.PROJECT_PT

Format

```
SDO_LRS.PROJECT_PT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    point         IN MDSYS.SDO_GEOMETRY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.PROJECT_PT(
    geom_segment      IN MDSYS.SDO_GEOMETRY,
    dim_array         IN MDSYS.SDO_DIM_ARRAY,
    point             IN MDSYS.SDO_GEOMETRY
    [, point_dim_array  IN MDSYS.SDO_GEOMETRY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the projection point of a point on a geometric segment.

Parameters

geom_segment
Geometric segment to be checked.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

point
Point to be projected.

point_dim_array
Dimensional information array corresponding to *point*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns the projection point (including its measure) of a specified point (*point*). The projection point is on the geometric segment.

If multiple projection points exist, the first projection point encountered from the start point is returned.

An exception is raised if *geom_segment* or *point* has an invalid geometry type or dimensionality, or if *geom_segment* and *point* are based on different coordinate systems.

The *_3D* format of this function (SDO_LRS.PROJECT_PT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

For more information about projecting a point onto a geometric segment, see Section 6.5.9.

Examples

The following example returns the point (9,4,9) on the geometric segment representing Route 1 that is closest to the specified point (9,3,NULL). (This example uses the definitions from the example in Section 6.6.)

```
-- Point 9,3,NULL is off the road; should return 9,4,9
SELECT  SDO_LRS.PROJECT_PT(route_geometry,
  MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) )
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.PROJECT_PT(ROUTE_GEOMETRY,MDSYS.SDO_GEOMETRY(3301,NULL,NULL,MDSYS.SDO_EL
--------------------------------------------------------------------------------
SDO_GEOMETRY(3301, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
9, 4, 9))
```

SDO_LRS.REDEFINE_GEOM_SEGMENT

Format

```
SDO_LRS.REDEFINE_GEOM_SEGMENT(
    geom_segment   IN OUT MDSYS.SDO_GEOMETRY
    [, start_measure  IN NUMBER,
    end_measure      IN NUMBER]);
```

or

```
SDO_LRS.REDEFINE_GEOM_SEGMENT(
    geom_segment   IN OUT MDSYS.SDO_GEOMETRY,
    dim_array          IN MDSYS.SDO_DIM_ARRAY
    [, start_measure  IN NUMBER,
    end_measure      IN NUMBER]);
```

Description

Populates the measures of all shape points based on the start and end measures of a geometric segment, overriding any previously assigned measures between the start point and end point.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Distance measured from the start point of a geometric segment to the start point of the linear feature. The default is the existing value (if any) in the measure dimension; otherwise, the default is 0.

end_measure
Distance measured from the end point of a geometric segment to the start point of the linear feature. The default is the existing value (if any) in the measure dimension; otherwise, the default is the cartographic length of the segment.

Usage Notes

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality, or if *start_measure* or *end_measure* is out of range.

The *_3D* format of this procedure (SDO_LRS.REDEFINE_GEOM_SEGMENT_3D) is available. For information about *_3D* formats of LRS functions and procedures, see Section 6.4.

For more information about redefining a geometric segment, see Section 6.5.2.

Examples

The following example redefines a geometric segment, effectively converting miles to kilometers in the measure values. (This example uses the definitions from the example in Section 6.6.)

```
-- First, display the original segment; then, redefine.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))

-- Redefine geom segment to "convert" miles to kilometers.
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

-- "Convert" mile measures to kilometers (27 * 1.609 = 43.443).
SDO_LRS.REDEFINE_GEOM_SEGMENT (geom_segment,
  dim_array,
```

```
   0,    -- Zero starting measure: LRS segment starts at start of route.
   43.443);  -- End of LRS segment. 27 miles = 43.443 kilometers.

-- Update and insert geometries into table, to display later.
UPDATE lrs_routes a SET a.route_geometry = geom_segment
   WHERE a.route_id = 1;

END;
/

PL/SQL procedure successfully completed.

-- Display the redefined segment, with all measures "converted".
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 3.218, 8, 4, 12.872, 12, 4, 19.308, 12, 10, 28.962, 8, 10, 35.398
, 5, 14, 43.443))
```

SDO_LRS.RESET_MEASURE

Format

```
SDO_LRS.RESET_MEASURE(
    geom_segment   IN OUT MDSYS.SDO_GEOMETRY
    [, dim_array       IN MDSYS.SDO_DIM_ARRAY]);
```

Description

Sets all measures of a geometric segment, including the start and end measures, to null values, overriding any previously assigned measures.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

Examples

The following example sets all measures of a geometric segment to null values. (This example uses the definitions from the example in Section 6.6.)

```
-- First, display the original segment; then, redefine.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))

-- Reset geom segment measures.
DECLARE
```

```
geom_segment MDSYS.SDO_GEOMETRY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';

SDO_LRS.RESET_MEASURE (geom_segment);

-- Update and insert geometries into table, to display later.
UPDATE lrs_routes a SET a.route_geometry = geom_segment
   WHERE a.route_id = 1;

END;
/

PL/SQL procedure successfully completed.

-- Display the segment, with all measures set to null.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, NULL, 2, 4, NULL, 8, 4, NULL, 12, 4, NULL, 12, 10, NULL, 8, 10, NULL, 5, 1
4, NULL))
```

SDO_LRS.REVERSE_GEOMETRY

Format

```
SDO_LRS.REVERSE_GEOMETRY(
    geom          IN MDSYS.SDO_GEOMETRY
    [, dim_array  IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a new geometric segment by reversing the measure values and the direction of the original geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function:

- Reverses the measure values of *geom_segment*

  That is, the start measure of *geom_segment* is the end measure of the returned geometric segment, the end measure of *geom_segment* is the start measure of the returned geometric segment, and all other measures are adjusted accordingly.
- Reverses the direction of *geom_segment*

Compare this function with SDO_LRS.REVERSE_MEASURE, which reverses only the measure values (not the direction) of a geometric segment.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality. The geometry type must be a line or multiline, and the dimensionality must be 3 (two dimensions plus the measure dimension).

The *_3D* format of this function (SDO_LRS.REVERSE_GEOMETRY_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example reverses the measure values and the direction of the geometric segment representing route 1. (This example uses the definitions from the example in Section 6.6.)

```
-- Reverse direction and measures (for example, to prepare for
-- concatenating with another road)
-- First, display the original segment; then, reverse.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))


SELECT  SDO_LRS.REVERSE_GEOMETRY(a.route_geometry, m.diminfo)
    FROM lrs_routes a, user_sdo_geom_metadata m
    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
      AND a.route_id = 1;

SDO_LRS.REVERSE_GEOMETRY(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_PO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 14, 27, 8, 10, 22, 12, 10, 18, 12, 4, 12, 8, 4, 8, 2, 4, 2, 2, 2, 0))
```

Note in the returned segment that the M values (measures) now go in descending order from 27 to 0, and the segment start and end points have the opposite X and Y values as in the original segment (5,14 and 2,2 here, as opposed to 2,2 and 5,14 in the original).

SDO_LRS.REVERSE_MEASURE

Format

```
SDO_LRS.REVERSE_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a new geometric segment by reversing the measure values, but not the direction, of the original geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function:

- Reverses the measure values of *geom_segment*

  That is, the start measure of *geom_segment* is the end measure of the returned geometric segment, the end measure of *geom_segment* is the start measure of the returned geometric segment, and all other measures are adjusted accordingly.

- Does not affect the direction of *geom_segment*

Compare this function with SDO_LRS.REVERSE_GEOMETRY, which reverses both the direction and the measure values of a geometric segment.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.REVERSE_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Note: The behavior of the SDO_LRS.REVERSE_MEASURE function changed between Release 8.1.7 and the current release. In Release 8.1.7, REVERSE_MEASURE reversed both the measures and the segment direction. However, if you want to have this same behavior with the current release, you must use the SDO_LRS.REVERSE_GEOMETRY function.

Examples

The following example reverses the measure values of the geometric segment representing route 1, but does not affect the direction. (This example uses the definitions from the example in Section 6.6.)

```
-- First, display the original segment; then, reverse.
SELECT a.route_geometry FROM lrs_routes a WHERE a.route_id = 1;

ROUTE_GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDIN
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 2, 8, 4, 8, 12, 4, 12, 12, 10, 18, 8, 10, 22, 5, 14, 27))


SELECT SDO_LRS.REVERSE_MEASURE(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.REVERSE_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO_POI
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 27, 2, 4, 25, 8, 4, 19, 12, 4, 15, 12, 10, 9, 8, 10, 5, 5, 14, 0))
```

Note in the returned segment that the M values (measures) now go in descending order from 27 to 0, but the segment start and end points have the same X and Y values as in the original segment (2,2 and 5,14).

SDO_LRS.SCALE_GEOM_SEGMENT

Format

```
SDO_LRS.SCALE_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    start_measure    IN NUMBER,
    end_measure     IN NUMBER,
    shift_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.SCALE_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array          IN MDSYS.SDO_DIM_ARRAY,
    start_measure    IN NUMBER,
    end_measure     IN NUMBER,
    shift_measure    IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the geometry object resulting from the scaling of a geometric segment.

Parameters

geom_segment
Geometric segment to be scaled.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

start_measure
Start measure of the scaled geometric segment.

end_measure
End measure of the scaled geometric segment.

shift_measure
Shift measure of the scaled geometric segment.

Usage Notes

This function performs a general scaling operation to the geometric segment. The new start and end measures are assigned, and all measures are populated by a linear mapping between old and new start and end measures. The shift measure is applied to the segment after scaling.

> **Note:** This general-purpose function has been deprecated and will not be supported in a future release of Spatial. You should instead use other functions for specific purposes, as described in Table 15–4.

Table 15–4 lists some common tasks and the suggested functions to use instead of SCALE_GEOM_SEGMENT.

Table 15–4 Functions to Use Instead of SCALE_GEOM_SEGMENT

Task	Suggested Function
Shift all measures by a specified amount (for example, to accommodate new construction at the start of a road that causes the original start point to be *n* measure units beyond the new start point).	SDO_LRS.TRANSLATE_MEASURE
Reverse the direction of a segment (for example, to allow one road segment to be concatenated with another coming from the opposite direction, because both segments to be concatenated must have the same direction).	SDO_LRS.REVERSE_GEOMETRY
Scale the measure information without performing a shift (for example, to change the measures from miles to kilometers).	SDO_LRS.REDEFINE_GEOM_SEGMENT

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality, or if *start_measure* or *end_measure* is out of range.

For more information about scaling a geometric segment, see Section 6.5.6.

Examples

The following examples illustrate some SCALE_GEOM_ELEMENT uses. (These examples use the definitions from the example in Section 6.6.)

```
-- Shift by 5 (for example, 5-mile segment added before original start)
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo, 0, 27, 5)
    FROM lrs_routes a, user_sdo_geom_metadata m
    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
      AND a.route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,0,27,5)(SDO_GTYPE, SDO_SRI
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 5, 2, 4, 7, 8, 4, 13, 12, 4, 17, 12, 10, 23, 8, 10, 27, 5, 14, 32))

-- "Convert" mile measures to kilometers (27 * 1.609 = 43.443)
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(route_geometry, 0, 43.443, 0)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(ROUTE_GEOMETRY,0,43.443,0)(SDO_GTYPE, SDO_SRID, SDO_P
--------------------------------------------------------------------------------
SDO_GEOMETRY(3302, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 0, 2, 4, 3.218, 8, 4, 12.872, 12, 4, 19.308, 12, 10, 28.962, 8, 10, 35.398
, 5, 14, 43.443))
```

SDO_LRS.SET_PT_MEASURE

Format

```
SDO_LRS.SET_PT_MEASURE(
    geom_segment  IN OUT MDSYS.SDO_GEOMETRY,
    point          IN MDSYS.SDO_GEOMETRY,
    measure        IN NUMBER) RETURN VARCHAR2;
```

or

```
SDO_LRS.SET_PT_MEASURE(
    geom_segment  IN OUT MDSYS.SDO_GEOMETRY,
    dim_array      IN MDSYS.SDO_DIM_ARRAY,
    point          IN MDSYS.SDO_GEOMETRY,
    pt_dim_array   IN MDSYS.SDO_DIM_ARRAY,
    measure        IN NUMBER) RETURN VARCHAR2;
```

or

```
SDO_LRS.SET_PT_MEASURE(
    point      IN OUT MDSYS.SDO_GEOMETRY,
    measure  IN NUMBER) RETURN VARCHAR2;
```

or

```
SDO_LRS.SET_PT_MEASURE(
    point       IN OUT MDSYS.SDO_GEOMETRY,
    dim_array  IN MDSYS.SDO_DIM_ARRAY,
    measure    IN NUMBER) RETURN VARCHAR2;
```

Description

Sets the measure value of a specified point.

Parameters

geom_segment
Geometric segment containing the point.

dim_array
Dimensional information array corresponding to *geom_segment* (in the second format) or *point* (in the fourth format), usually selected from one of the xxx_SDO_GEOM_METADATA views.

point
Point for which the measure value is to be set.

pt_dim_array
Dimensional information array corresponding to *point* (in the second format), usually selected from one of the xxx_SDO_GEOM_METADATA views.

measure
Measure value to be assigned to the specified point.

Usage Notes

The function returns TRUE if the measure value was successfully set, and FALSE if the measure value was not set.

If both *geom_segment* and *point* are specified, the behavior of the procedure depends on whether or not *point* is a shape point on *geom_segment*:

- If *point* is a shape point on *geom_segment*, the measure value of *point* is set.
- If *point* is not a shape point on *geom_segment*, the shape point on *geom_segment* that is nearest to *point* is found, and the measure value of that shape point is set.

The *_3D* format of this function (SDO_LRS.SET_PT_MEASURE_3D) is available; however, only the formats that include the *geom_segment* parameter are available for SDO_LRS.SET_PT_MEASURE_3D. For information about *_3D* formats of LRS functions, see Section 6.4.

An exception is raised if *geom_segment* or *point* is invalid.

Examples

The following example sets the measure value of point (8,10) to 20. (This example uses the definitions from the example in Section 6.6.)

```
-- Set the measure value of point 8,10 to 20 (originally 22).
```

```
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;
result VARCHAR2(32);

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

-- Set the measure value of point 8,10 to 20 (originally 22).
result := SDO_LRS.SET_PT_MEASURE (geom_segment,
  MDSYS.SDO_GEOMETRY(3301, NULL, NULL,
     MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
     MDSYS.SDO_ORDINATE_ARRAY(8, 10, 22)),
  20);

-- Display the result.
DBMS_OUTPUT.PUT_LINE('Returned value = ' || result);

END;
/
Returned value = TRUE

PL/SQL procedure successfully completed.
```

SDO_LRS.SPLIT_GEOM_SEGMENT

Format

```
SDO_LRS.SPLIT_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    split_measure IN NUMBER,
    segment_1     OUT MDSYS.SDO_GEOMETRY,
    segment_2     OUT MDSYS.SDO_GEOMETRY);
```

or

```
SDO_LRS.SPLIT_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    split_measure IN NUMBER,
    segment_1     OUT MDSYS.SDO_GEOMETRY,
    segment_2     OUT MDSYS.SDO_GEOMETRY);
```

Description

Splits a geometric segment into two geometric segments. (This is a procedure, not a function.)

Parameters

geom_segment
Geometric segment to be split.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

split_measure
Distance measured from the start point of a geometric segment to the split point.

segment_1
First geometric segment: from the start point of *geom_segment* to the split point.

segment_2
Second geometric segment: from the split point to the end point of *geom_segment*.

Usage Notes

An exception is raised if *geom_segment* or *split_measure* is invalid.

The directions and measures of the resulting geometric segments are preserved.

The *_3D* format of this procedure (SDO_LRS.SPLIT_GEOM_SEGMENT_3D) is available. For information about *_3D* formats of LRS functions and procedures, see Section 6.4.

For more information about splitting a geometric segment, see Section 6.5.4.

Examples

The following example defines the geometric segment, splits it into two segments, then concatenates those segments. (This example uses the definitions from the example in Section 6.6. The definitions of *result_geom_1*, *result_geom_2*, and *result_geom_3* are displayed in Example 6–4.)

```
DECLARE
geom_segment MDSYS.SDO_GEOMETRY;
line_string MDSYS.SDO_GEOMETRY;
dim_array MDSYS.SDO_DIM_ARRAY;
result_geom_1 MDSYS.SDO_GEOMETRY;
result_geom_2 MDSYS.SDO_GEOMETRY;
result_geom_3 MDSYS.SDO_GEOMETRY;

BEGIN

SELECT a.route_geometry into geom_segment FROM lrs_routes a
  WHERE a.route_name = 'Route1';
SELECT m.diminfo into dim_array from
  user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY';

-- Define the LRS segment for Route1.
SDO_LRS.DEFINE_GEOM_SEGMENT (geom_segment,
  dim_array,
  0,    -- Zero starting measure: LRS segment starts at start of route.
  27);  -- End of LRS segment is at measure 27.
```

```
SELECT a.route_geometry INTO line_string FROM lrs_routes a
  WHERE a.route_name = 'Route1';

-- Split Route1 into two segments.
SDO_LRS.SPLIT_GEOM_SEGMENT(line_string,dim_array,5,result_geom_1,result_geom_2);

-- Concatenate the segments that were just split.
result_geom_3 := SDO_LRS.CONCATENATE_GEOM_SEGMENTS(result_geom_1, dim_array,
result_geom_2, dim_array);

-- Insert geometries into table, to display later.
INSERT INTO lrs_routes VALUES(
  11,
  'result_geom_1',
  result_geom_1
);
INSERT INTO lrs_routes VALUES(
  12,
  'result_geom_2',
  result_geom_2
);
INSERT INTO lrs_routes VALUES(
  13,
  'result_geom_3',
  result_geom_3
);

END;
/
```

SDO_LRS.TRANSLATE_MEASURE

Format

```
SDO_LRS.TRANSLATE_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    translate_m   IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_LRS.TRANSLATE_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    translate_m   IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a new geometric segment by translating the original geometric segment (that is, shifting the start and end measures by a specified value).

Parameters

geom_segment
Cartographic representation of a linear feature.

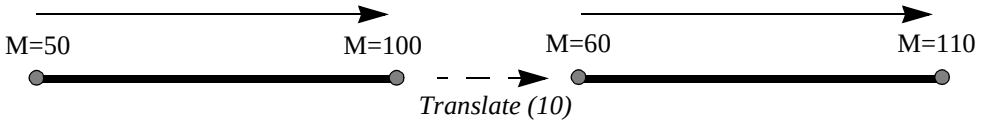
dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

translate_m
Distance measured from the start point of a geometric segment to the start point of the linear feature.

Usage Notes

This function adds *translate_m* to the start and end measures of *geom_segment*. For example, if *geom_segment* has a start measure of 50 and an end measure of 100, and if *translate_m* is 10, the returned geometric segment has a start measure of 60 and an end measure of 110, as shown in Figure 15–1.

Figure 15–1 Translating a Geometric Segment



An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.TRANSLATE_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example translates (shifts) by 10 the geometric segment representing Route 1. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.TRANSLATE_MEASURE(a.route_geometry, m.diminfo, 10)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.TRANSLATE_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO,10)(SDO_GTYPE, SDO_SRID, SD
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
2, 2, 10, 2, 4, 12, 8, 4, 18, 12, 4, 22, 12, 10, 28, 8, 10, 32, 5, 14, 37))
```

SDO_LRS.VALID_GEOM_SEGMENT

Format

```
SDO_LRS.VALID_GEOM_SEGMENT(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if a geometry object is a valid geometric segment.

Parameters

geom_segment
Geometric segment to be checked for validity.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if *geom_segment* is valid and FALSE if *geom_segment* is not valid.

Measure information is assumed to be stored in the last element of the SDO_DIM_ARRAY in the Oracle Spatial metadata.

This function only checks for geometry type and number of dimensions of the geometric segment. To further validate measure information, use the IS_GEOM_SEGMENT_DEFINED function.

The *_3D* format of this function (SDO_LRS.VALID_GEOM_SEGMENT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example checks if the geometric segment representing Route 1 is valid. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.VALID_GEOM_SEGMENT(route_geometry)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.VALID_GEOM_SEGMENT(ROUTE_GEOMETRY)
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.VALID_LRS_PT

Format

```
SDO_LRS.VALID_LRS_PT(
    point          IN MDSYS.SDO_GEOMETRY
    [, dim_array  IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if an LRS point is valid.

Parameters

point
Point to be checked for validity.

dim_array
Dimensional information array corresponding to *point*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if *point* is valid and FALSE if *point* is not valid.

This function checks if *point* is a point with measure information, and it checks for the geometry type and number of dimensions for the point geometry.

Ordinate information needs to be stored in SDO_ELEM_INFO_ARRAY and SDO_ORDINATE_ARRAY. The SDO_POINT field in the SDO_GEOMETRY definition of the point should not be used for LRS points, because SDO_POINT supports the definition of only three attributes (X, Y, Z).

The *_3D* format of this function (SDO_LRS.VALID_LRS_PT_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example checks if point (9,3,NULL) is a valid LRS point. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.VALID_LRS_PT(
    MDSYS.SDO_GEOMETRY(3001, NULL, NULL,
       MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
       MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)),
    m.diminfo)
    FROM lrs_routes a, user_sdo_geom_metadata m
    WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
      AND a.route_id = 1;

SDO_LRS.VALID_LRS_PT(MDSYS.SDO_GEOMETRY(3001,NULL,NULL,MDSYS.SDO_ELEM_INFO_ARRAY
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.VALID_MEASURE

Format

```
SDO_LRS.VALID_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    measure       IN NUMBER
    ) RETURN VARCHAR2;
```

or

```
SDO_LRS.VALID_MEASURE(
    geom_segment  IN MDSYS.SDO_GEOMETRY,
    dim_array     IN MDSYS.SDO_DIM_ARRAY,
    measure       IN NUMBER
    ) RETURN VARCHAR2;
```

Description

Checks if a measure falls within the measure range of a geometric segment.

Parameters

geom_segment
Geometric segment to be checked to see if it falls within the measure range of *measure*.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

measure
Geometric segment to be checked to see if *geom_segment* falls within its measure range.

Usage Notes

This function returns TRUE if *measure* falls within the measure range of *geom_segment* and FALSE if *measure* does not fall within the measure range of *geom_segment*.

An exception is raised if *geom_segment* has an invalid geometry type or dimensionality.

The *_3D* format of this function (SDO_LRS.VALID_MEASURE_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example checks if 50 is a valid measure on the Route 1 segment. The function returns FALSE because the measure range for that segment is 0 to 27. For example, if the route is 27 miles long and there is a mile marker at one-mile intervals, there is no 50-mile marker because the last marker is the 27-mile marker. (This example uses the definitions from the example in Section 6.6.)

```
SELECT  SDO_LRS.VALID_MEASURE(route_geometry, 50)
  FROM lrs_routes WHERE route_id = 1;

SDO_LRS.VALID_MEASURE(ROUTE_GEOMETRY,50)
--------------------------------------------------------------------------------
FALSE
```

SDO_LRS.VALIDATE_LRS_GEOMETRY

Format

```
SDO_LRS.VALIDATE_LRS_GEOMETRY(
    geom_segment  IN MDSYS.SDO_GEOMETRY
    [, dim_array      IN MDSYS.SDO_DIM_ARRAY]
    ) RETURN VARCHAR2;
```

Description

Checks if an LRS geometry is valid.

Parameters

geom_segment
Geometric segment to be checked.

dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function returns TRUE if *geom_segment* is valid and one of the following error codes if *geom_segment* is not valid:

- 13331 (invalid LRS geometry type)
- 13335 (measure information not defined)

The *_3D* format of this function (SDO_LRS.VALIDATE_LRS_GEOMETRY_3D) is available. For information about *_3D* formats of LRS functions, see Section 6.4.

Examples

The following example checks if the Route 1 segment is a valid LRS geometry. (This example uses the definitions from the example in Section 6.6.)

```
SELECT SDO_LRS.VALIDATE_LRS_GEOMETRY(a.route_geometry, m.diminfo)
   FROM lrs_routes a, user_sdo_geom_metadata m
   WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
```

```
   AND a.route_id = 1;

SDO_LRS.VALIDATE_LRS_GEOMETRY(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE
```

16

Migration Procedures

The procedures described in this chapter let you upgrade geometry tables from previous releases of Spatial Cartridge or Spatial Data Option.

This chapter contains descriptions of the migration procedures shown in Table 16–1.

Table 16–1 Migration Procedures

Procedure	Description
SDO_MIGRATE.FROM_815_TO_81X	Migrates data from Spatial release 8.1.5 to the current release.
SDO_MIGRATE.OGIS_METADATA_FROM	Generates a temporary table used when migrating OGIS (OpenGIS) metadata tables.
SDO_MIGRATE.OGIS_METADATA_TO	Reads a temporary table used when migrating OGIS metadata tables.
SDO_MIGRATE.TO_734	Migrates data from a previous release of Spatial Data Option to release 7.3.4.
SDO_MIGRATE.TO_81X	Migrates tables from Spatial Data Option release 7.3.4 or Spatial Cartridge release 8.0.4 to Oracle Spatial.
SDO_MIGRATE.TO_CURRENT	Migrates data from a previous Spatial release to the current release.

SDO_MIGRATE.FROM_815_TO_81X

Format

```
SDO_MIGRATE.FROM_815_TO_81X(
    tabname        IN VARCHAR2
    [, commit_int  IN NUMBER]);
```

Description

Migrates data from Spatial release 8.1.5 to the current release.

> **Note:** You should use the SDO_MIGRATE.TO_CURRENT procedure instead of this procedure.

Parameters

tabname
Table with geometry objects.

commit_int
Number of geometries to migrate before Spatial performs an internal commit operation. If *commit_int* is not specified, no internal commit operations are performed during the migration.

If you specify a *commit_int* value, you can use a smaller rollback segment than would otherwise be needed.

Usage Notes

See Section A.4 for important information about migrating from Spatial release 8.1.5.

All geometry objects in *tabname* will be migrated so that their SDO_GTYPE and SDO_ETYPE values are in the format of the current release:

- SDO_GTYPE values of 4 digits are created, using the format (*d*00*n*) shown in Table 2–1 in Section 2.2.1.
- SDO_ETYPE values are as discussed in Section 2.2.4.

The procedure also orders geometries so that exterior rings are followed by their interior rings, and saves them in the correct rotation (counterclockwise for exterior rings, and clockwise for interior rings).

Examples

The following example changes the definitions of geometry objects in the ROADS table from the release 8.1.5 format to the format of the current release.

```
SQL> execute sdo_migrate.from_815_to_81x('ROADS');
```

SDO_MIGRATE.OGIS_METADATA_FROM

Format

SDO_MIGRATE.OGIS_METADATA_FROM

Description

Called at the source database when migrating from one 8.1.5 database to another 8.1.5 database. The procedure migrates OGIS (OpenGIS) metadata entries from schemas owned by MDSYS.

Parameters

None.

Usage Notes

Consider the following when using this procedure:

- The tables involved are strictly maintained by the user, and not by Spatial. Details are available in the sdocat.sql file and the OpenGIS specification.
- Call this procedure once before migrating the data, and it will generate a temporary table called SDO_GC_MIG. Export the temporary table to the new database and call SDO_MIGRATE.OGIS_METADATA_TO to restore the data.

SDO_MIGRATE.OGIS_METADATA_TO

Format

SDO_MIGRATE.OGIS_METADATA_TO

Description

Used at the destination database when migrating from one 8.1.5 database to another 8.1.5 database. The procedure migrates OGIS (OpenGIS) metadata entries from schemas owned by MDSYS.

Parameters

None.

Usage Notes

Consider the following when using this procedure:

- The tables involved are strictly maintained by the user, and not by Spatial. Details are available in the sdocat.sql file and the OpenGIS specification.
- Call this procedure once after migrating the data. See SDO_MIGRATE.OGIS_METADATA_FROM.

SDO_MIGRATE.TO_734

Format

```
SDO_MIGRATE.TO_734(
    sn          IN VARCHAR2,
    layer       IN VARCHAR2,
    tess_type   IN VARCHAR2,
    param       IN INTEGER);
```

Description

Migrates data from a previous release of Spatial Data Option to release 7.3.4.

Parameters

sn
Schema name of the owner of *layer*.

layer
Name of the layer to be migrated.

tess_type
Type of tessellation (indexing) to be used: FIXED or VARIABLE.

param
Parameter whose significance depends on *tess_type*:

- If *tess_type* is FIXED, *param* is the SDO_LEVEL value.
- If *tess_type* is VARIABLE, *param* is the SDO_NUMTILES value.

Usage Notes

None.

Examples

For fixed-size tessellation:

```
SQL> execute sdo_migrate.to_734('HERMAN', 'ROADS', 'FIXED', 10);
```

For variable-sized tessellation:

```
SQL> execute sdo_migrate_to_734('HERMAN', 'ROADS', 'VARIABLE',4);
```

SDO_MIGRATE.TO_81X

Format

```
SDO_MIGRATE.TO_81X(
    layer          IN VARCHAR2,
    newtabname     IN VARCHAR2,
    gidcolumn      IN VARCHAR2,
    geocolname     IN VARCHAR2,
    layer_gtype    IN VARCHAR2,
    updateflag     IN VARCHAR2);
```

Description

Migrates data from a previous release of Spatial Cartridge or Spatial Data Option to the current release of Oracle Spatial.

Note: You should use the SDO_MIGRATE.TO_CURRENT procedure instead of this procedure.

Parameters

layer
Name of the layer to be migrated.

newtabname
Name of the new table to which you are migrating the data.

gidcolumn
Name of the column in which to store the GID from the old table.

geocolname
Name of the column in the new table where the geometry objects will be inserted.

layer_gtype
One of the following values: POINT or NOTPOINT (default).

If the layer you are migrating is composed solely of point data, set this parameter to POINT for optimal performance; otherwise, set this parameter to NOTPOINT. If you set the value to POINT and the layer contains any nonpoint geometries, the migration might produce invalid data.

updateflag
One of the following values: UPDATE or INSERT (default).

If you are migrating the layer into an existing populated attribute table, set this parameter to UPDATE; otherwise, set this parameter to INSERT.

Usage Notes

Consider the following when using this procedure:

- The new table must be created before calling this procedure.
- The procedure converts geometries from the relational model to the object-relational model.
- A commit operation is performed by this procedure.
- If any of the migration steps fails, nothing is migrated for the layer.
- *layer* is the underlying layer name, without the _SDOGEOM suffix.
- The old SDO_GID is stored in *gidcolumn*.
- SDO_GTYPE values of 4 digits are created, using the format (*d*00*n*) shown in Table 2–1 in Section 2.2.1.
- SDO_ETYPE values are created, using the values discussed in Section 2.2.4.
- The procedure orders geometries so that exterior rings are followed by their interior rings, and saves them in the correct rotation (counter clockwise for exterior rings, and clockwise for interior rings).

Examples

Insert point-only data into new rows:

```
execute sdo_migrate.to_81x('raptor', 'raptor', 'sdo_gid', 'feature', 'point');
```

Insert nonpoint data into new rows:

```
execute sdo_migrate.to_81x('BTU', 'BTU', 'sdo_gid', 'feature');
```

Update point-only data in existing rows:

```
execute sdo_migrate.to_81x('raptor', 'raptor', 'sdo_gid', 'feature',
  'point', 'update');
```

Update nonpoint data in existing rows:

```
execute sdo_migrate.to_81x('BTU', 'BTU', 'sdo_gid', 'feature',
  'notpoint', 'update');
```

SDO_MIGRATE.TO_CURRENT

Format (Any Object-Relational Model Implementation to Current)

```
SDO_MIGRATE.TO_CURRENT(
    tabname            IN VARCHAR2
    [, column_name  IN VARCHAR2]);
or
SDO_MIGRATE.TO_CURRENT(
    tabname          IN VARCHAR2,
    column_name  IN VARCHAR2
    [, commit_int    IN NUMBER]);
```

Format (Any Relational Model Implementation to Current)

```
SDO_MIGRATE.TO_CURRENT(
    layer            IN VARCHAR2,
    newtabname  IN VARCHAR2,
    gidcolumn       IN VARCHAR2,
    geocolname   IN VARCHAR2,
    layer_gtype    IN VARCHAR2,
    updateflag      IN VARCHAR2);
```

Description

Migrates data from a previous Spatial release to the current release. The format depends on whether you are migrating from the Spatial relational model (release 8.1.5 or lower) or object-relational model (release 8.1.6 or higher). See the Usage Notes for the model that applies to you.

You are encouraged to use this procedure instead of the SDO_MIGRATE.FROM_815_TO_81X or SDO_MIGRATE.TO_81X procedure.

Parameters

tabname
Table with geometry objects.

column_name
Column in *tabname* that contains geometry objects. If column_name is not specified or is specified as null, the column containing geometry objects is migrated.

commit_int
Number of geometries to migrate before Spatial performs an internal commit operation. If *commit_int* is not specified, no internal commit operations are performed during the migration.

If you specify a *commit_int* value, you can use a smaller rollback segment than would otherwise be needed.

layer
Name of the layer to be migrated.

newtabname
Name of the new table to which you are migrating the data.

gidcolumn
Name of the column in which to store the GID from the old table.

geocolname
Name of the column in the new table where the geometry objects will be inserted.

layer_gtype
One of the following values: POINT or NOTPOINT (default).

If the layer you are migrating is composed solely of point data, set this parameter to POINT for optimal performance; otherwise, set this parameter to NOTPOINT. If you set the value to POINT and the layer contains any nonpoint geometries, the migration might produce invalid data.

updateflag
One of the following values: UPDATE or INSERT (default).

If you are migrating the layer into an existing populated attribute table, set this parameter to UPDATE; otherwise, set this parameter to INSERT.

Usage Notes for Object-Relational Model Migration

See Section A.4 for important information about migrating from Spatial release 8.1.5.

All geometry objects in *tabname* will be migrated so that their SDO_GTYPE and SDO_ETYPE values are in the format of the current release:

- SDO_GTYPE values of 4 digits are created, using the format (*d*00*n*) shown in Table 2–1 in Section 2.2.1.
- SDO_ETYPE values are as discussed in Section 2.2.4.

The procedure also orders geometries so that exterior rings are followed by their interior rings, and saves them in the correct rotation (counter clockwise for exterior rings, and clockwise for interior rings).

Usage Notes for Relational Model Migration

Consider the following when using this procedure:

- The new table must be created before calling this procedure.
- The procedure converts geometries from the relational model to the object-relational model.
- A commit operation is performed by this procedure.
- If any of the migration steps fails, nothing is migrated for the layer.
- *layer* is the underlying layer name, without the _SDOGEOM suffix.
- The old SDO_GID is stored in *gidcolumn*.
- SDO_GTYPE values of 4 digits are created, using the format (*d*00*n*) shown in Table 2–1 in Section 2.2.1.
- SDO_ETYPE values are created, using the values discussed in Section 2.2.4.
- The procedure orders geometries so that exterior rings are followed by their interior rings, and saves them in the correct rotation (counter clockwise for exterior rings, and clockwise for interior rings).

Examples

The following example changes the definitions of geometry objects in the ROADS table from the release 8.1.5 or higher format to the format of the current release.

```
SQL> execute sdo_migrate.to_current('ROADS');
```

17

Tuning Functions and Procedures

This chapter contains descriptions of the tuning functions and procedures shown in Table 17–1.

Table 17–1 Tuning Functions and Procedures

Function/Procedure	Description
SDO_TUNE.ANALYZE_RTREE	Analyzes an R-tree index; generates statistics about the index use, and recommends a rebuild of the index if a rebuild would improve query performance significantly.
SDO_TUNE.AVERAGE_MBR	Calculates the average minimum bounding rectangle for geometries in a layer.
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE	Estimates the spatial index selectivity.
SDO_TUNE.ESTIMATE_TILING_LEVEL	Determines an appropriate tiling level for creating fixed-size index tiles.
SDO_TUNE.ESTIMATE_TILING_TIME	Estimates the tiling time for a layer, in seconds.
SDO_TUNE.ESTIMATE_TOTAL_NUMTILES	Estimates the total number of spatial tiles for a layer.
SDO_TUNE.EXTENT_OF	Determines the minimum bounding rectangle of the data in a layer.
SDO_TUNE.HISTOGRAM_ANALYSIS	Calculates statistical histograms for a spatial layer.
SDO_TUNE.MIX_INFO	Calculates geometry type information for a spatial layer, such as the percentage of each geometry type.

Table 17–1 Tuning Functions and Procedures

Function/Procedure	Description
SDO_TUNE.QUALITY_DEGRADATION	Returns the quality degradation for an R-tree index or the average quality degradation for all index tables for an R-tree index.
SDO_TUNE.RTREE_QUALITY	Returns the quality score for an R-tree index or the average quality score for all index tables for an R-tree index.

SDO_TUNE.ANALYZE_RTREE

Format

```
SDO_TUNE.ANALYZE_RTREE(
    schemaname  IN VARCHAR2,
    indexname   IN VARCHAR2);
```

Description

Analyzes an R-tree index; generates statistics about the index, and recommends a rebuild of the index if a rebuild would improve query performance significantly.

Parameters

schemaname
Name of the schema that contains the index specified in *indexname*.

indexname
Name of the Spatial R-tree index to be analyzed.

Usage Notes

The procedure computes an index quality score and compares it to the quality score when the index was created or most recently rebuilt (stored as SDO_RTREE_QUALITY in the xxx_INDEX_METADATA views, described in Section 2.5.1). If the comparison of the index quality scores shows that quality has degraded by 50% or more, the procedure recommends that the index be rebuilt.

For R-tree indexes with secondary indexes (created using the ALTER INDEX statement with the *add_index* keyword), the statistics for each index table are printed.

Because this procedure only prints the output to standard output, the SDO_TUNE.RTREE_QUALITY and SDO_TUNE.QUALITY_DEGRADATION functions are also provided, to return the quality score and quality degradation, respectively, for an R-tree index.

For more information about R-tree quality and its effect on query performance, see Section 1.7.1.1.

Examples

The following example analyzes the quality of the COLA_SPATIAL_IDX index.

```
EXECUTE SDO_TUNE.ANALYZE_RTREE('SCOTT', 'COLA_SPATIAL_IDX');
```

The display to standard output might be as follows:

```
--- Quality Statistics for Index table: MDRT_11A5$ ---
 Current Perf. Index : 1.000000
 Previous Perf. Index: 1.000000
 Index Quality: Good, No Rebuild Necessary
```

Related Topics

SDO_TUNE.QUALITY_DEGRADATION

SDO_TUNE.RTREE_QUALITY

SDO_TUNE.AVERAGE_MBR

Format

```
SDO_TUNE.AVERAGE_MBR(
    table_name     IN VARCHAR2,
    column_name  IN VARCHAR2,
    width             OUT NUMBER,
    height            OUT NUMBER);
```

Description

Calculates the average minimum bounding rectangle (MBR) for a geometry object column.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the average minimum bounding rectangle is to be computed.

width
Width of the average minimum bounding rectangle.

height
Height of the average minimum bounding rectangle.

Usage Notes

This procedure computes and stores the width and height of the average minimum bounding rectangle for all geometries in a geometry table. It calculates the average MBR by keeping track of the maximum and minimum X and Y values for all geometries in a geometry table.

AVERAGE_MBR is a procedure, not a function. (Procedures do not return values.)

Examples

The following example calculates the minimum bounding rectangle for the SHAPE column of the COLA_MARKETS table.

```
DECLARE
  table_name  VARCHAR2(32) := 'COLA_MARKETS';
  column_name  VARCHAR2(32) := 'SHAPE';
  width  NUMBER;
  height  NUMBER;
BEGIN
SDO_TUNE.AVERAGE_MBR(
  table_name,
  column_name,
  width,
  height);
DBMS_OUTPUT.PUT_LINE('Width = ' || width);
DBMS_OUTPUT.PUT_LINE('Height = ' || height);
END;
/
Width = 3.5
Height = 4.5
```

Related Topics

SDO_TUNE.EXTENT_OF

SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE

Format

```
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE(
    table_name      IN VARCHAR2,
    column_name  IN VARCHAR2,
    sample_ratio    IN INTEGER,
    tiling_level      IN INTEGER,
    num_tiles        IN INTEGER,
    window_obj      IN MDSYS.SDO_GEOMETRY,
    tiling_time       OUT NUMBER,
    filter_time       OUT NUMBER,
    query_time       OUT NUMBER
    ) RETURN NUMBER;
```

Description

Estimates the spatial index performance such as query selectivity and window query time for a column of type SDO_GEOMETRY.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the tiling time is to be estimated.

sample_ratio
Approximate ratio between the geometries in the original layer and those in the sample layer (to be generated in order to perform the estimate). The default is 20: that is, the sample layer will contain approximately 1/20 (5 percent) of the geometries in the original layer. The larger the *sample_ratio* value, the faster the function will run, but the less accurate will be the result (the estimate).

Note that Spatial obtains the sample by using the SAMPLE(sample_percent) feature internally. For a description of this feature, see the *sample_clause* description in the SELECT statement section of the *Oracle9i SQL Reference*.

tiling_level
Spatial index level at which the layer is to be tessellated.

num_tiles
Number of tiles for variable or hybrid tessellation. Should be 0 for fixed tessellation. The default is 0.

window_obj
Window geometry object.

tiling_time
Estimated tiling time in seconds.

filter_time
Estimated spatial index filter time in seconds.

query_time
Estimated window query time in seconds.

Usage Notes

The function returns a number between 0.0 and 1.0 representing estimated spatial index selectivity. The larger the number, the better the selectivity.

The *sample_ratio* parameter lets you control the trade-off between speed and accuracy. Note that *sample_ratio* is not exact, but reflects an average. For example, a *sample_ratio* value of 20 sometimes causes fewer than 5 percent of geometry objects to be sampled and sometimes more than 5 percent, but over time an average of 5 percent will be sampled.

A return value of 0.0 indicates an error.

Examples

The following example calculates the minimum bounding rectangle for the SHAPE column of the COLA_MARKETS table.

```
DECLARE
  table_name  VARCHAR2(32) := 'COLA_MARKETS';
  column_name  VARCHAR2(32) := 'SHAPE';
  sample_ratio  INTEGER := 15;
```

```
  tiling_level  INTEGER := 4;
  num_tiles  INTEGER := 10;
  window_obj  MDSYS.SDO_GEOMETRY :=
  MDSYS.SDO_GEOMETRY(
    2003,  -- 2-dimensional polygon
    NULL,
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon
    MDSYS.SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
  );
  tiling_time  NUMBER;
  filter_time   NUMBER;
  query_time  NUMBER;
  ret_number NUMBER;
BEGIN
ret_number := SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE(
  table_name,
  column_name,
  sample_ratio,
  tiling_level,
  num_tiles,
  window_obj,
  tiling_time,
  filter_time,
  query_time
);
END;
/
```

SDO_TUNE.ESTIMATE_TILING_LEVEL

Format

```
SDO_TUNE.ESTIMATE_TILING_LEVEL(
    table_name          IN VARCHAR2,
    column_name         IN VARCHAR2,
    num_tiles           IN INTEGER
    [, type_of_estimate  IN VARCHAR2]
    ) RETURN INTEGER;
```

Description

Estimates the appropriate SDO_LEVEL value to use when indexing with hybrid or fixed-size tiles.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the tiling level is to be estimated.

num_tiles
Maximum number of tiles that can be used to index the rectangle defined by *type_of_estimate*.

type_of_estimate
Keyword to specify the type of estimate:

- LAYER_EXTENT -- Uses the rectangle defined by your coordinate system.
- ALL_GID_EXTENT -- Uses the minimum bounding rectangle that encompasses all the geometric objects in the column. This estimate is recommended for most applications.
- AVG_GID_EXTENT (default) -- Uses a rectangle representing the average size of the individual geometric objects within the column. This option is the default

and performs the most analysis of the three types, but it takes the longest time to complete.

Usage Notes

The function returns an integer representing the level to use when creating a spatial index for the specified layer. The function returns NULL if the data is inconsistent.

If *type_of_estimate* is ALL_GID_EXTENT, a *maxtiles* value of 10000 is recommended for most applications.

Examples

The following example estimates the appropriate SDO_LEVEL value to use with the SHAPE column of the COLA_MARKETS table.

```
SELECT SDO_TUNE.ESTIMATE_TILING_LEVEL('COLA_MARKETS',  'SHAPE',
            10000, 'ALL_GID_EXTENT')
  FROM DUAL;

SDO_TUNE.ESTIMATE_TILING_LEVEL('COLA_MARKETS','SHAPE',10000,'ALL_GID_EXTENT')
--------------------------------------------------------------------------------
                                                                               7
```

Related Topics

- SDO_TUNE.EXTENT_OF

SDO_TUNE.ESTIMATE_TILING_TIME

Format

```
SDO_TUNE.ESTIMATE_TILING_TIME(
    table_name      IN VARCHAR2,
    column_name     IN VARCHAR2,
    sample_ratio    IN INTEGER,
    tiling_level    IN INTEGER,
    num_tiles       IN INTEGER
    ) RETURN NUMBER;
```

Description

Returns the estimated time (in seconds) to tessellate a column of type SDO_GEOMETRY.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the tiling time is to be estimated.

sample_ratio
Approximate ratio between the geometries in the original layer and those in the sample layer (to be generated to perform the estimate). The default is 20: that is, the sample layer will contain approximately 1/20 (5 percent) of the geometries in the original layer. The larger the *sample_ratio* value, the faster the function will run, but the less accurate will be the result (the estimate).

Note that Spatial obtains the sample by using the SAMPLE(sample_percent) feature internally. For a description of this feature, see the *sample_clause* description in the SELECT statement section of the *Oracle9i SQL Reference*.

tiling_level
Spatial index level at which the layer is to be tessellated.

num_tiles
Number of tiles for variable or hybrid tessellation. Should be 0 for fixed tessellation. The default is 0.

Usage Notes

A return value of 0 indicates an error.

The tiling time estimate is based on the tiling time of a small sample geometry table that is automatically generated from the original table column. (This generated table is deleted before the function completes.)

The *sample_ratio* parameter lets you control the trade-off between speed and accuracy. Note that *sample_ratio* is not exact, but reflects an average. For example, a *sample_ratio* value of 20 sometimes causes fewer than 5 percent of geometry objects to be sampled and sometimes more than 5 percent, but over time an average of 5 percent will be sampled.

The CREATE TABLE privilege is required for using this function.

Examples

The following example estimates the tiling time to tessellate the REGIONS column of the XYZ_MARKETS table.

```
DECLARE
  table_name  VARCHAR2(32) := 'XYZ_MARKETS';
  column_name  VARCHAR2(32) := 'REGIONS';
  sample_ratio  INTEGER := 15;
  tiling_level  INTEGER := 6;
  num_tiles  INTEGER := 10;
  ret_number  NUMBER;
BEGIN
ret_number := SDO_TUNE.ESTIMATE_TILING_TIME(
  table_name,
  column_name,
  sample_ratio,
  tiling_level,
  num_tiles
);
END;
/
```

SDO_TUNE.ESTIMATE_TOTAL_NUMTILES

Format

```
SDO_TUNE.ESTIMATE_TOTAL_NUMTILES(
    table_name      IN VARCHAR2,
    column_name     IN VARCHAR2,
    sample_ratio    IN INTEGER,
    tiling_level    IN INTEGER,
    num_tiles       IN INTEGER,
    num_largetiles  OUT INTEGER
    ) RETURN INTEGER;
```

Description

Estimates the total number of spatial tiles for a layer.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the total number of spatial tiles is to be estimated.

sample_ratio
Approximate ratio between the geometries in the original layer and those in the sample layer (to be generated to perform the estimate). The default is 20: that is, the sample layer will contain approximately 1/20 (5 percent) of the geometries in the original layer. The larger the *sample_ratio* value, the faster the function will run, but the less accurate will be the result (the estimate).

Note that Spatial obtains the sample by using the SAMPLE(sample_percent) feature internally. For a description of this feature, see the *sample_clause* description in the SELECT statement section of the *Oracle9i SQL Reference*.

tiling_level
Spatial index level at which the layer is to be tessellated.

num_tiles
Number of tiles for variable or hybrid tessellation. Should be 0 for fixed tessellation. The default is 0.

num_largetiles
Output parameter to contain the number of spatial tiles that are of the same size as group tiles for hybrid indexing. (For fixed indexing, *num_largetiles* will be the same as the returned value: the total number of spatial tiles.)

Usage Notes

The estimate is based on the total number of tiles for a small sample layer that is automatically generated from the original layer. (This generated table is deleted before the function completes.)

The *sample_ratio* parameter lets you control the trade-off between speed and accuracy. Note that *sample_ratio* is not exact, but reflects an average. For example, a *sample_ratio* value of 20 sometimes causes fewer than 5 percent of geometry objects to be sampled and sometimes more than 5 percent, but over time an average of 5 percent will be sampled.

The CREATE TABLE privilege is required for using this function.

Examples

The following example estimates the total number of spatial tiles required to index the REGIONS column of the XYZ_MARKETS table.

```
DECLARE
  table_name  VARCHAR2(32) := 'XYZ_MARKETS';
  column_name  VARCHAR2(32) := 'REGIONS';
  sample_ratio  INTEGER := 15;
  tiling_level  INTEGER := 4;
  num_tiles  INTEGER := 10;
  num_largetiles  INTEGER;
  ret_integer  INTEGER;
BEGIN
ret_integer := SDO_TUNE.ESTIMATE_TOTAL_NUMTILES(
  table_name,
  column_name,
  sample_ratio,
  tiling_level,
  num_tiles,
  num_largetiles
);
```

```
END;
/
```

SDO_TUNE.EXTENT_OF

Format

```
SDO_TUNE.EXTENT_OF(
    table_name      IN VARCHAR2,
    column_name  IN VARCHAR2
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the minimum bounding rectangle of all geometries in a column of type SDO_GEOMETRY.

Parameters

table_name
Geometry table.

column_name
Geometry column for which the minimum bounding rectangle is to be returned.

Usage Notes

The function returns NULL if the data is inconsistent.

> **Note:** This function is deprecated, and will not be supported in future versions of Spatial. You are instead encouraged to use the SDO_AGGR_MBR function, documented in Chapter 13, to return the MBR of geometries. The SDO_TUNE.EXTENT_OF function is limited to 2-dimensional geometries, whereas the SDO_AGGR_MBR function is not.

Examples

The following example calculates the minimum bounding rectangle for the objects in the SHAPE column of the COLA_MARKETS table.

```
SELECT SDO_TUNE.EXTENT_OF('COLA_MARKETS',  'SHAPE')
```

```
  FROM DUAL;

SDO_TUNE.EXTENT_OF('COLA_MARKETS','SHAPE')(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y,
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3), SDO_ORDINATE_
ARRAY(1, 1, 10, 11))
```

Related Topics

SDO_AGGR_MBR (in Chapter 13)

SDO_TUNE.ESTIMATE_TILING_LEVEL

SDO_TUNE.AVERAGE_MBR procedure

SDO_TUNE.HISTOGRAM_ANALYSIS

Format

```
SDO_TUNE.HISTOGRAM_ANALYSIS(
    table_name          IN VARCHAR2,
    column_name         IN VARCHAR2,
    result_table        IN VARCHAR2,
    type_of_histogram   IN VARCHAR2,
    max_value           IN NUMBER,
    intervals           IN INTEGER);
```

Description

Generates statistical histograms based on columns of type SDO_GEOMETRY.

Parameters

table_name
Geometry table.

column_name
Geometry object column for which the histogram is to be computed.

result_table
Result table to hold the histogram.

type_of_histogram
Keyword to specify the type of histogram:

- TILES_VS_LEVEL (default) -- Provides the number of tiles at different spatial index levels. (Available only with hybrid indexes.) This histogram is the default, and is used to evaluate the spatial index that is already built on the geometry column.
- GEOMS_VS_TILES -- Provides the number of geometries in different number-of-tiles ranges. This histogram is used to evaluate the spatial index that is already built on the geometry column.

- GEOMS_VS_AREA -- Provides the number of geometries in different size ranges. The shape of this histogram could be helpful in choosing a proper index type and index level
- GEOMS_VS_VERTICES -- Provides a histogram of the geometry count against the number of vertices. This histogram could help determine if spatial index selectivity is important for the layer. Because the number of vertices determines the performance of the secondary filter, selectivity of the primary filter could be crucial for layers that contain many complicated geometries.

max_value
The upper limit of the histogram. That is, the histogram runs in range (0, *max_value*).

intervals
Number of intervals between 0 and *max_value.*

Usage Notes

The procedure populates the result table with statistical histograms for a geometry table. (HISTOGRAM_ANALYSIS is a procedure, not a function. Procedures do not return values.)

Before calling this procedure, create the result table (*result_table* parameter) with VALUE and COUNT columns. For example:

```
CREATE TABLE histogram (value NUMBER, count NUMBER);
```

SDO_TUNE.MIX_INFO

Format

```
SDO_TUNE.MIX_INFO(
    table_name      IN VARCHAR2,
    column_name     IN VARCHAR2
    [, total_geom   OUT INTEGER,
    point_geom      OUT INTEGER,
    curve_geom      OUT INTEGER,
    poly_geom       OUT INTEGER,
    complex_geom    OUT INTEGER] );
```

Description

Provides information about each geometry type stored in a column of type SDO_GEOMETRY.

Parameters

table_name
Geometry table.

column_name
Geometry object column for which the geometry type information is to be calculated.

total_geom
Total number of geometry objects.

point_geom
Number of point geometry objects.

curve_geom
Number of curve string geometry objects.

poly_geom
Number of polygon geometry objects.

complex_geom
Number of complex geometry objects.

Usage Notes

This procedure calculates geometry type information for the table. It calculates the total number of geometries, as well as the number of point, curve string, polygon, and complex geometries.

Examples

The following example displays information about the mix of geometry objects in the SHAPE column of the COLA_MARKETS table.

```
EXECUTE SDO_TUNE.MIX_INFO('COLA_MARKETS',  'SHAPE');
Total number of geometries: 4
Point geometries:        0  (0%)
Curvestring geometries:   0  (0%)
Polygon geometries:      4  (100%)
Complex geometries:      0  (0%)
```

SDO_TUNE.QUALITY_DEGRADATION

Format

```
SDO_TUNE.QUALITY_DEGRADATION(
    schemaname  IN VARCHAR2,
    indexname       IN VARCHAR2
    [, indextable     IN VARCHAR2]
    ) RETURN NUMBER;
```

Description

Returns the quality degradation for an R-tree index or the average quality degradation for all index tables for an R-tree index.

Parameters

schemaname
Name of the schema that contains the index specified in *indexname*.

indexname
Name of the Spatial R-tree index.

indextable
Name of the index table associated with the index specified in *indexname*. (This parameter is appropriate only if multiple index tables have been created using the ALTER INDEX statement with the *add_index* keyword.)

Usage Notes

The **quality degradation** is a number indicating approximately how much longer it will take to execute any given query with the current index (or index table) compared to executing the same query when the index was created or most recently rebuilt. For example, if a typical query will probably take twice as much time as when the index was created or rebuilt, the quality degradation is 2.

If *indextable* is not specified, the function returns the average quality degradation for all index tables associated with *indexname* if multiple index tables have been created for the R-tree index. If multiple index tables have not been created (that is, if only

one index table exists for the index), the quality degradation for the index is returned.

Index names and index table names are available through the xxx_SDO_INDEX_INFO and xxx_SDO_INDEX_METADATA views, which are described in Section 2.5.1.

For more information about R-tree quality and its effect on query performance, see Section 1.7.1.1.

Examples

The following example returns the quality degradation for the COLA_SPATIAL_IDX index. In this example, the quality has not degraded at all, and therefore the degradation is 1; that is, queries will typically take the same time using the current index as using the original or previous index.

```
SELECT SDO_TUNE.QUALITY_DEGRADATION('SCOTT', 'COLA_SPATIAL_IDX') FROM DUAL;

SDO_TUNE.QUALITY_DEGRADATION('SCOTT','COLA_SPATIAL_IDX')
--------------------------------------------------------
                                                       1
```

Related Topics

SDO_TUNE.ANALYZE_RTREE

SDO_TUNE.RTREE_QUALITY

SDO_TUNE.RTREE_QUALITY

Format

```
SDO_TUNE.RTREE_QUALITY(
    schemaname  IN VARCHAR2,
    indexname       IN VARCHAR2
    [, indextable     IN VARCHAR2]
    ) RETURN NUMBER;
```

Description

Returns the quality score for an R-tree index table or the average quality score for all index tables for an R-tree index.

Parameters

schemaname
Name of the schema that contains the index specified in *indexname*.

indexname
Name of the Spatial R-tree index.

indextable
Name of the index table associated with the index specified in *indexname*. (This parameter is appropriate only if multiple index tables have been created using the ALTER INDEX statement with the *add_index* keyword.)

Usage Notes

If *indextable* is not specified, the function returns the average quality score for all index tables associated with *indexname* if multiple index tables have been created for the R-tree index. If multiple index tables have not been created (that is, if only one index table exists for the index), the quality score for the index is returned.

Index names and index table names are available through the xxx_SDO_INDEX_INFO and xxx_SDO_INDEX_METADATA views, which are described in Section 2.5.1.

This function can be useful in determining the quality of an R-tree and whether or not an R-tree index should be rebuilt in order to improve query performance. You can compare the index quality score returned by the function to the quality score at the time the index was created or most recently rebuilt (stored as SDO_RTREE_QUALITY in the xxx_INDEX_METADATA views, described in Section 2.5.1).

For more information about R-tree quality and its effect on query performance, see Section 1.7.1.1.

Examples

The following example returns the current quality score for the COLA_SPATIAL_IDX index.

```
SELECT SDO_TUNE.RTREE_QUALITY('SCOTT', 'COLA_SPATIAL_IDX') FROM DUAL;

SDO_TUNE.RTREE_QUALITY('SCOTT','COLA_SPATIAL_IDX')
--------------------------------------------------
                                                 1
```

Related Topics

SDO_TUNE.ANALYZE_RTREE

SDO_TUNE.QUALITY_DEGRADATION

A

Installation, Compatibility, and Migration

This appendix provides information concerning installation, compatibility, and migration between various Oracle Spatial product releases.

You must upgrade both the database server and Spatial at the same time if you wish to use older spatial applications with an Oracle9*i* release of Spatial. Spatial must always be synchronized with the Oracle9*i* database server on upgrade or downgrade. In both cases, Spatial must be reinstalled.

A.1 Introduction

Many of the Spatial release 9*i* features depend on new features in release 9*i* of the database server. Therefore, there are compatibility and migration issues that need to be addressed in this release of Spatial. This appendix outlines the database and application compatibility issues.

An upgrade or downgrade of the database server version requires a corresponding upgrade or downgrade of Spatial. If an Oracle8*i* (8.1.5, 8.1.6, or 8.1.7) database server is upgraded to an Oracle9*i* database server, Spatial must also be upgraded. Similarly, if an Oracle9*i* database server is downgraded, Spatial must be downgraded too. Lastly, if an Oracle9*i* database server is running in Oracle8*i* compatibility mode, features that are new for Spatial in release 9.0.1 (Oracle9*i*) will not work.

In summary:

- The Spatial release and the Oracle database server release must match.
- Upgrade and downgrade scripts must be run when upgrading or downgrading between Oracle8*i* and Oracle9*i*.

A.2 Installation of Spatial

This section applies to new users of Oracle Spatial. If you are upgrading from a previous release of Spatial, see Section A.4 for migration information.

When you install Oracle release 9.0.1, the option to install Spatial is preselected by default. If you accept this default, you do not need to perform the installation steps described in this section, because the MDSYS user is already created and locked automatically.

If you create an Oracle database using the Database Configuration Assistant (DBCA), Spatial is installed by default and you do not need to perform the installation steps described in this section.

If you did not select the option to install Spatial at installation time and you want to install Spatial later, follow these steps.

Note: Installation of Spatial for release 9.0.1 requires that the COMPATIBLE init.ora parameter is set to 9.0.0.0.0 or higher. This is required for the creation and definition of Spatial index types and operators. Thus, if the database was created with a compatibility parameter value of 8.n.n.n.n, the DBA must shut down the database and restart with COMPATIBLE=9.0.n.n.n.

---

1. Connect as SYS to the Oracle9*i* instance as SYSTEM AS SYSDBA.
2. Create the MDSYS user with a command in the following format:

   ```
   CREATE USER MDSYS IDENTIFIED BY <password>;
   ```

3. Grant the required privileges to the MDSYS user by running the following procedure:

   ```
   @ORACLE_HOME/md/admin/mdprivs.sql
   ```

4. Connect as MDSYS.
5. Install Spatial by running the following procedure:

   ```
   @ORACLE_HOME/md/admin/catmd.sql
   ```

After you install Spatial, it is strongly recommended that you lock the MDSYS user. The MDSYS user is created with administrator privileges; therefore, it is important to protect this account from unauthorized use. To lock the MDSYS user, connect as SYS and enter the following command:

```
ALTER USER MDSYS ACCOUNT LOCK;
```

A.3 Changing from Oracle9*i* to Oracle8*i* Compatibility Mode

If Spatial has been installed and the database compatibility needs to be reset to 8.1.n.n.n from 9.0.n.n.n, enter the following while connected as SYSTEM:

```
ALTER DATABASE RESET COMPATIBILITY
SHUTDOWN
<Change  the init.ora parameter COMPATIBLE=8.1.0.0.0>
STARTUP
```

Note that performing this operation disables function-based spatial indexing and partitioning support for spatial indexing.

A.4 Migrating from Spatial Release 8.1.5, 8.1.6, or 8.1.7

If you are upgrading from Spatial release 8.1.5, 8.1.6, or 8.1.7 to Spatial for release 9*i*, and if you have not chosen the automatic upgrade option, perform the following steps to migrate to Spatial for release 9*i*.

Note: The following steps are not necessary if you chose the Oracle Installer option for an automatic upgrade.

If you have linear referencing system (LRS) data, you must perform the steps in Section A.5 regardless of whether or not you chose an automatic upgrade.

---

1. Make sure that the Oracle RDBMS is upgraded to release 9*i*.
2. Connect as SYSTEM AS SYSDBA.
3. Grant the required privileges to the MDSYS user by running the following procedure:

   ```
   $ORACLE_HOME/md/admin/mdprivs.sql
   ```

4. Connect as MDSYS.
5. Perform the migration by running the following procedure:

   ```
   $ORACLE_HOME/md/admin/c81Xu900.sql
   ```

A.5 LRS Data Migration

If you have linear referencing data (that is, geometries with measure information), you must migrate that data to the Spatial release 9.0.1 format, as follows:

1. Drop any spatial indexes on the table with the linear referencing data.
2. Find out which dimension of the object has the linear referencing information.

   This could be the third or the fourth dimension, depending on the dimensionality of the data. For example, if the data has 3 dimensions (such as X, Y, and height), the LRS geometry object is 4D, and the LRS dimension in this case is usually 4.
3. Make sure that the data is in the format for release 8.1.6 or higher (that is, it has 4-digit SDO_GTYPE values).
4. Update the LRS geometry objects by setting the LRS dimension in the SDO_GTYPE field, as in the following examples.

   Example 1: The LRS dimension is 3 for the geometries in the GEOMETRY column of table LRS_DATA. Update the SDO_GTYPE as follows:

   ```
   UPDATE LRS_DATA a SET a.geometry.sdo_gtype = a.geometry.sdo_gtype + 300;
   ```

   Example 2: The LRS dimension is 4 for the geometries in the GEOMETRY column of table LRS_DATA. Update the SDO_GTYPE as follows:

   ```
   UPDATE LRS_DATA a SET a.geometry.sdo_gtype = a.geometry.sdo_gtype + 400;
   ```

B

Hybrid Indexing

Quadtree hybrid indexing uses a combination of fixed-size and variable-sized tiles for spatially indexing a layer. Variable-sized tile spatial indexing uses tiles of different sizes to approximate a geometry. For each geometry, you will have a set of fixed-size tiles that fully cover the geometry, and also a set of variable-sized tiles that fully cover the geometry.

For most applications, you should not use hybrid indexes, but should instead use quadtree fixed indexes or R-tree indexes. The rare circumstances where hybrid indexes should be considered are as follows:

- When joins are required between layers whose optimal fixed index level (SDO_LEVEL) values are significantly different (4 levels or more), it may be possible to get better performance by bringing the layer with a higher optimal SDO_LEVEL down to the lower SDO_LEVEL and adding the SDO_NUMTILES parameter to ensure adequate tiling of the layer.

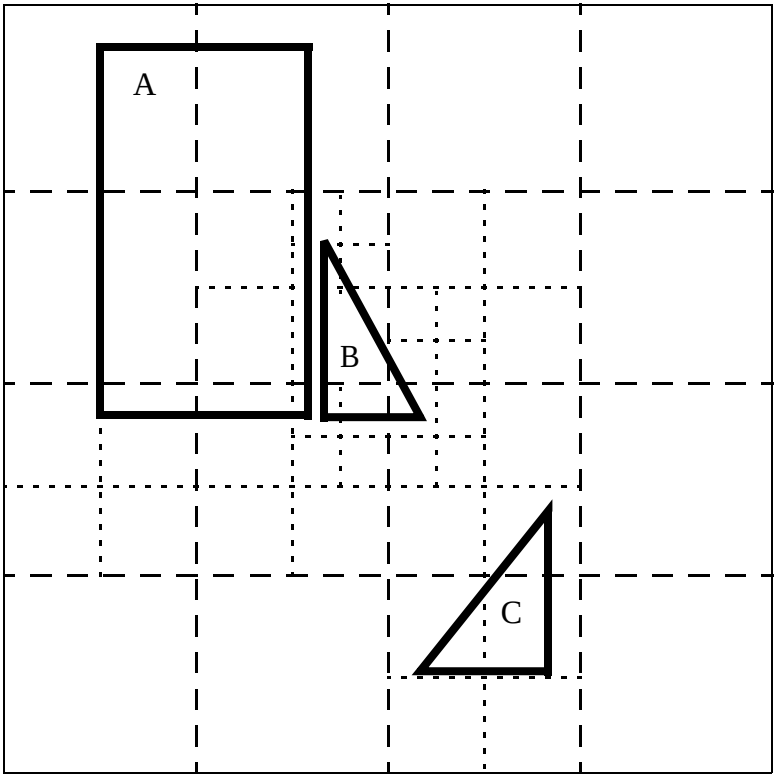
  The best starting value for SDO_NUMTILES in the new hybrid layer can be calculated by getting a count of the rows in the spatial index table and dividing this number by the number of rows with geometries in the layer, then rounding up. A spatial join ('QUERYTYPE=JOIN') is not a common requirement for applications, and it is comparable to a spatial cross product where each of the geometries in one layer will be compared with each of the geometries in the other layer.

- When both of the following are true for a single layer, hybrid indexing may be preferable: (1) the layer has a mixture of many geometries covering a very small area and many polygons covering a very large area; and (2) the optimal fixed tiling level for the very small geometries will result in an extremely large number of tiles to be generated for the very large geometries, causing the spatial index to grow to an unreasonable size.

If both of these conditions are true, it may be better to use the SDO_NUMTILES parameter to get coverage for the smaller geometries, while keeping the fixed tile size relatively large for the large geometries by using a smaller SDO_LEVEL value.

In Figure B–1, the variable-sized cover tiles closely approximate each geometry. This results in good selectivity. The number of variable tiles needed to cover a geometry is controlled using the SDO_NUMTILES parameter.

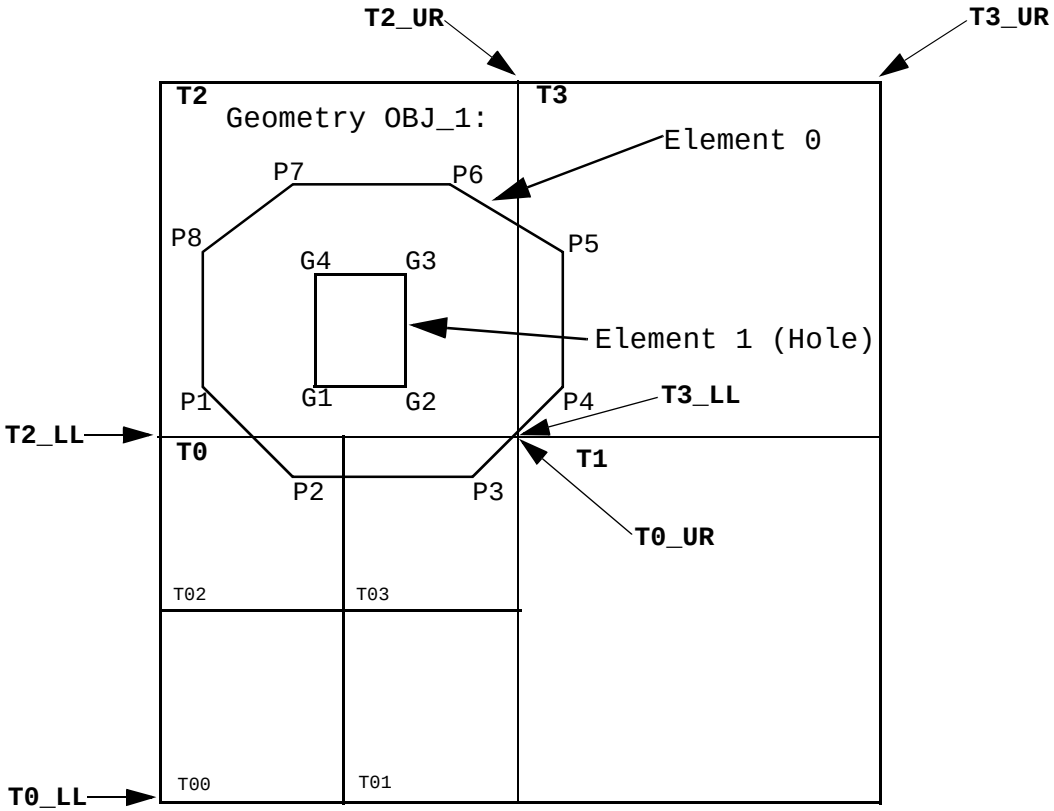
Figure B–1 Variable-Sized Tile Spatial Indexing



A variable tile is subdivided if it interacts with the geometry, and subdivision will not result in tiles that are smaller than a predetermined size. This size, or tiling resolution, is determined by a default SDO_MAXLEVEL value.

Figure B–2 illustrates how geometry OBJ_1 is approximated with hybrid indexing (SDO_LEVEL = 1 and SDO_NUMTILES = 4). These are not recommended values for SDO_LEVEL and SDO_NUMTILES; they were chosen to simplify this example. The cover tiles are stored in the SDOINDEX table as shown in Table B–1.

Figure B–2 Decomposition of the Geometry



In Figure B–2, note that for simplicity the tiles have been numbered, and *LL* and *UR* indicate *lower left* and *upper right*, respectively. For example, T2_LL indicates the lower left corner of tile T2. (This designation scheme does not reflect the actual format use in Spatial.)

In Figure B–2, note which fixed-size tiles are associated with geometry OBJ_1. Only three (T0, T2, T3) of the four large tiles (T0, T1, T2, T3) generated by the tessellation

actually interact with the geometry. Only those three are stored in the SDOINDEX table. In examining which variable-sized tiles are used, tile T0 shows a further tessellation to four smaller tiles, two of which (T02, T03) are used to cover a portion of the geometry. The variable-sized tiles are stored in the SDO_CODE column in the Spatial index table. The fixed-size tiles are stored in the SDO_GROUPCODE column. The spatial index structure is discussed in Section 2.5.

Table B–1 shows the tiles from Figure B–2 that are stored in the SDOINDEX table.

Table B–1 Section of the SDOINDEX Table

SDO_ROWID <RAW>	SDO_CODE <RAW>	SDO_ MAXCODE <RAW>	SDO_ GROUPCODE <RAW>	SDO_META <RAW>
GID_OBJ_1	T02	<binary data>	T0	<binary data>
GID_OBJ_1	T03	<binary data>	T0	<binary data>
GID_OBJ_1	T2	<binary data>	T2	<binary data>
GID_OBJ_1	T3	<binary data>	T3	<binary data>

As with the fixed-size tile model, all elements in a geometry are tessellated in one step. In a multielement geometry like OBJ_1, Element 1 (the hole shown in Figure B–2) is covered by a redundant tile (T2) from the tessellation of Element 0, but this tile is stored only once.

The SDO_TUNE package has some functions that help determine appropriate SDO_ LEVEL and SDO_NUMTILES values.

B.1 Creating a Hybrid Index

This section describes hybrid indexing, which uses both fixed-size and variable-sized tiles as a spatial indexing mechanism. For each geometry, you will have a set of fixed-size tiles that fully covers the geometry, and a set of variable-sized tiles that fully covers the geometry. The terms hybrid indexing, hybrid tiling, and hybrid tessellation are used interchangeably in this section.

To use hybrid tiling, the SDO_LEVEL and SDO_NUMTILES keywords in the PARAMETERS clause must contain valid values. Both SDO_LEVEL and SDO_ NUMTILES must be greater than 1.

The SDO_NUMTILES value determines the number of variable tiles that will be used to fully cover a geometry being indexed. Typically this value is small. For points, SDO_NUMTILES is always one. For other element types, you might set SDO_NUMTILES to a value around 8. The larger the SDO_NUMTILES value, the better the tiles will approximate the geometry being covered. A larger SDO_NUMTILES value improves the selectivity of the primary filter, but it also increases the number of index entries per geometry (see Section 4.2.2.1 and Section 4.2.2.2 for a discussion of primary and secondary filters). The SDO_NUMTILES value should be larger for long, linear spatial entities, such as major highways or rivers, than for area-related spatial entities such as county or state boundaries.

The SDO_LEVEL value determines the size of the fixed tiles used to fully cover the geometry being indexed. Setting the proper SDO_LEVEL value may appear more like art than science. Performing some simple data analysis and testing puts the process back in the realm of science. One approach would be to use the SDO_TUNE.ESTIMATE_TILING_LEVEL function to determine an appropriate starting SDO_LEVEL value, and then compare the performance with slightly higher or lower values.

In Example B–1, assume that data has been loaded into a table called ROADS, and the USER_SDO_GEOM_METADATA view has an entry for ROADS.SHAPE. (Assume also that no spatial index has already been created on the ROADS.SHAPE column.) You can use the following SQL statement to create a hybrid index named ROADS_HYBRID.

Example B–1 Creating a Hybrid Index

```
CREATE INDEX ROADS_HYBRID ON ROADS(SHAPE)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX PARAMETERS('SDO_LEVEL=6 SDO_NUMTILES=12');
```

B.2 Tuning Considerations with Hybrid Indexes

Hybrid indexing allows indexes to be built using the tiling mechanism by specifying the SDO_LEVEL. Additionally, hybrid indexing introduces the ability to specify the minimum number of tiles to be created for each geometry during the indexing process by specifying the indexing parameter SDO_NUMTILES.

If the number of tiles created for a geometry using the SDO_LEVEL value is less than the value specified by the SDO_NUMTILES value, then the indexing process continues by creating more tiles for the geometry until the SDO_NUMTILES value has been reached. The ability to specify the minimum number of tiles for each geometry is important for a number of reasons:

- It ensures that all geometries will have at least as many index entries as the value of SDO_NUMTILES, regardless of the tiling level.
- It can reduce (as compared to fixed indexing) the space required for index data to get full indexing coverage of all geometries.
- Special performance enhancing algorithms have been coded within Spatial to make use of hybrid indexes.

If hybrid indexing is used and if the layer being indexed is point-only data, the SDO_NUMTILES value should be set to 1.

C

Locator

Oracle9*i* Locator (also referred to as Locator) supports the geocoding, storage, and retrieval of geocoded spatial data in Oracle9*i* databases. Locator is not designed to be an end-user application, but is a set of spatial capabilities for application developers.

Locator has been enhanced for release 9.0.1. In general, it includes the data types, operators, and indexing capabilities of Oracle Spatial, along with a limited set of the functions and procedures of Spatial. The Locator features consist of the following:

- An object type (SDO_GEOMETRY) that describes and supports any type of geometry
- A spatial indexing capability that lets you create R-tree or quadtree-based spatial indexes on geometry data
- A geocode result object type that describes the geocode result definition
- A call interface described by two geocode result functions used for geocoding spatial data that also contains the output geocode result object and the geometry object
- Spatial operators that use the spatial index for performing spatial queries: SDO_FILTER, SDO_RELATE, SDO_NN, SDO_NN_DISTANCE, and SDO_WITHIN_DISTANCE

> **Note:** For Oracle8*i*, LOCATOR_WITHIN_DISTANCE was the only Locator operator supported. For Oracle9*i*, LOCATOR_WITHIN_DISTANCE and SDO_WITHIN_DISTANCE are synonyms (same signatures and behavior), and both are supported.

For information about spatial concepts, the SDO_GEOMETRY object type, and indexing and loading spatial data, see Chapters 1 through 4 in this manual. For reference and usage information about features supported by Locator, see the chapters listed in Table C–1.

Table C–1 Spatial Features Supported for Locator

Chapter	Supported Locator Features
Chapter 7	Geocoding
Chapter 9	SQL statements for creating, altering, and deleting indexes
Chapter 10	SDO_GEOMETRY object type methods
Chapter 11	Spatial operators

Glossary

area

An extent or region of dimensional space.

attribute

Descriptive information characterizing a geographical feature such as a point, line, or area.

attribute data

Nondimensional data that provides additional descriptive information about multidimensional data, for example a class or feature such as a bridge or a road.

authalic sphere

A sphere that has the same surface area as a particular oblate ellipsoid of revolution representing the figure of the Earth.

batch geocoding

An operation that simultaneously geocodes many records from one table. *See also* geocoding.

boundary

1. The lower or upper extent of the range of a dimension, expressed by a numeric value.
2. The line representing the outline of a polygon.

Cartesian coordinate system

A coordinate system in which the location of a point in *n*-dimensional space is defined by distances from the point to the reference plane. Distances are measured

parallel to the planes intersecting a given reference plane. *See also* coordinate system.

contain

To describe a geometric relationship where one object encompasses another and the inner object does not touch any boundaries of the outer. The outer object *contains* the inner object. *See also* inside.

convex hull

A simple convex polygon that completely encloses the associated geometry object.

coordinate

A set of values uniquely defining a point in an *n*-dimensional coordinate system.

coordinate system

A reference system for the unique definition for the location of a point in *n*-dimensional space. Also called a *spatial reference system*.

cover

To describe a geometric relationship in which one object encompasses another and the inner object touches the boundary of the outer object in one or more places.

data dictionary

A repository of information about data. A data dictionary stores relational information on all the objects in a database.

datum transformation

See transformation.

decompose

To separate or resolve into constituent parts or elements, or into simpler compounds.

dimensional data

Data that has one or more dimensional components and is described by multiple values.

direction

The direction of an LRS geometric segment is indicated from the start point of the geometric segment to the end point. Measures of points on a geometric segment always increase along the direction of the geometric segment.

disjoint

A geometric relationship where two objects do not interact in any way. Two *disjoint* objects do not share any element or piece of their geometry.

equal

A geometric relationship in which two objects are considered to represent the same geometric figure. The two objects must be composed of the same number of points, however, the ordering of the points defining the two objects' geometries may differ (clockwise or counterclockwise).

extent

A rectangle bounding a map, the size of which is determined by the minimum and maximum map coordinates.

feature

An object with a distinct set of characteristics in a spatial database.

geocoding

The process of converting tables of address data into standardized address, location, and possibly other data.

geodetic coordinates

Angular coordinates (longitude and latitude), closely related to spherical polar coordinates, and are defined relative to a particular Earth geodetic datum. Also referred to as geographic coordinates.

geodetic datum

A means of representing the figure of the Earth, usually as an oblate ellipsoid of revolution, that approximates the surface of the Earth locally or globally, and is the reference for the system of geodetic coordinates.

geographic coordinates

See geodetic coordinates.

geographical information system (GIS)

A computerized database management system used for the capture, conversion, storage, retrieval, analysis, and display of spatial data.

geographically referenced data

See spatiotemporal data.

geometry

The geometric representation of the shape of a spatial feature in some coordinate space.

georeferenced data

See spatiotemporal data.

GIS

See geographical information system (GIS).

grid

A data structure composed of points located at the nodes of an imaginary grid. The spacing of the nodes is constant in both the horizontal and vertical directions.

hole

A polygon can include subelements that negate sections of its interior. For example, consider a polygon representing a map of buildable land with an inner polygon (a hole) representing where a lake is located.

homogeneous

Spatial data of one feature type such as points, lines, or regions.

hyperspatial data

In mathematics, any space having more than the three standard *x*, *y*, and *z* dimensions, also referred to as multidimensional data.

index

Identifier that is not part of a database and used to access stored information.

inside

To describe a geometric relationship where one object is surrounded by a larger object and the inner object does not touch the boundary of the outer. The smaller object is *inside* the larger. *See also* contain.

key

A field in a database used to obtain access to stored information.

keyword

Synonym for reserved word.

latitude

North / South position of a point on the Earth defined as the angle between the normal to the Earth's surface at that point and the plane of the equator.

layer

A collection of geometries having the same attribute set and stored in a geometry column.

line

A geometric object represented by a series of points, or inferred as existing between two coordinate points.

linear feature

Any spatial object that can be treated as a logical set of linear segments.

local coordinates

Cartesian coordinates in a non-Earth (non-georeferenced) coordinate system.

longitude

East / West position of a point on the Earth defined as the angle between the plane of a reference meridian and the plane of a meridian passing through an arbitrary point.

measure

The linear distance (in the LRS measure dimension) measured from the start point of the geometric segment.

measure range

The measure values at the start and end measures of a geometric segment.

multidimensional data

See hyperspatial data.

offset

The perpendicular distance between a point along a geometric segment and the geometric segment. Offsets are positive if points are on the left side along the segment direction and are negative if they are on the right side. Points are on a geometric segment if their offsets to the segment are zero.

polygon

A class of spatial objects having a nonzero area and perimeter, and representing a closed boundary region of uniform characteristics.

primary filter

The operation that permits fast selection of candidate records to pass along to the secondary filter. The primary filter compares geometry approximations to reduce computation complexity and is considered a lower-cost filter. Because the primary filter compares geometric approximations, it returns a superset of the exact result set. *See also* secondary filter *and* two-tier query model.

projected coordinates

Planar Cartesian coordinates that result from performing a mathematical mapping from a point on the Earth's surface to a plane. There are many such mathematical mappings, each used for a particular purpose.

projection

The point on the LRS geometric segment with the minimum distance to the specified point.

proximity

A measure of inter-object distance.

query

A set of conditions or questions that form the basis for the retrieval of information from a database.

query window

Area within which the retrieval of spatial information and related attributes is performed.

RDBMS

See Relational Database Management System (RDBMS).

recursion

A process, function, or routine that executes continuously until a specified condition is met.

region

An extent or area of multidimensional space.

Relational Database Management System (RDBMS)

A computer program designed to store and retrieve shared data. In a relational system, data is stored in tables consisting of one or more rows, each containing the same set of columns. Oracle9*i* is an object-relational database management system. Other types of database systems are called hierarchical or network database systems.

resolution

The number of subdivision levels of data.

scale

The ratio of the distance on a map, photograph, or image to the corresponding image on the ground, all expressed in the same units.

secondary filter

The operation that applies exact computations to geometries that result from the primary filter. The secondary filter yields an accurate answer to a spatial query. The secondary filter operation is computationally expensive, but it is only applied to the primary filter results, not the entire data set. *See also* primary filter *and* two-tier query model.

shape points

Points that are specified when an LRS segment is constructed, and that are assigned measure information.

sort

The operation of arranging a set of items according to a key that determines the sequence and precedence of items.

spatial

A generic term used to reference the mathematical concept of *n*-dimensional data.

spatial data

Data that is referenced by its location in *n*-dimensional space. The position of spatial data is described by multiple values. *See also* hyperspatial data.

spatial database

A database containing information indexed by location.

spatial data model

A model of how objects are located on a spatial context.

Spatial data dictionary

An extension of the Oracle9*i* data dictionary. It keeps track of the number of partitions created in a spatial table. The Spatial data dictionary is owned by user MDSYS. The data dictionary is used only by the deprecated partitioned point routines.

spatial data structures

A class of data structures designed to store spatial information and facilitate its manipulation.

spatial join

A query in which each of the geometries in one layer is compared with each of the geometries in the other layer. Comparable to a spatial cross product.

spatial query

A query that includes criteria for which selected features must meet location conditions.

spatial reference system

See coordinate system.

spatiotemporal data

Data that contains time and / or location components as one of its dimensions, also referred to as geographically referenced data or georeferenced data.

SQL*Loader

A utility to load formatted data into spatial tables.

tessellation

The process of covering a geometry with rectangular tiles without gaps or overlaps.

tiling

See tessellation.

touch

A geometric relationship where two objects share a common point on their boundaries, but their interiors do not intersect.

transformation

The conversion of coordinates from one coordinate system to another coordinate system. If the coordinate system is georeferenced, transformation can involve datum transformation: the conversion of geodetic coordinates from one geodetic datum to another geodetic datum, usually involving changes in the shape, orientation, and center position of the reference ellipsoid.

two-tier query model

The query model used by Spatial to resolve spatial queries and spatial joins. Two distinct filtering operations (primary and secondary) are performed to resolve queries. The output of both operations yields the exact result set. *See also* primary filter *and* secondary filter.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z