Oracle® Spatial

User's Guide and Reference

Release 8.1.7

September 2000

Part Number A85337-01

ORACLE®

Oracle Spatial User's Guide and Reference

Part Number A85337-01

Release 8.1.7

Copyright © 1997, 2000, Oracle Corporation. All rights reserved.

Primary Author: Chuck Murray

Contributors: Dan Abugov, Bruce Blackwell, Dan Geringer, Ravi Kothuri, L.J. Qian, Siva Ravada, Jayant Sharma, Frank Wang, Jack Wang, and Ran Wei

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle and SQL*Loader are registered trademarks, and Oracle7 and Oracle8*i* are trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments xix

Preface xxi

Audience xxi
Organization xxi
Changes for Release 8.1.7 xxiii
Features Released Separately xxiv
Related Documents xxiv
Conventions xxv

1 Spatial Concepts

1.1 What Is Oracle Spatial? 1-1
1.2 Object-Relational and Relational Models 1-1
1.2.1 Benefits of the Object-Relational Model 1-2
1.2.2 Benefits of the Relational Model 1-3
1.3 Introduction to Spatial Data 1-3
1.4 Geometric Types for Relational and Object-Relational Models 1-4
1.5 Data Model 1-6
1.5.1 Element 1-6
1.5.2 Geometry 1-7
1.5.3 Layer 1-7
1.5.4 Tolerance 1-7
1.6 Query Model 1-8
1.7 Indexing of Spatial Data 1-9

1.7.1 R-tree Indexing 1-10
1.7.1.1 Before Creating an R-tree Index 1-11
1.7.2 Quadtree Indexing 1-12
1.7.2.1 Tessellation of a Layer During Indexing 1-13
1.7.2.2 Fixed Indexing 1-13
1.7.2.3 Hybrid Indexing 1-18
1.8 Spatial Relations and Filtering 1-22
1.9 Partitioned Point Data 1-25
1.10 Examples 1-25

Part I Object-Relational Model

2 The Object-Relational Schema

2.1 Simple Example: Inserting, Indexing, and Querying Spatial Data 2-1
2.2 SDO_GEOMETRY Object Type 2-6
2.2.1 SDO_GTYPE 2-6
2.2.2 SDO_SRID 2-8
2.2.3 SDO_POINT 2-8
2.2.4 SDO_ELEM_INFO 2-8
2.2.5 SDO_ORDINATES 2-12
2.2.6 Usage Considerations 2-13
2.3 Geometry Examples Using the Object-Relational Model 2-13
2.3.1 Rectangle 2-13
2.3.2 Polygon with a Hole 2-14
2.3.3 Compound Element 2-16
2.3.4 Compound Polygon 2-17
2.4 Geometry Metadata Structure 2-18
2.4.1 TABLE_NAME 2-19
2.4.2 COLUMN_NAME 2-19
2.4.3 DIMINFO 2-19
2.4.4 SRID 2-20
2.5 Spatial Index-Related Structure 2-20
2.5.1 Spatial Index Views 2-20
2.5.2 Spatial Index Table Definition 2-22
2.5.3 R-Tree Index Sequence Object 2-23

3 Loading and Indexing Spatial Object Types

3.1 Load Process 3-1
3.1.1 Bulk Loading 3-1
3.1.1.1 Bulk Loading the SDO_GEOMETRY Object 3-2
3.1.1.2 Bulk Loading Point-Only Data in the SDO_GEOMETRY Object 3-3
3.1.2 Transactional Insert Using SQL 3-3
3.1.2.1 Polygon with Hole 3-4
3.1.2.2 Compound Line String 3-6
3.1.2.3 Compound Polygon 3-7
3.1.2.4 Compound Polygon with Holes 3-9
3.1.2.5 Transactional Insert of Point-Only Data 3-10
3.2 Index Creation 3-11
3.2.1 Determining Index Creation Behavior (Quadtree Indexes) 3-11
3.2.2 Spatial Indexing with Fixed-Size Tiles (Quadtree Indexes) 3-12
3.2.3 Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles 3-15
3.2.4 R-tree Index Parameter Considerations 3-16
3.2.4.1 SDO_FANOUT 3-16
3.2.4.2 SDO_RTR_PCTFREE 3-16
3.2.5 Cross-Schema Index Creation 3-16

4 Querying Spatial Data

4.1 Query Model 4-1
4.2 Spatial Query 4-1
4.2.1 Primary Filter 4-4
4.2.2 Primary and Secondary Filters 4-5
4.2.3 Within-Distance Operator 4-7
4.2.4 Nearest Neighbor Operator 4-8
4.3 Spatial Join 4-9
4.4 Cross-Schema Operator Invocation 4-9

5 Indexing Statements

ALTER INDEX 5-2
ALTER INDEX REBUILD 5-5
ALTER INDEX RENAME TO 5-8

CREATE INDEX 5-9
DROP INDEX 5-14

6 Spatial Operators

SDO_FILTER 6-2
SDO_NN 6-6
SDO_RELATE 6-8
SDO_WITHIN_DISTANCE 6-13

7 Geometry Functions

SDO_GEOM.RELATE 7-4
SDO_GEOM.SDO_AREA 7-7
SDO_GEOM.SDO_BUFFER 7-9
SDO_GEOM.SDO_CENTROID 7-11
SDO_GEOM.SDO_CONVEXHULL 7-13
SDO_GEOM.SDO_DIFFERENCE 7-15
SDO_GEOM.SDO_DISTANCE 7-18
SDO_GEOM.SDO_INTERSECTION 7-20
SDO_GEOM.SDO_LENGTH 7-23
SDO_GEOM.SDO_POINTONSURFACE 7-25
SDO_GEOM.SDO_UNION 7-27
SDO_GEOM.SDO_XOR 7-30
SDO_GEOM.VALIDATE_GEOMETRY 7-33
SDO_GEOM.VALIDATE_LAYER 7-36
SDO_GEOM.WITHIN_DISTANCE 7-39

8 Coordinate System Functions

SDO_CS.TRANSFORM 8-2
SDO_CS.TRANSFORM_LAYER 8-5

9 Linear Referencing Functions

SDO_LRS.CLIP_GEOM_SEGMENT 9-5
SDO_LRS.CONCATENATE_GEOM_SEGMENTS 9-7
SDO_LRS.CONNECTED_GEOM_SEGMENTS 9-10
SDO_LRS.CONVERT_TO_LRS_DIM_ARRAY 9-12
SDO_LRS.CONVERT_TO_LRS_GEOM 9-14
SDO_LRS.CONVERT_TO_LRS_LAYER 9-16
SDO_LRS.CONVERT_TO_STD_DIM_ARRAY 9-18
SDO_LRS.CONVERT_TO_STD_GEOM 9-20
SDO_LRS.CONVERT_TO_STD_LAYER 9-22
SDO_LRS.DEFINE_GEOM_SEGMENT 9-24
SDO_LRS.DYNAMIC_SEGMENT 9-27
SDO_LRS.FIND_MEASURE 9-29
SDO_LRS.GEOM_SEGMENT_END_MEASURE 9-31
SDO_LRS.GEOM_SEGMENT_END_PT 9-33
SDO_LRS.GEOM_SEGMENT_LENGTH 9-35
SDO_LRS.GEOM_SEGMENT_START_MEASURE 9-37
SDO_LRS.GEOM_SEGMENT_START_PT 9-39
SDO_LRS.GET_MEASURE 9-41
SDO_LRS.IS_GEOM_SEGMENT_DEFINED 9-43
SDO_LRS.LOCATE_PT 9-45
SDO_LRS.MEASURE_RANGE 9-47
SDO_LRS.MEASURE_TO_PERCENTAGE 9-49
SDO_LRS.PERCENTAGE_TO_MEASURE 9-51
SDO_LRS.PROJECT_PT 9-53
SDO_LRS.REDEFINE_GEOM_SEGMENT 9-55
SDO_LRS.REVERSE_MEASURE 9-57
SDO_LRS.SCALE_GEOM_SEGMENT 9-59
SDO_LRS.SPLIT_GEOM_SEGMENT 9-62
SDO_LRS.TRANSLATE_MEASURE 9-65
SDO_LRS.VALID_GEOM_SEGMENT 9-67

SDO_LRS.VALID_LRS_PT 9-69
SDO_LRS.VALID_MEASURE 9-71

10 Migration Procedures

SDO_MIGRATE.FROM_815_TO_81X 10-2
SDO_MIGRATE.TO_734 10-3
SDO_MIGRATE.TO_81X 10-5
SDO_MIGRATE.OGIS_METADATA_FROM 10-8
SDO_MIGRATE.OGIS_METADATA_TO 10-9

11 Tuning Functions and Procedures

SDO_TUNE.AVERAGE_MBR 11-2
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE 11-4
SDO_TUNE.ESTIMATE_TILING_LEVEL 11-7
SDO_TUNE.ESTIMATE_TILING_TIME 11-9
SDO_TUNE.ESTIMATE_TOTAL_NUMTILES 11-11
SDO_TUNE.EXTENT_OF 11-14
SDO_TUNE.HISTOGRAM_ANALYSIS 11-16
SDO_TUNE.MIX_INFO 11-18

Part II Relational Model

12 The Relational Schema

12.1 Database Structures for the Relational Implementation 12-1

13 Loading Spatial Data (Relational Model)

13.1 Load Model 13-1
13.2 Load Process 13-2
13.2.1 Bulk Loading 13-2
13.2.2 Transactional Insert Using SQL 13-4
13.3 Index Creation 13-6
13.3.1 Choosing a Tessellation Algorithm 13-6

13.3.2 Spatial Indexing with Fixed-Size Tiles ... 13-7
13.3.3 Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles ... 13-10

14 Querying Spatial Data (Relational Model)

14.1 Query Model ... 14-1
14.2 Spatial Index Data Structures ... 14-1
14.3 Spatial Query ... 14-4
14.3.1 Dynamic Query Window ... 14-5
14.3.2 Primary Filter Query ... 14-6
14.3.3 Secondary Filter Query ... 14-7
14.4 Spatial Join ... 14-8

15 Administrative Functions and Procedures for Relational Model

SDO_ADMIN.POPULATE_INDEX ... 15-3
SDO_ADMIN.POPULATE_INDEX_FIXED ... 15-5
SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS ... 15-8
SDO_ADMIN.SDO_CODE_SIZE ... 15-10
SDO_ADMIN.SDO_VERSION ... 15-11
SDO_ADMIN.UPDATE_INDEX ... 15-12
SDO_ADMIN.UPDATE_INDEX_FIXED ... 15-14
SDO_ADMIN.VERIFY_LAYER ... 15-16

16 Tuning Functions and Procedures for Relational Model

SDO_TUNE.AVERAGE_MBR ... 16-2
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE ... 16-3
SDO_TUNE.ESTIMATE_TILING_LEVEL ... 16-5
SDO_TUNE.ESTIMATE_TILING_TIME ... 16-8
SDO_TUNE.EXTENT_OF ... 16-9
SDO_TUNE.HISTOGRAM_ANALYSIS ... 16-10
SDO_TUNE.MIX_INFO ... 16-12

17 Geometry Functions and Procedures for Relational Model

SDO_GEOM.RELATE 17-2
SDO_GEOM.VALIDATE_GEOMETRY 17-5
SDO_GEOM.VALIDATE_LAYER 17-6

18 Window Functions and Procedures for Relational Model

SDO_WINDOW.BUILD_WINDOW 18-3
SDO_WINDOW.BUILD_WINDOW_FIXED 18-5
SDO_WINDOW.CLEAN_WINDOW 18-7
SDO_WINDOW.CLEANUP_GID 18-8
SDO_WINDOW.CREATE_WINDOW_LAYER 18-9

A Tuning Tips and Sample SQL Scripts

A.1 Tuning Tips A-1
A.1.1 Data Modeling A-1
A.1.2 Understanding the Tiling Level A-2
A.1.3 Using Hybrid Indexes (Object-Relational Model Only) A-3
A.1.4 Database Sizing A-4
A.1.5 Visualizing the Spatial Index (Drawing Tiles) A-5
A.1.5.1 Drawing Tiles from the Object-Relational Model A-5
A.1.5.2 Drawing Tiles from the Relational Model A-7
A.1.6 Setting the SORT_AREA_SIZE Value A-9
A.1.7 Tuning Point Data with the Relational Model A-9
A.1.7.1 Efficient Queries for Relational Point Data A-9
A.1.7.2 Efficient Schema for Relational Point Layers A-10
A.1.7.3 Script for Using Table Partitioning with Relational Point Data A-11
A.1.8 Tuning Spatial Join Queries Using the Relational Model A-11
A.1.8.1 Using the NO_MERGE, INDEX, and USE_NL Hints A-11
A.1.8.2 Spatial Join Queries with Point Layers A-12
A.1.9 Using Customized Geometry Types in the Relational Model A-14
A.1.10 Partitioning Spatial Data Using the Relational Model A-14
A.1.11 Parallel Loading and Indexing of Spatial Data Using the Relational Model A-14
A.2 Scripts for Spatial Indexing Using the Relational Model A-16

A.2.1 cr_spatial_index.sql Script A-16
A.2.2 crlayer.sql Script A-17
A.3 Tools and Related Products A-17
A.3.1 Oracle *inter*Media Locator A-17
A.3.1.1 Geocoding Support A-17
A.3.1.2 Compatibility with Spatial Objects A-18
A.3.1.3 Sample *inter*Media Locator Code A-18
A.3.2 Spatial Viewer on UNIX/Motif for Relational Model A-18
A.3.2.1 Installation and Setup A-18
A.3.2.2 Connecting to a Database and Viewing Geometries A-19
A.3.2.3 Using the Sample Viewer A-20
A.3.3 Spatial Visualizer on Windows NT for the Object-Relational Model A-20
A.3.3.1 Compiling and Running the Sample Program A-20
A.3.3.2 Usage Notes A-21

B Installation, Compatibility, and Migration Issues

B.1 Introduction B-1
B.2 Installation Details B-2
B.2.1 Changing from 8.1 to 8.0 Compatibility Mode B-2
B.3 Compatibility Details B-3
B.4 Data Migration Issues B-4
B.5 Migrating from Spatial Release 8.1.5 or 8.1.6 to Release 8.1.7 ..... B-5
B.5.1 Migrating from Spatial Release 8.1.5 to Release 8.1.6 ..... B-5
B.5.1.1 Data Migration to Release 8.1.6 ..... B-6
B.5.1.2 Compatibility Between Releases 8.1.5 and 8.1.6 ..... B-6
B.5.2 Migrating from Spatial Release 8.1.6 to Release 8.1.7 ..... B-7

C Generic Geocoding Interface

C.1 Locator Implementation: Benefits and Limitations C-2
C.2 Generic Geocoding Client C-2
C.3 Geocoder Metadata C-3
C.3.1 Server Properties C-4
C.3.2 Geocoding Input and Output Specification C-5
C.3.2.1 Multiple Matches and Rejected Records C-9
C.4 Metadata Helper Class C-9

C.5 Single-Record and Interactive Geocoding C-9
C.6 Java Geocoder Service Interface C-10
C.7 Enabling Third-Party Geocoders..... C-11

D Coordinate Systems (Spatial Reference Systems)

D.1 Why Integrate Coordinate System Information?..... D-1
D.2 Terms and Concepts..... D-2
D.2.1 Coordinate System (Spatial Reference System) D-2
D.2.2 Cartesian Coordinates D-2
D.2.3 Geodetic Coordinates (Geographic Coordinates)..... D-2
D.2.4 Projected Coordinates..... D-2
D.2.5 Geodetic Datum..... D-2
D.2.6 Authalic Sphere D-3
D.2.7 Transformation (Datum Transformation)..... D-3
D.3 Coordinate Systems Data Structures D-3
D.3.1 MDSYS.CS_SRS Table..... D-3
D.3.1.1 Well-Known Text (WKTEXT)..... D-4
D.3.2 Other Objects..... D-5
D.4 Coordinate Systems Functions and Procedures..... D-7
D.5 Restrictions and Problems in the Current Release..... D-7
D.5.1 Geometries with Longitude and Latitude Coordinates..... D-7
D.6 Example of Coordinate Systems..... D-8
D.7 Error Messages for Coordinate Systems D-14

E Linear Referencing System

E.1 Terms and Concepts..... E-1
E.1.1 Geometric Segments (LRS Segments)..... E-2
E.1.2 Shape Points E-2
E.1.3 Direction of a Geometric Segment E-3
E.1.4 Measure (Linear Measure) E-3
E.1.5 Offset E-3
E.1.6 Measure Populating..... E-4
E.1.7 Measure Range of a Geometric Segment E-6
E.1.8 Projection E-6
E.1.9 LRS Point E-6

E.1.10 Linear Features E-6
E.2 LRS Data Model E-7
E.3 Indexing of LRS Data E-8
E.4 LRS Operations E-9
E.4.1 Defining a Geometric Segment E-9
E.4.2 Redefining a Geometric Segment E-10
E.4.3 Clipping a Geometric Segment E-11
E.4.4 Splitting a Geometric Segment E-12
E.4.5 Concatenating Two Connected Geometric Segments E-12
E.4.6 Scaling a Geometric Segment E-14
E.4.7 Locating a Point on a Geometric Segment E-15
E.4.8 Projecting a Point onto a Geometric Segment E-17
E.4.9 Converting Geometric Segments E-17
E.5 Example E-19
E.6 Error Messages for Linear Referencing System E-31

Glossary

List of Examples

2–1 Simple Example: Inserting, Indexing, and Querying Spatial Data 2-3
3–1 Control File for a Bulk Load.. 3-2
3–2 Control File for a Bulk Load of Point-Only Data 3-3
3–3 Procedure to Perform Transactional Insert Operation.. 3-4
3–4 PL/SQL Block Invoking Procedure to Insert a Geometry.. 3-4
3–5 Creating a Fixed Index.. 3-14
3–6 Creating a Hybrid Index... 3-16
4–1 Primary Filter with a Temporary Query Window 4-4
4–2 Primary Filter with a Transient Instance of the Query Window................................... 4-5
4–3 Primary Filter with a Stored Query Window.. 4-5
4–4 Secondary Filter Using a Temporary Query Window 4-6
4–5 Secondary Filter Using a Stored Query Window 4-6
13–1 Raw Data Format 13-2
13–2 Control File to Load Data into the Geometry Table 13-3
13–3 Raw Data Format... 13-3
13–4 Control File to Load from a Single Flat File.. 13-4
13–5 Transactional Insert... 13-4
13–6 Transactional Insert for a Large Geometry.. 13-5
15–1 Populate an Index.. 15-4
15–2 Populate an Index with Fixed-Size Tiles 15-7
15–3 Populate an Index with Fixed-Size Tiles Based on Point Data 15-9
15–4 Update an Index 15-13
15–5 Update an Index with Fixed-Size Tiles.. 15-15
15–6 Verify a Layer... 15-16
16–1 Recommended Tile Level for One-Degree Latitude/Longitude Cells......................... 16-6
16–2 Recommended Tile Level Based on the GIDs of All Geometries 16-6
16–3 Recommended Tile Level Based on Average Extent of All Geometries 16-7
A–1 View Fixed-Size Tiles for All Geometries A-6
A–2 View Variable-Sized Tiles for All Geometries... A-6
A–3 View Fixed-Size Tiles for One Geometry... A-6
A–4 View Variable-Sized Tiles for One Geometry A-7
A–5 View Fixed-Sized Tiles for All Geometries Using the Relational Model A-8
A–6 View Fixed-Size Tiles for a Specific Geometry Using the Relational Model A-8
B–1 Insert Trigger for Metadata Consistency B-7
D–1 Simplified Example of Coordinate Systems D-8
D–2 Output of SELECT Statements in Coordinate Systems Example D-11
E–1 Including LRS Measure Dimension in Spatial Metadata... E-7
E–2 Simplified Example: Highway... E-21
E–3 Simplified Example: Output of SELECT Statements... E-26

List of Figures

1–1 Geometric Primitive Types 1-5
1–2 Additional Geometry Types Using the Object-Relational Model 1-6
1–3 Query Model 1-8
1–4 Quadtree Decomposition and Morton Codes 1-13
1–5 Fixed-Size Tiling with Many Small Tiles 1-15
1–6 Fixed-Size Tiling with Fewer Large Tiles 1-16
1–7 Tessellated Geometry 1-17
1–8 Variable-Sized Tile Spatial Indexing 1-20
1–9 Decomposition of the Geometry 1-21
1–10 The 9-Intersection Model........ 1-23
1–11 Distance Buffers for Points, Lines, and Polygons 1-25
2–1 Areas of Interest for Simple Example........ 2-2
2–2 Rectangle........ 2-13
2–3 Geometry with a Hole 2-14
2–4 Compound Element........ 2-16
2–5 Compound Polygon........ 2-17
3–1 Polygon with Hole 3-5
3–2 Line String Consisting of Arcs and Straight Line Segments 3-6
3–3 Compound Polygon........ 3-8
3–4 Compound Polygon with a Hole 3-9
3–5 Sample Domain 3-12
3–6 Fixed-Size Tiling at Level 1 3-13
3–7 Fixed-Size Tiling at Level 2 3-13
4–1 Tessellated Layer with Multiple Objects........ 4-2
4–2 Tessellated Layer with a Query Window 4-3
7–1 SDO_GEOM.SDO_DIFFERENCE........ 7-16
7–2 SDO_GEOM.SDO_INTERSECTION........ 7-21
7–3 SDO_GEOM.SDO_UNION........ 7-28
7–4 SDO_GEOM.SDO_XOR 7-31
9–1 Reversing a Geometric Segment 9-58
9–2 Translating a Geometric Segment........ 9-66
12–1 Complex Polygon........ 12-5
13–1 Sample GIS Domain 13-8
13–2 Fixed-Size Tiling at Level 1 13-8
13–3 Fixed-Size Tiling at Level 2 13-9
14–1 Tessellated Layer with Multiple Objects........ 14-2
14–2 Tessellated Layer with a Query Window 14-5
14–3 Spatial Join of Two Layers 14-8
C–1 Oracle Geocoding Framework C-3
E–1 Geometric Segment........ E-2

E–2 Describing a Point Along a Segment with a Measure and an Offset E-4
E–3 Measures, Distances, and Their Mapping Relationship E-4
E–4 Measure Populating of a Geometric Segment E-5
E–5 Measure Populating With Disproportional Assigned Measures E-5
E–6 Linear Feature, Geometric Segments, and LRS Points E-7
E–7 Creating a Geometric Segment E-8
E–8 Defining a Geometric Segment E-10
E–9 Redefining a Geometric Segment E-11
E–10 Clipping, Splitting, and Concatenating Geometric Segments E-12
E–11 Measure Assignment in Geometric Segment Operations.. E-13
E–12 Segment Direction with Concatenation E-14
E–13 Scaling a Geometric Segment... E-15
E–14 Locating a Point Along a Segment with a Measure and an Offset.............................. E-16
E–15 Ambiguity in Location Referencing with Offsets E-16
E–16 Multiple Projection Points E-17
E–17 Conversion from Standard to LRS Line String.. E-19
E–18 Simplified LRS Example: Highway E-20

List of Tables

1–1 SDOINDEX Table Using Fixed-Size Tiles 1-18
1–2 Section of the SDOINDEX Table 1-22
2–1 Valid SDO_GTYPE Values 2-7
2–2 Values and Semantics in SDO_ELEM_INFO 2-10
2–3 Columns in the xxx_SDO_INDEX_METADATA Views 2-21
2–4 Columns in a Spatial Index Data Table 2-22
5–1 Spatial Index Creation and Usage Statements 5-1
5–2 SDO_LEVEL and SDO_NUMTILES Combinations 5-12
6–1 Spatial Usage Operators 6-1
7–1 Geometric Functions for the Object-Relational Model 7-1
8–1 Functions and Procedures for Coordinate Systems 8-1
8–2 Table to Hold Transformed Layer 8-6
9–1 Functions for Creating and Editing Geometric Segments 9-1
9–2 Functions for Querying Geometric Segments 9-2
9–3 Functions for Converting Geometric Segments 9-3
10–1 Migration Procedures 10-1
11–1 Tuning Functions and Procedures 11-1
12–1 <layername>_SDOLAYER Table 12-2
12–2 <layername>_SDODIM Table or View 12-2
12–3 <layername>_SDOGEOM Table or View 12-2
12–4 <layername>_SDOINDEX Table 12-2
13–1 <layername>_SDOLAYER Table 13-1
13–2 <layername>_SDODIM Table or View 13-1
13–3 <layername>_SDOGEOM Table or View 13-2
13–4 <layername>_SDOINDEX Table 13-2
13–5 Choosing a Tessellation Algorithm 13-7
14–1 <layername>_SDOLAYER Table 14-3
14–2 <layername>_SDOGEOM Table or View 14-3
14–3 <layername>_SDOINDEX Table 14-4
15–1 Administrative Procedures for Spatially Indexed Data 15-1
16–1 Tuning Functions and Procedures 16-1
17–1 Geometric Functions and Procedures 17-1
18–1 Window Functions and Procedures 18-1
D–1 MDSYS.CS_SRS Table D-3
D–2 Supported Map Projections D-5
D–3 Supported Ellipsoids D-6
E–1 Highway Features and LRS Counterparts E-20

Send Us Your Comments

Oracle Spatial User's Guide and Reference, Release 8.1.7

Part Number A85337-01

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: 603.897.3316   Attn: Spatial Documentation
- Postal service:
  Oracle Corporation
  Oracle Spatial Documentation
  One Oracle Drive
  Nashua, NH 03062-2698
  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 user and reference information for the Spatial product, and extensions to Oracle8*i* Enterprise Edition.

Spatial requires Oracle8*i* Enterprise Edition. Oracle8*i* and Oracle8*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 Oracle8*i* table partitioning, you must have the Enterprise Edition and the Partitioning Option.

For information about the differences between Oracle8*i* and Oracle8*i* Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i*.

Audience

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

Organization

This guide is divided into two parts. Part I deals with the object-relational storage model, and Part II describes the relational storage model. The following table lists the elements in this guide:

Chapter 1	Introduces spatial data concepts.
Part I	The following chapters describe the object-relational spatial model:
Chapter 2	Explains the object-relational schema.

Chapter 3	Explains loading and indexing spatial data.
Chapter 4	Explains methods for querying a spatial database.
Chapter 5	Provides the syntax and semantics for the indexing functions.
Chapter 6	Provides the syntax and semantics for operators used with the spatial object data type.
Chapter 7	Provides the syntax and semantics for the geometric functions and procedures.
Chapter 8	Provides the syntax and semantics for the linear referencing system (LRS) functions.
Chapter 9	Provides the syntax and semantics for the migration functions.
Chapter 10	Provides the syntax and semantics for the migration functions.
Chapter 11	Provides the syntax and semantics for the tuning functions and procedures.
Part II	The following chapters describe the relational spatial model:
Chapter 12	Explains the relational schema.
Chapter 13	Explains spatial data loading.
Chapter 14	Explains methods for querying a spatial database.
Chapter 15	Provides the syntax and semantics for the administrative functions and procedures.
Chapter 16	Provides the syntax and semantics for the tuning functions and procedures.
Chapter 17	Provides the syntax and semantics for the geometric functions and procedures.
Chapter 18	Provides the syntax and semantics for the window functions and procedures.
Appendix A	Describes sample SQL scripts and tuning tips.
Appendix B	Describes installation, compatibility, and migration issues.
Appendix C	Describes the Spatial Generic Geocoding Interface.
Appendix D	Provides conceptual and usage information for using coordinate systems (spatial reference systems) with Oracle Spatial.

Appendix E	Provides conceptual and usage information for using the Oracle Spatial linear referencing system (LRS).
Glossary	Provides definitions of terms used in this guide.

Changes for Release 8.1.7

The following changes have been made to this guide for release 8.1.7:

- Information about support for coordinate systems (spatial reference systems) and the linear referencing system (LRS) has been added. This information previously appeared in separate documents available through the Oracle Technology Network (OTN).
- Information about spatial R-tree indexes has been added. This information previously appeared in a separate document available through the Oracle Technology Network.
- The behavior of the SDO_GEOM.VALIDATE_LAYER procedure (documented in Chapter 7) has changed. Rows are written to the result table only for invalid geometries, and no row is written if a geometry is valid. In the previous release, a row was written to the result table for each geometry, and for valid geometries the row contained the primary key value and the string TRUE. This change is designed to minimize the size of the result table when the layer contains many geometries.
- The SDO_POLY_xxx functions, which were deprecated at release 8.1.6, have been removed from this guide. You should use instead the corresponding "generic" (not restricted to polygons) SDO_xxx functions documented in Chapter 7: SDO_DIFFERENCE, SDO_INTERSECTION, SDO_UNION, and SDO_XOR.
- Minor corrections and clarifications have been made where necessary.

Note: The **relational model** (documented in Part II) **will not be included** in future releases of this guide, but will instead be provided in a separate document to be announced. You are encouraged to use only the object-relational model (documented in Part I) for Oracle Spatial applications.

Features Released Separately

The following features or capabilities are of interest to spatial application developers, but are not part of Oracle Spatial. Software and documentation for the following are available through the Oracle Technology Network.

- GeoImage
- Workspace Management (long transactions)

To access the Oracle Technology Network, go to

```
http://technet.oracle.com
```

Related Documents

For more information, see the following documents:

- *Oracle interMedia Locator User's Guide and Reference*
- *Getting to Know Oracle8i*
- *Oracle8i Administrator's Guide*
- *Oracle8i Application Developer's Guide - Fundamentals*
- *Oracle8i Error Messages* - Spatial messages are in the range of 13000 to 13499; however, for release 8.1.7 coordinate systems messages are in Section D.7 and linear referencing system (LRS) messages are in Section E.6 of this guide.
- *Oracle8i Concepts*
- *Oracle8i Performance Guide and Reference*
- *Oracle8i Utilities*

For additional information about Oracle Spatial, including white papers and other collateral, visit the official Spatial Web site at

```
http://www.oracle.com/database/options/spatial/
```

If that Web address has changed since the publication of this guide, visit the Oracle home page at

```
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.

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 Oracle8*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 Oracle8*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 and spatial join queries
- Administrative utilities

The spatial attribute 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 and Relational Models

Spatial supports two mechanisms, or models, for representing geometries:

- The **object-relational** model uses a table with a single column of type MDSYS.SDO_GEOMETRY and a single row per geometry instance.

- The **relational** model uses a table with a predefined set of columns of type NUMBER and one or more rows for each geometry instance.

These models roughly correspond to two alternatives described in the OpenGIS ODBC/SQL specification for geospatial features. The object-relational model corresponds to a "SQL with Geometry Types" implementation of spatial feature tables, and the relational model corresponds to an implementation of spatial feature tables using numeric SQL types for geometry storage. Implementation-specific details are described in Part I "Object-Relational Model" and Part II "Relational Model" of this guide.

You should select the object-relational model in all cases except where the relational model is necessary for current needs. Basically, the object-relational model is preferable in cases where replication and distributed databases are not required.

Note: *If read-only replication is acceptable:* Oracle8*i* does not currently provide database replication support for tables containing one or more columns of an object data type. In many application environments, however, it may be acceptable to have read-only replicas that do not need to be perfectly up to date with the production database. In these environments, users can take advantage of the Oracle8*i* managed standby capability to get, in effect, read-only replication of database instances or tablespaces, and thus the object-relational model can be used.

1.2.1 Benefits of the Object-Relational Model

The following are some of the benefits of using the object-relational model, as described in Part I of this guide:

- Additional geometry types are supported: arcs, circles, compound polygons, compound line strings, and optimized rectangles.
- Ease of use is improved for creating and maintaining indexes and for performing spatial queries.
- Indexing is maintained by the Oracle8*i* database server.
- Geometries are modeled in a single row, single column.
- Performance is greatly improved.

1.2.2 Benefits of the Relational Model

The following are some of the benefits of using the relational model, as described in Part II of this guide:

- Database replication is supported.
- Distributed databases are supported.
- Table partitioning and parallel index loading are supported.

When Oracle introduces replication and distributed support for objects in a future release, there will be no benefits to using the relational model.

Note: In the next release of Oracle Spatial, the relational model will no longer be documented in this guide, but will instead be documented in a separate document whose title and location will be announced.

Spatial applications using the relational model will continue to work; however, if you are not already using the object-relational model for all Spatial applications, you are encouraged to do so before the next release.

1.3 Introduction to Spatial Data

Oracle Spatial is designed to make spatial data management easier and more natural to users of applications such as a Geographic Information System (GIS). 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 three systems are only in the scale of the data, not its complexity. They 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. The spatial attribute, referred to as the 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, which may be one of point, line string, or polygon.

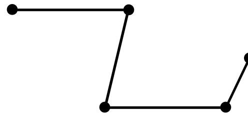
1.4 Geometric Types for Relational and Object-Relational Models

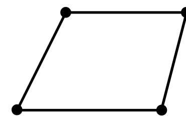
The relational model of Spatial supports three geometric primitive types and geometries composed of collections of these types. The primitive types are as follows:

- 2-D point and point cluster
- 2-D line strings
- 2-D n-point polygons

2-D 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. Figure 1–1 illustrates the supported geometric primitive types.

Figure 1–1 Geometric Primitive Types





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.

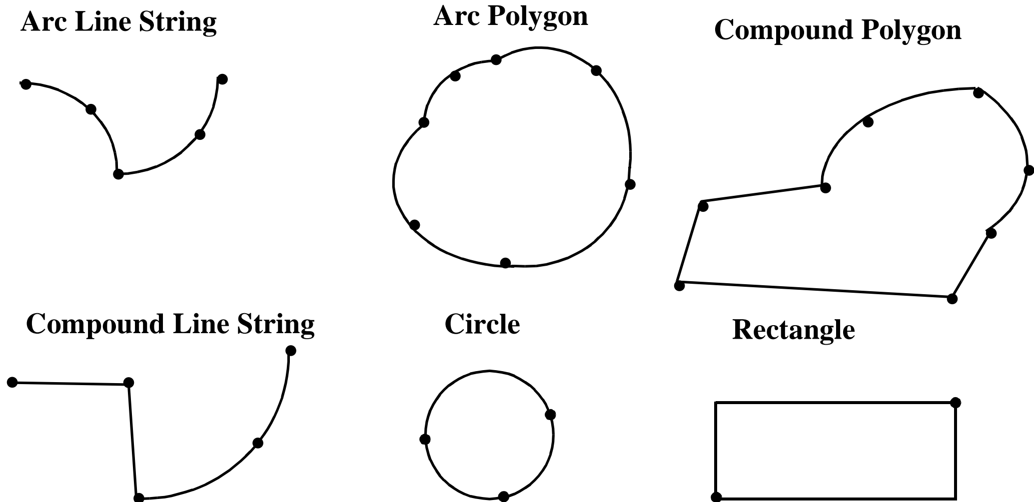
Thus, the object-relational implementation supports the types listed in Figure 1–1, as well as the types shown in Figure 1–2.

The object-relational model adds the following types to those previously listed:

- 2-D arc line strings (All arcs are generated as circular arcs.)
- 2-D arc polygons
- 2-D compound polygons
- 2-D compound line strings
- 2-D circles
- 2-D optimized rectangles

Thus, the object-relational implementation supports the types listed in Figure 1–1, as well as the types shown in Figure 1–2.

Figure 1–2 Additional Geometry Types Using the Object-Relational Model



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. In the relational model, each geometry is required to be uniquely identified by a geometry identifier (GID) associating it with the other attributes of the feature. This is not required in the object-relational model.

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.

In the relational model, a complex geometry such as a polygon with holes would be stored as a sequence of polygon elements. All subelements of a multielement polygon are wholly contained within the outermost element. This is not required using the object-relational model.

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 heterogeneous 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 Tolerance

Many Spatial functions accept a *tolerance* parameter. If the distance between two points is less than or equal to the tolerance, 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 you ask, *Find all restaurants within 5 kilometers and use a tolerance of 0.1* (or greater, such as 0.5), Maria's Pizzeria will be included; however, if you specify a tolerance less than 0.1 (such as 0.05), Maria's Pizzeria will not be included.

Tolerance values for Spatial functions are typically very small, for example, 0.0005 (5E-4). With a tolerance of 5E-4 and the query in the preceding paragraph, a restaurant 5.0005 kilometers away is returned but a restaurant 5.00051 kilometers away is not returned.

1.6 Query Model

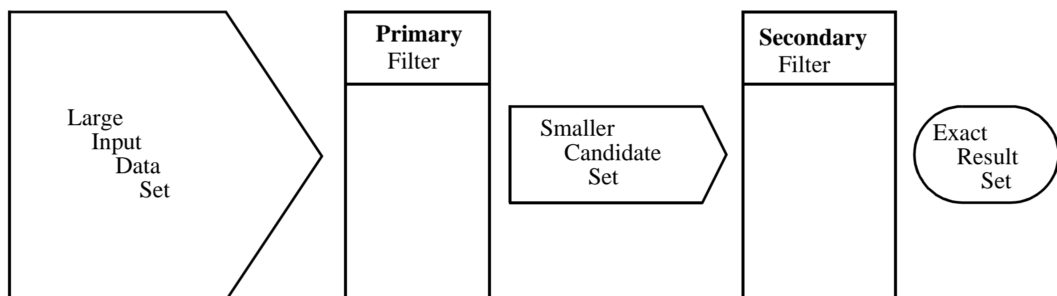
Spatial uses a *two-tier* query model to resolve spatial queries and spatial joins. The term is used to indicate that two distinct operations are performed to resolve queries. The output of both operations yields the exact result set.

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 1–3 illustrates the relationship between the primary and secondary filters.

Figure 1–3 Query Model



As shown in Figure 1–3, 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 linear quadtree-based spatial index to implement the primary filter. This is described in detail in following sections.

The function SDO_GEOM.RELATE is used as a secondary filter. It evaluates the topological relationship, such as whether two given geometries are touching, covering each other, or have any interaction.

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 overlaps 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 overlap 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 take on values from a linearly ordered integer domain, while 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 5), 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 6.

In choosing whether to use an R-tree or quadtree index for a spatial application, consider the following.

With R-tree indexes:

- The approximation of geometries cannot be fine-tuned. (Spatial uses the minimum bounding rectangles, as described in Section 1.7.1.)
- Index creation and tuning are easier than with quadtree indexes.
- Less storage is required than with quadtree indexes, except for point-only data, where there is no significant difference.
- If your application workload includes nearest-neighbor queries (SDO_NN operator), R-tree indexes are faster.
- If there is heavy update activity to the spatial column, an R-tree index may not be a good choice.

With quadtree indexes:

- The approximation of geometries can be fine-tuned by setting the tiling level and number of tiles.
- Tuning is more complex than with R-tree indexes, and setting the appropriate tuning parameter values can affect performance significantly.
- More storage is required than with R-tree indexes, except for point-only data, where there is no significant difference.
- Heavy update activity does not affect the performance of a quadtree 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). For a layer of

geometries, an R-tree index consists of a hierarchical index on the MBRs of the geometries in the layer. This 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.

If you create a spatial index without specifying any indexing 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;
```

If you create a spatial index without specifying the SDO_LEVEL or SDO_NUMTILES keyword in the PARAMETERS clause, an R-tree index is created. For detailed information about options when creating a spatial index, see the documentation for the CREATE INDEX statement in Chapter 5.

1.7.1.1 Before Creating an R-tree Index

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 the system rollback segment cannot be extended, place that rollback segment offline, create a public rollback segment of the appropriate size, and place that rollback segment online. For information about performing these operations on a rollback segment, see the Oracle8i 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;
```

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

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

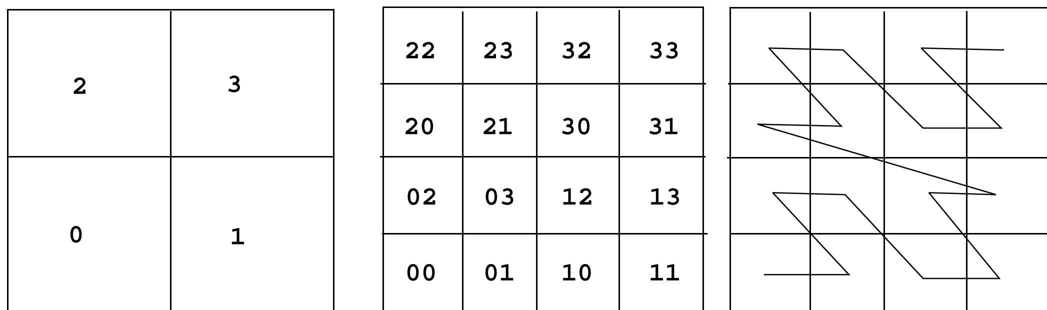
fixed-size tiles and the other set contains variable-sized tiles. Hybrid indexing is described in Section 1.7.2.3.

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 1–4. 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 1–4 Quadtree Decomposition and Morton Codes



1.7.2.2 Fixed Indexing

Fixed-size tile spatial indexing is the preferred indexing method for the relational model. This method 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.

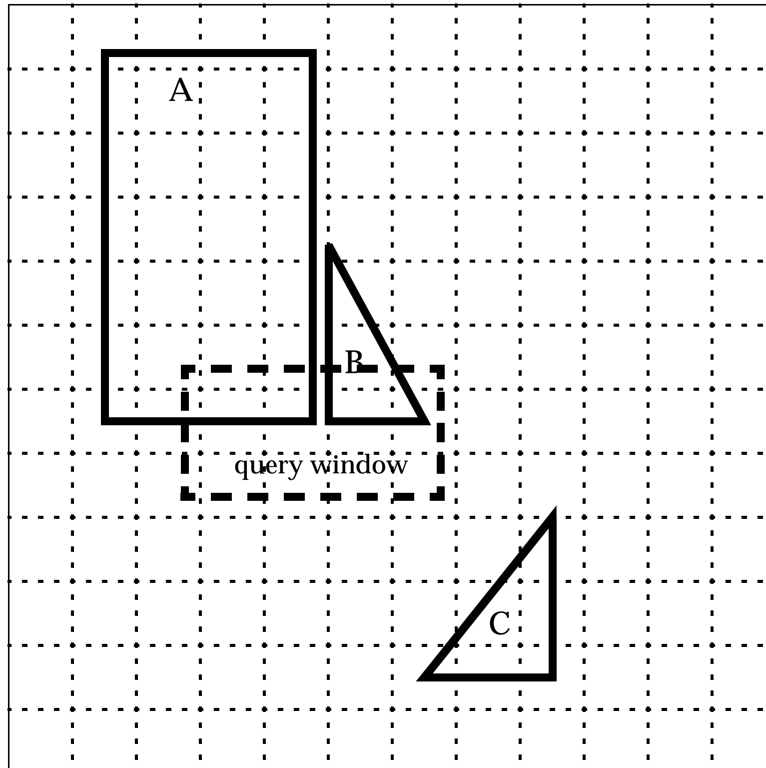
Two geometries are likely to interact, and hence pass the primary filter stage, if they share one or more tiles. The SQL statement for the primary filter stage is:

```
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 1–5 and Figure 1–6 illustrate the relationships between tile size, selectivity, and the number of cover tiles.

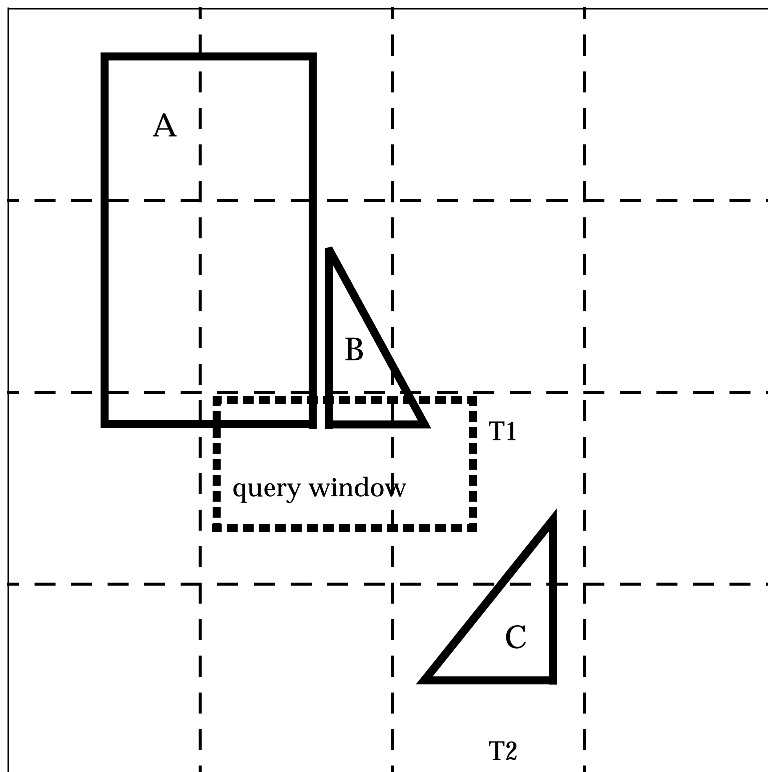
With a small fixed-size tile as shown in Figure 1–5, selectivity is good, but a large number of tiles is needed to cover large geometries. A window query would easily identify geometries A and B, but would reject C.

Figure 1–5 Fixed-Size Tiling with Many Small Tiles



With a large fixed-size tile as shown in Figure 1–6, fewer tiles are needed to cover the geometries, but the selectivity is not as good. The same window query as in Figure 1–5 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 1–6.

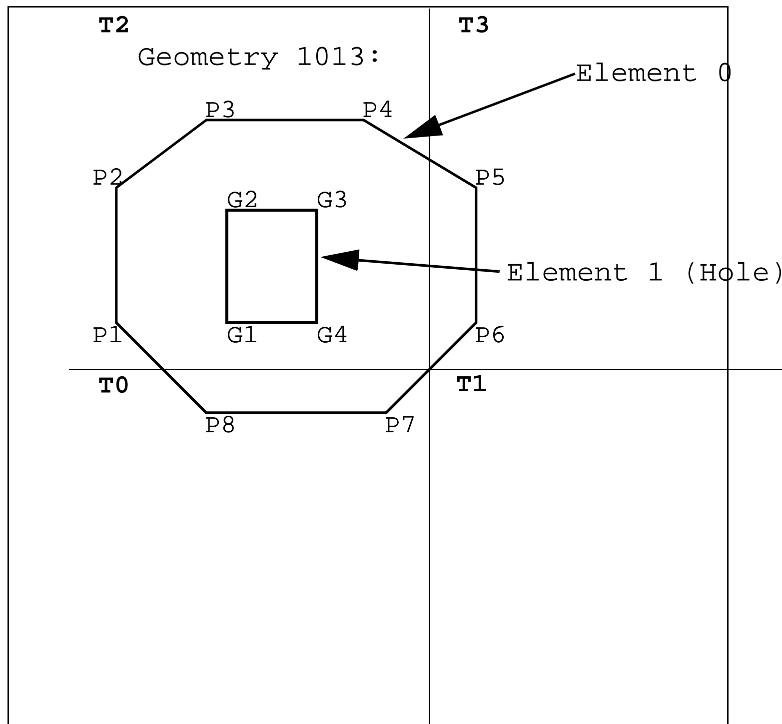
Figure 1–6 Fixed-Size Tiling with Fewer Large Tiles



The SDO_TUNE.ESTIMATE_TILING_LEVEL function helps determine an appropriate tiling level for your data set.

Figure 1–7 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 1–7 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 1–1. In this example, three fixed-size tiles are used. The table structure is shown for illustrative purposes only. The column names of this table differ depending on which implementation method, relational or object-relational, is in use. In the relational model, you must directly access the

index tables. In the object-relational model, this is both unnecessary and not recommended.

Table 1–1 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 like 1013, Element 1 is already covered by tile T2 from the tessellation of Element 0. If, however, the specified tiling resolution were such that tile T2 were further subdivided and one of these smaller tiles were completely contained in Element 1, then that tile would be excluded because it would not interact with the geometry.

1.7.2.3 Hybrid Indexing

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 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.

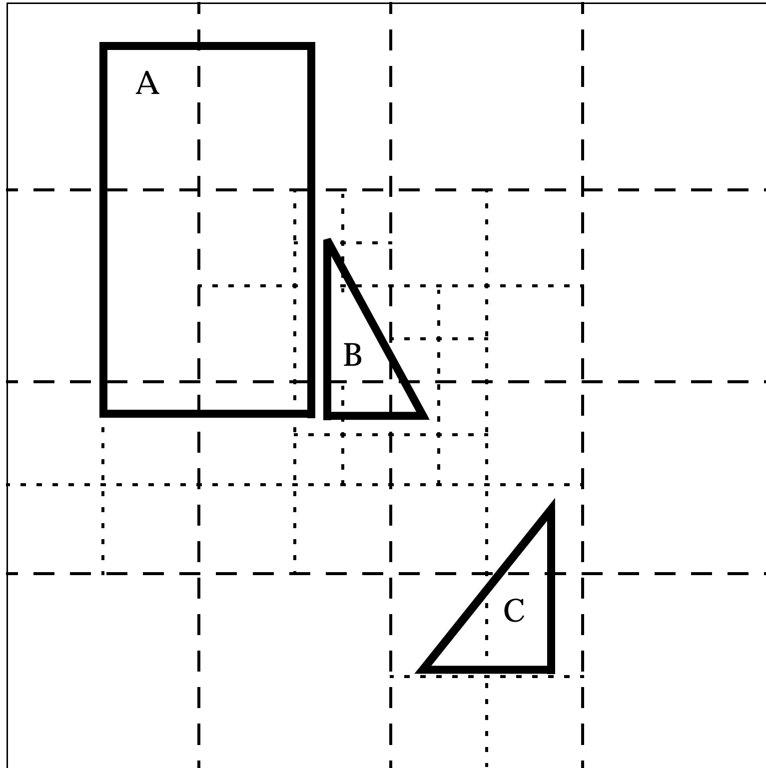
  The best starting value for SDO_NUMTILES in the new hybrid layer can be calculated by getting a count of the rows in the spatial index table and dividing this number by the number of rows with geometries in the layer, then rounding up. A spatial join ('QUERYTYPE=JOIN') is not a common requirement for 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 1–8, 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.

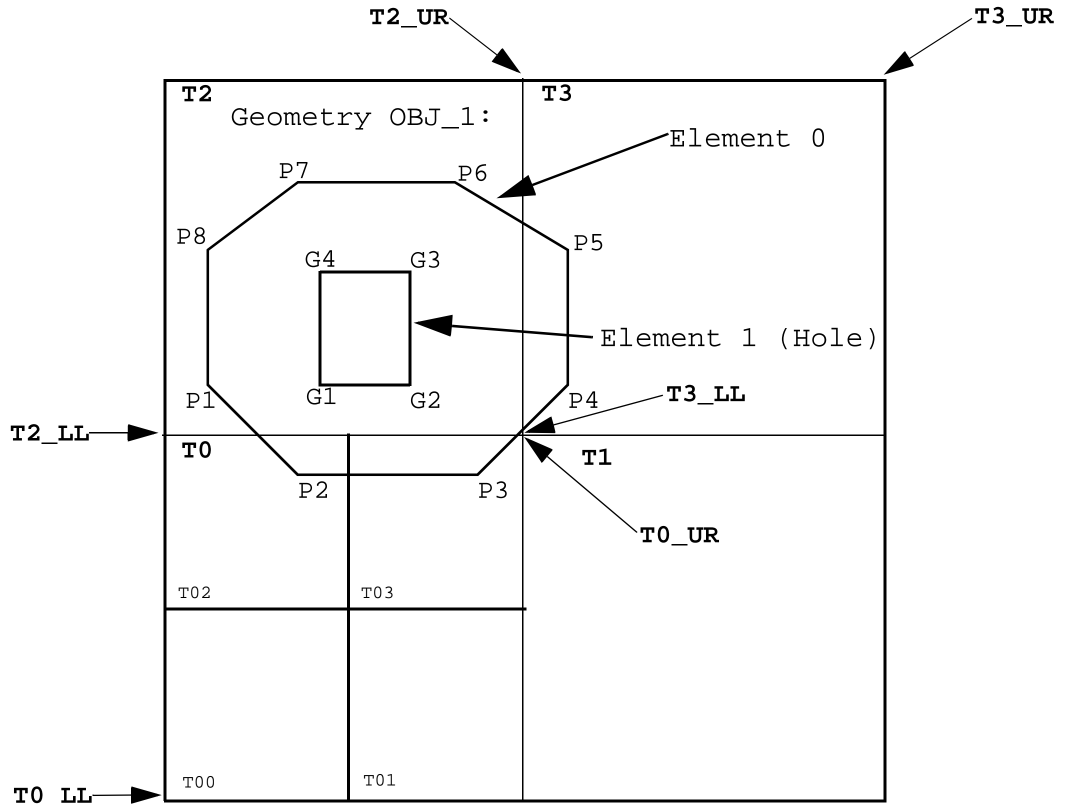
Figure 1–8 Variable-Sized Tile Spatial Indexing



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 1–9 illustrates how geometry OBJ_1, represented using the object-relational implementation, 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 1–2.

Figure 1–9 Decomposition of the Geometry



In Figure 1–9, 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 1–9, 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 1–2 shows the tiles from Figure 1–9 that are stored in the SDOINDEX table.

Table 1–2 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 1–9) 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. Appendix A contains suggestions on when hybrid indexing may be beneficial, and how to select values for the two required parameters.

1.8 Spatial Relations and Filtering

Spatial uses filter methods 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:

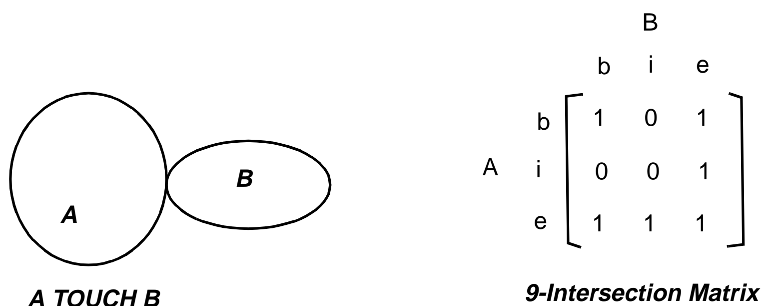
- The SDO_RELATE operator evaluates topological criteria.
- The SDO_WITHIN_DISTANCE operator determines if two spatial objects are within a Euclidean distance of each other.
- The SDO_NN operator identifies the nearest neighbors for a spatial object.

The syntax of these operators is given in Chapter 6.

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 1–10 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 1–10 The 9-Intersection Model

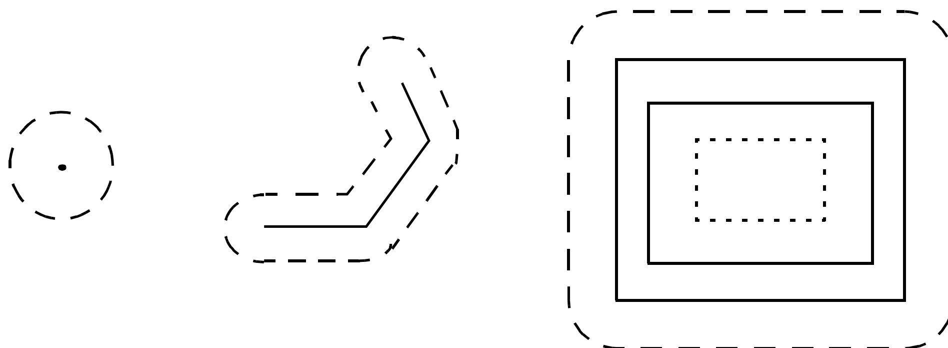


Some of the topological relationships identified in the seminal work by Professor Max Egenhofer (University of Maine, Orono) and colleagues have names associated with them. Spatial uses the following names:

- 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 relation 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.
- ANYINTERACT -- The objects are non-disjoint.

The SDO_WITHIN_DISTANCE operator determines if two spatial objects, A and B, are within a Euclidean 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 point, line, and area objects. Notice how the buffer is rounded near the corners of the objects.

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



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 Partitioned Point Data

Point data, unlike line and polygon data, has the unique characteristic of always using only one tile per point. For applications handling point data sets that are several tens of gigabytes or larger, performance gains can be achieved by using Oracle8*i* table partitioning features.

Table partitioning is available only with the Partitioning Option of Oracle8*i* Enterprise Edition. If the Partitioning Option is available to you, the preferred method is to use Oracle8*i* table partitioning in conjunction with spatial indexing (using the relational model). See *Oracle8i Concepts* for a description of Oracle8*i* partitioning. See Section A.1.7.3 for a description of a sample script that uses table partitioning with point data.

1.10 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 D.6)
- Linear referencing system (LRS) (Section E.5)

Part I

Object-Relational Model

Oracle Spatial supports two models for representing geometries: relational and object-relational. The two models are mutually exclusive. See Section 1.2 for a description of how to choose the model best suited for your application.

You do not need prior knowledge of the relational model to use the newer object-relational model.

This part of the User's Guide and Reference contains the following chapters that describe the object-relational model:

- Chapter 2, "The Object-Relational Schema"
- Chapter 3, "Loading and Indexing Spatial Object Types"
- Chapter 4, "Querying Spatial Data"
- Chapter 5, "Indexing Statements"
- Chapter 6, "Spatial Operators"
- Chapter 7, "Geometry Functions"
- Chapter 8, "Coordinate System Functions"
- Chapter 9, "Linear Referencing Functions"
- Chapter 10, "Migration Procedures"
- Chapter 11, "Tuning Functions and Procedures"

2 The Object-Relational Schema

The object-relational implementation of Oracle Spatial consists of a set of object data types, an index method type, and operators on 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.

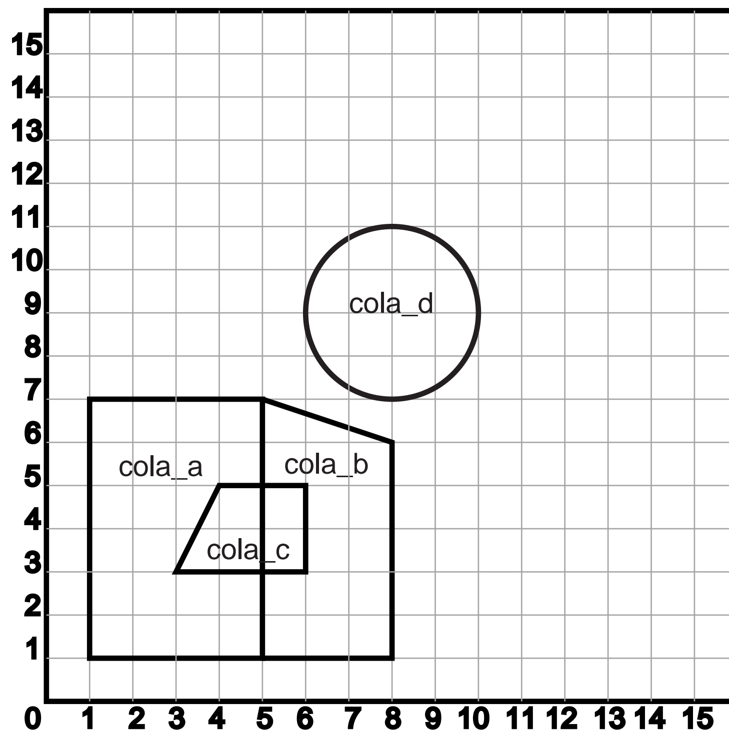
2.1 Simple Example: Inserting, Indexing, and Querying Spatial Data

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)
  )
);

-- 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, virtually zero tolerance
```

```
    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
PARAMETERS('SDO_LEVEL = 8');

-------------------------------------------------------------------
-- PERFORM SOME SPATIAL QUERIES --
-------------------------------------------------------------------
-- Return the topological intersection of two geometries.
SELECT SDO_GEOM.SDO_INTERSECTION(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';

-- Do two geometries have any spatial relationship?
SELECT SDO_GEOM.RELATE(c_b.shape, m.diminfo, 'anyinteract',
    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';

-- Return the areas of all cola markets.
SELECT c.name, SDO_GEOM.SDO_AREA(c.shape, m.diminfo)
  FROM cola_markets c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE';

-- Return the area of just cola_a.
SELECT c.name, SDO_GEOM.SDO_AREA(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';

-- Return the distance between two geometries.
SELECT SDO_GEOM.SDO_DISTANCE(c_b.shape, m.diminfo, 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';

-- Is a geometry valid?
SELECT c.name, SDO_GEOM.VALIDATE_GEOMETRY(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';

-- 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).

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. Table 2–1 shows the valid SDO_GTYPE values.

Table 2–1 Valid SDO_GTYPE Values

Value	Geometry Type	Description
*d*000	UNKNOWN_ GEOMETRY	Spatial ignores this geometry.
*d*001	POINT	Geometry contains one point.
*d*002	LINESTRING	Geometry contains one line string.
*d*003	POLYGON	Geometry contains one polygon with or without holes.[1]
*d*004	COLLECTION	Geometry is a heterogeneous collection of elements.[2]
*d*005	MULTIPOINT	Geometry has multiple points.
*d*006	MULTILINESTRING	Geometry has multiple line strings.
*d*007	MULTIPOLYGON	Geometry has multiple, disjoint polygons (more than one exterior boundary).

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

[2] All polygons in the collection must be disjoint.

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

> **Note:** The pre-release 8.1.6 format of a 1-digit value is still supported. If a 1-digit value is used, however, Oracle Spatial determines the number of dimensions and stores the appropriate 4-digit value in the DIMINFO column of the metadata views described in Section 2.4.

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 E.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.

Values *d*008-*d*099 are reserved for future use.

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 D.3.1), 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 Appendix D.

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 Appendix E.

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 0 through 5, as well as the following: 1003 and 2003 (variants of 3), and 1005 and 2005 (variants of 5).

  > **Note:** For polygon ring elements in a single geometry, you can use either 1-digit or 4-digit SDO_ETYPE values for all elements; however, you cannot mix 1-digit and 4-digit SDO_ETYPE values.

  SDO_ETYPE values 1, 2, and 3 are considered *simple elements*. They are defined by a single triplet entry in the SDO_ELEM_INFO array. Moreover, the following are considered variants of type 3, with the first digit indicating *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)

  You should specify an SDO_ETYPE value of 3 if you do not know if the simple polygon is exterior or interior; otherwise, you should specify 1003 or 2003.

  SDO_ETYPE values 4 and 5 are considered *compound elements*. They contain at least one header triplet with a series of triplet values that belong to the compound element. Moreover, the following are considered variants of type 5, with the first digit indicating *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)

  You should specify an SDO_ETYPE value of 5 if you do not know if the compound polygon is exterior or interior; otherwise, you should specify 1005 or 2005.

  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 or 5), this field specifies how many subsequent triplet values are part of the element.

  If the SDO_ETYPE is not a compound element (1, 2, or 3), 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	0	Unsupported element type. Ignored by the Spatial functions and procedures.
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 starting 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.
3	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.
3	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.
3	3	Rectangle type. A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it.
3	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	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 and Figure 2–4 for an example of a geometry using this type.
5	*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.

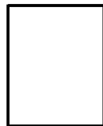
2.3 Geometry Examples Using the Object-Relational Model

This section contains examples of several geometry types.

2.3.1 Rectangle

Figure 2–2 illustrates a rectangle.

Figure 2–2 Rectangle



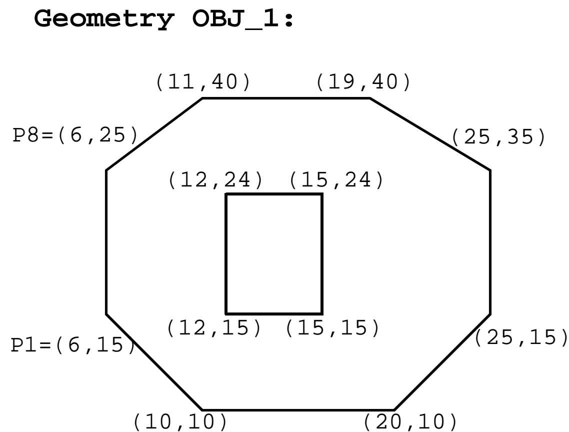
In the SDO_GEOMETRY definition of the geometry illustrated in Figure 2–2:

- 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 = (12,15,15,24). These identify the lower-left and upper-right ordinates of the rectangle.

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 Geometry 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, 12, meaning that the first point is 12,15).
- SDO_ORDINATES = (6,15, 10,10, 20,10, 25,15, 25,35, 19,40, 11,40, 6,25, 6,15, 12,15, 12,24, 15,24, 15,15, 12,15)
- The area (SDO_GEOM.SDO_AREA function) of the polygon is the area of the exterior polygon minus the area of the interior polygon.
- The perimeter (SDO_GEOM.SDO_LENGTH function) of the polygon is the perimeter of the exterior polygon plus the perimeter of the interior polygon.

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

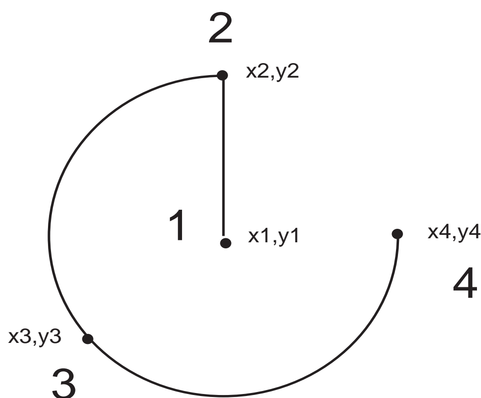
The elements in SDO_ELEM_INFO and SDO_ORDINATES must be in one of the following orders (depending on whether you specify Polygon A or Polygon B first):

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

2.3.3 Compound Element

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 1 and 2 describe the straight line segment and points 2, 3, and 4 describe the circular arc. The SDO_ELEM_INFO array contains 3 triplets for this compound line string. These are {(1,4,2), (1,2,1), (3,2,2)}. The SDO_ORDINATES array contains (X1,Y1, X2, Y2, X3, Y3, X4,Y4).

Figure 2–4 Compound Element



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

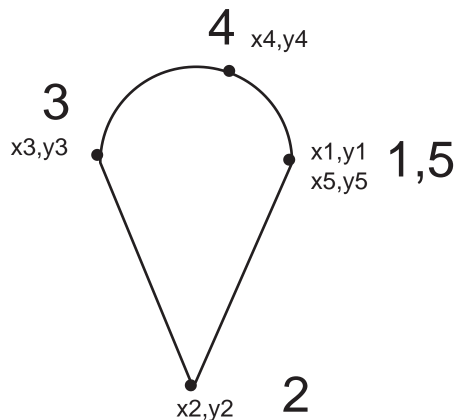
The second triplet indicates that the line string is made up of straight line segments and that the ordinates for this line string start at offset 1. The end point of this line string is determined by the starting offset of the second line string, 3 in this instance. Assuming the vertices are 2-dimensional, the coordinates for the end point of the first line string are at ordinates 3 and 4.

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.

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 1, 2, and 3 describe one acute angle-shaped line string, and points 3, 4, and 5 describe the circular arc. Points 1 and 5 are the same point. 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



NU-3747A-AI

The first triplet indicates that this element is a compound line string made up of two line strings, which are described using 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, 5 in this instance. Assuming 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 line string is made up of circular arcs 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.

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 B.

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.

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. It is imperative that the DIMINFO varying length array is ordered by dimension in the same way the ordinates for the points in SDO_ORDINATES varying length array are ordered. That is, 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 3.1.2 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 Appendix D) for all geometries in the column, or NULL if no specific coordinate system should be associated with the geometries.

2.5 Spatial Index-Related Structure

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 three metadata views per schema (user). These views are read-only to users; they are created and maintained by the Spatial indexing routines.

- 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 B.

The USER_SDO_INDEX_METADATA, ALL_SDO_INDEX_METADATA, and DBA_SDO_INDEX_METADATA 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_METADATA Views

Column Name	Data Type	Purpose
SDO_INDEX_OWNER	VARCHAR2	The 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	The 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_TSNAME	VARCHAR2	The schema name of the SDO_INDEX_TABLE.
SDO_COLUMN_NAME	VARCHAR2	The column name on which this index is built.
SDO_RTREE_HEIGHT	NUMBER	Height of the R-tree (R-tree index).
SDO_RTREE_NUM_NODES	NUMBER	Number of nodes in the R-tree (R-tree index).
SDO_RTREE_DIMENSIONALITY	NUMBER	Number of dimensions indexed (R-tree index).
SDO_RTREE_FANOUT	NUMBER	Maximum number of children in each R-tree node (R-tree index).
SDO_RTREE_ROOT	VARCHAR2	Rowid corresponding to the root node of the R-tree in the index table (R-tree index).
SDO_RTREE_SEQ_NAME	VARCHAR2	Sequence name associated with the R-tree (R-tree index).
SDO_LEVEL	NUMBER	The fixed tiling level at which to tile all objects in the geometry column (quadtree index).
SDO_NUMTILES	NUMBER	Suggested number of tiles per object that should be used to approximate the shape (quadtree index).

Table 2–3 Columns in the xxx_SDO_INDEX_METADATA Views (Cont.)

Column Name	Data Type	Purpose
SDO_MAXLEVEL	NUMBER	The maximum level for any tile for any object (quadtree index). It will always be greater than the SDO_LEVEL value.
SDO_COMMIT_INTERVAL	NUMBER	The number of geometries (rows) to process, during index creation, before committing the insertion of spatial index entries into the SDOINDEX table. See Section A.1.4 for more information about SDO_COMMIT_INTERVAL.
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.

2.5.2 Spatial Index Table Definition

Each quadtree spatial index table (each SDO_INDEX_TABLE entry as described in Table 2–3 in Section 2.5.1) contains the columns shown in Table 2–4.

Table 2–4 Columns in a 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	Row ID 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).

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 2–3 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. For example, if the index table name is E1_RT$, the sequence name is E1_RT$S.

3

Loading and Indexing Spatial Object Types

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 and creating a spatial index for it.

The following steps will enable you to query spatial data efficiently:

1. Load data into a column of type SDO_GEOMETRY.
2. Create spatial indexes on columns of type SDO_GEOMETRY.

3.1 Load Process

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 inserts

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

3.1.1 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 *Oracle8i Utilities*.)

3.1.1.1 Bulk Loading the SDO_GEOMETRY Object

The following example 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 roads for a region. Example 3–1 shows the control file for loading the roads and attributes.

Example 3–1   Control File for a Bulk Load

```
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 '|/'
       (X            FLOAT EXTERNAL),
     SDO_ORDINATES   VARRAY TERMINATED BY '|/'
       (Y            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|/
```

3.1.1.2 Bulk Loading Point-Only Data in the SDO_GEOMETRY Object

Example 3–2 shows a control file for loading a table with point data.

Example 3–2 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|
```

3.1.2 Transactional Insert Using SQL

Oracle Spatial uses standard Oracle8*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 Oracle8*i* 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 3–3 creates a procedure to perform the insert operation.

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

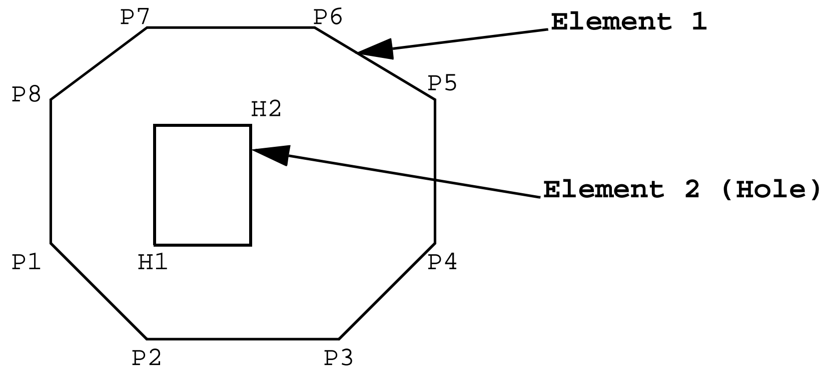
Example 3–4 PL/SQL Block Invoking Procedure to Insert a Geometry

```
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;
/
```

3.1.2.1 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 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.

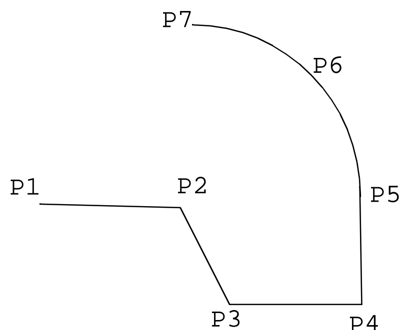
Assume that two dimensions are named X and Y, their bounds are 0 to 100, and the tolerance for both dimensions is 0.005. The SQL statement for loading the USER_SDO_GEOM_METADATA metadata view is:

```
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);
```

3.1.2.2 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.

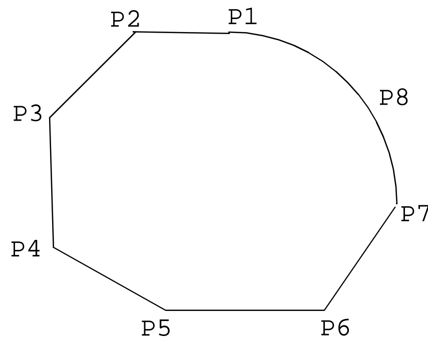
Assume that two dimensions are named X and Y, their bounds are 0 to 100, and tolerance for both dimensions is 0.005. The SQL statement to insert the metadata into the USER_SDO_GEOM_METADATA view is:

```
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);
```

3.1.2.3 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



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);
```

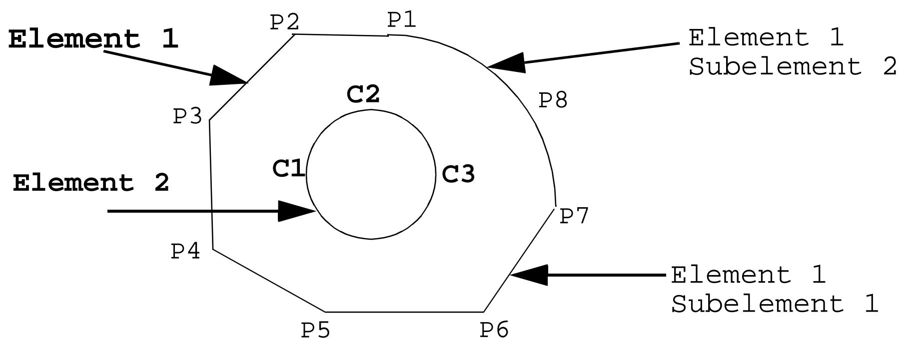
Assume that two dimensions are named X and Y, their bounds are 0 to 100, and tolerance for both dimensions is 0.005. The SQL statement to insert the metadata into the USER_SDO_GEOM_METADATA view is:

```
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);
```

3.1.2.4 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);
```

3.1.2.5 Transactional Insert 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)
                          );
```

3.2 Index Creation

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.

3.2.1 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.

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.

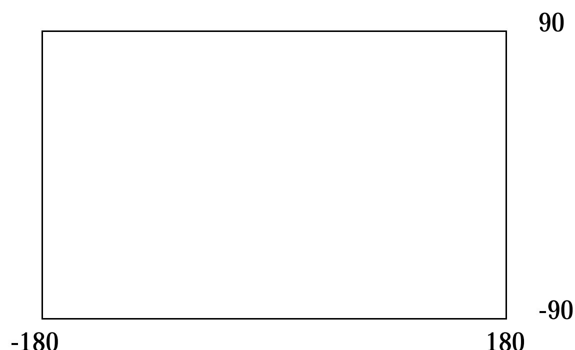
3.2.2 Spatial Indexing with Fixed-Size Tiles (Quadtree Indexes)

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. (These rare circumstances are explained in Section 1.7.2.3, and hybrid indexing is discussed in Section 3.2.3. 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,[1] and -90 to 90 degrees for latitude, as represented in Figure 3–5.

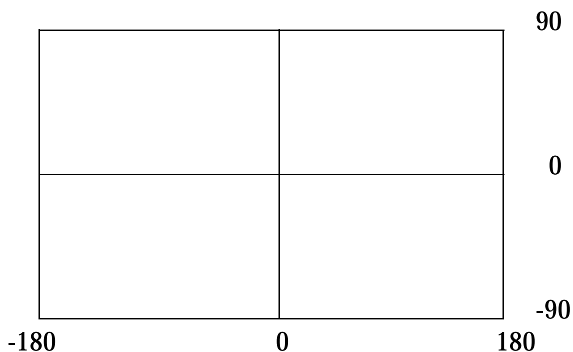
Figure 3–5 Sample Domain



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 3–6.

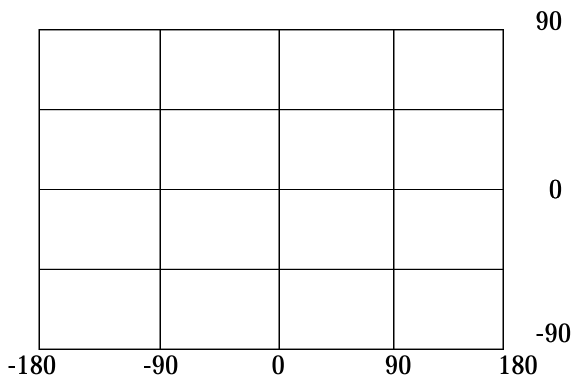
[1] The transference of the domain onto a sphere or other projection is left up to an application, unless a coordinate system is specified, as explained in Appendix D.)

Figure 3–6 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 3–7 shows fixed-size tiling at level 2. In this figure, each tile is 90 degrees by 45 degrees.

Figure 3–7 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 3–7 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 11, 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 3–5, 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 3–5 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.

3.2.3 Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles

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.

Note: With quadtree indexes, you should use fixed indexing for most applications, except for the rare circumstances where hybrid indexing should be considered. These rare circumstances are explained in Section 1.7.2.3. You should also consider using R-tree indexing (see Section 1.7) before deciding on hybrid indexing.

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 4.2.1 and Section 4.2.2 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. This technique and others are described in Appendix A.

In Example 3–6, 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 3–6 Creating a Hybrid Index

```
CREATE INDEX ROADS_HYBRID ON ROADS(SHAPE)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX PARAMETERS('SDO_LEVEL=6 SDO_NUMTILES=12');
```

3.2.4 R-tree Index Parameter Considerations

This section describes considerations and recommendations for parameters related to R-tree indexes. For basic information about all available parameters, see the CREATE INDEX statement description in Chapter 5.

3.2.4.1 SDO_FANOUT

The default value for SDO_FANOUT is best for most applications. However, a larger value of 60 for SDO_FANOUT is recommended for very large databases (more than 1 million rows).

3.2.4.2 SDO_RTR_PCTFREE

The default value for SDO_RTR_PCTFREE is best for most applications. However, a value of 0 for SDO_RTR_PCTFREE is recommended if no updates will be performed to the geometry column.

3.2.5 Cross-Schema Index Creation

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 qtree on B.T1(geometry)
     INDEXTYPE IS mdsys.spatial_index
     PARAMETERS('sdo_level=10 sdo_numtiles=4');
   ```

4

Querying Spatial Data

This chapter describes how the structures of a Spatial layer in the object-relational model are used to resolve spatial queries and spatial joins. For the sake of clarity, the examples all use fixed-size tiling, but hybrid indexing is actually recommended for the object-relational model.

4.1 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.

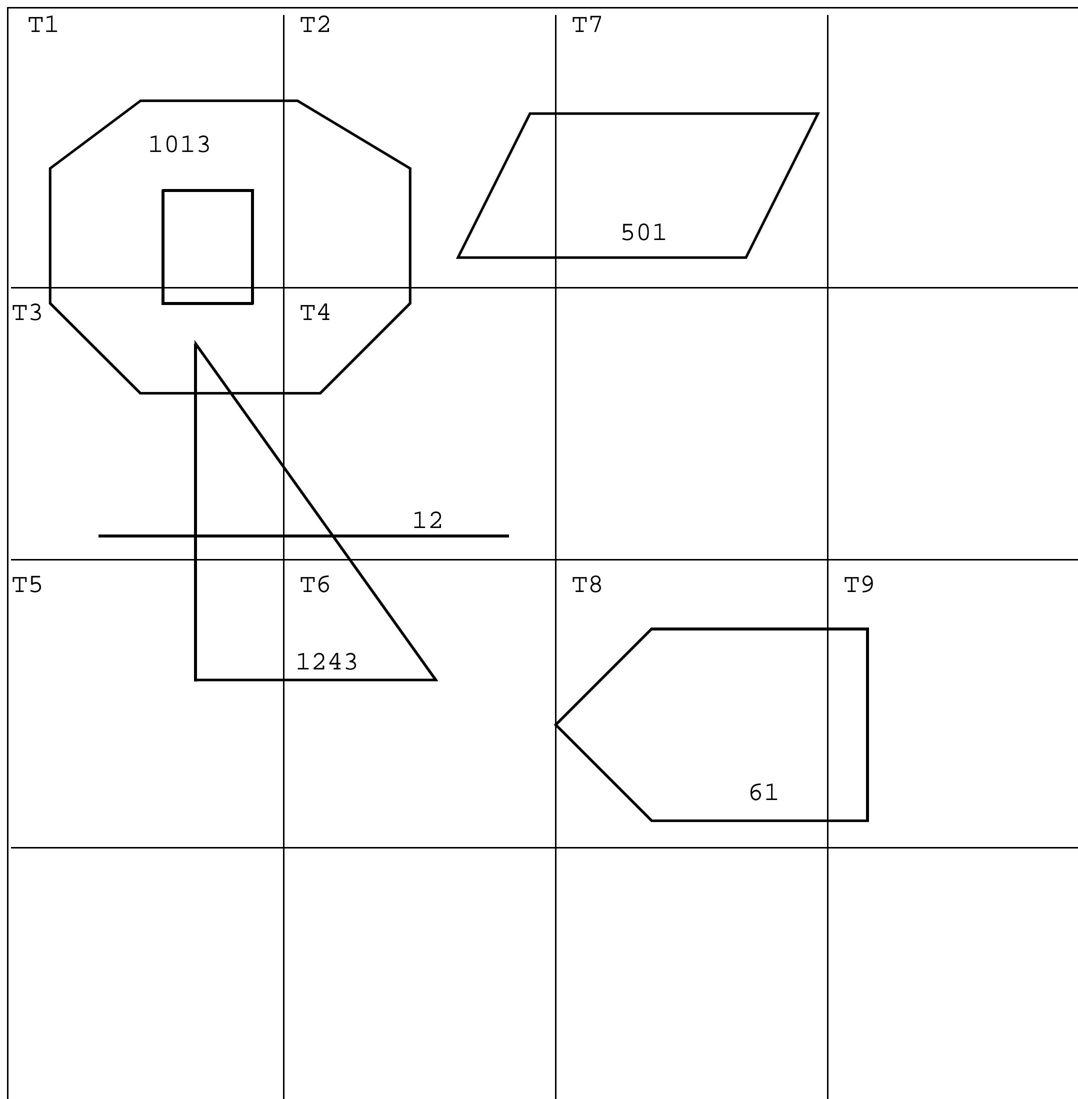
- The **primary filter** permits fast selection of candidate records to pass along to the secondary filter. The primary filter uses geometry approximations (or index tiles) to reduce computational complexity and is considered a lower-cost filter.
- The **secondary filter** applies exact computational 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 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**).

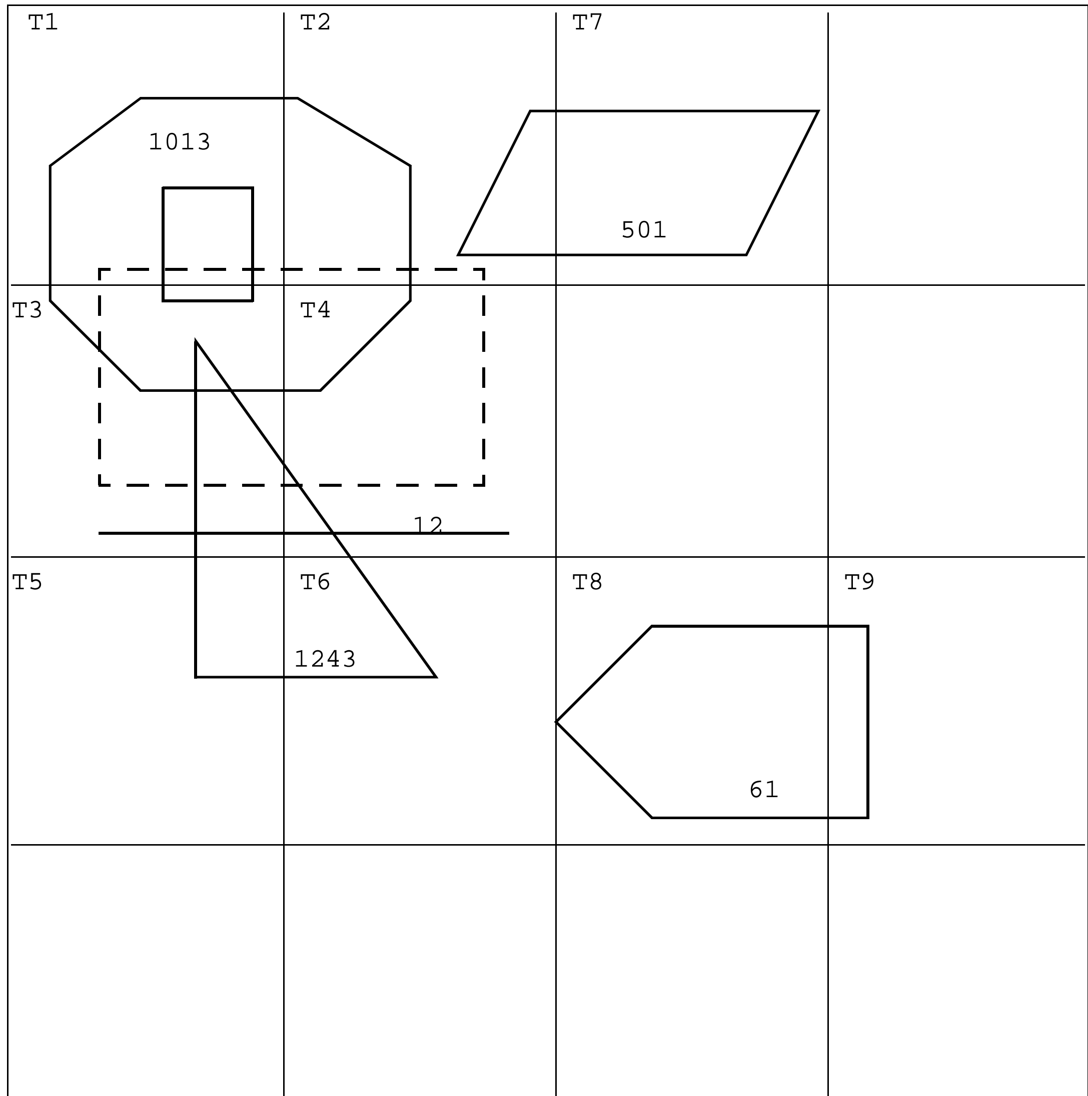
Consider the following layer containing several objects in Figure 4–1. Each object is labeled with its SDO_GID. The relevant tiles are labeled with 'T*n*'.

Figure 4–1 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 4–2 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 4–2 Tessellated Layer with a Query Window



4.2.1 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 6 for a list of parameters.

The following examples perform a primary filter operation only. They will return all the geometries shown in Figure 4–2 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 4–1 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 4–1 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 4–2, a transient instance of type SDO_GEOMETRY was constructed for the query window instead of specifying the window parameters in the query itself.

Example 4–2 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 4–3 assumes the query window was inserted into a table called WINDOWS, with an ID of WINS_1.

Example 4–3 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 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 6 for a list of parameters.

The following examples perform both primary and secondary filter operations. They return all the geometries in Figure 4–2 that lie within or overlap the query window. The result of these examples is objects 1243 and 1013.

Example 4–4 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 4–4 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 4–5 assumes the query window was inserted into a table called WINDOWS, with an ID of WINS_1.

Example 4–5 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.

4.2.3 Within-Distance Operator

The SDO_WITHIN_DISTANCE operator is used to determine the set of objects in a table that are within *n* Euclidean distance units from a reference object *aRefGeom*. 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,
                    aRefGeom   MDSYS.SDO_GEOMETRY,
                    params     VARCHAR2)
```

Where:

- *geometry1* is a column of type MDSYS.SDO_GEOMETRY in a table. *geometry1* must be spatially indexed.
- *aRefGeom* 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 6 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. Spatial treats the coordinate space as Cartesian. If your data consists of latitude and longitude pairs, then you cannot use the SDO_WITHIN_DISTANCE operator to provide correct results unless all your data is near the equator. If all the data is not near the equator, you must project the latitude/longitude data into a locally-conformal Cartesian plane before using the SDO_WITHIN_DISTANCE operator.

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.

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';
```

4.2.4 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)
```

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 6 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 6.)

```
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';
```

4.3 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');
```

4.4 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

Indexing Statements

This chapter describes the statements used when working with the spatial object data type. The statements are listed in Table 5–1.

Table 5–1 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

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_fanout	Specifies the fanout value, which reflects the node capacity of the index tree. (R-tree indexes only.) If queries that use the index are likely to return thousands of rows or more, you may want to specify a value greater than the default, such as 50 or 60. Data type is NUMBER. Default = 35.
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 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');
```

Related Topics

- ALTER INDEX REBUILD
- ALTER INDEX RENAME TO
- CREATE INDEX

ALTER INDEX REBUILD

Syntax

```
ALTER INDEX [schema.]index REBUILD
  [PARAMETERS (‘rebuild_params [physical_storage_params]’ ) ]
```

Purpose

Rebuilds a spatial 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	Specifies special processing for point data. If the layer you are indexing is all points, set this parameter to POINT for optimal performance. 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_fanout	Specifies the fanout value, which reflects the node capacity of the index tree. (R-tree indexes only.) If queries that use the index are likely to return thousands of rows or more, you may want to specify a value greater than the default, such as 50 or 60. Data type is NUMBER. Default = 35.
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.

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 INDEX RENAME TO

Syntax

ALTER INDEX [schema.]index RENAME TO <new_index_name>

Purpose

Alters the name of a spatial index.

Keywords and Parameters

new_index_name	Specifies the new name of the index.

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

The *new_index_name* string must not be longer than 18 characters.

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
layer_gtype	Specifies special processing for point data. (Quadtree indexes only.) If the layer you are indexing is all points, set this parameter to POINT for optimal performance. 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_fanout	Specifies the fanout value, which reflects the node capacity of the index tree. (R-tree indexes only.) If queries that use the index are likely to return thousands of rows or more, you may want to specify a value greater than the default, such as 50 or 60. Data type is NUMBER. Default = 35.
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 1.7.1.1. See also the considerations and recommendations for certain R-tree index parameters in Section 3.2.4.

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 E.2.)

Other options available for regular indexes (such as ASC and DESC) are not applicable for spatial indexes.

The *index_name* string must not be longer than 18 characters.

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 5–2.

Table 5–2 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.

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 quadtree index named QTREE.

```
CREATE INDEX qtree ON POLY_4PT(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('sdo_numtiles=4 sdo_level=6
    sdo_commit_interval=500 tablespace=system initial=10K
    next=10K pctincrease=10 minextents=10 maxextents=20');
```

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

6

Spatial Operators

This chapter describes the operators used when working with the spatial object data type. The operators are listed in Table 6–1.

Table 6–1 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_RELATE	Determines whether or not two geometries interact in a specified way.
SDO_WITHIN_DISTANCE	Determines if two geometries are within a specified Euclidean 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.

layer_gtype Specifies special processing for point data.

If the columns you are comparing have only point data, set this parameter to POINT for optimal performance.
Data type is VARCHAR2. Default = NOTPOINT.

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.
- Both *geometry1* and *geometry2* must have the same type of index (R-tree or quadtree). If the geometries have quadtree indexes, the indexes must have the same *sdo_level* value.

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);
```

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. Data type is VARCHAR2.

Keyword	Description
sdo_num_res	Specifies the number of results (nearest neighbors). If not specified, the default is 1. For example: 'sdo_num_res=10'

Returns

This operator returns the *sdo_num_res* number of objects from *geometry1* that are closest to *geometry2* in the query. In determining how close 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, 'sdo_num_res=<some_val>') = 'TRUE'.

You should not make any assumptions about the order of the returned results. For example, the first of several returned objects is not guaranteed to be the one closest to *geometry2*.

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 closest 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.

SDO_NN is not supported for spatial joins.

In some situations the SDN_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 Oracle8i Performance Guide and Reference.

Examples

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). (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
---------- --------------------------------
         4 cola_d
         2 cola_b
```

Related Topics

None.

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. Multiple masks are combined with a 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'.
layer_gtype	Specifies special processing for point data. If the columns you are comparing have only point data, set this parameter to POINT for optimal performance. Data type is VARCHAR2. Default = NOTPOINT.

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.
- Both *geometry1* and *geometry2* must have the same type of index (R-tree or quadtree). If the geometries have quadtree indexes, the indexes must have the same *sdo_level* value.

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(T.column, aGeom, params);

Description

Uses the spatial index to identify the set of spatial objects that are within some specified Euclidean distance of a given object (such as an area of interest or point of interest).

Keywords and Parameters

T.column	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 *T.column*. 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 Euclidean distance value. 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 *T.column*.
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.
layer_gtype	Allows special processing for point data. If the objects in *T.column* have only point data, set this parameter to POINT for optimal performance. Do not set this parameter to POINT if *T.column* contains any *n* on-point objects. Data type is VARCHAR2. Default = NOTPOINT.

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.

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'
```

T.column must have a spatial index built on it.

SDO_WITHIN_DISTANCE is not supported for spatial joins. See Section 4.2.3 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

7

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_AREA, SDO_BUFFER, SDO_CENTROID, SDO_CONVEXHULL, SDO_LENGTH, SDO_POINTONSURFACE
- Two-object operations: SDO_DISTANCE, SDO_DIFFERENCE, SDO_INTERSECTION, SDO_UNION, SDO_XOR

The geometry functions are listed Table 7–1, and some usage information follows the table.

Table 7–1 Geometric Functions for the Object-Relational Model

Function	Description
SDO_GEOM.RELATE	Determines how two objects interact.
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.
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.

Table 7–1 Geometric Functions for the Object-Relational Model (Cont.)

Function	Description
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_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,
    tolerance  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).

tolerance
Tolerance value (see Section 1.5.4).

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 *tolerance* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

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, m.diminfo, 'anyinteract', 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.RELATE(C_B.SHAPE,M.DIMINFO,'ANYINTERACT',C_D.SHAPE,M.DIMINFO)
--------------------------------------------------------------------------------
FALSE
```

Related Topics

None.

SDO_GEOM.SDO_AREA

Format

```
SDO_GEOM.SDO_AREA(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_AREA(
    geom      IN MDSYS.SDO_GEOMETRY,
    tolerance IN NUMBER
    ) 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).

tolerance
Tolerance value (see Section 1.5.4).

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 *tolerance* 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 c.name, SDO_GEOM.SDO_AREA(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_AREA(C.SHAPE,M.DIMINFO)
-------------------------------- ------------------------------------
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, 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_AREA(C.SHAPE,M.DIMINFO)
-------------------------------- ------------------------------------
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,
     distance  IN NUMBER,
     ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_BUFFER(
     geom       IN MDSYS.SDO_GEOMETRY,
     distance   IN NUMBER,
     tolerance  IN NUMBER
     ) 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).

distance
Euclidean distance value.

tolerance
Tolerance value (see Section 1.5.4).

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 *tolerance* 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 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 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))
```

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(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_CENTROID(
    geom       IN MDSYS.SDO_GEOMETRY,
    tolerance  IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns the centroid of a polygon. (The centroid is also known as the "center of gravity.")

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).

tolerance
Tolerance value (see Section 1.5.4).

Usage Notes

This function returns a null value if *geom* is not a polygon or if *geom* is a multipolygon.

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).

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(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_CONVEXHULL(
    geom       IN MDSYS.SDO_GEOMETRY,
    tolerance  IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a polygon-type object that represents the convex hull 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).

tolerance
Tolerance value (see Section 1.5.4).

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.

The geometry object (*geom*) cannot be a circle.

This function returns a null value if *geom* is of point type or has fewer than three points or vertices.

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).

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,
     tolerance  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.

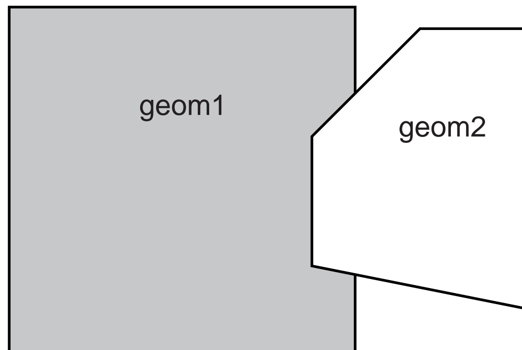
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tolerance
Tolerance value (see Section 1.5.4).

Usage Notes

In Figure 7–1, the shaded area represents the polygon returned when SDO_DIFFERENCE is used with a square (*geom1*) and another polygon (*geom2*).

Figure 7–1 SDO_GEOM.SDO_DIFFERENCE



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 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
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_DISTANCE(
    geom1     IN MDSYS.SDO_GEOMETRY,
    geom2     IN MDSYS.SDO_GEOMETRY,
    tolerance IN NUMBER
    ) 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).

tolerance

Tolerance value (see Section 1.5.4).

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 *tolerance* 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 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, m.diminfo, 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.SDO_DISTANCE(C_B.SHAPE,M.DIMINFO,C_D.SHAPE,M.DIMINFO)
--------------------------------------------------------------
                                                    .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,
     tolerance  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.

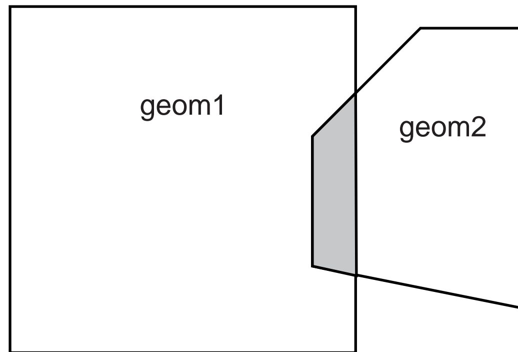
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tolerance
Tolerance value (see Section 1.5.4).

Usage Notes

In Figure 7–2, the shaded area represents the polygon returned when SDO_INTERSECTION is used with a square (*geom1*) and another polygon (*geom2*).

Figure 7–2 SDO_GEOM.SDO_INTERSECTION



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 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, 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_INTERSECTION(C_A.SHAPE,M.DIMINFO,C_C.SHAPE,M.DIMINFO)(SDO_GTYPE, SD
--------------------------------------------------------------------------------
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
    ) RETURN NUMBER;
```

or

```
SDO_GEOM.SDO_LENGTH(
    geom       IN MDSYS.SDO_GEOMETRY,
    tolerance  IN NUMBER
    ) 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).

tolerance
Tolerance value (see Section 1.5.4).

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 *tolerance* 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_POINTONSURFACE

Format

```
SDO_GEOM.SDO_POINTONSURFACE(
     geom  IN MDSYS.SDO_GEOMETRY,
     dim     IN MDSYS.SDO_DIM_ARRAY
     ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_GEOM.SDO_POINTONSURFACE(
     geom       IN MDSYS.SDO_GEOMETRY,
     tolerance  IN NUMBER
     ) RETURN MDSYS.SDO_GEOMETRY;
```

Description

Returns a point that is guaranteed to be on the surface of a polygon geometry object.

Parameters

geom
Polygon geometry object.

dim
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tolerance
Tolerance value (see Section 1.5.4).

Usage Notes

This function returns a point geometry object representing a point that is guaranteed to be on the surface of *geom*.

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 *tolerance* 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,
     tolerance  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).

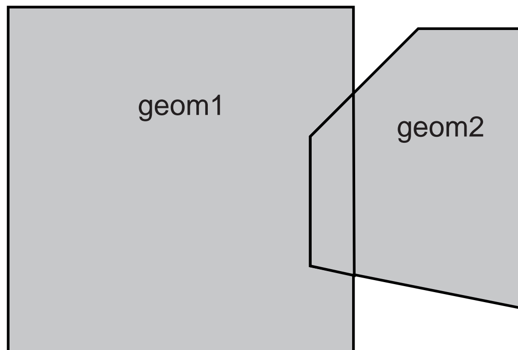
tolerance

Tolerance value (see Section 1.5.4).

Usage Notes

In Figure 7–3, the shaded area represents the polygon returned when SDO_UNION is used with a square (*geom1*) and another polygon (*geom2*).

Figure 7–3 SDO_GEOM.SDO_UNION



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 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,
    tolerance  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.

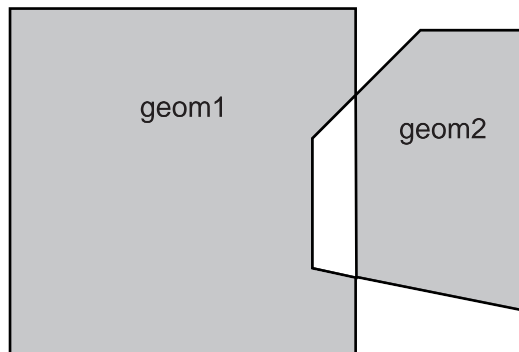
dim2
Dimensional information array corresponding to *geom2*, usually selected from one of the xxx_SDO_GEOM_METADATA views (see Section 2.4).

tolerance
Tolerance value (see Section 1.5.4).

Usage Notes

In Figure 7–4, the shaded area represents the polygon returned when SDO_XOR is used with a square (*geom1*) and another polygon (*geom2*).

Figure 7–4 SDO_GEOM.SDO_XOR



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 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, 11, 1003, 1), SDO
_ORDINATE_ARRAY(5, 5, 5, 3, 6, 3, 6, 5, 5, 5, 1, 7, 1, 1, 5, 1, 5, 3, 3, 3, 4, 5
, 5, 5, 5, 7, 1, 7))
```

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 (5, 5) and the other starting and ending at (1, 7).

Related Topics

- SDO_GEOM.SDO_DIFFERENCE
- SDO_GEOM.SDO_INTERSECTION
- SDO_GEOM.SDO_UNION

SDO_GEOM.VALIDATE_GEOMETRY

Format

```
SDO_GEOM.VALIDATE_GEOMETRY(
    geom  IN MDSYS.SDO_GEOMETRY,
    dim   IN MDSYS.SDO_DIM_ARRAY
    ) 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

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

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).
- 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.

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, 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.VALIDATE_GEOMETRY(C.SHAPE,M.DIMINFO)
--------------------------------------------------------------------------------
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);
```

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.

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.) In each row, 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, no rows are 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;

no rows selected
```

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,
    distance 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,
    distance  IN NUMBER,
    geom2     IN MDSYS.SDO_GEOMETRY,
    tolerance 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).

distance

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).

tolerance

Tolerance value (see Section 1.5.4).

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 *tolerance* is used, all geometry objects must be defined using 4-digit SDO_GTYPE values (explained in Section 2.2.1).

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

8 Coordinate System 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 Appendix D.

Table 8–1 lists the coordinate systems functions and procedures.

Table 8–1 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).

The rest of this chapter provides reference information on the functions and procedures, listed in alphabetical order.

Note: Error messages for coordinate system functions are documented in Section D.7. (They are not included in the *Oracle8i Error Messages* manual for release 8.1.7.)

SDO_CS.TRANSFORM

Format

```
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    dim_array   IN MDSYS.SDO_DIM_ARRAY,
    to_srid     IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    dim_array   IN MDSYS.SDO_DIM_ARRAY,
    to_srname   IN VARCHAR2
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    tolerance   IN NUMBER,
    to_srid     IN NUMBER
    ) RETURN MDSYS.SDO_GEOMETRY;
```

or

```
SDO_CS.TRANSFORM(
    geom        IN MDSYS.SDO_GEOMETRY,
    tolerance   IN NUMBER,
    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 D.3.1).

dim_array
Dimensional information array corresponding to *geom*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

tolerance
Tolerance value (see Section 1.5.4).

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 D.3.1).

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 D.3.1).

Usage Notes

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 D.3.1).

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 D.6.)

```
-- Return the transformation of cola_c using to_srid 8199
SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 8199)
  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_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(3.00074116, 3.00289624, 6.0006707, 3.00289431, 6.00067234, 5.00305745, 4.0007
1964, 5.00305956, 3.00074116, 3.00289624))
```

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 D.3.1).

Usage Notes

An exception is raised if any of the following occurs:

- *table_in* does not exist, or *column_in* does not exist in the table.
- *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 8–2.

Table 8–2 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 *Oracle8i SQL Reference* manual.
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 table to a representation that uses SRID value 8199. The transformed geometries are stored in the newly created table named COLA_MARKETS_8199. (This example uses the definitions from the example in Section D.6.)

```
-- Transform the entire SHAPE layer and put results in the table
-- named cola_markets_8199, which the procedure will create.
EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS','SHAPE','COLA_MARKETS_8199',8199);
```

Example D–2 in Section D.6 includes a display of the geometry object coordinates in both tables (COLA_MARKETS and COLA_MARKETS_8199).

9

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 Appendix E.

Table 9–1 lists functions related to creating and editing geometric segments.

Table 9–1 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 9–1 Functions for Creating and Editing Geometric Segments (Cont.)

Function	Description
SDO_LRS.CONCATENATE_GEOM_SEGMENTS	Concatenates two geometric segments into one segment.
SDO_LRS.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).

Table 9–2 lists functions related to querying geometric segments.

Table 9–2 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.
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.

Table 9–2 Functions for Querying Geometric Segments (Cont.)

Function	Description
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.

Table 9–3 lists functions related to converting geometric segments.

Table 9–3 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.
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.

Table 9–3 Functions for Converting Geometric Segments (Cont.)

Function	Description
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 E.4.9.

The rest of this chapter provides reference information on the functions, listed in alphabetical order.

Note: Error messages for linear referencing functions are documented in Section E.6. (They are not included in the *Oracle8i Error Messages* manual for release 8.1.7.)

SDO_LRS.CLIP_GEOM_SEGMENT

Format

```
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:** 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 geometric segments, see Section E.4.3

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 E.5.)

```
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(a.route_geometry, m.diminfo, 5, 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.CLIP_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,5,10)(SDO_GTYPE, SDO_SRID,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, 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,
     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.

Usage Notes

An exception is raised if *geom_segment_1* or *geom_segment_2* is invalid, or if the end point of the first segment and the start point of the second segment are not spatially connected.

The direction of the resulting 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.

For more information about concatenating geometric segments, see Section E.4.5

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 E.5. The definitions of ***result_geom_1***, ***result_geom_2***, and ***result_geom_3*** are displayed in Example E–3.)

```
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,
    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.

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* is invalid.

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, m.diminfo,
                                        b.route_geometry, m.diminfo)
  FROM lrs_routes a, lrs_routes b, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND m.column_name = 'ROUTE_GEOMETRY'
    AND a.route_id = 1;

SDO_LRS.CONNECTED_GEOM_SEGMENTS(A.ROUTE_GEOMETRY,M.DIMINFO,B.ROUTE_GEOMETRY,M.DI
--------------------------------------------------------------------------------
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;
```

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.

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 (SDO_TOLERANCE value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

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, and sets the SDO_DIMNAME value in this added SDO_DIM_ELEMENT to M. 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.

For more information about conversion functions, see Section E.4.9.

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 E.5.)

```
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,
    dim_array       IN MDSYS.SDO_DIM_ARRAY
    [, 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.

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* is invalid or if *start_measure* or *end_measure* is out of range.

For more information about conversion functions, see Section E.4.9.

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 E.5.)

```
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;
```

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.

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 (SDO_TOLERANCE value in the SDO_DIM_ELEMENT definition) of the ordinate in the measure dimension.

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.

For more information about conversion functions, see Section E.4.9.

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 E.5.) 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
    ) 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.

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*.

If *dim_array* is already a standard dimensional array (that is, does not contain dimensional information), the *dim_array* is returned.

For more information about conversion functions, see Section E.4.9.

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 E.5.)

```
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.

For more information about conversion functions, see Section E.4.9.

Examples

The following example converts the geometric segment representing Route 1 to standard format. (This example uses the definitions from the example in Section E.5.)

```
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.

For more information about conversion functions, see Section E.4.9.

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 E.5.) 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,
    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. (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.

end_measure
Distance measured from the end point of a geometric segment to the start point of the linear feature.

Usage Notes

An exception is raised if *geom_segment* is invalid or if *start_measure* or *end_measure* is out of range.

All unassigned measures of the geometric segment will be populated automatically.

For more information about defining a geometric segment, see Section E.4.1

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 E.5. The definitions of *result_geom_1*, *result_geom_2*, and *result_geom_3* are displayed in Example E–3.)

```
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.DYNAMIC_SEGMENT

Format

```
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 E.4.3

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 E.5.)

```
SELECT  SDO_LRS.DYNAMIC_SEGMENT(a.route_geometry, m.diminfo, 5, 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.DYNAMIC_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,5,10)(SDO_GTYPE, SDO_SRID,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 4, 5, 8, 4, 8, 10, 4, 10))
```

SDO_LRS.FIND_MEASURE

Format

```
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* or *point* is invalid.

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 E.5.)

```
-- 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* is invalid.

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 E.5.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_END_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.GEOM_SEGMENT_END_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO)
------------------------------------------------------------
                                                          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* is invalid.

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 E.5.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_END_PT(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.GEOM_SEGMENT_END_PT(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 0, 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* is invalid.

Examples

The following example returns the length of the geometric segment representing Route 1. (This example uses the definitions from the example in Section E.5.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(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.GEOM_SEGMENT_LENGTH(A.ROUTE_GEOMETRY,M.DIMINFO)
-------------------------------------------------------
                                                     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* is invalid.

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 E.5.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(a.route_geometry, m.diminfo)
    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.GEOM_SEGMENT_START_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------
                                                             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* is invalid.

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 E.5.)

```
SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(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.GEOM_SEGMENT_START_PT(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, S
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 0, 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 a point on a geometric segment.

Parameters

point
Point whose measure along its geometric segment 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 a point on the geometric segment.

If *point* is not valid, an “invalid LRS segment” 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 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.

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 order along the segment direction.

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 E.5.)

```
SELECT SDO_LRS.IS_GEOM_SEGMENT_DEFINED(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_GEOM_SEGMENT_DEFINED(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE
```

SDO_LRS.LOCATE_PT

Format

```
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*.

Usage Notes

This function returns the referenced point.

An exception is raised if *geom_segment* is invalid or if the location is out of range.

For more information about locating a point on a geometric segment, see Section E.4.7.

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 E.5.)

```
SELECT  SDO_LRS.LOCATE_PT(a.route_geometry, m.diminfo, 9, 0)
    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.LOCATE_PT(A.ROUTE_GEOMETRY,M.DIMINFO,9,0)(SDO_GTYPE, SDO_SRID, SDO_POINT
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 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*.

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 E.5.)

```
SELECT SDO_LRS.MEASURE_RANGE(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.MEASURE_RANGE(A.ROUTE_GEOMETRY,M.DIMINFO)
-------------------------------------------------
                                               27
```

SDO_LRS.MEASURE_TO_PERCENTAGE

Format

```
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 E.5.) 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.PERCENTAGE_TO_MEASURE

Format

```
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* is invalid 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 E.5.) 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,
     dim_array     IN MDSYS.SDO_DIM_ARRAY,
     point         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.

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* is invalid.

For more information about projecting a point onto a geometric segment, see Section E.4.8.

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 E.5.)

```
-- Point 9,3,NULL is off the road; should return 9,4,9
SELECT  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)) )
    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.PROJECT_PT(A.ROUTE_GEOMETRY,M.DIMINFO,MDSYS.SDO_GEOMETRY(3001,NULL,NULL,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 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,
    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 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 distance of the segment.

Usage Notes

An exception is raised if *geom_segment* is invalid or if *start_measure* or *end_measure* is out of range.

For more information about redefining a geometric segment, see Section E.4.2.

Examples

The following example redefines a geometric segment. (This example uses the definitions from the example in Section E.5.)

```
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';

-- Define the LRS segment for Route1.
SDO_LRS.REDEFINE_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.
END;
/
```

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 original geometric segment.

Parameters

geom_segment
Cartographic representation of a linear feature.

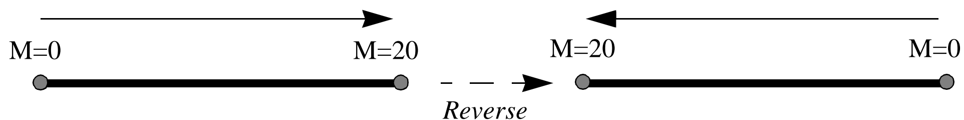
dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

Usage Notes

This function reverses the start and end measures (and consequently the direction) of *geom_segment*. That is, the start measure of *geom_segment* is the end measure of the returned geometric segment, and the end measure of *geom_segment* is the start measure of the returned geometric segment.

Figure 9–1 shows the reversal of the start and end measures and the segment direction resulting from this function.

Figure 9–1 Reversing a Geometric Segment



An exception is raised if *geom_segment* is invalid.

Examples

The following example reverses the geometric segment representing route 1. (This example uses the definitions from the example in Section E.5.)

```
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(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 14, 0, 8, 10, 5, 12, 10, 9, 12, 4, 15, 8, 4, 19, 2, 4, 25, 2, 2, 27))
```

SDO_LRS.SCALE_GEOM_SEGMENT

Format

```
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.

Common uses of this function are:

- Shifting 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)
- Reversing 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)
- Scaling the measure information without performing a shift (for example, to change the measures from miles to kilometers)

The examples illustrate these uses.

An exception is raised if *geom_segment* is invalid or if *start_measure* or *end_measure* is out of range.

For more information about scaling a geometric segment, see Section E.4.6.

Examples

The following examples illustrate the common SCALE_GEOM_ELEMENT uses described in the Usage Notes. (These examples use the definitions from the example in Section E.5.)

```
-- 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))

-- Reverse direction (for example, to concatenate with another road)
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo,27, 0, 0)
    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,27,0,0)(SDO_GTYPE, SDO_SRI
```

```
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 14, 0, 8, 10, 5, 12, 10, 9, 12, 4, 15, 8, 4, 19, 2, 4, 25, 2, 2, 27))

-- "Convert" mile measures to kilometers (27 * 1.609 = 43.443)
SELECT     SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo,
                                      0, 43.443, 0)
    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,0,43.443,0)(SDO_GTYPE, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, 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.SPLIT_GEOM_SEGMENT

Format

```
SDO_LRS.SPLIT_GEOM_SEGMENT(
     geom_segment     IN MDSYS.SDO_GEOMETRY,
     dim_array        IN MDSYS.SDO_DIM_ARRAY,
     split_measure    IN NUMBER,
     geom_segment_1   OUT MDSYS.SDO_GEOMETRY,
     geom_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.

geom_segment_1
First geometric segment: from the start point of *geom_segment* to the split point.

geom_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.

For more information about splitting a geometric segment, see Section E.4.4.

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 E.5. The definitions of ***result_geom_1***, ***result_geom_2***, and ***result_geom_3*** are displayed in Example E–3.)

```
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,
    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.

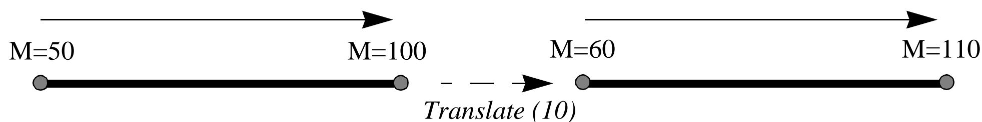
dim_array
Dimensional information array corresponding to *geom_segment*, usually selected from one of the xxx_SDO_GEOM_METADATA views.

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 9–2.

Figure 9–2 Translating a Geometric Segment



An exception is raised if ***geom_segment*** is invalid.

Examples

The following example translates (shifts) by 10 the geometric segment representing Route 1. (This example uses the definitions from the example in Section E.5.)

```
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.

Examples

The following example checks if the geometric segment representing Route 1 is valid. (This example uses the definitions from the example in Section E.5.)

```
SELECT  SDO_LRS.VALID_GEOM_SEGMENT(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.VALID_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
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).

Examples

The following example checks if point (9,3,NULL) is a valid LRS point.

```
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,
    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* is invalid.

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.)

```
SELECT  SDO_LRS.VALID_MEASURE(a.route_geometry, m.diminfo, 50)
  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_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO,50)
--------------------------------------------------------------------------------
FALSE
```

10 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 10–1.

Table 10–1 Migration Procedures

Procedure	Description
SDO_MIGRATE.FROM_815_TO_81X	Migrates data from Spatial release 8.1.5 to the current release.
SDO_MIGRATE.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 7.3.4 or Spatial Cartridge 8.0.4 to Oracle Spatial.
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.FROM_815_TO_81X

Format

SDO_MIGRATE.FROM_815_TO_81X(

tabname IN VARCHAR2);

Description

Migrates data from Spatial release 8.1.5 to the current release.

Parameters

tabname
Table with geometry objects.

Usage Notes

See Section B.5 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 release 8.1.6 format.

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.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.

Parameters

layer
Name of the layer to be migrated.

newtabname
Name of the new table to which you are migrating the data.

gidcolumn
The 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.

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.
- This procedure converts from Spatial Data Option release 7.3.4 or from Spatial Cartridge releases 8.0.4 and 8.0.5.
- 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.
- This procedure assigns SDO_GTYPE values of *d*001, *d*002, or *d*003. (See Table 2–1 in Section 2.2.1 for the SDO_GTYPE values.) If the data has multiple points, line strings, or disjoint polygons, then you should update the SDO_GTYPE values to *d*005, *d*006, or *d*007, respectively, after migration.

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 into existing rows:

```
execute sdo_migrate.to_81x('raptor', 'raptor', 'sdo_gid', 'feature',
  'point', 'update');
```

Update nonpoint data into existing rows:

```
execute sdo_migrate.to_81x('BTU', 'BTU', 'sdo_gid', 'feature',
```

```
'notpoint', 'update');
```

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.

11

Tuning Functions and Procedures

This chapter contains descriptions of the tuning functions and procedures shown in Table 11–1.

Table 11–1 Tuning Functions and Procedures

Function/Procedure	Description
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.

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 and Subqueries" section of the Oracle8i 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,
    maxtiles          IN INTEGER,
    type_of_estimate  IN VARCHAR2 := 'AVG_GID_EXTENT'
    ) 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.

maxtiles
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 -- 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
- Section A.1.2, "Understanding the Tiling Level"
- Section A.1.5, "Visualizing the Spatial Index (Drawing Tiles)"

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 and Subqueries" section of the Oracle8i 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 and Subqueries" section of the Oracle8i 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.

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_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 -- 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%)
.
```

Part II

Relational Model

Oracle Spatial supports two models for representing geometries: relational and object-relational. The two models are mutually exclusive. See Section 1.2 for a description of how to choose the model best suited for your application.

Note: In the next release of Oracle Spatial, the relational model will no longer be documented in this guide, but will instead be documented in a separate document whose title and location will be announced.

Spatial applications using the relational model will continue to work; however, if you are not already using the object-relational model for all Spatial applications, you are encouraged to do so before the next release.

This part of the User's Guide and Reference contains the following chapters, describing the relational model:

- Chapter 12, "The Relational Schema"
- Chapter 13, "Loading Spatial Data (Relational Model)"
- Chapter 14, "Querying Spatial Data (Relational Model)"
- Chapter 15, "Administrative Functions and Procedures for Relational Model"
- Chapter 16, "Tuning Functions and Procedures for Relational Model"
- Chapter 17, "Geometry Functions and Procedures for Relational Model"
- Chapter 18, "Window Functions and Procedures for Relational Model"

12

The Relational Schema

Before release 8.1, the Spatial product always used four database tables to store and index spatial data. This database structure is modeled on the first of three OpenGIS Features for SQL Implementation options, namely, using numeric SQL types for geometry storage. This schema is different from the spatial objects model introduced in Spatial release 8.1 and described in Part II of this guide. However, there are still some advantages, discussed in Section 1.2.2, to using this model.

Note: In the next release of Oracle Spatial, the relational model will no longer be documented in this guide, but will instead be documented in a separate document whose title and location will be announced.

Spatial applications using the relational model will continue to work; however, if you are not already using the object-relational model for all Spatial applications, you are encouraged to do so before the next release.

12.1 Database Structures for the Relational Implementation

The four tables, used to store and index geometries, are collectively referred to as a **layer**. A template SQL script is provided to facilitate the creation of these tables. See Section A.2.2 for details.

Table 12–1 through Table 12–4 describe the schema of a Spatial layer.

Table 12–1 <layername>_SDOLAYER Table

SDO_ORDCNT	SDO_LEVEL	SDO_NUMTILES	SDO_MAXLEVEL[1]	SDO_COORDSYS[2]
<number>	<number>	<number>	<number>	<varchar>

[1] SDO_MAXLEVEL is an optional column.
[2] SDO_COORDSYS is an optional column.

Table 12–2 <layername>_SDODIM Table or View

SDO_DIMNUM	SDO_LB	SDO_UB	SDO_TOLERANCE	SDO_DIMNAME
<number>	<number>	<number>	<number>	<varchar>

Table 12–3 <layername>_SDOGEOM Table or View

SDO_GID	SDO_ESEQ	SDO_ETYPE	SDO_SEQ	SDO_X1	SDO_Y1	...	SDO_Xn	SDO_Yn
<number>	<number>	<number>	<number>	<number>	<number>	...	<number>	<number>

Table 12–4 <layername>_SDOINDEX Table

SDO_GID	SDO_CODE	SDO_MAXCODE [1]	SDO_GROUPCODE [2]	SDO_META
<number>	<raw>	<raw>	<raw>	<raw>

[1] SDO_MAXCODE is not required for the recommended fixed-size tile indexing algorithm.
[2] SDO_GROUPCODE is not required for the recommended fixed-size tile indexing algorithm.

The columns of each table are defined as follows:

<layername>_SDOLAYER

- **SDO_ORDCNT**: The SDO_ORDCNT column is the total number of ordinates per row in the <layername>_SDOGEOM table. That is, the total number of data value columns, and not the number of points or coordinates. SDO_ORDCNT should not be multiplied by the total number of dimensions per coordinate as it is already a total.
- **SDO_LEVEL**: The SDO_LEVEL column stores the number of times the tiles that interact with a geometry should be decomposed. It is the termination criterion for fixed tiling. Use the SDO_TUNE.ESTIMATE_TILING_LEVEL procedure to determine an appropriate tiling level for your data.

- **SDO_NUMTILES**: The SDO_NUMTILES column is the number of variable-sized tiles used to tessellate each object in the <layername>_ SDOGEOM table. This column must be set to NULL when using fixed-size tiles.
- **SDO_MAXLEVEL**: The SDO_MAXLEVEL column indicates the maximum level to which a variable-sized tile can be decomposed. It is the termination criterion for the variable component of hybrid tiling.
- **SDO_COORDSYS**: The SDO_COORDSYS column is optional; you can indicate the name of the coordinate system, using a standard such as POSC or OGIS.

<layername>_SDODIM

- **SDO_DIMNUM**: The SDO_DIMNUM column is the dimension to which this row refers, starting with 1 and increasing.
- **SDO_LB**: The SDO_LB column is the lower bound of the ordinate in this dimension. For example, if the dimension is latitude, the lower bound would be -90.
- **SDO_UB**: The SDO_UB column is the upper bound of the ordinate in this dimension. For example, if the dimension is latitude, the upper bound would be 90.
- **SDO_TOLERANCE**: The SDO_TOLERANCE column is the distance two points can be apart and still be considered the same due to round-off errors. Tolerance must be greater than zero. If you want zero tolerance, enter a number such as 0.00005, where the number of zeros to the right of the decimal point matches the precision of your data. The extra 5 will round up to the last decimal digit.
- **SDO_DIMNAME**: The SDO_DIMNAME column is used for the usual name applied to this dimension, such as *longitude*, *latitude*, *X,* or *Y.*

<layername>_SDOGEOM

- **SDO_GID**: The SDO_GID column is a unique numeric identifier for each geometry in a layer.
- **SDO_ESEQ**: The SDO_ESEQ column enumerates each element in a geometry, that is, the **E**lement **SEQ**uence number.
- **SDO_ETYPE**: The SDO_ETYPE column is the geometric primitive type of the element. For this release of Spatial, the valid values are SDO_

GEOM.POINT_TYPE, SDO_GEOM.LINESTRING_TYPE, or SDO_GEOM.POLYGON_TYPE (ETYPE values 1, 2, and 3, respectively). The SDO_ETYPE values 4 and 5, supported in the object-relational schema, are not supported. Setting the ETYPE to zero indicates that this element should be ignored. See Section A.1.9 for information on ETYPE=0.

- **SDO_SEQ**: The SDO_SEQ column records the order (the **SEQ**uence number) of each row of data making up the element.
- **SDO_X1**: The X value of the first coordinate.
- **SDO_Y1**: The Y value of the first coordinate.
- **SDO_Xn**: The X value of the *nth* coordinate.
- **SDO_Yn**: The Y value of the *nth* coordinate.

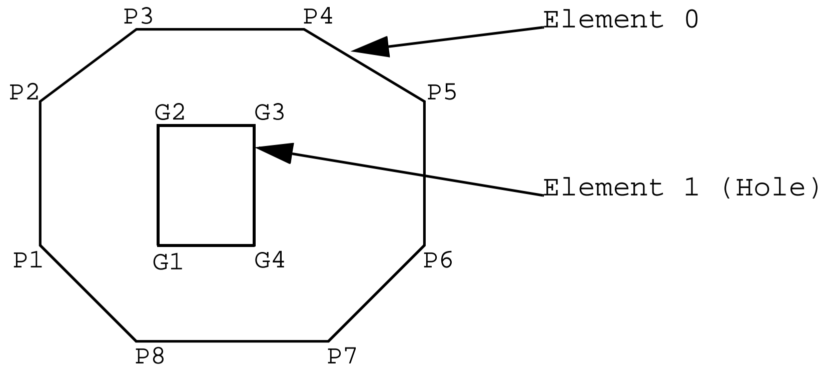
<layername>_SDOINDEX

- **SDO_GID**: The SDO_GID column is a unique numeric identifier for each geometry in a layer. This can be thought of as a foreign key back to the <layername>_SDOGEOM table.
- **SDO_CODE**: The SDO_CODE column is the bit-interleaved ID of a tile that covers SDO_GID. This column should be created as type RAW(255).
- **SDO_MAXCODE**: The SDO_MAXCODE column describes a variable-sized logical tile, which is the smallest tile (with the longest tile ID) in the current quadrant. The SDO_MAXCODE column is SDO_CODE padded out one place farther than the longest allowable code name for this index. This column is not used for fixed-size tiles.
- **SDO_GROUPCODE**: The SDO_GROUPCODE column is a prefix of SDO_CODE. It represents a variable-sized tile at level <layername>_SDOLAYER.SDO_LEVEL that contains or is equal to the tile represented by SDO_CODE. This column is not used for fixed-size tiles.
- **SDO_META**: The SDO_META column is not required for spatial queries. It provides information necessary to find the bounds of a tile. See Section A.1.5 for one possible use of this column.

Spatial provides stored procedures that assume the existence of the layer schema as described in this section. While layer tables may contain additional columns, they are required to contain at least the columns described in this section with the same column names and data types.

Figure 12–1 illustrates how a geometry is stored in the database using Spatial and the OGIS V1 schema model. The geometry to be stored is a complex polygon with a hole in it.

Figure 12–1 Complex Polygon



<layername>_SDOLAYER

SDO_ORDCNT (number)
4

<layername>_SDODIM

SDO_ DIMNUM (number)	SDO_LB (number)	SDO_UB (number)	SDO_ TOLERANCE (number)	SDO_ DIMNAME (varchar)
1	0	100	.05	X axis
2	0	100	.05	Y axis

<layername>_SDOGEOM

SDO_GID (number)	SDO_ESEQ (number)	SDO_ETYPE (number)	SDO_SEQ (number)	SDO_X1 (number)	SDO_Y1 (number)	SDO_X2 (number)	SDO_Y2 (number)
1013	0	3	0	P1(X)	P1(Y)	P2(X)	P2(Y)
1013	0	3	1	P2(X)	P2(Y)	P3(X)	P3(Y)
1013	0	3	2	P3(X)	P3(Y)	P4(X)	P4(Y)
1013	0	3	3	P4(X)	P4(Y)	P5(X)	P5(Y)
1013	0	3	4	P5(X)	P5(Y)	P6(X)	P6(Y)
1013	0	3	5	P6(X)	P6(Y)	P7(X)	P7(Y)
1013	0	3	6	P7(X)	P7(Y)	P8(X)	P8(Y)
1013	0	3	7	P8(X)	P8(Y)	P1(X)	P1(Y)
1013	1	3	0	G1(X)	G1(Y)	G2(X)	G2(Y)
1013	1	3	1	G2(X)	G2(Y)	G3(X)	G3(Y)
1013	1	3	2	G3(X)	G3(Y)	G4(X)	G4(Y)
1013	1	3	3	G4(X)	G4(Y)	G1(X)	G1(Y)

In this example, the <layername>_SDOGEOM table is shown as an 8-column table with 4 ordinates per row. In actual usage, Spatial supports *n*-wide[1] tables. The coordinates for the outer polygon in this example could have been loaded into a single row containing values for coordinates P1 to P8, and then repeating P1 to close the polygon. The coordinates would be stored in the SDO_X1 and SDO_Y1 through SDO_X9 and SDO_Y9 columns.

The data in the <layername>_SDOINDEX table is described in further detail in Section 1.7. The SDOINDEX table contains entries of the form [SDO_GID, SDO_CODE] where each SDO_CODE represents a tile that interacts with a geometry identified by SDO_GID. For a given SDO_GID value, there may be one or more SDO_CODE values. Each SDO_CODE value may be associated with one or more SDO_GID values.

[1] A <layername>_SDOGEOM table can have up to 1000 columns. The maximum number of data columns is 1000, minus 4 for the other required spatial columns, and minus any other user-defined columns. For polygons and line strings, storing 16 to 20 ordinates per row is suggested for performance reasons, but not required. The objective is to minimize the number of null values stored in the <layername>_SDOGEOM table.

13

Loading Spatial Data (Relational Model)

This chapter describes how to load spatial data into a database, including storing the data in a table and creating a spatial index for it. This chapter refers to the relational Spatial model only.

13.1 Load Model

There are two steps involved in loading raw data into a spatial database such that it can be queried efficiently:

1. Loading the data into spatial tables
2. Creating or updating the index on the spatial tables

Table 13–1 through Table 13–4 show the format of the tables or views needed to store and index spatial data. Note that these tables show the relational schema.

Table 13–1 <layername>_SDOLAYER Table

SDO_ORDCNT	SDO_LEVEL	SDO_NUMTILES	SDO_MAXLEVEL	SDO_COORDSYS
<number>	<number>	<number>	<number>	<varchar>

Table 13–2 <layername>_SDODIM Table or View

SDO_DIMNUM	SDO_LB	SDO_UB	SDO_TOLERANCE	SDO_DIMNAME
<number>	<number>	<number>	<number>	<varchar>

Table 13–3 <layername>_SDOGEOM Table or View

SDO_GID	SDO_ESEQ	SDO_ETYPE	SDO_SEQ	SDO_X1	SDO_Y1	...	SDO_Xn	SDO_Yn
<number>	<number>	<number>	<number>	<number>	<number>	...	<number>	<number>

Table 13–4 <layername>_SDOINDEX Table

SDO_GID	SDO_CODE	SDO_MAXCODE	SDO_GROUPCODE	SDO_META
<number>	<raw>	<raw>	<raw>	<raw>

13.2 Load Process

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 SQL*Loader to load the data.

- Transactional inserts

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

13.2.1 Bulk Loading

Bulk loading can be used to import large amounts of legacy or ASCII data into a spatial database. Bulk loading is accomplished using SQL*Loader.

Example 13–1 shows the format of the raw data and control file that would be required to load the data into the SDOGEOM table with the layer name ROADS. You can choose any format of ASCII data as long you can write a SQL*Loader control file to load that data into the tables.

Assume that the ASCII data consists of a file with delimited columns and separate rows fixed by the limits of the table, with the format shown in Example 13–1:

Example 13–1 Raw Data Format

```
geometry rows:    GID, ESEQ, ETYPE, SEQ, LON1, LAT1, LON2, LAT2
```

The coordinates in the geometry rows represent the end points of line segments, which taken together, represent a polygon. Example 13–2 shows the control file for loading the data into the geometry table.

Example 13–2 Control File to Load Data into the Geometry Table

```
LOAD DATA INFILE *
INTO TABLE ROADS_SDOGEOM
FIELDS TERMINATED BY WHITESPACE TRAILING NULLCOLS
(SDO_GID INTEGER EXTERNAL,
SDO_ESEQ INTEGER EXTERNAL,
SDO_ETYPE INTEGER EXTERNAL,
SDO_SEQ INTEGER EXTERNAL,
SDO_X1 FLOAT EXTERNAL,
SDO_Y1 FLOAT EXTERNAL,
SDO_X2 FLOAT EXTERNAL,
SDO_Y2 FLOAT EXTERNAL)

BEGINDATA
1 0 3 0 -122.401200   37.805200 -122.401900   37.805200
1 0 3 1 -122.401900   37.805200 -122.402400   37.805500
1 0 3 2 -122.402400   37.805500 -122.403100   37.806000
1 0 3 3 -122.403100   37.806000 -122.404400   37.806800
1 0 3 4 -122.404400   37.806800 -122.401200   37.805200
1 1 3 0 -122.405900   37.806600 -122.407549   37.806394
1 1 3 1 -122.407549   37.806394 -122.408300   37.806300
1 1 3 2 -122.408300   37.806300 -122.409100   37.806200
1 1 3 3 -122.409100   37.806200 -122.405900   37.806600
2 0 2 0 -122.410800   37.806000 -122.412300   37.805800
2 0 2 1 -122.412300   37.805800 -122.414100   37.805600
2 0 2 2 -122.414100   37.805600 -122.412300   37.805800
2 0 2 3 -122.412300   37.805800 -122.410800   37.806000
3 0 1 0 -122.567474   38.643564
3 0 1 1 -126.345345   39.345345
```

Be sure that table ROADS_SDOGEOM exists in the schema before attempting the load operation.

In Example 13–3, the data resides in a single flat file and the data set consists of point, line string, and polygon data. The data uses fixed-position columns and overloaded table rows.

Example 13–3 Raw Data Format

```
SDO_GID  SDO_ESEQ  SDO_ETYPE  SDO_SEQ  SDO_X1  SDO_Y1  SDO_X2  SDO_Y2
```

The corresponding control file for this format of input data is shown in Example 13–4.

Example 13–4 Control File to Load from a Single Flat File

```
LOAD DATA INFILE *
INTO TABLE NEW_SDOGEOM
(SDO_GID POSITION (1:5) INTEGER EXTERNAL,
SDO_ESEQ POSITION (7:10) INTEGER EXTERNAL,
SDO_ETYPE POSITION (12:15) INTEGER EXTERNAL,
SDO_SEQ POSITION (17:21) INTEGER EXTERNAL,
SDO_X1 POSITION (23:35) FLOAT EXTERNAL,
SDO_Y1 POSITION (37:48) FLOAT EXTERNAL,
SDO_X2 POSITION (50:62) FLOAT EXTERNAL,
SDO_Y2 POSITION (64:75) FLOAT EXTERNAL)

BEGINDATA
1     0    3    0    -122.401200   37.805200   -122.401900   37.805200
1     0    3    1    -122.401900   37.805200   -122.402400   37.805500
1     0    3    2    -122.402400   37.805500   -122.403100   37.806000
1     0    3    3    -122.403100   37.806000   -122.404400   37.806800
1     0    3    4    -122.404400   37.806800   -122.401200   37.805200
1     1    3    0    -122.405900   37.806600   -122.407549   37.806394
1     1    3    1    -122.407549   37.806394   -122.408300   37.806300
1     1    3    2    -122.408300   37.806300   -122.409100   37.806200
1     1    3    3    -122.409100   37.806200   -122.405900   37.806600
2     0    2    0    -122.410800   37.806000   -122.412300   37.805800
2     0    2    1    -122.412300   37.805800   -122.414100   37.805600
2     0    2    2    -122.414100   37.805600   -122.412300   37.805800
2     0    2    3    -122.412300   37.805800   -122.410800   37.806000
3     0    1    0    -122.567474   38.643564
3     0    1    1    -126.345345   39.345345
```

13.2.2 Transactional Insert Using SQL

Spatial uses standard Oracle8*i* tables that can be accessed or loaded with standard SQL syntax. Example 13–5 loads data for a geometry (GID 17) consisting of a polygon with four sides that contains both a hole and a point. Notice that the first coordinate of the polygon (5, 20) is repeated at the end to close the polygon.

Example 13–5 Transactional Insert

```
INSERT INTO SAMPLE_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
```

```
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (17, 0, 3, 0, 5, 20, 5, 30, 10, 30, 10, 20, 5, 20);

   -- hole
INSERT INTO SAMPLE_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (17, 1, 3, 0, 8, 21, 8, 24, 9, 24, 9, 21, 8, 21);

   -- point
INSERT INTO SAMPLE_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1)
  VALUES (17, 2, 1, 0, 9, 29);
```

The SQL INSERT statement inserts one row of data per call. In Example 13–5, the table had enough columns to store the polygon in a single row. However, if your table had fewer columns (or your polygon had more points), you would have to perform multiple inserts to match the table structure; the data would not wrap automatically to the next row. To load a large geometry, repeat the SDO_GID, SDO_ESEQ, and SDO_ETYPE, and increment the SDO_SEQ for each line as shown in Example 13–6.

Example 13–6 Transactional Insert for a Large Geometry

```
INSERT INTO SAMPLE2_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (18, 0, 3, 0, 1, 15, 1, 16, 2, 17, 3, 17, 4, 18);

INSERT INTO SAMPLE2_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (18, 0, 3, 1, 4, 18, 5, 18, 6, 19, 7, 18, 6, 17);

INSERT INTO SAMPLE2_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (18, 0, 3, 2, 6, 17, 7, 16, 7, 15, 6, 14, 7, 13);

INSERT INTO SAMPLE2_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3,
                              SDO_Y3, SDO_X4, SDO_Y4, SDO_X5, SDO_Y5)
  VALUES (18, 0, 3, 3, 7, 13, 6, 12, 5, 13, 4, 13, 3, 14);
```

```
INSERT INTO SAMPLE2_SDOGEOM (SDO_GID, SDO_ESEQ, SDO_ETYPE, SDO_SEQ,
                              SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3, SDO_
Y3)
  VALUES (18, 0, 3, 4, 3, 14, 2, 14, 1, 15);
```

13.3 Index Creation

Once data has been loaded into the spatial tables through either bulk or transactional loading, a spatial index needs to be created on the tables for efficient access to the data.

Create an Oracle table called <layername>_SDOINDEX as follows:

```
SQL>  create table <layername>_SDOINDEX
   2  (
   3    SDO_GID number,
   4    SDO_CODE raw(255)
   5    );
```

For a bulk load, you can call the SDO_ADMIN.POPULATE_INDEX procedure once to tessellate the geometry table and add the generated tiles to the spatial index table. The argument to this procedure is simply the name of the layer. The level to which the geometry should be tessellated and whether to use the fixed or the hybrid indexing technique is determined by values in the <layername>_ SDOLAYER table.

If data is updated in or deleted from a specific geometry table, you can call the SDO_ADMIN.UPDATE_INDEX procedure to update the index for one SDO_GID. The arguments to this procedure are the name of the layer and the SDO_GID of the designated geometry.

See Chapter 15 for a complete description of the SDO_ADMIN.POPULATE_INDEX and SDO_ADMIN.UPDATE_INDEX procedures.

13.3.1 Choosing a Tessellation Algorithm

Spatial provides two methods for spatial indexing, fixed and hybrid. Fixed indexing is recommended for the relational Spatial model.

Which tessellation algorithm is used by the SDO_ADMIN.POPULATE_INDEX and SDO_ADMIN.UPDATE_INDEX procedures is determined by the values of the

SDO_LEVEL and SDO_NUMTILES columns in the <layername>_SDOLAYER table as shown in Table 13–5.

Table 13–5 Choosing a Tessellation Algorithm

SDO_LEVEL	SDO_NUMTILES	Action
NULL	NULL	Error.
>= 1	NULL	Fixed indexing with fixed-size tiles (recommended).
>= 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 tiles to generate per geometry.
NULL	>= 1	Not supported.

13.3.2 Spatial Indexing with Fixed-Size Tiles

Fixed-size cover tiles are recommended for indexing a geometry stored using the relational model.

The fixed-size tile algorithm is expressed as a level referring to the number of tessellations performed. To use fixed-size tile indexing, set the SDO_NUMTILES column in the <layername>_SDOLAYER table to NULL and the SDO_LEVEL column to the desired tiling level. The relationship between the tiling level and the resulting size of the tiles is dependent on the domain of the layer.

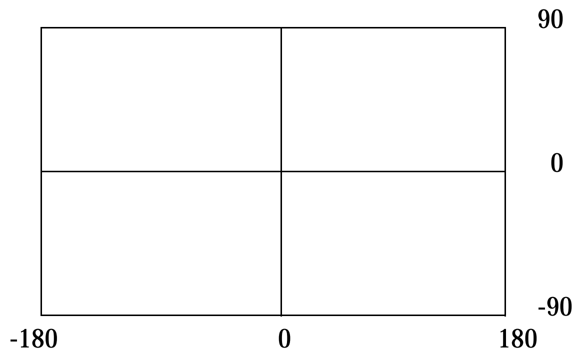
The domain used for indexing is defined by the upper and lower boundaries of each dimension stored in the <layername>_SDODIM table. A typical domain in a GIS application could be -90 to 90 degrees for latitude, and -180 to 180 degrees for longitude,[1] as represented in Figure 13–1.

[1] The transference of the domain onto a sphere or Mercator projection is left to GIS (or other) application programmers. Spatial treats the domain as a conventional X by Y rectangle.

Figure 13–1 Sample GIS Domain

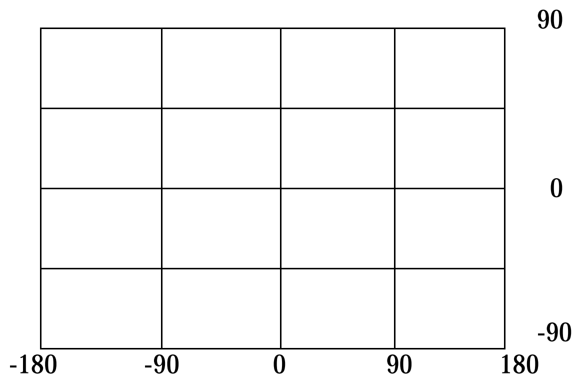
If the SDO_LEVEL column is set to 1, then the tiles created by the indexing mechanism are the same size as the tiles at the first level of tessellation. Each tile would be 180 degrees by 90 degrees as shown in Figure 13–2.

Figure 13–2 Fixed-Size Tiling at Level 1



The formula for the number of fixed-size tiles is 4^n where *n* is the number of tessellations stored in the SDO_LEVEL column. Figure 13–3 shows fixed-size tiling at level 2. In this figure, each tile is 90 degrees by 45 degrees.

Figure 13–3 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 13–3 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
```

Thus, at level 2 the tiles are 90x45 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. The SDO_TUNE.ESTIMATE_TILING_LEVEL function can be used to determine an appropriate level for indexing with fixed-size tiles. See Chapter 16 for a description of this procedure.

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 any database design, is essential in a spatial database where the data often represents actual physical locations.

In the following example, assume that data has been loaded into a layer called ROADS, and you want to create a spatial index on that data. This is accomplished by first creating a table ROADS_SDOINDEX and invoking the following procedure:

```
sdo_admin.populate_index('ROADS');
```

The value in the SDO_LEVEL column of the ROADS_SDOLAYER table can be used as a tuning parameter while tessellating objects. Increasing the level increases the number of tiles to provide a more precise fit of the tiles over the object. See the description of the ESTIMATE_TILING_LEVEL function in Chapter 16 for information on estimating the tiling level in several different ways.

After the SDO_ADMIN.POPULATE_INDEX procedure has been called to fill the spatial index, you should also create a concatenated index using the SDO_CODE and SDO_GID columns. The concatenated index helps the join to the <layername>_SDOGEOM table during a query. The SDO_GID values from the primary filter will come from the index instead of from the table.

If a geometry with an SDO_GID 5944 has been added to the spatial tables, update the index with the following procedure:

```
sdo_admin.update_index('ROADS', 5944);
```

Like the CREATE INDEX statement in SQL, the SDO_ADMIN.POPULATE_INDEX procedure performs an implicit commit operation. The SDO_ADMIN.UPDATE_INDEX procedure, however, does not. Therefore, SDO_ADMIN.UPDATE_INDEX transactions can be rolled back.

The SDO_ADMIN.POPULATE_INDEX procedure operates as a single transaction. To reduce the amount of rollback space required to execute this procedure, you can write a routine that loops and calls the SDO_ADMIN.UPDATE_INDEX procedure. See Section A.2.1 for more information.

13.3.3 Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles

This section describes a variation on the linear quadtree (Morton encoding) scheme that uses both fixed-size and variable-sized tiles as a spatial indexing mechanism. The terms *hybrid indexing*, *hybrid tiling*, and *hybrid tessellation* will be used interchangeably in this section. Spatial indexing with purely variable-sized tiles is not recommended for production systems and is not supported in this release.

To use hybrid tiling, the SDO_LEVEL and SDO_NUMTILES columns in the <layername>_SDOLAYER table must contain valid values. That is, both SDO_LEVEL and SDO_NUMTILES must be greater than one.

The SDO_NUMTILES column determines the number of tiles that will be used to cover a geometry being indexed. Typically, this value is small, such as 4 or 8 tiles. However, the larger the number of tiles, the better the tiles will fit the geometry being covered. This increases the selectivity of the primary filter, but also increases the number of index entries per geometry. (See Section 14.3.2 and Section 14.3.3 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-based spatial entities such as county or state boundaries.

The SDO_LEVEL column determines the size of the fixed tiles used in hybrid indexing. Setting the proper SDO_LEVEL value may appear more like art than science. Performing some simple data analysis and testing, however, puts the process back in the realm of science. One approach would be 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. This and other techniques are described in Appendix A.

Assume that the ROADS layer has already been loaded. Furthermore, assume that there is one row with valid values for the ROADS_SDOLAYER.SDO_LEVEL and ROADS_SDOLAYER.SDO_NUMTILES columns. To create the spatial index on ROADS, first create a table ROADS_SDOINDEX with appropriate columns:

```
SQL>  create table ROADS_SDOINDEX
   2  (
   3    SDO_GID number,
   4    SDO_CODE raw(255),
   5    SDO_GROUPCODE raw(255),
   6    SDO_MAXCODE raw(20),
   7    SDO_META raw(255),
   8    );
```

Then, invoke `SDO_ADMIN.POPULATE_INDEX('ROADS')` to build the spatial index.

After the SDO_ADMIN.POPULATE_INDEX procedure has been called to fill the spatial index, you should also create a concatenated index on the SDO_CODE and SDO_GID columns. The concatenated index helps the join to the <layername>_SDOGEOM table during a query. The SDO_GID values from the primary filter will come from the index instead of from the table.

If a geometry with an SDO_GID 5944 has been added to the spatial tables, update the index with the following procedure:

```
sdo_admin.update_index('ROADS', 5944);
```

Like the CREATE INDEX statement in SQL, the SDO_ADMIN.POPULATE_INDEX procedure performs an implicit commit operation. The SDO_ADMIN.UPDATE_INDEX procedure, however, does not. Therefore, SDO_ADMIN.UPDATE_INDEX transactions can be rolled back.

The SDO_ADMIN.POPULATE_INDEX procedure operates as a single transaction. To reduce the amount of rollback space required to execute this procedure, you can write a routine that loops and calls SDO_ADMIN.UPDATE_INDEX. See Section A.2.1 for more information.

14

Querying Spatial Data (Relational Model)

This chapter describes how the structures of a Spatial layer are used to resolve spatial queries and spatial joins. For the sake of clarity, the examples all use fixed tiling. This chapter refers to the relational Spatial model only.

14.1 Query Model

Spatial uses a two-tier query model to resolve spatial queries and spatial joins. A **two-tier query** means that two distinct operations are performed to resolve queries. The output of both operations yields the exact result set.

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

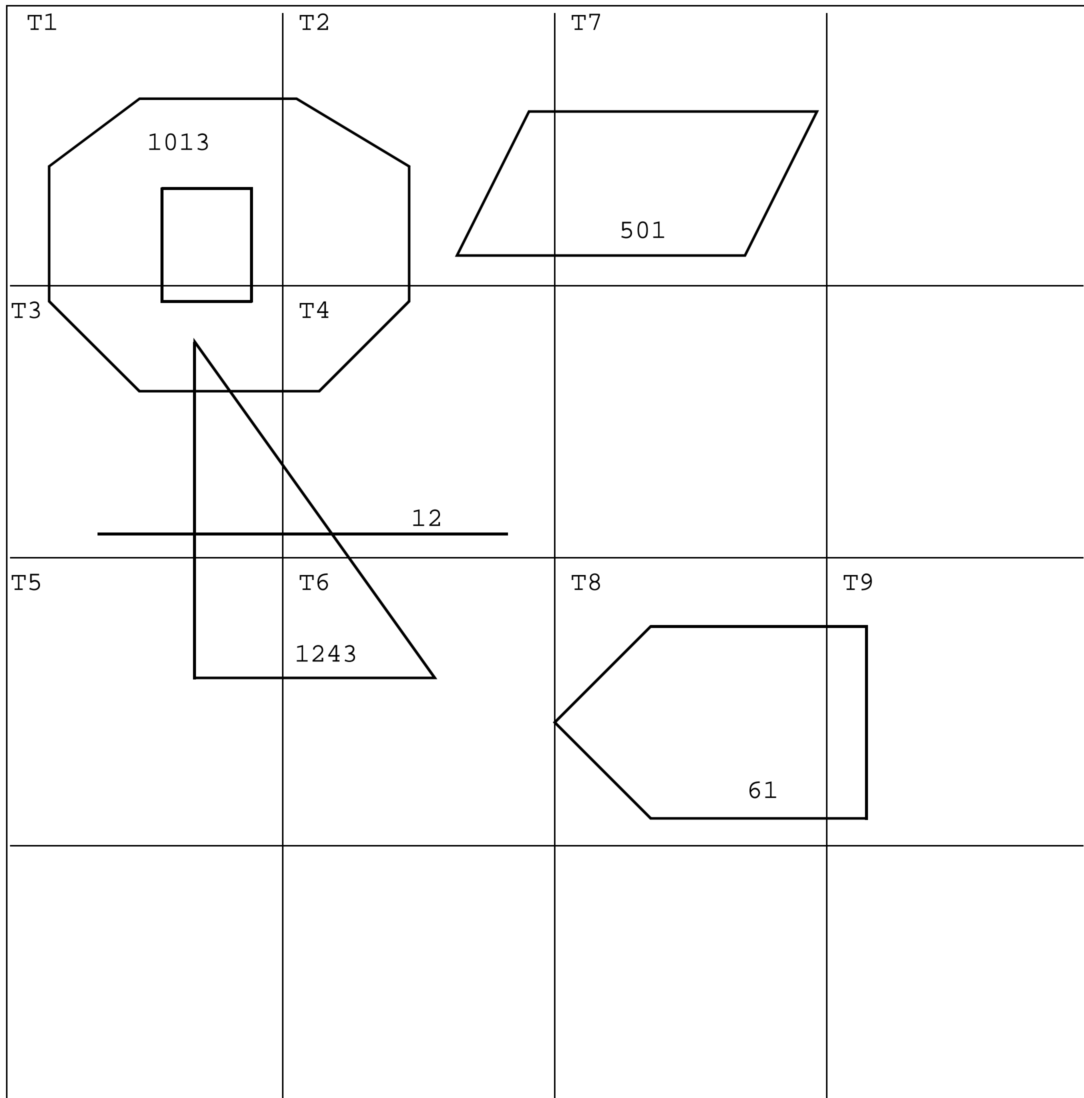
- The **primary filter** permits fast selection of a small number of candidate records to pass along to the secondary filter. The primary filter uses approximations in order to reduce computational complexity and is considered a lower-cost filter.
- The **secondary filter** applies exact computational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

14.2 Spatial Index Data Structures

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

Consider the following layer containing several objects in Figure 14–1. Each object is labeled with its SDO_GID. The relevant tiles are labeled with T*n*.

Figure 14–1 Tessellated Layer with Multiple Objects



The Spatial layer tables would have the following information stored in them for these geometries, as shown in Table 14–1, Table 14–2, and Table 14–3.

Table 14–1 <layername>_SDOLAYER Table

SDO_ORDCNT (number)	SDO_LEVEL (number)	SDO_NUMTILES (number)
4	2	NULL

Table 14–2 <layername>_SDOGEOM Table or View

SDO_GID (number)	SDO_ESEQ (number)	SDO_ETYPE (number)	SDO_SEQ (number)	SDO_X1 (number)	SDO_Y1 (number)	SDO_X2 (number)	SDO_Y2 (number)
1013	0	3	0	P1(X)	P1(Y)	P2(X)	P2(Y)
1013	0	3	1	P2(X)	P2(Y)	P3(X)	P3(Y)
1013	0	3	2	P3(X)	P3(Y)	P4(X)	P4(Y)
1013	0	3	3	P4(X)	P4(Y)	P5(X)	P5(Y)
1013	0	3	4	P5(X)	P5(Y)	P6(X)	P6(Y)
1013	0	3	5	P6(X)	P6(Y)	P7(X)	P7(Y)
1013	0	3	6	P7(X)	P7(Y)	P8(X)	P8(Y)
1013	0	3	7	P8(X)	P8(Y)	P1(X)	P1(Y)
1013	1	3	0	G1(X)	G1(Y)	G2(X)	G2(Y)
1013	1	3	1	G2(X)	G2(Y)	G3(X)	G3(Y)
1013	1	3	2	G3(X)	G3(Y)	G4(X)	G4(Y)
1013	1	3	3	G4(X)	G4(Y)	G1(X)	G1(Y)
501	0	3	0	A1(X)	A1(Y)	A2(X)	A2(Y)
501	0	3	1	A2(X)	A2(Y)	A3(X)	A3(Y)
501	0	3	2	A3(X)	A3(Y)	A4(X)	A4(Y)
501	0	3	3	A4(X)	A4(Y)	A1(X)	A1(Y)
1243	0	3	0	B1(X)	B1(Y)	B2(X)	B2(Y)
1243	0	3	1	B2(X)	B2(Y)	B3(X)	B3(Y)
1243	0	3	2	B3(X)	B3(Y)	B1(X)	B1(Y)
12	0	2	0	D1(X)	D1(Y)	D2(X)	D2(Y)
61	0	3	0	C1(X)	C1(Y)	C2(X)	C2(Y)
61	0	3	1	C2(X)	C2(Y)	C3(X)	C3(Y)
61	0	3	2	C3(X)	C3(Y)	C4(X)	C4(Y)
61	0	3	3	C4(X)	C4(Y)	C5(X)	C5(Y)
61	0	3	4	C5(X)	C5(Y)	C1(X)	C1(Y)

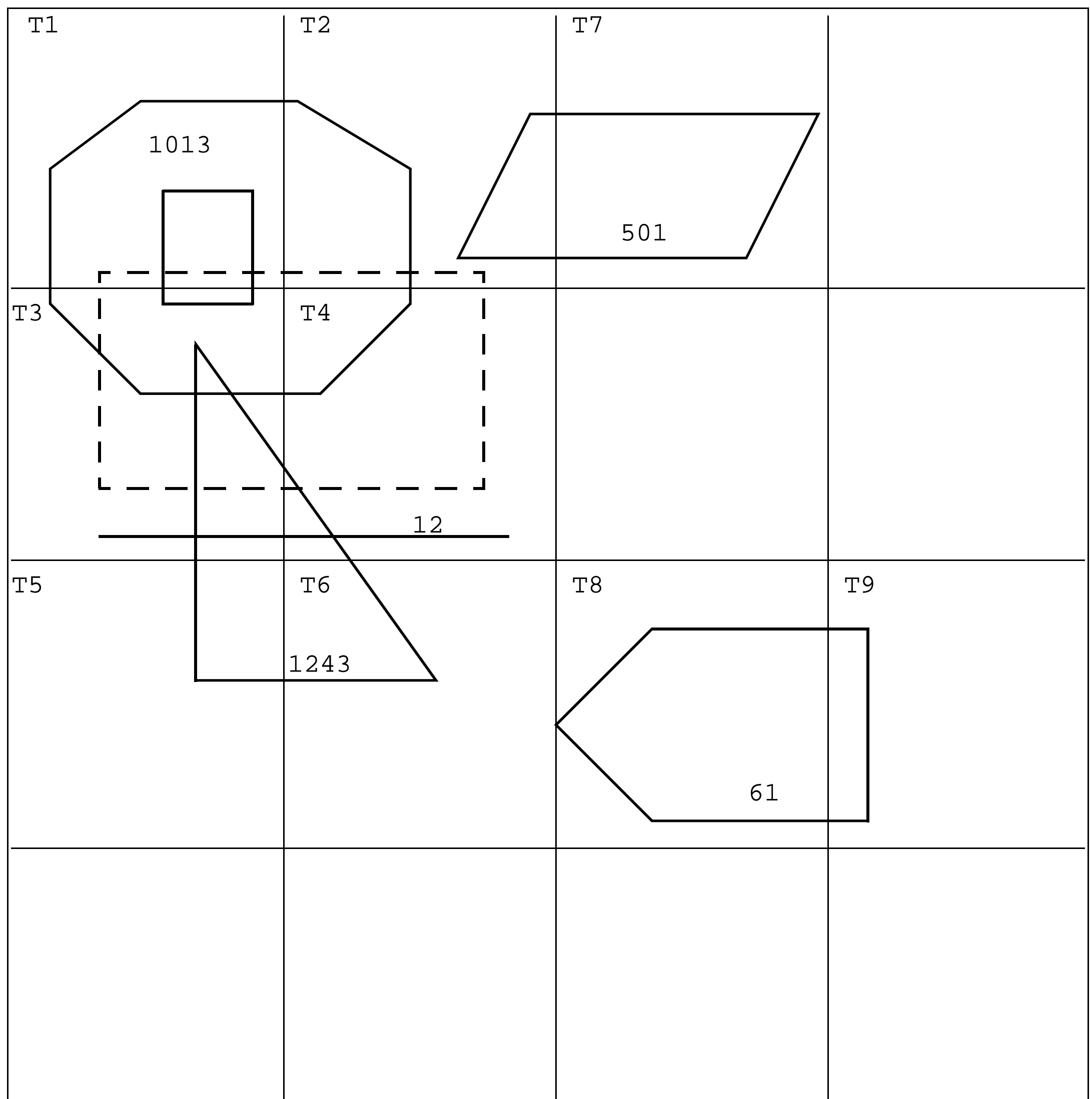
Table 14–3 <layername>_SDOINDEX Table

SDO_GID (number)	SDO_CODE (raw)
1013	T1
1013	T2
1013	T3
1013	T4
501	T2
501	T7
1243	T3
1243	T4
1243	T5
1243	T6
12	T3
12	T4
61	T8
61	T9

14.3 Spatial Query

A typical spatial query is to request all objects that lie within a defined fence or window. A **query window** is shown in Figure 14–2 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 and indexed before it is used.

Figure 14–2 Tessellated Layer with a Query Window



14.3.1 Dynamic Query Window

If a query window does not already exist in the database, you must first insert it

and create an index for it. Because not all Oracle users necessarily have insert privileges, Spatial includes the SDO_WINDOW PL*SQL package. See Chapter 18, "Window Functions and Procedures for Relational Model", for more information.

The SDO_WINDOW package is not automatically installed when you install Spatial. This allows a DBA to control the schema under which this package operates. Choose an Oracle user who has insert privilege and compile the SDO_WINDOW package under that user. For example, you could choose the MDSYS Oracle user:

```
sqlplus mdsys/password
SQL> @$ORACLE_HOME/md/admin/sdowin.sql
SQL> @$ORACLE_HOME/md/admin/prvtwin.plb
```

After compiling, the routines are available for use. When you call a routine in this package, and the routine performs an INSERT operation, the insertion will occur under the MDSYS schema. Note that it is not a requirement to use the MDSYS account. You can select any Oracle user with insert privileges.

If you need to perform other INSERT, UPDATE, or DELETE operations, and you cannot guarantee that the user of your application has those privileges, you can write your own PL*SQL package similar to the SDO_WINDOW package. You will have to compile your package under a user with the required database privileges.

14.3.2 Primary Filter Query

To resolve the window query shown in Figure 14–2, build a layer for the query fence if it is not already defined:

```
SQL> EXECUTE MDSYS.SDO_WINDOW.CREATE_WINDOW_LAYER (fencelayer, DIMNUM1, LB1,
UB1, TOLERANCE1, DIMNAME1, DIMNUM2, LB2, UB2, TOLERANCE2, DIMNAME2);
```

Next, insert the ordinates for the query fence into the layer tables:

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE(MDSYS.SDO_WINDOW.BUILD_WINDOW_FIXED(comp_user,
     fencelayer, SDO_ETYPE, TILE_SIZE, X1,Y1, X2,Y2, X3,Y3, X4,Y4, X1,Y1));
```

Query SDO_LEVEL from the <fencelayer>_SDOLAYER table to pass the correct TILE_SIZE to the SDO_WINDOW.BUILD_WINDOW_FIXED procedure.

Now you can construct a query that joins the index of the query window to the appropriate layer index and determines all elements that have these tiles in common. The following SQL query form is used:

```
SELECT DISTINCT A.SDO_GID
```

```
FROM <layer1>_SDOINDEX A, <fencelayer>_SDOINDEX B
WHERE A.SDO_CODE = B.SDO_CODE
  AND B.SDO_GID = {GID returned from SDO_WINDOW.BUILD_WINDOW_FIXED};
```

The result set of this query is the primary filter set. In this case, the result set is:

```
{ 1013,501,1243,12 }
```

14.3.3 Secondary Filter Query

The secondary filter performs exact geometry calculations of the tiles selected by the primary filter. The following example shows the primary and secondary filters:

```
SELECT SDO_X1, SDO_Y1, SDO_X2, SDO_Y2, SDO_X3, SDO_Y3, SDO_X4, SDO_Y4
FROM <layer1>_SDOGEOM,
(
SELECT SDO_GID GID1
   FROM (
          SELECT DISTINCT A.SDO_GID
             FROM <layer1>_SDOINDEX A,
                  <fencelayer>_SDOINDEX B
             WHERE A.SDO_CODE = B.SDO_CODE
              AND B.SDO_GID = {GID returned from SDO_WINDOW.BUILD_WINDOW_FIXED}

         )
   WHERE SDO_GEOM.RELATE('<layer1>', SDO_GID, 'ANYINTERACT', '<fence>', 1) =
'TRUE'
   )
WHERE SDO_GID = GID1;
```

This query would return all the geometry IDs that lie within or overlap the window. In this example, the results of the secondary filter would be:

```
{1243,1013}
```

The example in this section uses the SDO_GEOM.RELATE secondary filter. For better performance, use the overloaded version of this function, which explicitly lists the coordinates of the query window whenever possible. See Chapter 17 for details on using this function.

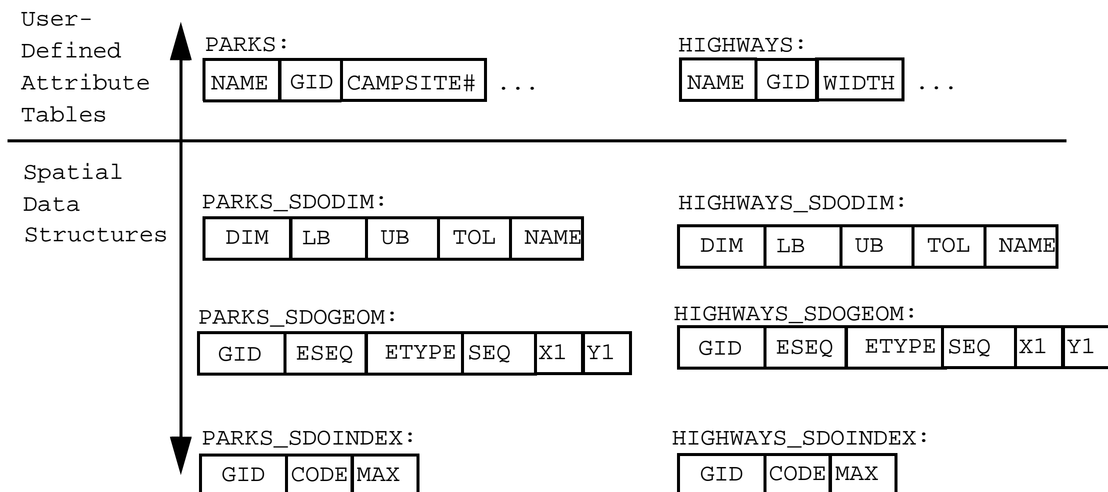
14.4 Spatial Join

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 between two layers; specifically, two <layername>_SDOINDEX tables are joined.

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

This query could be resolved by joining a layer that stores national park geometries with one that stores highway geometries. Figure 14–3 illustrates how the join would be accomplished for this example using the OGIS V1 schema model.

Figure 14–3 Spatial Join of Two Layers



The primary filter would identify pairs of park GIDs and highway GIDs that cross in the index. The query that performs the primary filter join (assuming fixed-size tile indexing) is as follows:

```
SELECT DISTINCT A.SDO_GID,B.SDO_GID
     FROM PARKS_SDOINDEX A, HIGHWAYS_SDOINDEX B
     WHERE A.SDO_CODE = B.SDO_CODE
```

The result set of the primary filter must be passed through the secondary filter to get the exact set of parks/highways GID pairs that cross. The full query is shown in the following example:

```
SELECT DISTINCT GID_B
  FROM (
         SELECT /*+ index(a PARKS_SDOINDEX_SDO_CODE_INDEX)
                    index(b HIGHWAYS_SDOINDEX_SDO_CODE_INDEX)
                   use_nl(a b)
                   no_merge */
            DISTINCT A.SDO_GID GID_A, B.SDO_CODE GID_B
         FROM PARKS_SDOINDEX A, HIGHWAYS_SDOINDEX B
        WHERE A.SDO_CODE = B.SDO_CODE
        )
  WHERE SDO_GEOM.RELATE ( 'PARKS',  GID_A,
                          'ANYINTERACT',
                          'HIGHWAYS', GID_B)   <> 'FALSE';
```

Primary Filter

Secondary Filter

Suppose the original query had asked, *Which 4-lane highways cross national parks?* You could modify the preceding SQL statement to join back to the HIGHWAYS table where HIGHWAYS.WIDTH=4. This combination of spatial and relational attributes in a single query is one of the essential reasons for using Spatial.

15

Administrative Functions and Procedures for Relational Model

The SDO_ADMIN procedures create and maintain spatial structures in the database, and they are used to perform the following tasks:

- Tessellate entries in a geometry table and place them in a spatial index table
- Verify spatial index information

This chapter contains descriptions of the administrative functions and procedures used for working with spatially indexed geometric data. This chapter refers to the relational Spatial model only.

Table 15–1 lists the administrative functions and procedures for working with spatially indexed geometry-based data.

Table 15–1 Administrative Procedures for Spatially Indexed Data

Procedure or Function	Description
SDO_ADMIN.POPULATE_INDEX	Generates a spatial index for the geometry table using either a set number of tiles or a fixed-size tile.
SDO_ADMIN.POPULATE_INDEX_FIXED	Generates a spatial index using fixed-size tiles. This is a deprecated procedure.
SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS	Generates a spatial index using fixed-size tiles for a layer composed solely of point data.
SDO_ADMIN.SDO_CODE_SIZE	Determines the required sizes for SDO_CODE and SDO_MAXCODE.
SDO_ADMIN.SDO_VERSION	Returns the release number of the installed version of Spatial.

Table 15–1 Administrative Procedures for Spatially Indexed Data (Cont.)

Procedure or Function	Description
SDO_ADMIN.UPDATE_INDEX	Updates the spatial index based on changes to the geometry table.
SDO_ADMIN.UPDATE_INDEX_FIXED	Updates a spatial index with fixed-size tiles. This is a deprecated procedure.
SDO_ADMIN.VERIFY_LAYER	Checks for the existence of geometry and spatial index tables.

SDO_ADMIN.POPULATE_INDEX

Format

SDO_ADMIN.POPULATE_INDEX (*layername*)

Description

Tessellates a list of geometric objects created by selecting all the entries in the geometry table that do not have corresponding entries in the spatial index table.

This procedure can generate either fixed-size or variable-sized tiles depending on values stored in the <layername>_SDOLAYER table.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the names of the geometry and spatial index tables. Data type is VARCHAR2.

Usage Notes

Consider the following when using this procedure:

- The <layername>_SDOINDEX table must be created before calling this procedure. Use the SQL CREATE TABLE statement to create the spatial index table.
- For performance reasons, create an index on the SDO_GID column in the <layername>_SDOGEOM table before calling this procedure.
- This procedure generates either fixed-size or variable-sized tiles depending on values stored in the <layername>_SDOLAYER table as follows:

SDO_LEVEL	SDO_NUMTILES	Action
NULL	NULL	Error.
>= 1	NULL	Perform fixed-size tiling (recommended for relational model).

SDO_LEVEL	SDO_NUMTILES	Action
>= 1	>= 1	Perform hybrid indexing. The SDO_LEVEL column defines the partition bucket size. The SDO_NUMTILES column defines the number of tiles to generate per geometry. Note: Hybrid indexing is for experimentation purposes only in the relational model.
NULL	>= 1	Not supported.

- If the <layername>_SDOINDEX table is empty, the procedure selects all the geometries in the geometry table and generates index entries for them. If the index table is not empty, the procedure determines which entries in the geometry table do not have index entries, and generates them.
- SDO_ADMIN.POPULATE_INDEX behaves similarly to the CREATE INDEX statement in SQL. An implicit commit operation is executed after the procedure is called.
- SDO_ADMIN.POPULATE_INDEX operates as a single transaction. To reduce the number of rollback operations required to execute this procedure, you can write a routine that loops and calls SDO_ADMIN.UPDATE_INDEX repeatedly. See Section A.2.1 for more information.

Example 15–1 tessellates all the geometric objects in the LAYER1_SDOGEOM table and adds the generated tiles to the LAYER1_SDOINDEX table.

Example 15–1 Populate an Index

```
SQL> EXECUTE SDO_ADMIN.POPULATE_INDEX('layer1');
SQL> COMMIT;
```

Related Topics

- SDO_ADMIN.UPDATE_INDEX

SDO_ADMIN.POPULATE_INDEX_FIXED

Format

SDO_ADMIN.POPULATE_INDEX_FIXED (*layername, tile_size, [synch_flag,] [sdo_tile_flag,] [sdo_maxcode_flag]*)

Description

Provided for compatibility with Spatial Cartridge release 8.0.3 tables, but it has been replaced by enhanced features in the SDO_ADMIN.POPULATE_INDEX procedure, in order to support schema changes as shown in Section 12.1.

This procedure tessellates a list of geometric objects created by selecting all the entries in the geometry table that do not have corresponding entries in the spatial index table. This procedure can also tessellate all the geometric objects in a geometry table or view and add the tiles to the spatial index table.

Use this procedure to tessellate the geometries into fixed-size tiles.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
tile_size	Specifies the number of tessellations required to achieve the desired tile size (see the Usage Notes). Data type is INTEGER.
synch_flag	Specifies whether to tessellate every geometric object in the geometry table, or only those that do not have corresponding entries in the spatial index table. If TRUE, only those geometric objects in the geometry table that do not have any corresponding tiles in the spatial index table are tessellated. If FALSE, all the geometric objects in the geometry table are tessellated, and new tiles are added to the spatial index table. Data type is BOOLEAN. Default = TRUE.
sdo_tile_flag	For internal use only. Not supported in this release. Default = FALSE.
sdo_ maxcode_flag	Specifies whether or not the SDO_MAXCODE column is populated. If TRUE, SDO_MAXCODE is populated. If FALSE, the column is not populated. Set this flag to FALSE for the recommended fixed-size tiling. Data type is BOOLEAN. Default = TRUE.

Usage Notes

> **Note:** This procedure is likely to be removed in a future release of Spatial.

Consider the following when using this procedure:

- The SQL CREATE TABLE statement is used to create the spatial index table, <layername>_SDOINDEX, before calling this procedure.
- The layer is tessellated into equal-sized tiles based on the number passed in the tile_size parameter. The value of *tile_size* specifies how many times to tessellate the layer. See Section 13.3.2.
- For performance reasons, set the synch_flag to FALSE when the spatial index table contains zero rows.
- For performance reasons, create an index on the SDO_GID column in the <layername>_SDOGEOM table before calling this procedure.
- SDO_ADMIN.POPULATE_INDEX_FIXED behaves similarly to the CREATE INDEX statement in SQL. An implicit commit operation is executed after the procedure is called.
- SDO_ADMIN.POPULATE_INDEX_FIXED operates as a single transaction. To reduce the number of rollback operations required to execute this procedure, you can write a routine that loops and calls SDO_ADMIN.UPDATE_INDEX_FIXED repeatedly. See Section A.2.1 for more information.

Example 15–2 tessellates all the geometric objects in the LAYER1_SDOGEOM table using fixed-size tiles, and adds the generated tiles to the LAYER1_SDOINDEX table.

Example 15–2 Populate an Index with Fixed-Size Tiles

```
SQL> EXECUTE SDO_ADMIN.POPULATE_INDEX_FIXED('layer1',4,FALSE,FALSE,FALSE);
```

Related Topics

- SDO_ADMIN.UPDATE_INDEX_FIXED
- SDO_TUNE.ESTIMATE_TILING_LEVEL

SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS

Format

SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS (*layername, sdo_tile_flag, commit_count*)

Description

Builds an index with fixed-size tiles for a geometry layer consisting solely of point data. Because a point is indexed using a single tile, special optimizations are possible.

Keywords and Parameters

layername	Specifies the name of the data set layer. Data type is VARCHAR2.
sdo_tile_flag	Specifies whether or not to generate the SDO_TILE column. Data type is BOOLEAN. Default = FALSE.
commit_count	Specifies how many points to index before updating and committing the data. Data type is NUMBER. Default = 50.

Usage Notes

Consider the following when using this procedure:

- The <layername>_SDOLAYER, <layername>_SDOGEOM, and <layername>_SDODIM tables must be populated before calling this procedure.
- The <layername>_SDOINDEX table must be created before calling this procedure. Use the SQL CREATE TABLE statement to create the spatial index table.
- For performance reasons, create an index on the SDO_GID column in the <layername>_SDOGEOM table before calling this procedure.
- If the <layername>_SDOINDEX table is empty, the procedure selects all the geometries in the geometry table and generates index entries for them. If the index table is not empty, the procedure determines which entries in the geometry table do not have index entries, and generates them.

Example 15–3 tessellates all the points in the LAYER1_SDOGEOM table and adds the generated tiles to the LAYER1_SDOINDEX table. This example commits after every 100 points.

Example 15–3 Populate an Index with Fixed-Size Tiles Based on Point Data

```
SQL> EXECUTE SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS('layer1', FALSE, 100 );
```

Related Topics

- SDO_ADMIN.UPDATE_INDEX

SDO_ADMIN.SDO_CODE_SIZE

Format

SDO_ADMIN.SDO_CODE_SIZE (*layername*)

Description

Determines the size that the SDO_MAXCODE column should be in the <layername>_SDOINDEX table.

Keywords and Parameters

layername	Specifies the name of the data set layer. Data type is VARCHAR2.

Returns

This function returns the required size in bytes for the SDO_MAXCODE column. Data type is INTEGER.

Usage Notes

The SDO_CODE column is used to store the bit-interleaved cell ID of a tile that covers a geometry. The SDO_MAXCODE column is SDO_CODE padded out one place farther than the longest allowable code name for the index. Both columns are defined as RAW data types, with a maximum of 255 bytes. Use the SDO_ADMIN. SDO_CODE_SIZE function to fine-tune the size of the column.

Always declare the SDO_CODE column to raw(255).

Related Topics

None.

SDO_ADMIN.SDO_VERSION

Format

SDO_ADMIN.SDO_VERSION

Description

Returns the current installed version of Spatial.

Keywords and Parameters

None.

Returns

This function returns a string describing the version of Spatial installed on the local system.
Data type is VARCHAR2.

Usage Notes

The following version strings can be returned:

8.0.5.0.0
8.1.0.0.0
8.1.3.0.0
8.1.5.0.0
8.1.6.0.0

This information is useful when migrating data between systems, or when upgrading. See Appendix B for more information about migration.

Related Topics

None.

SDO_ADMIN.UPDATE_INDEX

Format

SDO_ADMIN.UPDATE_INDEX (*layername, GID*)

Description

Tessellates a single geometric object in a geometry table or view and adds the tiles to the spatial index table. If the object already exists and has index entries, those entries are deleted and replaced by the newly generated tiles.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry table. Data type is VARCHAR2.
GID	Specifies the geometric object identifier. Data type is NUMBER.

Usage Notes

Consider the following when using this procedure:

- The <layername>_SDOINDEX table must exist before calling this procedure. Use the SQL CREATE TABLE statement to create the spatial index table.
- For performance reasons, create an index on the SDO_GID column in the <layername>_SDOGEOM table before calling this procedure.
- The values of the SDO_LEVEL and SDO_NUMTILES columns must be set in the <layername>_SDOLAYER table before calling this procedure. This procedure generates either fixed-size or hybrid tiles depending on values stored in the <layername>_SDOLAYER table as follows:

SDO_LEVEL	SDO_NUMTILES	Action
NULL	NULL	Error.
>= 1	NULL	Perform indexing with fixed-size tiles (recommended for the relational model).

SDO_LEVEL	SDO_NUMTILES	Action
>= 1	>= 1	Perform hybrid indexing. The SDO_LEVEL column defines the partition bucket size. The SDO_NUMTILES column defines the number of tiles to generate per geometry. Note: Hybrid indexing is for experimentation purposes only in the relational model.
NULL	>= 1	Not supported.

- SDO_ADMIN.UPDATE_INDEX does not perform an implicit commit operation after it executes; therefore, the transaction can be rolled back.

Example 15–4 tessellates the polygon for geometry 25 and adds the generated tiles to the LAYER1_SDOINDEX table.

Example 15–4 Update an Index

```
SQL> EXECUTE SDO_ADMIN.UPDATE_INDEX('layer1', 25);
SQL> COMMIT;
```

Related Topics

- SDO_ADMIN.POPULATE_INDEX

SDO_ADMIN.UPDATE_INDEX_FIXED

Format

SDO_ADMIN.UPDATE_INDEX_FIXED (*layername, GID, tile_size, [replace_flag,] [sdo_tile_flag] [sdo_maxcode_flag]*)

Description

Provided for compatibility with Spatial Cartridge release 8.0.3 tables, but it has been replaced by enhanced features in the SDO_ADMIN.UPDATE_INDEX procedure to support schema changes as shown in Section 12.1.

This procedure tessellates a single geometric object in a geometry table or view and adds the fixed-sized tiles to the spatial index table. By default, these tiles will replace existing ones for the same geometry; or optionally, existing tiles can be left alone.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry table. Data type is VARCHAR2.
GID	Specifies the geometric object identifier. Data type is NUMBER.
tile_size	Specifies the number of tessellations required to achieve the desired fixed-size tiles. Each tessellation subdivides the tiles from the previous level into four smaller tiles. Data type is INTEGER.
replace_flag	Specifies whether or not to delete tiles for the GID before adding new ones. If TRUE, tiles are deleted before new entries are inserted into the spatial index table. If FALSE, new tiles are added to the spatial index table. Data type is BOOLEAN. Default = TRUE.
sdo_tile_flag	For internal use only. Not supported in this release. Data type is BOOLEAN. Default = FALSE.
sdo_maxcode_flag	Specifies whether or not the SDO_MAXCODE column is populated. If TRUE, SDO_MAXCODE is populated. If FALSE, the column is not populated. Set this flag to FALSE for the recommended indexing with fixed-size tiles. Data type is BOOLEAN. Default = TRUE.

Usage Notes

> **Note:** This procedure is likely to be removed in a future release of Spatial.

Consider the following when using this procedure:

- For performance reasons, set the replace_flag to FALSE when the spatial index table contains no entries for the specified GID.
- For performance reasons, create an index on the SDO_GID column in the <layername>_SDOGEOM table before calling this procedure.
- SDO_ADMIN.UPDATE_INDEX_FIXED does not perform an implicit commit operation after it executes; therefore, this transaction can be rolled back.

Example 15–5 tessellates the polygon for geometry 25 and adds the generated tiles to the LAYER1_SDOINDEX table.

Example 15–5 Update an Index with Fixed-Size Tiles

```
SQL> EXECUTE SDO_ADMIN.UPDATE_INDEX_FIXED ('layer1',25,4,FALSE,FALSE,FALSE);
```

Related Topics

- SDO_ADMIN.POPULATE_INDEX_FIXED
- SDO_TUNE.ESTIMATE_TILING_LEVEL

SDO_ADMIN.VERIFY_LAYER

Format

SDO_ADMIN.VERIFY_LAYER (*layername,[maxtiles]*)

Description

Checks for the existence of the geometry and spatial index tables.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
maxtiles	For internal use only. Not supported in this release.

Usage Notes

If this procedure does not find the geometry and spatial index tables, it generates the following error: SDO 13113 (Oracle table does not exist.)

Example 15–6 verifies the LAYER1 data set layer.

Example 15–6 Verify a Layer

```
SQL> EXECUTE SDO_ADMIN.VERIFY_LAYER('layer1');
```

Related Topics

None.

16

Tuning Functions and Procedures for Relational Model

This chapter contains descriptions of the tuning functions and procedures shown in Table 16–1. This chapter refers to the relational Spatial model only.

Table 16–1 Tuning Functions and Procedures

Function/Procedure	Description
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.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.

SDO_TUNE.AVERAGE_MBR

Format

SDO_TUNE.AVERAGE_MBR (*layername, width, height*)

Description

Calculates the average minimum bounding rectangle (MBR) for all geometries in a layer.

Keywords and Parameters

layername	Specifies the name of the data set layer to examine. Data type is VARCHAR2.
width	Returns the width of the average MBR. Data type is OUT NUMBER.
height	Returns the height of the average MBR. Data type is OUT NUMBER.

Returns

This function returns the width and height of the average MBR for all geometries in a layer.

Usage Notes

This function calculates the average MBR by keeping track of the maximum and minimum X and Y values for all geometries in a layer.

SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE

Format

SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE (*layername, sample_ratio, tiling_level, num_tiles, window_layer, window_gid, tiling_time, filter_time, query_time*)

Description

Estimates the spatial index performance such as query selectivity and window query time for a layer.

Keywords and Parameters

layername	Specifies the name of the data set layer to examine. Data type is VARCHAR2.
sample_ratio	Specifies the size ratio between the original layer and the sample layer to be generated. Data type is INTEGER. Default is 20.
tiling_level	Specifies the spatial index level at which the layer is to be tessellated. Data type is INTEGER.
num_tiles	Specifies the number of tiles for variable or hybrid tessellation. Data type is INTEGER.
window_layer	Specifies the name of the spatial layer in which the window geometry is stored. Data type is VARCHAR2.
window_gid	Specifies the window geometry ID. Data type is NUMBER.
tiling_time	Returns the estimated tiling time in seconds. Data type is OUT NUMBER.
filter_time	Returns the estimated spatial index filter time in seconds. Data type is OUT NUMBER.
query_time	Returns the estimated window query time in seconds. Data type is OUT NUMBER.

Returns

This function returns a number between 0.0 and 1.0 representing estimated spatial index selectivity. It also returns the estimated tiling time, filter time, and query time.

Data type for these variables is NUMBER.

Usage Notes

- A larger selectivity number indicates better selectivity. A selectivity of 0.0 indicates an error.
- A larger sample_ratio means faster but less accurate estimation.

SDO_TUNE.ESTIMATE_TILING_LEVEL

Format

SDO_TUNE.ESTIMATE_TILING_LEVEL *(layername, maxtiles, type_of_estimate)*

Description

Estimates the appropriate tiling level to use when indexing with fixed-size tiles.

Keywords and Parameters

layername
Specifies the name of the data set layer to examine.
Data type is VARCHAR2.

maxtiles
Specifies the maximum number of tiles that can be used to index the rectangle defined by the type_of_estimate parameter.
Data type is INTEGER.

type_of_estimate
Indicates by keyword one of three different models. Specify the type of estimate with one of the following keywords:

- LAYER_EXTENT -- Use the rectangle defined by your coordinate system.
- ALL_GID_EXTENT -- Use the minimum bounding rectangle that encompasses all the geometric objects within the layer. Recommended for most applications.
- AVG_GID_EXTENT -- Use a rectangle representing the average size of the individual geometries within the layer. This option performs the most extensive analysis of the three types.

Returns

This function returns an integer representing the level to use when creating a spatial index for the specified layer.

Usage Notes

The SDO_ADMIN.POPULATE_INDEX and SDO_ADMIN.UPDATE_INDEX procedures are used to create or update the spatial index using fixed-size or hybrid indexing. Store the value returned by the SDO_TUNE.ESTIMATE_TILING_LEVEL function in the SDO_LEVEL column of the <layername>_SDOLAYER table before building the spatial index.

The maxtiles parameter specifies the maximum number of tiles that should be used to define a grid covering the rectangular extent of interest. This extent could be:

- Defined in the <layername>_SDODIM table, which defines the bounds of the coordinate system
- Defined by the minimum and maximum coordinates for the given data set (as returned by the SDO_TUNE.EXTENT_OF procedure)
- Defined by computing the average bounds of the objects in the <layername>_SDOGEOM table

The code shown in Example 16–1 generates a recommendation based on the extent of the defined coordinate system (-90 to +90 latitude and -180 to +180 longitude). This example returns a level whose tiles are not smaller than one-degree cells.

Example 16–1 Recommended Tile Level for One-Degree Latitude/Longitude Cells

```
set serveroutput on
declare
   tiling_level integer;
begin
   tiling_level := mdsys.sdo_tune.estimate_tiling_level('WORLD_CITIES',
           360*180, 'LAYER_EXTENT');
    dbms_output.put_line('VALUE is '|| tiling_level);
end;
```

For most applications, however, it is more effective to call the SDO_ TUNE.ESTIMATE_TILING_LEVEL function using the ALL_GID_EXTENT estimate type with a *maxtiles* value of 10,000. In Example 16–2, assume the data set consists of block groups for San Francisco and that the <layername>_SDODIM table defines the extent to be one that covers all of California. Because the data set is localized to a small subregion of this extent, ALL_GID_EXTENT is the appropriate estimate type. The recommended tiling level in this case will be such that at most, 10,000 tiles will be required to completely cover the extent of San Francisco block groups.

Example 16–2 Recommended Tile Level Based on the GIDs of All Geometries

```
set serveroutput on
declare
   tiling_level integer;
begin
   tiling_level:=   mdsys.sdo_tune.estimate_tiling_level('SF_BLOCK_GROUPS',
         10000, 'ALL_GID_EXTENT');
   dbms_output.put_line('VALUE is' ,|| tiling_level);
```

```
end;
```

The third type of estimate helps determine the tiling level that should be used such that on average, the maxtiles parameter defines the number of tiles to cover the extent of a single geometry in the layer. This estimate type requires the most computation of the three because the bounding rectangle of every geometry is used in calculating the average extent. In Example 16–3, eight tiles on average are used to cover any block group in San Francisco.

Example 16–3 Recommended Tile Level Based on Average Extent of All Geometries

```
set serveroutput on
declare
   tiling_level integer;
begin
   tiling_level := mdsys.sdo_tune.estimate_tiling_level('SF_BLOCK_GROUPS', 8,
         'AVG_GID_EXTENT');
   dbms_output.put_line('Tiling level value is ' || tiling_level);
end;
```

Related Topics

- SDO_ADMIN.POPULATE_INDEX
- SDO_ADMIN.UPDATE_INDEX
- SDO_TUNE.EXTENT_OF
- Section A.1.2, "Understanding the Tiling Level"
- Section A.1.5, "Visualizing the Spatial Index (Drawing Tiles)"

SDO_TUNE.ESTIMATE_TILING_TIME

Format

SDO_TUNE.ESTIMATE_TILING_TIME (*layername, sample_ratio, tiling_level, num_tiles*)

Description

Returns the estimated time to tessellate a layer.

Keywords and Parameters

layername	Specifies the name of the data set layer to examine. Data type is VARCHAR2.
sample_ratio	Specifies the size ratio between the original layer and the sample layer to be generated. Data type is INTEGER. Default is 20.
tiling_level	Specifies the spatial index level at which the layer is to be tessellated. Data type is INTEGER.
num_tiles	Specifies the number of tiles for variable or hybrid tessellation. Data type is INTEGER.

Returns

This function returns the estimated tiling time in seconds. A return of 0 indicates an error.

Data type is NUMBER.

Usage Notes

None.

SDO_TUNE.EXTENT_OF

Format

SDO_TUNE.EXTENT_OF *(layername, min_X, max_X, min_Y, max_Y)*

Description

Determines the extent of all geometries in a layer.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
min_X	Minimum X value of the bounding rectangle. Data type is NUMBER.
max_X	Maximum X value of the bounding rectangle. Data type is NUMBER.
min_Y	Minimum Y value of the bounding rectangle. Data type is NUMBER.
max_Y	Maximum Y value of the bounding rectangle. Data type is NUMBER.

Returns

This function returns the coordinates of the minimum bounding rectangle for all geometric data in a layer.

Data type is NUMBER for the four return values.

Usage Notes

None.

Related Topics

- SDO_TUNE.ESTIMATE_TILING_LEVEL function

SDO_TUNE.HISTOGRAM_ANALYSIS

Format

SDO_TUNE.HISTOGRAM_ANALYSIS (*layername, result_table, type_of_histogram, max_value, intervals*)

Description

Generates statistical histograms based on a layer.

Keywords and Parameters

layername — Specifies the name of the spatial data set layer to examine.
Data type is VARCHAR2.

result_table — Specifies the name of the result table where the histogram will be stored.
Data type is VARCHAR2.

type_of_histogram — Specifies one of the following types of histograms:

- TILES_VS_LEVEL (default)
- GEOMS_VS_AREA
- GEOMS_VS_TILES
- GEOMS_VS_VERTICES

Data type is VARCHAR2.

max_value — Specifies the upper limit of the histogram.
Data type is NUMBER.

intervals — Specifies the number of intervals between 0 and max_value.
Data type is INTEGER.

Returns

This procedure populates the result table with statistical histograms for a spatial layer.

Usage Notes

You must create the result table before calling this procedure. The table has the following format:

```
CREATE TABLE histogram (value NUMBER, count NUMBER);
```

The following types of histograms are available:

TILES_VS_LEVEL	Provides the number of tiles at different spatial index levels. (Available only with hybrid indexes.) This histogram is used to evaluate the spatial index that is already built on the layer.
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_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_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.

SDO_TUNE.MIX_INFO

Format

SDO_TUNE.MIX_INFO (*layername*)

Description

Provides the number of geometries of each type stored in the layer.

Keywords and Parameters

layername	Specifies the name of the spatial data set layer to examine. Data type is VARCHAR2.

Returns

This function calculates geometry type information for the layer. It returns the number of geometries of different type, as well as the percentages of points, line strings, polygons, and complex geometries.

Usage Notes

None.

17

Geometry Functions and Procedures for Relational Model

This chapter contains descriptions of the geometric functions and procedures shown in Table 17–1. This chapter refers to the relational Spatial model only.

Table 17–1 Geometric Functions and Procedures

Function/Procedure	Description
SDO_GEOM.RELATE	Determines how two objects interact.
SDO_GEOM.VALIDATE_GEOMETRY	Determines if a geometry is valid.
SDO_GEOM.VALIDATE_LAYER	Determines if all geometries in a layer are valid.

SDO_GEOM.RELATE

Format

SDO_GEOM.RELATE (*layername1, SDO_GID1, mask, [layername2,] SDO_GID2*)

SDO_GEOM.RELATE (*layername1, SDO_GID1, mask, X_tolerance, Y_tolerance,*
SDO_ETYPE, num_ordinates, X_ordinate1, Y_ordinate1 [,...,Xn, Yn]
[,SDO_ETYPE, num_ordinates, X_ordinate1, Y_ordinate1 [,...,Xn, Yn]])

Description

Examines two geometry objects to determine their spatial relationship. This function is available in two forms. See the Usage Notes for more information.

Keywords and Parameters

layername1, layername2	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
SDO_GID1, SDO_GID2	Specifies the geometry object identifier. Data type is NUMBER.
mask	Specifies a list of relationships to check. See the list of keywords in the Usage Notes.
X_tolerance, *Y_tolerance*	Specifies the distance two points can be apart and still be considered the same due to rounding errors. Tolerance must be greater than zero. If you want zero tolerance, enter a number such as 0.000005, where the number of zeros to the right of the decimal point matches the precision of your data. Data type is NUMBER.
SDO_ETYPE	Specifies the type of geometry element. Data type is INTEGER, corresponding to the following constants: 1 SDO_GEOM.POINT_TYPE 2 SDO_GEOM.LINESTRING_TYPE 3 SDO_GEOM.POLYGON_TYPE

num_ordinates	Specifies the number of ordinates for this element. Data type is NUMBER.
X_ordinateN, *Y_ordinateN*	Specifies the X and Y values of a vertex (coordinate pair) in a geometry. Data type is NUMBER.

Returns

The SDO_GEOM.RELATE function can return three types of answers:

1. 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 function returns FALSE.
2. If you pass the DETERMINE keyword in the mask, the function returns the one relationship keyword that best matches the geometries. The DETERMINE keyword can only be used when SDO_GEOM.RELATE is in the SELECT clause of the SQL statement.
3. If you pass the ANYINTERACT keyword in the mask, the function returns TRUE if the two geometries are not disjoint.

The data type is VARCHAR2.

Usage Notes

Use the first form of the function to examine two stored geometric objects.

Use the second form of the function to compare a stored object against a user-defined object. You can specify up to 123 vertices for a single-element geometry. If the geometry has multiple elements, the total number of arguments passed, including SDO_ETYPE, num_ordinates, and the list of vertex coordinates, cannot exceed 255 values.

The following 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.

Mask values 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.

Related Topics

None.

SDO_GEOM.VALIDATE_GEOMETRY

Format

SDO_GEOM.VALIDATE_GEOMETRY *(layername,SDO_GID*)

Description

Provides a consistency check for valid geometry types. This function checks the representation of the geometry from the tables against the element definitions.

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
SDO_GID	Specifies the geometric object identifier. Data type is NUMBER.

Returns

This function returns one of the following:

- TRUE if the geometry is valid.
- FALSE if the geometry is invalid for some unknown reason.
- An Oracle error number indicating the problem with the geometry.

The data type is VARCHAR2.

Usage Notes

This function checks for the following:

- Polygons must have at least three points and must be closed.
- Line strings must have at least two points.
- When an SDO_ESEQ spans multiple rows, the last point of the previous row must be the first point on the next row.

Related Topics

None.

SDO_GEOM.VALIDATE_LAYER

Format

SDO_GEOM.VALIDATE_LAYER *(layername, result_table)*

Description

Examines a layer to determine if the stored geometries follow the defined rules for geometric objects.

Keywords and Parameters

layername	Specifies the name of the layer to examine. Data type is VARCHAR2.
result_table	Specifies the name of the result table. Data type is VARCHAR2.

Returns

This function fills the result table with validation results.

Usage Notes

Create an empty result table before calling this function. The format of the result table is: (sdo_gid number, result varchar2).

This function checks for the following:

- Polygons must have at least three points and must be closed.
- Line strings must have at least two points.
- When an SDO_ESEQ spans multiple rows, the last point of the previous row must be the first point on the next row.

Related Topics

None.

18

Window Functions and Procedures for Relational Model

If a query window does not already exist in the database, you must first insert it and create an index for it. The SDO_WINDOW functions and procedures are used to create temporary geometry objects to be used in comparisons with stored geometries. You can create query windows with any number of coordinates.

Because not all Oracle users may have insert privileges, the SDO_WINDOW package is not automatically installed when you install Spatial. This allows a DBA to control the schema under which these functions and procedures operate. Choose an Oracle user who has insert privilege and compile the SDO_WINDOW package under that user. For example, you could choose the MDSYS Oracle user:

```
% sqlplus mdsys/password
SQL> @$ORACLE_HOME/md/admin/sdowin.sql
SQL> @$ORACLE_HOME/md/admin/prvtwin.plb
```

This chapter contains descriptions of the window functions and procedures listed in Table 18–1. This chapter refers to the relational Spatial model only.

Table 18–1 Window Functions and Procedures

Function/Procedures	Description
SDO_WINDOW.BUILD_WINDOW	Builds a query window.
SDO_WINDOW.BUILD_WINDOW_FIXED	Builds a query window using fixed-size tiles.
SDO_WINDOW.CLEAN_WINDOW	Removes the tables used for a query window.
SDO_WINDOW.CLEANUP_GID	Removes the query window without removing the tables.

Table 18–1 Window Functions and Procedures

Function/Procedures	Description
SDO_WINDOW.CREATE_WINDOW_LAYER	Creates the tables needed for a query window layer.

SDO_WINDOW.BUILD_WINDOW

Format

SDO_WINDOW.BUILD_WINDOW(*comp_name*, *layername*, *SDO_ETYPE*, *SDO_NUMTILES*, *X1*, *Y1*, *[...Xn*, *Yn]*)

Description

Builds the window for the query and returns an SDO_GID that serves as a handle. The window is tessellated into hybrid tiles. Hybrid indexing is not recommended for the relational Spatial model.

Keywords and Parameters

comp_name

Specifies the name of the user who compiled this package. This user must have appropriate privileges to read and write into the database.
Data type is VARCHAR2.

layername

Specifies the name of the window layer into which the coordinates will be inserted.
Data type is VARCHAR2.

SDO_ETYPE

Specifies the type of geometry objects.
Data type is INTEGER, corresponding to the following constants:

1 or SDO_GEOM.POINT_TYPE

2 or SDO_GEOM.LINESTRING_TYPE

3 or SDO_GEOM.POLYGON_TYPE

SDO_NUMTILES

Value must be NULL for Spatial release 8.0.4 and later.
Data type is NUMBER.

X ordinateN,
Y ordinateN

Specifies the X and Y values of a vertex (coordinate pair) in a geometry.
Up to 125 pairs may be added in a single call.
Data type is NUMBER.

Returns

This function returns the SDO_GID of the new geometry.

Data type is NUMBER.

Usage Notes

This function inserts the coordinates into the <layername>_SDOGEOM table, tessellates the geometry (creates the index), and returns a unique SDO_GID corresponding to the geometry.

You do not need special privileges to execute this function. However, the user who compiles it does need appropriate privileges to read and write into the database.

When working with Spatial release 8.0.3 tables, the SDO_NUMTILES parameter indicates the number of tiles into which the window should be tessellated. For release 8.0.4 or later, the function reads that information automatically from the <layername>_SDOLAYER table.

Related Topics

SDO_WINDOW.BUILD_WINDOW_FIXED

SDO_WINDOW.BUILD_WINDOW_FIXED

Format

SDO_WINDOW.BUILD_WINDOW_FIXED (*comp_name, layername, SDO_ETYPE, SDO_TILESIZE, X1, Y1, [...Xn, Yn]*)

Description

Builds the window for the query and returns an SDO_GID that serves as a handle. The window is tessellated into fixed-size tiles.

Keywords and Parameters

comp_name	Specifies the name of the user who compiled this package. This user must have appropriate privileges to read and write into the database. Data type is VARCHAR2.
layername	Specifies the name of the window layer into which the coordinates will be inserted. Data type is VARCHAR2.
SDO_ETYPE	Specifies the type of geometry element. Data type is INTEGER, corresponding to the following constants: 1 or SDO_GEOM.POINT_TYPE 2 or SDO_GEOM.LINESTRING_TYPE 3 or SDO_GEOM.POLYGON_TYPE
SDO_TILESIZE	Specifies the number of tessellations required to achieve the desired fixed-size tiles. Data type is NUMBER.
X ordinateN, *Y ordinateN*	Specifies the X and Y values of a vertex (coordinate pair) in a geometry. Up to 125 pairs may be added in a single call. Data type is NUMBER.

Returns

This function returns the SDO_GID of the new geometry.

Data type is NUMBER.

Usage Notes

This function inserts the coordinates into the <layername>_SDOGEOM table, tessellates the geometry (creates the index), and returns a unique SDO_GID corresponding to the geometry.

You do not need special privileges to execute this function. However, the user who compiles it does need appropriate privileges to read and write into the database.

Query SDO_LEVEL from the <layername>_SDOLAYER table to pass the correct SDO_TILE_SIZE value to this function.

Related Topics

None.

SDO_WINDOW.CLEAN_WINDOW

Format

SDO_WINDOW.CLEAN_WINDOW (*layername*);

Description

Removes the four tables created in the layer for the query window.

Keywords and Parameters

layername	Specifies the name of the window layer that must be removed. Data type is VARCHAR2.

Usage Notes

Typically, you would build a layer once, and then build multiple windows and perform multiple queries using that layer. After finishing all queries, you can execute the SDO_WINDOW.CLEAN_WINDOW procedure to remove the tables.

Related Topics

SDO_WINDOW.CLEANUP_GID

SDO_WINDOW.CLEANUP_GID

Format

SDO_WINDOW.CLEANUP_GID (gid, *layer, do_commit*);

Description

Removes the query window from the layer tables.

Keywords and Parameters

gid	Specifies the geometric object identifier of the query window. Data type is NUMBER.
layer	Specifies the name of the window layer associated with the query window. Data type is VARCHAR2.
do_commit	Specifies whether a commit operation is performed (TRUE) or is not performed (FALSE, the default) after the cleanup. Data type is BOOLEAN.

Usage Notes

Typically, you would create a query layer once, and then build multiple query windows and perform multiple queries using that layer. The SDO_WINDOW.CLEANUP_GID procedure removes a single query window from the layer. Use this procedure to avoid the overhead of removing and re-creating the tables repeatedly.

After finishing all queries, you can execute the SDO_WINDOW.CLEAN_WINDOW procedure to remove the tables.

Related Topics

SDO_WINDOW.CLEAN_WINDOW

SDO_WINDOW.CREATE_WINDOW_LAYER

Format

SDO_WINDOW.CREATE_WINDOW_LAYER (*layername*, *SDO_LEVEL*, *SDO_NUMTILES*,
SDO_DIMNUM1, *SDO_LB1*, *SDO_UB1*, *SDO_TOLERANCE1*, *SDO_DIMNAME1*,
SDO_DIMNUM2, *SDO_LB2*, *SDO_UB2*, *SDO_TOLERANCE2*, *SDO_DIMNAME2*)

Description

Creates the necessary tables that constitute a layer used for defining a query window.

Keywords and Parameters

layername	Specifies the name of the window layer to be created. The layer name is used to construct the four tables associated with the layer. Data type is VARCHAR2.
SDO_LEVEL	Specifies the number of times the layer should be tessellated during the indexing phase. Data type is INTEGER.
SDO_NUMTILES	Specifies the number of tiles to generate during indexing. Data type is INTEGER.
SDO_DIMNUM1, *SDO_DIMNUM2*	Specifies the number of the dimension, starting with 1. Data type is NUMBER.
SDO_LB1, *SDO_UB1*, *SDO_LB2*, *SDO_UB2*	Specifies the lower and upper bounds of this dimension. Data type is NUMBER.
SDO_TOLERANCE1, *SDO_TOLERANCE2*	Specifies the allowable variance of ordinate values within each dimension. Data type is NUMBER.
SDO_DIMNAME1, *SDO_DIMNAME2*	Specifies the name of the dimension. Data type is VARCHAR2.

Usage Notes

Because the <layername>_SDODIM table is initialized with the dimension and the bound information, only those queries that are in the same dimension should be queried against this layer. If you wish to issue a query with respect to a different dimension, you must create a new layer.

Related Topics

None.

A

Tuning Tips and Sample SQL Scripts

This appendix provides supplemental information to help you set up, maintain, and tune a spatial database. The scripts and tuning suggestions provided are intended as guidelines that can be adapted to the specific needs of your database.

For a description of the Oracle Spatial models (object-relational and relational) and guidelines for choosing a model, see Section 1.2.

A.1 Tuning Tips

The following information can be used as a guideline for tuning a spatial database. Unless otherwise specified, the following sections refer to both the object-relational and relational models.

A.1.1 Data Modeling

Data modeling is very important when designing a spatial database. You should group geometries into layers based on the similarity of their attributes. Assume your data model uses line strings to represent both roads and rivers. The attributes of a road and the attributes of a river are different. Therefore, these geometries should be modeled in two different layers.

In practice, however, if the user of your application will always ask to see both the roads and rivers in a particular region (area of interest), then it may be appropriate to model roads and rivers in the same layer with a common set of attributes.

It is equally important to understand how the data in the various layers will be queried. If the user of your application is interested in querying the data based on a relationship between the layers, then you should index the layers with the same fixed-size tiling level. For example, a query such as *Which roads cross rivers?* can achieve better performance if the roads and rivers layers are tiled at the same level.

On the other hand, if two layers are indexed with different SDO_LEVEL values, a spatial join of these two layers will not result in optimal query performance. For example, if ROADS is indexed using (SDO_LEVEL=8 SDO_NUMTILES=12) and COUNTIES is indexed using (SDO_LEVEL=10 SDO_NUMTILES =6), the following query has less than optimal performance:

```
SELECT a.name from Roads a, Counties b WHERE
   MDSYS.SDO_RELATE(a.geometry, b.geometry,
   'MASK=ANYINTERACT QUERYTYPE=JOIN')='TRUE';
```

If the layers are frequently used in a spatial join, then they should be indexed using the same SDO_LEVEL value. In the preceding example, better performance results if both layers are indexed with the same SDO_LEVEL value (for example, SDO_LEVEL=8).

A.1.2 Understanding the Tiling Level

The following example explains how tiling is used in Spatial.

Assume you want to find all the roads (line strings) that overlap a county boundary (polygon) in a spatial database containing 10 million roads. Ignoring Spatial features for a moment, in purely mathematical terms, the problem translates into comparing all the line segments that make up each road to the line segments and area of the county boundary, to see if there is any intersection. This geometry-to-geometry comparison is very expensive.

Spatial simplifies this calculation by approximating each geometry with tiles. The primary filter in Spatial translates the problem to show all the roads that have a tile equal to a tile that approximates the polygon. The result of this is a superset of the final answer.

The secondary filter (a true geometry-to-geometry comparison) can now be applied to the candidates that returned from the Spatial primary filter, instead of to every road in the database.

Picking the correct tile size for fixed-size tiling is one of the most important factors in attaining good performance. If the tile size you select is too small, you could end up generating thousands of tiles per geometry. Also, the process of tiling a query window may become very time consuming.

At the same time, you do not want to choose tiles that are too big. This would defeat the purpose of the Spatial primary filter. If the tiles are too big, then too many geometries are returned from the primary filter and are sent to the more costly secondary filter.

Keep in mind that the tile size you choose should also depend on whether or not the query window (area of interest) is already defined in the database. If the query window is defined in the database (that is, if the spatial tables and spatial indexes already exist), then you should choose a smaller tile size (that is, use a larger value for SDO_LEVEL). Assume that the state and highway layers are already defined in the database. You could perform a spatial join query, such as *Which interstate highways go through the state?*, without incurring the overhead of tiling because the query window is already defined in the database. If, on the other hand, you are creating the query window dynamically, you have to factor in the time it takes to define and index the query window. In this case, you should choose a larger tile size (that is, a smaller value for SDO_LEVEL) to reduce the time it takes to define and index the query window.

The SDO_TUNE.ESTIMATE_TILING_LEVEL function can be used on your data set to get an initial tiling level estimate. This may not be your final answer, but it will be a good level for starting your analysis. In general, it is recommended that you take a random sample of your data and check the query performance at different levels of tiling. This will give an indication of what is the best tiling level for the total data set.

A.1.3 Using Hybrid Indexes (Object-Relational Model Only)

Note: For most applications, you should not use hybrid indexes, but should instead use fixed indexes or R-tree indexes. The rare circumstances where hybrid indexes should be considered are described in Section 1.7.2.3.

Before deciding on the type of index to use for a spatial application, be sure you understand the indexing concepts and guidelines discussed in Section 1.7.

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 then 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.

A.1.4 Database Sizing

Properly choosing rollback segments and tablespaces is important for getting good performance from Spatial. Therefore, it is very important to read the *Oracle8i Administrator's Guide* and understand the concepts of tablespaces and rollbacks.

Here are some general guidelines to consider:

- Always make sure that you have enough rollback space to create a spatial index.
- Create separate tablespaces for data layers, indexes, and rollback segments.
- Properly define initial extents, next extents, and pctincrease values for data layer tables.
- Use the SDO_GEOM.VALIDATE_GEOMETRY function to ensure correctness of geometries in the data sets. Entering incorrect data may lead to unexpected behavior in index creation and in the SDO_GEOM.RELATE function.
- Visualizing the indexing tiles, as described in Section A.1.5, can lead to a greater understanding of the tuning process with respect to the size of the tiles.
- As values of SDO_LEVEL and SDO_NUMTILES are increased, so are the storage requirements for the index table and the indexes associated with it, as well as the size of the rollback segment required for the CREATE INDEX statement if SDO_COMMIT_INTERVAL is not specified.

The following guidelines refer to only the relational model:

- Define the initial extent to be as small as possible when you create the <layername>_SDOLAYER and the <layername>_SDODIM tables. These tables contain a few rows each, and a small initial extent will reduce the amount of wasted space.

- Always build a B-tree index on the SDO_GID column of the <layername>_SDOGEOM table before attempting to call the SDO_ADMIN.POPULATE_INDEX_FIXED, SDO_ADMIN.UPDATE_INDEX_FIXED, SDO_ADMIN.POPULATE_INDEX, or SDO_ADMIN.UPDATE_INDEX procedure.
- For fixed-size tiling, always build a B-tree index on the SDO_CODE column of the <layername>_SDOINDEX table before trying any queries using this table.
- Always build a B-tree index on the SDO_GID column of the <layername>_SDOINDEX table if individual SDO_GID values will be used as query windows for other Spatial layers.
- For variable-sized tiling, always build a B-tree index on the SDO_GROUPCODE column of the <layername>_SDOINDEX table before trying any queries using this table.

A.1.5 Visualizing the Spatial Index (Drawing Tiles)

To select an appropriate tiling level, it may help to visualize the tiles covering your geometries. Through visualization, you can determine how many tiles are used for each object, the size of the tiles, and how well the edges of your geometry are covered. The basic algorithm is:

1. Select the edges of the tiles represented by the index entries.
2. Plot the tiles on a two-dimensional grid.
3. Plot your geometries on the same grid.

A.1.5.1 Drawing Tiles from the Object-Relational Model

Two Spatial internal functions have been made visible to describe the tiles. These functions were part of a previous release of Oracle Spatial Data Option, and are currently reserved for internal use only. These functions are not recommended for general use, except for this visualization example. Use the following syntax for the internal functions:

```
hhcellbndry (sdo_code || substrb(sdo_code,-1,1)||''020000'',
      sdo_dimnum, sdo_lb, sdo_ub,
     hhlength(sdo_code || substrb(sdo_code,-1,1)||''020000'') {'MIN' | 'MAX'})
```

Note that in ''020000'' two pairs of single quotation marks are used, not two double quotation marks.

In the following examples, the dimension boundaries were assumed to be -180 to 180, and -90 and 90. Also, an index named TEST_INDEX_HL2N6 and a table named TEST are used in the examples.

The SQL queries shown in Example A–1 and Example A–2 can be used to decode all the index entries in an index table. The examples return the coordinates of the lower-left and upper-right corners of each tile.

Example A–1 View Fixed-Size Tiles for All Geometries

```
SELECT HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 1,-180.0, 180.0,
                    HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MIN') min_x,
       HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 1,-180.0, 180.0,
                    HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MAX') max_x,
       HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 2, -90.0,  90.0,
                    HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MIN') min_y,
       HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 2, -90.0,  90.0,
                    HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MAX') max_y
FROM (SELECT distinct sdo_groupcode, sdo_fixed_meta
      FROM TEST_INDEX_HL2N6$ a,
           SDO_INDEX_METADATA b
      WHERE b.sdo_index_table = 'TEST_INDEX_HL2N6$');
```

Example A–2 View Variable-Sized Tiles for All Geometries

```
SELECT
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 1,-180.0, 180.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MIN') min_x,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 1,-180.0, 180.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MAX') max_x,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 2, -90.0,  90.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MIN') min_y,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 2, -90.0,  90.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MAX') max_y
FROM (SELECT distinct sdo_code, sdo_meta
       FROM TEST_INDEX_HL2N6$ a);
```

The SQL queries shown in Example A–3 and Example A–4 can be used to decode the index entries for a specific geometry stored in an index table.

Example A–3 View Fixed-Size Tiles for One Geometry

```
SELECT HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 1,-180.0, 180.0,
                    HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MIN') min_x,
       HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 1,-180.0, 180.0,
```

```
                HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MAX') max_x,
        HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 2, -90.0,  90.0,
                HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MIN') min_y,
        HHCELLBNDRY(sdo_groupcode || sdo_fixed_meta, 2, -90.0,  90.0,
                HHLENGTH(sdo_groupcode || sdo_fixed_meta), 'MAX') max_y
        FROM (SELECT distinct sdo_groupcode, sdo_fixed_meta
              FROM TEST_INDEX_HL2N6$ a,
                   SDO_INDEX_METADATA b
              WHERE b.sdo_index_table = 'TEST_INDEX_HL2N6$'
                AND a.sdo_rowid = 'AAAA59AAFAADzAZAAA');
```

Example A–4 View Variable-Sized Tiles for One Geometry

```
SELECT
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 1,-180.0, 180.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MIN') min_x,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 1,-180.0, 180.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MAX') max_x,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 2, -90.0,  90.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MIN') min_y,
  HHCELLBNDRY(sdo_code || substrb(sdo_code,-1,1)||''020000'', 2, -90.0,  90.0,
    HHLENGTH(sdo_code || substrb(sdo_code,-1,1)||''020000''), 'MAX') max_y
FROM TEST_INDEX_HL2N6$
WHERE sdo_rowid = 'AAAA59AAFAADzAZAAA';
```

A.1.5.2 Drawing Tiles from the Relational Model

The spatial index is represented internally as a linear quadtree. The structure used to represent the linear quadtree is composed of two components: a data component and a metadata component. The data component of the linear quadtree is stored in the SDO_CODE column, and the metadata component is stored in the SDO_META column.

The SDO_META column is not required for spatial queries. However, by combining the SDO_META column with the SDO_CODE column, the tiles of any geometry or of the entire data set can be decoded. This capability allows the tiles to be visualized.

Two Spatial internal functions have been made visible to describe the tiles. These functions were part of a previous release of Oracle Spatial Data Option, and are currently reserved for internal use only. The functions are not recommended for general use, except for this visualization example. Use the following syntax for the internal functions:

```
hhcellbndry (sdo_code || sdo_meta, sdo_dimnum, sdo_lb, sdo_ub,
```

```
hhlength(sdo_code || sdo_meta) {'MIN' | 'MAX'})
```

In the following examples, the dimension boundaries were assumed to be -180 to 180, and -90 and 90. The dimensional information is stored in the <layername>_SDODIM table.

If you used SDO_ADMIN.UPDATE_INDEX_FIXED or SDO_ADMIN.POPULATE_INDEX_FIXED to generate your spatial index, replace `sdo_code || sdo_meta` with `sdo_tile` in the SQL statements that follow.

The SQL query shown in Example A–5 can be used to decode all the index entries in a <layername>_SDOINDEX table. The example returns the coordinates of the lower-left and upper-right corners of each tile.

Example A–5 View Fixed-Sized Tiles for All Geometries Using the Relational Model

```
SELECT hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
           hhlength (sdo_code || sdo_meta), 'MIN') min_x,
       hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
           hhlength (sdo_code || sdo_meta), 'MAX') max_x,
       hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
           hhlength (sdo_code || sdo_meta), 'MIN') min_y,
       hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
           hhlength (sdo_code || sdo_meta), 'MAX') max_y
FROM (SELECT DISTINCT sdo_code, sdo_meta FROM <layername>_sdoindex);
```

The SQL query shown in Example A–6 can be used to decode the index entries for a specific geometry stored in a <layername>_SDOINDEX table.

Example A–6 View Fixed-Size Tiles for a Specific Geometry Using the Relational Model

```
SELECT hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
          hhlength (sdo_code || sdo_meta), 'MIN') min_x,
       hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
          hhlength (sdo_code || sdo_meta), 'MAX') max_x,
       hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
          hhlength (sdo_code || sdo_meta), 'MIN') min_y,
       hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
          hhlength (sdo_code || sdo_meta), 'MAX') max_y
FROM <layername>_sdoindex
WHERE sdo_gid = <geometry id>;
```

See Section A.3.2.3 for another method of viewing tiles.

A.1.6 Setting the SORT_AREA_SIZE Value

When the Oracle8*i* database server processes SQL statements that require sorting, such as statements containing an ORDER BY or DISTINCT clause, the database server stores the result set in a temporary storage area. The result set is then sorted. If the SORT_AREA_SIZE value is insufficient for holding the result set in memory, then some data may be written to disk and an entry is written in the redo log.

Many Spatial operators issue SQL statements internally that contain DISTINCT and ORDER BY clauses. If the SORT_AREA_SIZE initialization parameter is too small for processing the secondary filters, then some sorting may occur on disk, which causes entries to be written in the redo log. This may affect performance. For better performance, increase the SORT_AREA_SIZE parameter to force sorting to occur in memory.

To determine if sort operations associated with Spatial are happening in memory or on disk, execute the following SQL statement before and after running spatial queries on an otherwise inactive database:

```
SELECT name, value
   FROM v$sysstat
   WHERE name IN ('sorts (memory)', 'sorts (disk)');
```

If the value associated with disk sort operations is higher after the queries, then the slower (costlier) disk sorting is being used.

A.1.7 Tuning Point Data with the Relational Model

Point data, unlike line and polygon data, has the unique characteristic of containing one tile per point. This section describes how to improve the performance of queries on point data.

A.1.7.1 Efficient Queries for Relational Point Data

When querying point data with a rectangular query window, you can take advantage of the nature of these geometries to improve performance.

A rectangle can be defined by its lower-left and upper-right coordinates (Xmin, Ymin and Xmax, Ymax). A point has a single set of coordinates (Px, Py). When your area of interest is a rectangle, instead of using the SDO_GEOM.RELATE function in the secondary filter, you can use simple SQL comparison operators as follows:

```
SELECT sdo_gid, sdo_x1, sdo_y1
FROM  cities_sdogeom,
      (SELECT a.sdo_gid gid1
```

```
      FROM cities_sdoindex a,
           window_sdoindex b
      WHERE b.sdo_gid = [area of interest id]
        AND a.sdo_code = b.sdo_code)
           WHERE sdo_gid = gid1
            AND sdo_x1 BETWEEN Xmin AND Xmax
            AND sdo_y1 BETWEEN Ymin AND Ymax;
```

The DISTINCT clause is not necessary in the primary filter of the query because a point contains only a single tile in the spatial index.

A.1.7.2 Efficient Schema for Relational Point Layers

Because a point is always referenced by only one tile in a spatial index, for improved performance you can place the columns normally found in the <layername>_SDOINDEX table in the <layername>_SDOGEOM table. This will save you the cost of joining the <layername>_SDOINDEX and <layername>_SDOGEOM tables.

You still need to create an updatable view for the <layername>_SDOINDEX table that selects the appropriate columns from the <layername>_SDOGEOM table. This is because functions such as SDO_ADMIN.UPDATE_INDEX_FIXED and SDO_ADMIN.POPULATE_INDEX_FIXED expect a <layername>_SDOINDEX table to exist. Create the view using INSTEAD OF triggers for insert, delete, and update operations such that the appropriate columns in the <layername>_SDO_GEOM table are updated. The following example shows how to use INSTEAD OF triggers:

```
CREATE OR REPLACE TRIGGER mytrig INSTEAD OF INSERT ON points_sdoindex
     REFERENCING new AS n
     FOR EACH ROW
     BEGIN
          UPDATE points_sdogeom SET points_sdogeom.sdo_code = :n.sdo_gid
          WHERE sdo_gid = :n.sdo_gid;
     END;
CREATE OR REPLACE TRIGGER mydeltrig INSTEAD OF DELETE ON points_sdoindex
    REFERENCING old AS n
    FOR EACH ROW
    BEGIN
          UPDATE points_sdogeom SET points_sdogeom.sdo_code = NULL
          WHERE  points_sdogeom.sdo_gid = :n.sdo_gid;
    END;
```

The following example shows a window query of a layer containing point data when the window layer contains one rectangle:

```
SELECT a.sdo_gid, sdo_x1, sdo_y1
FROM  points_sdogeom a,
      window_sdoindex b
WHERE b.sdo_gid = [area of interest id]
  AND a.sdo_code = b.sdo_code
  AND sdo_x1 BETWEEN Xmin AND Xmax
  AND sdo_y1 BETWEEN Ymin AND Ymax;
```

A.1.7.3 Script for Using Table Partitioning with Relational Point Data

Because point data is always indexed using a single tile, it is well suited for partitioning. The following script shows an example of using the Oracle8*i* partitioning feature with Spatial point data:

```
ORACLE_HOME/MD/demo/examples/scripts/partition_points.sql
```

A.1.8 Tuning Spatial Join Queries Using the Relational Model

There are some helpful hints you can place in your spatial join queries to improve performance. The remainder of this section describes some of the hints you can use. For more information on hints, see *Oracle8i Tuning*.

A.1.8.1 Using the NO_MERGE, INDEX, and USE_NL Hints

A spatial join takes place between two layers. When the two layers being joined are line or polygon layers, the spatial join query contains two DISTINCT clauses: one in the inner SELECT clause and the other in the outer SELECT clause. The Oracle optimizer ignores the inner DISTINCT clause to save on the cost of sorting. However, if the inner DISTINCT clause is ignored, the secondary filter gets called many more times than it needs to be. This can have a significant impact on performance because the secondary filter is an expensive operation. Use the NO_ MERGE hint to prevent the optimizer from ignoring the inner DISTINCT clause.

In a spatial join, all the tiles from one layer are compared to all the tiles from another layer. The Oracle database server performs a full table scan on one <layername>_SDOINDEX table, (preferably the smaller of the two), and an index lookup on the other <layername>_SDOINDEX table. Use the INDEX and USE_NL hints to force the optimizer to perform the full table scan on the smaller of the two <layername>_SDOINDEX tables being compared.

The following example shows a spatial join between line (road) and polygon (county) data. The query answers the question, *Which counties intersect major roads?*

```
SELECT /*+ cost
           ordered use_nl(COUNTY_sdogeom)
```

```
            index (COUNTY_sdogeom NAME_OF_SDO_GID_INDEX)
        */
        COUNTY_sdogeom.SDO_GID,
        COUNTY_sdogeom.SDO_ESEQ,
        COUNTY_sdogeom.SDO_SEQ,
        COUNTY_sdogeom.SDO_X1,COUNTY_sdogeom.SDO_Y1,
        COUNTY_sdogeom.SDO_X2,COUNTY_sdogeom.SDO_Y2,
        COUNTY_sdogeom.SDO_X3,COUNTY_sdogeom.SDO_Y3,
        COUNTY_sdogeom.SDO_X4,COUNTY_sdogeom.SDO_Y4,
        COUNTY_sdogeom.SDO_X5,COUNTY_sdogeom.SDO_Y5,
        COUNTY_sdogeom.SDO_X6,COUNTY_sdogeom.SDO_Y6,
        COUNTY_sdogeom.SDO_X7,COUNTY_sdogeom.SDO_Y7,
        COUNTY_sdogeom.SDO_X8,COUNTY_sdogeom.SDO_Y8
FROM (SELECT DISTINCT gid_a gid1
      FROM (SELECT /*+ index (a NAME_OF_SDO_CODE_INDEX)
                       index (b NAME_OF_SDO_CODE_INDEX)
                       use_nl (a b)
                       no_merge */
            DISTINCT a.sdo_gid gid_a,
                     b.sdo_gid gid_b
            FROM COUNTY_SDOINDEX a,
                 MAJOR_ROAD_SDOINDEX b
            WHERE a.sdo_code = b.sdo_code)
      WHERE sdo_geom.relate('COUNTY', gid_a, 'ANYINTERACT',
                            'MAJOR_ROAD',gid_b) <> 'FALSE'),
      COUNTY_sdogeom
WHERE COUNTY_sdogeom.sdo_gid = gid1;
```

A.1.8.2 Spatial Join Queries with Point Layers

The following example shows a spatial join between line (road) and point (street address) data. The query answers the question, *Which addresses are on a major road?*

```
SELECT /*+ cost
            ordered use_nl (STREET_ADDRESS_sdogeom)
            index (STREET_ADDRESS_sdogeom NAME_OF_SDO_GID_INDEX)
        */
        STREET_ADDRESS_sdogeom.SDO_GID,
        STREET_ADDRESS_sdogeom.SDO_X1,
        STREET_ADDRESS_sdogeom.SDO_Y1
FROM (SELECT DISTINCT gid_a gid1
      FROM (SELECT /*+ index (a NAME_OF_SDO_CODE_INDEX)
                       index (b NAME_OF_SDO_CODE_INDEX)
                       use_nl (a b) */
                     a.sdo_gid gid_a,
```

```
                b.sdo_gid gid_b
         FROM STREET_ADDRESS_SDOINDEX a,
              MAJOR_ROAD_SDOINDEX b
         WHERE a.sdo_code = b.sdo_code)
   WHERE sdo_geom.relate('STREET_ADDRESS', gid_a, 'ANYINTERACT',
                         'MAJOR_ROAD',gid_b) <> 'FALSE'),
   COUNTY_sdogeom
WHERE COUNTY_sdogeom.sdo_gid = gid1;
```

The inner DISTINCT clause is not necessary for spatial joins where one of the layers contains point data. Therefore, the NO_MERGE hint is not necessary. This is because points contain only one tile in the spatial index.

The following example shows a spatial join between polygon (county) and point (street address) data. The query generates a report that displays how many addresses are associated with each county.

If you can assume that each street address is associated with a single county, you can significantly speed up this query. Because points contain only a single tile in the spatial index, any street address tile that matches only one county tile in the primary filter does not need to go through the expensive secondary filter.

```
SELECT county_gid, count(street_gid)
FROM  (SELECT poly.sdo_gid county_gid, street.sdo_gid street_gid
       FROM  STREET_ADDRESS_sdoindex street,
             (SELECT sdo_code county_sdo_code,
                     count(sdo_gid) interacts
              FROM CENSUS_COUNTY_sdoindex
              GROUP by sdo_code
             ) counts,
             CENSUS_COUNTY_sdoindex  poly
       WHERE street.sdo_code = counts.county_sdo_code
         AND poly.sdo_code   = street.sdo_code
         AND (counts.interacts = 1
              OR
              sdo_geom.relate('STREET_ADDRESS', street.sdo_gid,
                              'ANYINTERACT',
                              'CENSUS_COUNTY',poly.sdo_gid) <> 'FALSE'
             )
      )
GROUP BY county_gid;
```

A.1.9 Using Customized Geometry Types in the Relational Model

The relational spatial model supports three geometry types: points, lines, and polygons. If your data contains another type, such as a circle or arc, then you must choose the supported type that best approximates your desired type (or upgrade to the object-relational model). For example, in the relational model, a circle can be defined as a multisided polygon. Obviously, the more coordinates in the element, the better the approximation will be.

Although customized types are not supported, you do not have to lose the information about customized types. After storing the approximated element, create another element in that geometry with SDO_ETYPE=0. Spatial ignores elements of SDO_ETYPE=0. You can then write your own routines to handle your specialized geometry type.

A.1.10 Partitioning Spatial Data Using the Relational Model

The Oracle8*i* partitioning feature lets you spread out your spatial data and create spatial indexes in a very controlled manner. Such control allows a database administrator to isolate data that may be causing I/O performance issues. Note that this optimization works only for the relational implementation.

The most obvious way to partition relational spatial data is to base the partitions on the geometry ID (GID) column. Select the full list of available GIDs in a given layer and sort them to produce an ordered list. Next, examine the list to determine whether or not the GIDs would provide a good set of balanced partitions. In cases where one or two GIDs dominate the layer, partitioning by GID will not yield a balanced distribution. In such cases, you may want to consider adding a new alphanumeric column to the layer, and use this column to create balanced partitions. Although this requires an extra effort, it may result in significant performance improvements.

For more information, including examples and sample parsing times, see the online text file: *ORACLE_HOME*/md/demo/examples/scripts/parallel.doc.

A.1.11 Parallel Loading and Indexing of Spatial Data Using the Relational Model

On a multiprocessor system, you can use parallel execution to improve both loading times and spatial index creation times. Note that this optimization works only for the relational implementation.

When using partitioned tables, as described in Section A.1.10, you can achieve further performance gains by loading and indexing geometries in parallel. The partitioned tables can be loaded by selecting from nonpartitioned source tables, or

using the SQL*Loader utility. Parallel threads (one for each partition) can be submitted to load the partitioned table. For information on parallel loading, see the description of the SQL*Loader utility in Oracle8i Utilities.

You can also create spatial indexes in parallel by creating a number of views or layers. Create each layer with a range of GIDs, with corresponding <layername>_ SDOLAYER and <layername>_SDODIM tables. For example, the following statements create the necessary views for the first 300 GIDs in a table.

```
CREATE VIEW a_sdogeom AS SELECT * FROM a_sdogeom
  WHERE sdo_gid BETWEEN 1 and 300;
CREATE VIEW a_sdodim AS SELECT * FROM a_sdodim;
CREATE VIEW a_sdolayer AS SELECT * FROM a_sdolayer;
```

Next, create the index table as a partitioned table. Create a partition for each range of GIDs for which you created a view.

```
CREATE TABLE a_sdoindex
  (sdo_gid   NUMBER,
   sdo_code  RAW(255),
   sdo_meta  RAW(255))
INITRANS 4
STORAGE (initial     2M
         next        1M
         pctincrease  0
         freelist groups 12
         freelists   19)
PARTITION BY RANGE (sdo_gid)
(PARTITION a_idx1 VALUES LESS THAN (300)
 TABLESPACE sdo_data
  .
  .
  . );
```

To create the index, submit commands to execute the SDO_ADMIN.POPULATE_ INDEX procedure for each of the partitions. The threads will independently build their corresponding indexes, with significant performance improvements over the nonpartitioned, single-threaded model.

For more information, including examples and sample parsing times, see the online text file: *ORACLE_HOME*/md/demo/examples/scripts/parallel.doc.

A.2 Scripts for Spatial Indexing Using the Relational Model

Spatial provides sample SQL script files to show how to use dynamic SQL in a PL/SQL block to create layer tables for spatially indexed data. The scripts are available after installation in the *ORACLE_HOME*/md/admin directory.

The following sections describe the cr_spatial_index.sql and crlayer.sql scripts.

A.2.1 cr_spatial_index.sql Script

The cr_spatial_index.sql script file shows an example of updating the spatial index for a layer, and executing a commit operation after every 50 GIDs have been entered.

The procedures SDO_ADMIN.POPULATE_INDEX and SDO_ADMIN.POPULATE_INDEX_FIXED operate as a single transaction. To reduce the number of rollback operations required to execute these procedures, you can write a routine similar to that in cr_spatial_index.sql. This script loops and calls SDO_ADMIN.UPDATE_INDEX_FIXED for each GID, committing after every 50 GIDs.

```
-- cr_spatial_index.sql
--
-- Note: if geometries do not span more than 1 row, you can remove
-- the DISTINCT qualifier from the SELECT statement.
--
declare
   cursor c1 is SELECT DISTINCT sdo_gid from POLYGON_SDOGEOM;
   gid number;
   i number;
begin
     i := 0;
     for r in c1 loop
       begin
        gid:= r.sdo_gid;
        sdo_admin.update_index_fixed('POLYGON', gid, 15, FALSE, FALSE, FALSE);
        exception when others then
          dbms_output.put_line('error for gid'||to_char(gid)||':  '||SQLERRM );
       end;
       i:=  i + 1;
       if i = 50 then
          commit;
          i:= 0;
       end if;
     end loop;
commit;
```

```
end;
/
```

When you call the SDO_ADMIN.UPDATE_INDEX_FIXED procedure for a large data set, you may get a "snapshot too old" error message from the Oracle database server. You can avoid this error by creating more or larger rollback segments. You can also try to increase the number of GIDs before committing the transaction.

Note: The cr_spatial_index.sql script is not available in your *ORACLE_HOME*/md/admin directory after installation. You must create this script yourself.

A.2.2 crlayer.sql Script

The crlayer.sql script file is a template used to create all the tables for a layer and populate the metadata in the <layername>_SDODIM and <layername>_SDOLAYER tables.

A.3 Tools and Related Products

The following sections describe sample programs and related products that, while not required for the storage or maintenance of spatial data, can make those tasks simpler.

A.3.1 Oracle *inter*Media Locator

Oracle *inter*Media Locator is a related product that supports online Internet-based geocoding facilities for location-aware applications and proximity queries.

A.3.1.1 Geocoding Support

Geocoding is the process for converting a nonstandardized street address or postal code into a standardized address (optionally certified by the USPS), with latitude and longitude information. In addition, census information such as block groups, postal carrier routes, and block codes can be retrieved as a result of this process.

The *inter*Media Locator option provides an interface to the online geocoding service provided by Qualitative Marketing Service, Inc. (QMS). You can use PL/SQL stored procedures to geocode an address, and record and fetch all the information into two predefined objects from the QMS Web site. The first object is of type SDO_GEOMETRY, and it contains the spatial longitude and latitude information stored as

point data. The second object returned is GEOCODE_RESULT, which contains text fields of a standardized address and other fields mentioned previously, such as postal carrier route or block code.

For more information about this online service, see the following Web site:

```
http://www.centrus-software.com/oracle
```

For more information about *inter*Media Locator, see the *Oracle interMedia Locator User's Guide and Reference.*

A.3.1.2 Compatibility with Spatial Objects

*inter*Media Locator is a subset of Oracle Spatial and, therefore, is completely compatible with Spatial objects. The index uses the same set of metadata tables, for instance. One difference is that *inter*Media Locator locates only points, while Spatial supports multiple geometry types.

The LOCATOR_WITHIN_DISTANCE operator is similar to the SDO_ GEOM.WITHIN_DISTANCE operator.

The *inter*Media Locator version of the WITHIN_DISTANCE operator takes a new parameter in the last string: *units=[mile,meter,ft]*. This allows you to search by units. The Spatial version uses only an estimate on the surface of the Earth, and not exact distance or driving distance.

A.3.1.3 Sample *inter*Media Locator Code

Sample scripts are available in the following directory after you install Oracle *inter*Media Locator:

```
$ORACLE_HOME/md/demo/geocoder
```

To migrate data between products, type `ocimig`, and prompts will guide you through the process, which is similar to using SQL*Loader or the Export/Import utilities.

A.3.2 Spatial Viewer on UNIX/Motif for Relational Model

A sample geometry viewer, sdodemo, is available for UNIX systems using a Motif interface. This viewer displays geometries stored using the relational model.

A.3.2.1 Installation and Setup

The following steps are required to set up and run the Motif application:

1. Set the environment variables:

```
setenv MD_VIEWER <full_pathname>/sdo_motif_demo/src
setenv XENVIRONMENT $MD_VIEWER/app-defaults/resource_file
alias sdodemo $MD_VIEWER/bin/demo
```

2. Run the following as MDSYS:

```
$ORACLE_HOME/md/admin/sdowin.sql
$ORACLE_HOME/md/admin/prvtwin.plb
$MD_VIEWER/sql_scripts/my_window.sql
$MD_VIEWER/sql_scripts/my_win.sql
```

3. If you are using a Sun Solaris system, a compiled version of $MD_ VIEWER/bin/demo has been shipped with Spatial. Go to step 4.

   If you are using a UNIX operating system other than Solaris, you need to recompile the viewer. A makefile is included only for Sun Solaris systems. You may need to make some system-specific modifications.

```
cd $MD_VIEWER
make -f makefile8.sun clean
make -f makefile8.sun
```

4. Create an alias for the sample program:

```
alias sdodemo $MD_VIEWER/bin/demo
```

5. Run the sample program:

```
sdodemo
```

A.3.2.2 Connecting to a Database and Viewing Geometries

When you run the sample sdodemo program, you will be prompted for an Oracle user name, password, and alias if the database resides on a remote machine.

Two windows appear, one where geometries are drawn, and a second with several buttons. Click **CHOOSE LAYER** and select a layer.

The extent of the map will initially be the values stored in the <layername>_ SDODIM table for the current layer. You can then click **ZOOM TO EXTENT**, and the map extent will be set to the true extent of your data. Note that the time it takes to perform ZOOM TO EXTENT depends on the amount of data in your <layername>_SDOGEOM table.

A.3.2.3 Using the Sample Viewer

The text for all queries is displayed in the UNIX shell where you are running the sdodemo program.

There are three radio buttons at the top of the control panel. These buttons determine which query is executed when you click **PERFORM QUERY**:

- PRIM & SEC: Performs a primary and secondary filter.
- PRIMARY FILTER ONLY: Performs a primary filter only query.
- DRAW ALL: Selects everything in the <layername>_SDOGEOM table. This does not perform a spatial query.

To perform a spatial query:

1. Click either **PRIM & SEC** or **PRIMARY FILTER ONLY**.
2. Click **SELECT BOX**, **SELECT CIRCLE**, or **SELECT POLYGON**, and draw the area of interest on the map.
3. Click **PERFORM QUERY**. The geometries will be displayed on the base map.

You can look at individual geometries by clicking **SHOW GID**. You can also click **SHOW ALL TILES** to look at index tiles. This can help you tune your spatial index. See Section A.1.5 for another method of drawing tiles.

A.3.3 Spatial Visualizer on Windows NT for the Object-Relational Model

The Spatial Visualizer is a sample program used to demonstrate two things. First, it is an example of using dynamic linking libraries to wrap Oracle Call Interface (OCI) and Spatial functions into C++ classes. Second, the program provides a simple visualizer that can display Spatial objects.

A.3.3.1 Compiling and Running the Sample Program

To compile the Spatial Visualizer sample program, first unzip the following file into your work directory: *ORACLE_HOME*/md/demos/NT/DEMO_Visualizer.zip. This creates the following subdirectories:

- include: Contains header files.
- bin and lib: Contain output files.
- SDOConnCur: Contains a project for creating a dynamic link library (DLL).
- VisualSDO: Contains another project for creating an executable (EXE) file.

Next, make sure your Visual C++ IDE has the correct directory settings for using OCI and common header files. To ensure this, click **Tools** > **Options** > **Directories**, and then perform the following tasks:

1. Click **Include files** to add the OCI include path (for example, C:\ORANT\OCI80\include) and the common include path for your projects (for example, Myprojects\include).
2. Click **Library files** to add the OCI library path (for example, C:\ORANT\OCI80\lib\msvc) and the common library path for your projects (for example, Myprojects\lib).
3. Type `SDOConnCur\SDOConnCur.dsw` and click **Open** to compile SDOConnCur.dll.
4. Type `VisualSDO\VisualSDO.dsw` and click **Open** to create VisualSDO.exe.

A.3.3.2 Usage Notes

Consider the following when using this sample program:

- ’SDOConnCur’: This project creates a DLL (SDOConnCur.dll) to wrap OCI and SDO functions into C++ classes, so that users of this DLL can benefit from Oracle Call Interface (OCI) without knowing how to make OCI calls.
- 'VisualSDO’: This project creates an executable file (VisualSDO.exe) based on SDOConnCur.dll. It is a simple visualizer that can display Oracle Spatial geometry objects.
- All the files and directories under *ORACLE_HOME*/md/demos/NT are components of the Spatial Visualizer demonstration program. They should be used for demonstration purposes only.
- The workspaces are created with Visual C++ 6.0, and might not be compatible with previous versions.
- The ZIP file (DEMO_Visualizer.zip) contains all the contents under this directory. Due to system dependencies, copy the ZIP file only to a Windows NT system.

B

Installation, Compatibility, and Migration Issues

This appendix provides information concerning installation, compatibility, and migration between various Oracle Spatial product releases.

Beginning with Spatial Data Option 7.3.3, all interfaces are supported in each subsequent release. A spatial application built for and using the 7.3.3 Spatial Data Option interfaces will work with a release 8.0.4, 8.0.5, 8.1.6, or 8.1.7 database server. The implementations of these interfaces have changed, and therefore PL/SQL packages from prior releases of the Spatial cartridge will not work with later releases of the Oracle8*i* database server. Therefore, you must upgrade both the database server and Spatial at the same time if you wish to use older spatial applications with an Oracle8*i* release of Spatial.

Spatial must always be synchronized with the Oracle8*i* database server on upgrade or downgrade. In both cases, Spatial must be reinstalled.

B.1 Introduction

Spatial release 8.1 requires Oracle8*i* Enterprise Edition and the Objects Option. Spatial release 8.1 was redesigned to use various Oracle8*i* object and extensibility features. Many of the Spatial release 8.1 features depend on new features in release 8.1 of the database server. Therefore, there are many compatibility and migration issues that need to be addressed in this release of Spatial. This appendix outlines the database and application compatibility issues.

Database compatibility issues exist because Spatial uses extensible indexing and object types in release 8.1; and therefore if a release 8.1 database instance is downgraded to release 8.0.5, the spatial objects must be deleted and re-created. In this case, the data must be exported and imported into the release 8.0.5 database.

This, and other requirements, result in application incompatibility. A release 8.1 Spatial application will likely use the new spatial operators and therefore will not work with a release 8.0.5 instance unless it can identify the Spatial version and dynamically change its spatial queries.

An upgrade or downgrade of the database server version requires a corresponding upgrade or downgrade of Spatial. If a release 8.0.5 database server is upgraded to release 8.1, Spatial must also be upgraded. The reason has to do with using dynamic SQL in PL/SQL and with invoker's rights in release 8.1. Similarly, if a release 8.1 server is downgraded, Spatial must be downgraded too. Lastly, if a release 8.1 server is running in release 8.0 compatibility mode, Spatial will experience various failures unless it is reconfigured for release 8.0.5. You can reconfigure the product by running the downgrade script: c813d805.sql.

In summary:

- The Spatial release and the Oracle8*i* database server release must match.
- Upgrade and downgrade scripts must be run when upgrading or downgrading between releases 8.0.5 and 8.1.
- Spatial will work in release 8.0 compatibility mode for a release 8.1 database server if, and only if, the downgrade script is run and users or applications attempt to use only the relational implementation of the product.

B.2 Installation Details

To install Spatial, the script catmd.sql in the *ORACLE_HOME*/md/admin directory must be run as user MDSYS. The MDSYS user should be created with the set of privileges listed in *ORACLE_HOME*/MD/mdprivs.sql, and with both default and temporary tablespaces.

Installation of Spatial requires that the COMPATIBLE init.ora parameter is set to 8.1.0.0.0 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 8.0.n.n.n, the DBA must shut down the database and restart with COMPATIBLE=8.1.n.n.n.

B.2.1 Changing from 8.1 to 8.0 Compatibility Mode

If Spatial has been installed and the database compatibility needs to be reset to 8.0.n.n.n from 8.1.n.n.n, do the following:

1. Determine if there is any user data that contains instances of the type MDSYS.SDO_GEOMETRY. That is, determine if any user table has a column of type MDSYS.SDO_GEOMETRY and has data in it.
2. If there are instances, delete all spatial indexes on these columns. Delete the data in these columns or delete these columns and tables. If there are no instances, go on to the next step.
3. Run the script c813d805.sql in *ORACLE_HOME*/md/admin. This will delete all spatial objects that require 8.1 compatibility. That is, all the object-relational implementation objects for Oracle Spatial will be deleted. The relational implementation available in release 8.0.n.n.n will remain installed and accessible.
4. While connected as SYSTEM, enter the following:

```
ALTER DATABASE RESET COMPATIBILITY
SHUTDOWN
Change  the init.ora parameter COMPATIBLE=8.0.0.0.0
STARTUP
```

After running *ORACLE_HOME*/MD/c813d805.sql, resetting the database compatibility to 8.1.n.n.n from 8.0.n.n.n requires running the script *ORACLE_HOME*/MD/c805u813.sql to reinstall and enable the object-relational implementation of Spatial.

B.3 Compatibility Details

All releases of the Spatial product provide a set of predefined spatial data types, topological operators such as RELATE, and a spatial indexing mechanism. The Oracle8i (release 8.1.5) Spatial release differed from pre-Oracle8*i* releases in that it:

- Used object types (a varray-based type called SDO_GEOMETRY to store ordinates)
- Supported new spatial data types, namely arcs and circles
- Had new spatial operators (SDO_WITHIN_DISTANCE) and functions (SDO_POLY_UNION, SDO_BUFFER, SDO_POLY_INTERSECT, and SDO_POLY_XOR)
- Utilized dynamic SQL in PL/SQL
- Allowed invoker’s rights
- Tessellated a geometry as a whole rather than one element at a time

All interfaces preceding Oracle8*i* are maintained, but the package bodies have been changed to use the preceding features. Thus, for Oracle8*i*, the Spatial packages must be reinstalled to use these interfaces even if the compatibility parameter is set to 8.0.

No data migration is needed, and the 7.3.4/8.0.4 spatial applications will work without modification. Any OCI-specific migration issues must be handled in the same manner as they would have to be for any OCI application.

The release 7.3.4/8.0.4 to 8.1.5 upgrade requirements are the same. Upgrade both Oracle8*i* and Spatial. Perform all the necessary steps for an upgrade. Your spatial applications will continue to work as before.

Downgrading from 8.0.5 or earlier releases to a previous release of the database server and Spatial requires no special steps specific to the Spatial implementation. However, this situation is different for Oracle8*i*. In Oracle8*i*, Spatial uses objects and extensible indexing. Therefore, it creates database objects specific to Oracle8*i* that are not compatible with previous releases of the database server. When you downgrade the database server and Spatial from Oracle8*i* to release 8.0.5, a spatial-specific downgrade script must be executed to remove all the spatial geometry type, index type, and spatial operator definitions.

B.4 Data Migration Issues

Beginning with release 7.3.3, all subsequent releases can work with spatial data from previous releases. That is, no data migration is required. The situation is different in Oracle8*i* because Spatial now allows two storage mechanisms. If you want the features specific to Oracle8*i*, such as extensible indexing and spatial operators, you must migrate your spatial data from the release 7.3.3 columns-of-numbers style to the SDO_GEOMETRY storage scheme. Spatial provides a stored procedure and sample code that demonstrates one way of migrating data and metadata.

Migrating data on downgrades is more complex. Spatial provides OCI demonstration programs to read SDO_GEOMETRY instances and store them in a release 8.0.5 spatial schema for comparable data. The demo also addresses issues related to the changes in the way metadata is stored in Oracle8*i* compared to previous releases. The complexity arises from the following:

- From release 7.3.4 onward, Spatial has an UNSUPPORTED_GEOMETRY type that is always used in conjunction with a bounding box or polygon, which is used for indexing purposes and which encloses the spatial object. This did not exist in release 7.3.3.

- From release 8.1.5 onward, Spatial supports arcs, circles, arc strings, and geometries made up of a mixture of arc and line segments.

You cannot store arcs and circles in any release earlier than Oracle8*i*, and you cannot use data from a release 7.3.4 or later spatial layer in release 7.3.3 if it contains instances of type UNSUPPORTED_GEOMETRY (etype=0).

B.5 Migrating from Spatial Release 8.1.5 or 8.1.6 to Release 8.1.7

To migrate from Spatial release 8.1.5 to release 8.1.7, you must first migrate from release 8.1.5 to release 8.1.6, as described in Section B.5.1, and then migrate to release 8.1.7, as described in Section B.5.2. Be sure that you perform any actions needed for data migration and compatibility for release 8.1.6 before migrating to release 8.1.7.

To migrate from Spatial release 8.1.6 to release 8.1.7, run the c816u817.sql script, as described in Section B.5.2.

B.5.1 Migrating from Spatial Release 8.1.5 to Release 8.1.6

Spatial release 8.1.5 uses objects and index types to create spatial indexes. However, currently there is no tool to convert release 8.1.5 spatial indexes to the new format for release 8.1.6. Therefore, you must delete all the spatial indexes built in release 8.1.5 database and re-create them in a release 8.1.6 database.

Follow these steps to upgrade from release 8.1.5 to release 8.1.6 of Spatial:

1. Make sure that the Oracle RDBMS is upgraded to release 8.1.6.
2. Find out how the current spatial indexes are built (including what the index parameters are), so that you can re-create all the indexes after the upgrade.

   To see which users have spatial indexes and the spatial index parameters, enter the following SQL statement:

   ```
   SELECT PARAMETERS, INDEX_NAME FROM user_indexes,
        sdo_index_metadata_table
        WHERE INDEX_NAME = SDO_INDEX_NAME;
   ```

   Save this information before upgrading to Spatial 8.1.6.
3. Run the following script: *ORACLE_HOME*/md/c815u816.sql

   This script deletes all the spatial indexes and installs types and packages related to Spatial 8.1.6. It also migrates the data from all the SDO_GEOM_METADATA

tables in each user's schema to the new xxx_SDO_GEOM_METADATA views managed by Spatial.

However, this metadata migration is done only for those layers that have a spatial index built on them. If you need to do this metadata migration separately, you can use the following SQL statement (once for each user with spatial data):

```
INSERT INTO USER_SDO_GEOM_METADATA
   SELECT TABLE_NAME, COLUMN_NAME, DIMINFO, NULL
   FROM SDO_GEOM_METADATA;
```

For example, if you connect as user Herman and execute that statement, it migrates all the metadata in user Herman's SDO_GEOM_METADATA table.

4. Migrate the geometry objects to release 8.1.6-style types by executing the SDO_MIGRATE.FROM_815_TO_81X procedure on each table that has geometry data. (This step is strongly recommended, as explained in Section B.5.1.1.)
5. Manually re-create all the spatial indexes that are required.

B.5.1.1 Data Migration to Release 8.1.6

Spatial release 8.1.6 introduced new SDO_GTYPE and SDO_ETYPE values to better manage the geometry data. To take advantage of these benefits, it is strongly recommended that the data be migrated to the new SDO_GTYPE and SDO_ETYPE values.

To initiate this data migration, execute the SDO_MIGRATE.FROM_815_TO_81X procedure on each table that has geometry data. This procedure updates all the geometries to set the SDO_GTYPE and SDO_ETYPE values.

B.5.1.2 Compatibility Between Releases 8.1.5 and 8.1.6

Spatial release 8.1.6 changed the way the geometry metadata is managed. In release 8.1.5, the metadata is managed by the users by keeping the metadata in SDO_GEOM_METADATA tables in each user's schema. In release 8.1.6, the metadata is centrally managed under the MDSYS schema, and the user can access and manipulate the metadata through metadata views.

A release 8.1.5 Spatial application will fail against a release 8.1.6 database if it tries to access the metadata. Therefore, if you need to run a release 8.1.5 application with a release 8.1.6 database, you need to keep the SDO_GEOM_METADATA table and make sure that the USER_SDO_GEOM_METADATA view and the SDO_GEOM_METADATA table are consistent all the time.

A recommended method for ensuring consistency is to create SQL triggers on the SDO_GEOM_METADATA table to perform a corresponding insert, update, or delete operation on USER_SDO_GEOM_METADATA for each such operation on SDO_GEOM_METADATA. For example, if user SCOTT has spatial data and needs to keep the SDO_GEOM_METADATA table consistent with the new metadata views, SCOTT can create a trigger (shown in Example B–1) that inserts data into the USER_SDO_GEOM_METADATA view whenever SCOTT inserts data into the SDO_GEOM_METADATA table.

Example B–1 Insert Trigger for Metadata Consistency

```
CREATE TRIGGER scott_sdo_meta_ins_trig
before insert on sdo_geom_metadata
referencing new as n
FOR EACH ROW
BEGIN

 INSERT INTO user_sdo_geom_metadata
 VALUES(:n.table_name, :n.column_name, :n.diminfo, NULL);
END;
/
```

User SCOTT can create similar triggers for delete and update operations on the SDO_GEOM_METADATA table.

B.5.2 Migrating from Spatial Release 8.1.6 to Release 8.1.7

To migrate from Spatial release 8.1.6 to release 8.1.7, run the following script: *ORACLE_HOME*/md/c816u817.sql

C

Generic Geocoding Interface

This appendix 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.

C.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.

C.2 Generic Geocoding Client

A fast, scalable, highly available, and secure Java Virtual Machine (Java VM, or JVM) is integrated in the Oracle8*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 Oracle8*i* object types.

Therefore, any client geocoder component written in Java can be embedded in the Oracle8*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

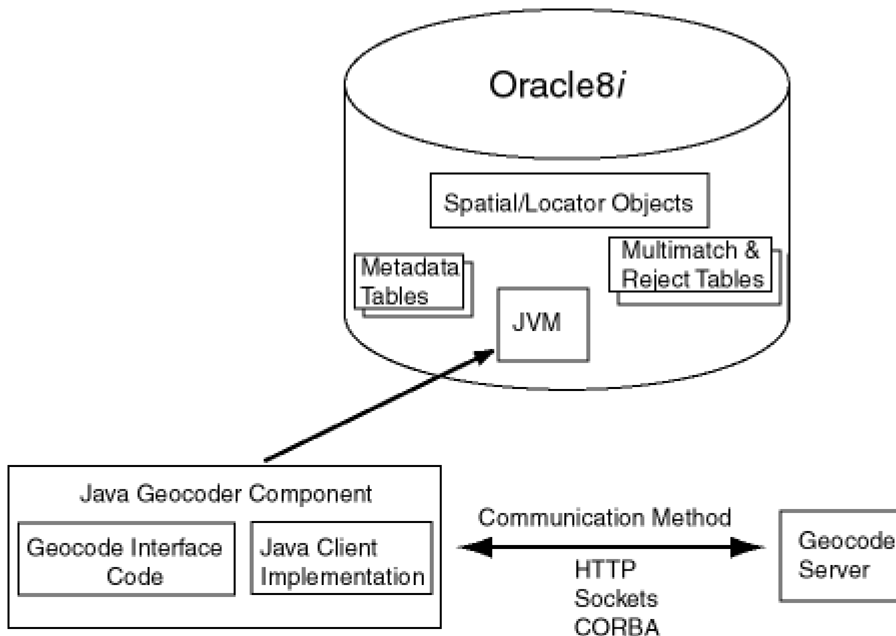
defined and metadata tables to be populated. The object types, metadata schema, and the geocoder interface are described in further detail in the following sections.

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 C–1 shows the Oracle geocoding framework.

Figure C–1 Oracle Geocoding Framework



C.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 C.3.1) and a schema object (described in Section C.3.2).
- Each object is identified by a unique *task_id* value.

C.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.

C.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.

C.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)
);
```

C.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.

C.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 C.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.

C.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.

C.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 Oracle8*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.

D

Coordinate Systems (Spatial Reference Systems)

This appendix describes the coordinate system transformation capabilities of Oracle Spatial. The coordinate systems application programming interface (API) integrates support into Oracle8*i* for storing and manipulating SDO_GEOMETRY objects in a variety of coordinate systems. (Coordinate systems are sometimes called *spatial reference systems*.)

For reference information about coordinate systems functions and procedures, see Chapter 8.

D.1 Why Integrate Coordinate System Information?

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. For definitions of SDO_GEOMETRY objects, users were instructed to set the SDO_SRID value (intended for future coordinate system support use) to a null value, and in fact this instruction appeared in the *Oracle8i Spatial User's Guide and Reference* for release 8.1.6. The Spatial functions and operators always assumed a coordinate system that had the properties of an orthogonal Cartesian system. With such a system, if Earth-based geometries are stored in latitude and longitude coordinates, Spatial functions and operators sometimes do not provide correct results in these coordinates.

With coordinate system support in Oracle Spatial, you can freely convert data from one coordinate system to another coordinate system, and Spatial functions, operators, and utilities provide correct and unambiguous results in whatever coordinate system the data is stored, particularly relating to measurements on the Earth's surface. Moreover, Spatial operators for queries and joins perform accurate

computations with data that uses different coordinate systems. (However, see Section D.5 for any restrictions and problems in the current release.)

D.2 Terms and Concepts

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

D.2.1 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.

D.2.2 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.

D.2.3 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 D.2.5).

D.2.4 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.

D.2.5 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.

D.2.6 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.

D.2.7 Transformation (Datum Transformation)

Transformation, specifically datum transformation, is 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.

D.3 Coordinate Systems Data Structures

The coordinate systems functions and procedures use information provided in a table and other objects supplied with Oracle Spatial:

- A table, MDSYS.CS_SRS, defines the valid coordinate systems.
- Other data structures define the valid map projections and ellipsoids.

The MDSYS.CS_SRS table 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`).

D.3.1 MDSYS.CS_SRS Table

The MDSYS.CS_SRS reference table is included with Oracle Spatial, and it is used by coordinate systems functions and procedures. This table contains over 900 rows, one for each valid coordinate system.

> **Note:** You should not modify, delete, or add any information in the MDSYS.CS_SRS table. (Support is planned for user-defined coordinate systems in a future release.)

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

Table D–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.

Table D–1 MDSYS.CS_SRS Table (Cont.)

Column Name	Data Type	Description
SRID	INTEGER	The unique ID number (Spatial Reference ID) for a coordinate system.
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 D.3.1.1.
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.

D.3.1.1 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. 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.

D.3.2 Other Objects

Underlying the CS_SRS table are data and code to represent the ellipsoids and projections in common use around the world. Table D–2 lists the supported map projections.

Table D–2 Supported Map Projections

ID	Projection Name	ID	Projection Name
0	Geographic (longitude/latitude)	3	Albers Conical Equal Area
4	Lambert Conformal Conic	5	Mercator
7	Polyconic	8	Equidistant Conic
9	Transverse Mercator	10	Stereographic
11	Lambert Azimuthal Equal Area	12	Azimuthal Equidistant
13	Gnomonic	14	Orthographic
15	General Vertical Near-Side Perspective	16	Sinusiodal
17	Equirectangular	18	Miller Cylindrical
19	Van der Grinten	20	Hotine Oblique Mercator
21	Robinson	22	Space Oblique Mercator
23	Alaska Conformal	24	Interrupted Goode Homolosine
25	Mollweide	26	Interrupted Mollweide
27	Hammer	28	Wagner IV
29	Wagner VII	30	Oblated Equal Area
31	Non-Earth	32	Transverse Mercator Danish System 45 Bornholm

Table D–2 Supported Map Projections (Cont.)

ID	Projection Name	ID	Projection Name
33	Transverse Mercator Danish System 34 Jylland-Fyn	34	Transverse Mercator Sjaelland
35	Transverse Mercator Finnish KKJ	36	Eckert IV
37	Eckert VI	38	Gall
39	Lambert Conformal Conic (Belgium 1972)	40	New Zealand Map Grid
41	Cylindrical Equal Area	42	Swiss Oblique Mercator
43	Bonne	44	Cassini

Table D–3 lists the supported ellipsoids.

Table D–3 Supported Ellipsoids

ID	Ellipsoid Name	ID	Ellipsoid Name
0	Clarke 1866	1	WGS 72
2	Australian	3	Krassovsky
4	International 1924	5	Hayford
6	Clarke 1880	7	GRS 80
8	Clarke 1866 (Michigan)	9	Airy 1930
10	Bessel 1841	11	Everest
12	Sphere	13	Airy 1930 (Ireland 1965)
14	Bessel 1841 (Schwarzeck)	15	Clarke 1880 (Arc 1950)
16	Clarke 1880 (Merchich)	17	Everest (Kertau)
18	Fischer 1960 (Mercury)	19	Fischer 1960 (South Asia)
20	Fischer 1968	21	GRS 67
22	Helmert 1906	23	Hough
24	South American 1969	25	War Office
26	WGS 60	27	WGS 66
28	WGS 84	29	Clarke 1880 (IGN)
30	IAG 75	31	MERIT 83

Table D–3 Supported Ellipsoids (Cont.)

ID	Ellipsoid Name	ID	Ellipsoid Name
32	New International 1967	33	Walbeck
34	Bessel 1841 (NGO 1948)	35	Clarke 1858
36	Clarke 1880 (Jamaica)	37	Clarke 1880 (Palestine)
38	Everest (Timbalai)	39	Everest (Kalianpur)
40	Indonesian	41	NWL 9D
42	NWL 10D	43	OSU86F
44	OSU91A	45	Plessis 1817
46	Struve 1860	48	Sphere (Unity)

D.4 Coordinate Systems Functions and Procedures

The current release of Oracle Spatial includes the following functions and procedures:

- 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).

Reference information about these functions and procedures is in Chapter 8.

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

D.5 Restrictions and Problems in the Current Release

The current release of Oracle Spatial provides the first phase of support for coordinate systems. Further support is planned for future releases.

The following restrictions and problems apply to the current release.

D.5.1 Geometries with Longitude and Latitude Coordinates

In the current release, Spatial functions and operators do not necessarily return precisely correct results with geometries whose coordinates are expressed as longitude and latitude values. 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.

As a workaround, first transform the geometries of interest to a projection coordinate system that is conformant to the local space of the geometries. Then, use the Spatial functions and operators with the transformed geometries.

In a future release, support is planned for correct results in all cases with Spatial functions and operators using geometries with longitude/latitude coordinates.

D.6 Example of Coordinate Systems

This section presents a simplified example that uses coordinate system functions and procedures. It refers to concepts that were explained in this appendix and uses functions documented in Chapter 8.

Example D–1 uses the same geometry data (cola markets) as in Section 2.1, except that instead of null SRID values, the 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 D–1 uses the geometries illustrated in Figure 2–1 in Section 2.1.

Example D–1 does the following:

- 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*), using the SRID value 8307
- Updates the USER_SDO_GEOM_METADATA view to reflect the dimension of the areas, using the SRID value 8307
- Creates a spatial index (*cola_spatial_idx*)
- Performs some transformation operations (single geometry and entire layer)

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

Example D–1 Simplified Example of Coordinate Systems

```
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
    8307,  -- SRID for 'Longitude / Latitude (WGS 84)' coordinate system
    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)
  )
);

-- 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
    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 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)
  )
);

-- 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
    8307,
    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 (i.e., table-column combination; here: cola_markets and shape).

INSERT INTO USER_SDO_GEOM_METADATA
  VALUES (
  'cola_markets',
  'shape',
  MDSYS.SDO_DIM_ARRAY(   -- 20X20 grid, virtually zero tolerance
    MDSYS.SDO_DIM_ELEMENT('X', 0, 20, 0.005),
    MDSYS.SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
     ),
  8307   -- SRID for ''Longitude / Latitude (WGS 84)' coordinate system
);

-------------------------------------------------------------------
-- CREATE THE SPATIAL INDEX --
-------------------------------------------------------------------
CREATE INDEX cola_spatial_idx
ON cola_markets(shape)
```

```
INDEXTYPE IS MDSYS.SPATIAL_INDEX
PARAMETERS('SDO_LEVEL = 8');

--------------------------------------------------------------------
-- TEST COORDINATE SYSTEMS FUNCTIONS AND PROCEDURES --
--------------------------------------------------------------------

-- 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 c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' 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 c, user_sdo_geom_metadata m
  WHERE m.table_name = 'COLA_MARKETS' AND m.column_name = 'SHAPE'
  AND c.name = 'cola_c';

-- Transform the entire SHAPE layer and put results in the table
-- named cola_markets_8199, which the procedure will create.
EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS','SHAPE','COLA_MARKETS_8199',8199);

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

-- Select all from the new (layer transformed) table.
SELECT * from cola_markets_8199;

-- Show metadata for the new (layer transformed) table.
DESCRIBE cola_markets_8199;
```

Example D–2 shows the output of the SELECT statements in Example D–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, 7) and (1.00078606, 1.00272755, 5.00069866, 7.00321633) for *cola_a*.

Example D–2 Output of SELECT Statements in Coordinate Systems Example

```
SQL> -- Return the transformation of cola_c using to_srid 8199 ('Longitude /
Latitude (Arc 1950)')
SQL> SELECT c.name, SDO_CS.TRANSFORM(c.shape, m.diminfo, 8199)
  2    FROM cola_markets c, user_sdo_geom_metadata m
```

```
  3    WHERE m.table_name = 'COLA_MARKETS' 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(3.00074116, 3.00289624, 6.0006707, 3.00289431, 6.00067234, 5.00305745, 4.0007
1964, 5.00305956, 3.00074116, 3.00289624))


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 c, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'COLA_MARKETS' 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(3.00074116, 3.00289624, 6.0006707, 3.00289431, 6.00067234, 5.00305745, 4.0007
1964, 5.00305956, 3.00074116, 3.00289624))


SQL> -- Transform the entire SHAPE layer and put results in the table
SQL> -- named cola_markets_8199, which the procedure will create.

SQL> EXECUTE SDO_CS.TRANSFORM_LAYER('COLA_MARKETS',  'SHAPE',  'COLA_MARKETS_
8199', 8199);

PL/SQL procedure successfully completed.

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

    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, 3), SDO_ORDINATE_ARR
AY(1, 1, 5, 7))

         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, 4), SDO_ORDINATE_ARR
AY(8, 7, 10, 9, 8, 11))


SQL>
SQL> -- Select all from the new (layer transformed) table.

SQL> SELECT * from cola_markets_8199;

SDO_ROWID
------------------
GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
AAAA1BAABAAACcHAAA
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3), SDO_ORDINATE_ARR
AY(1.00078606, 1.00272755, 5.00069866, 7.00321633))

AAAA1BAABAAACcHAAB
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(5.00069355, 1.00272665, 8.00062193, 1.00272605, 8.00062526, 6.00313458, 5.000
69866, 7.00321633, 5.00069355, 1.00272665))


SDO_ROWID
------------------
```

```
GEOMETRY(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
AAAA1BAABAAACcHAAC
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(3.00074116, 3.00289624, 6.0006707, 3.00289431, 6.00067234, 5.00305745, 4.0007
1964, 5.00305956, 3.00074116, 3.00289624))

AAAA1BAABAAACcHAAD
SDO_GEOMETRY(2003, 8199, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 4), SDO_ORDINATE_ARR
AY(8.00062651, 7.00321213, 10.0005803, 9.00335882, 8.00063347, 11.0035044))


SQL> -- Show metadata for the new (layer transformed) table.
SQL> DESCRIBE cola_markets_8199;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SDO_ROWID                                          ROWID
 GEOMETRY                                           MDSYS.SDO_GEOMETRY
```

D.7 Error Messages for Coordinate Systems

This section lists the coordinate systems error messages, including the cause and recommended user action for each.

ORA-13276 internal error [%s] in coordinate transformation

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

ORA-13278 failure to convert SRID to native format

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

ORA-13281 failure in execution of sql statement to retrieve WKT

Cause: OCI internal error or SRID does not match a table entry.

Action: Check that a valid SRID is being used.

ORA-13282 failure on initialization of coordinate transformation

Cause: Parsing error on source or destination coordinate system WKT or incompatible coordinate systems.

Action: Check validity of WKT for table entries and the legitimacy of the requested transformation.

ORA-13283 failure to get new geometry object for conversion in place

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

ORA-13284 failure to copy geometry object for conversion in place

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

ORA-13285 Geometry coordinate transformation error

Cause: A coordinate pair was out of valid range for a conversion/projection.

Action: Check that data to be transformed is consistent with the desired conversion/projection.

ORA-13287 can't transform unknown gtype

Cause: A geometry with an SDO_GTYPE value of <= 0 was specified for transformation. Only an SDO_GTYPE value >= 1 is allowed.

Action: Specify an appropriate SDO_GTYPE value.

ORA-13288 Point coordinate transformation error

Cause: An internal error occurred while transforming points.

Action: Check the accompanying error messages.

ORA-13294 Cannot transform geometry containing circular arcs

Cause: It is impossible to transform a 3-point representation of a circular arc without distortion.

Action: Make sure a geometry does not contain circular arcs.

ORA-13300 Single point transform error

Cause: Low-level coordinate transformation error trap.

Action: Check the accompanying error messages.

ORA-13303 failure to retrieve a geometry object from a table

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

ORA-13304 failure to insert a transformed geometry object in a table

Cause: OCI internal error.

Action: Contact Oracle Support Services with the exact error message text.

E

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 E.5. However, you may want to read the rest of this appendix first, to understand the concepts that the example illustrates.

For reference information about LRS functions, see Chapter 9.

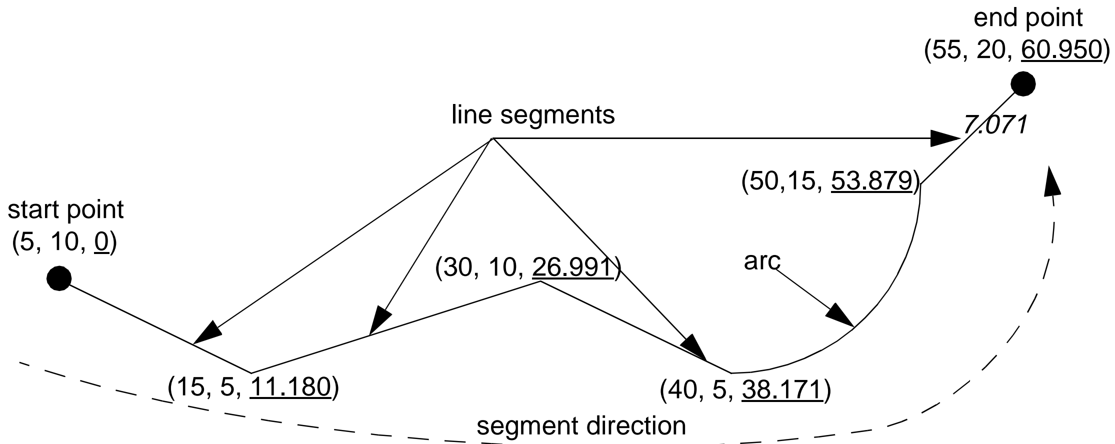
E.1 Terms and Concepts

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

E.1.1 Geometric Segments (LRS Segments)

Geometric segments are basic LRS elements in Oracle Spatial. They are Oracle line string geometries. An Oracle line string is an ordered, non-branching, and continuous geometry. 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 E–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 E–1).

Figure E–1 Geometric Segment



E.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 the LRS segment is defined.

E.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. Measures of points on a geometric segment always increase along the direction of the geometric segment.

E.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 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.

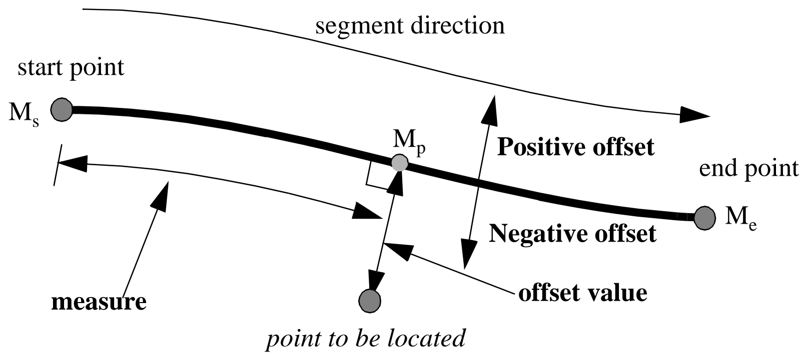
Some LRS functions use *offset* instead of measure to represent measured distance along linear features. Although some other linear referencing systems might use offset to mean what the Oracle Spatial LRS refers to as measure, offset has a different meaning in Oracle Spatial from measure, as explained in Section E.1.5.

E.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.

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



E.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 E–3 and the example in Figure E–4.

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

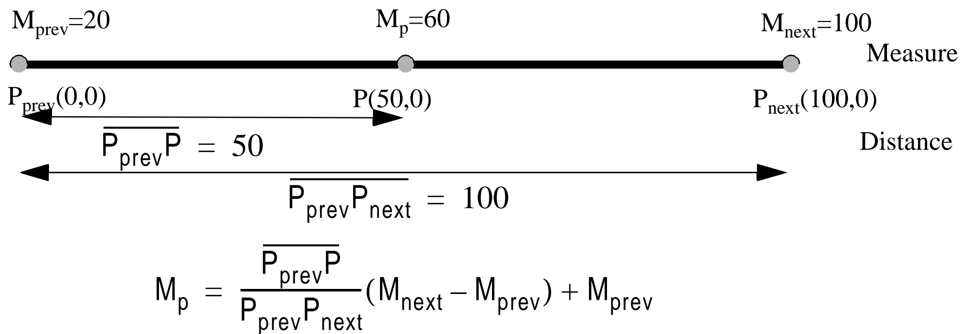
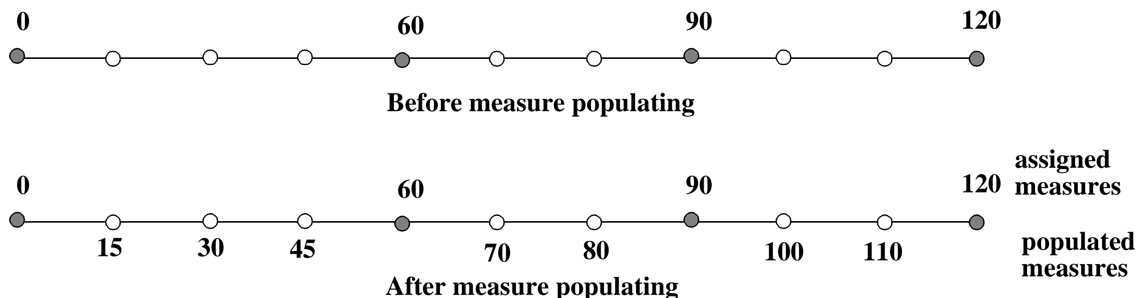


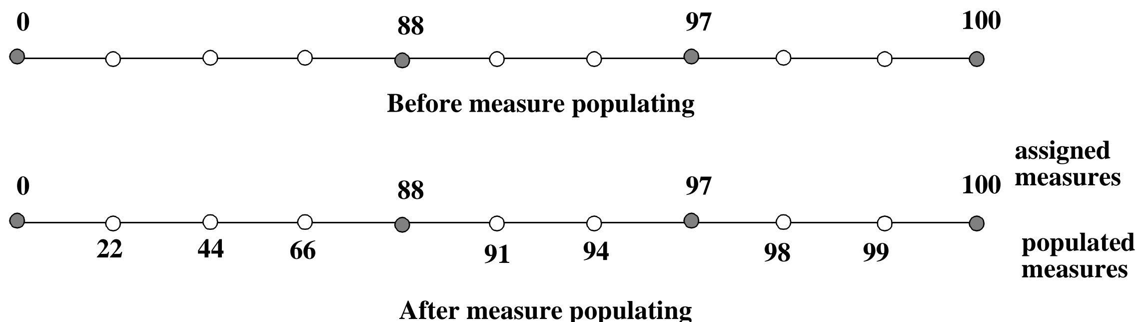
Figure E–4 Measure Populating of a Geometric Segment



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 E–5 shows the measure population that results when assigned measure values are not proportional and reflect widely varying gaps.

Figure E–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.

E.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.

E.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.

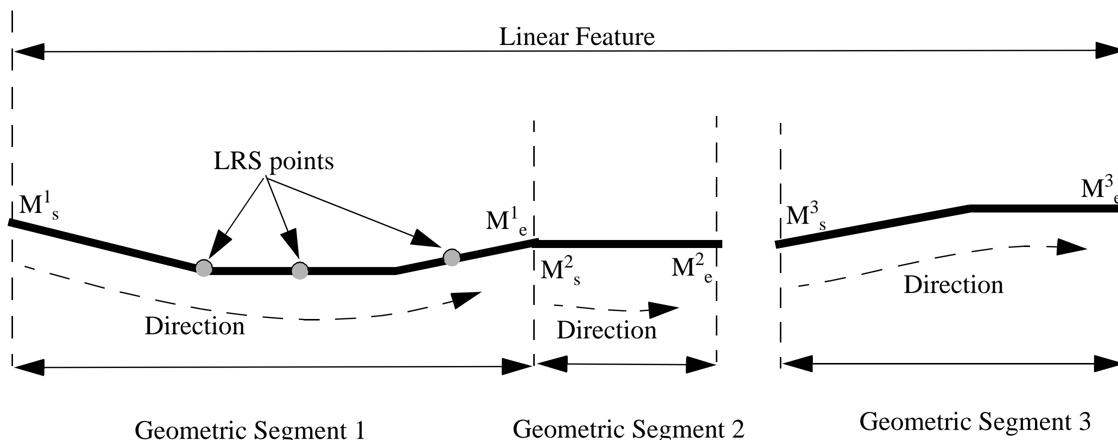
E.1.9 LRS Point

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

E.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 E–6.

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



E.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.

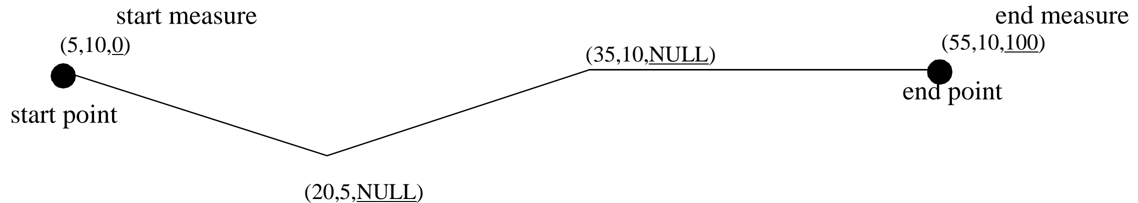
Oracle Spatial LRS support affects the Spatial metadata and data (the geometries). Example E–1 shows how a measure dimension can be added to 2-dimensional geometries in the Spatial metadata. The measure dimension must be the last element of the SDO_DIM_ARRAY in a spatial object definition (shown in bold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

Example E–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 E–7.

Figure E–7 Creating a Geometric Segment



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

```
SDO_GEOMETRY(3002, 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 or created, 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 supports the object-relational model of Oracle Spatial. 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 3001.

E.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.

Information about the CREATE INDEX statement and its parameters and keywords is in Chapter 5.

E.4 LRS Operations

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

E.4.1 Defining a Geometric Segment

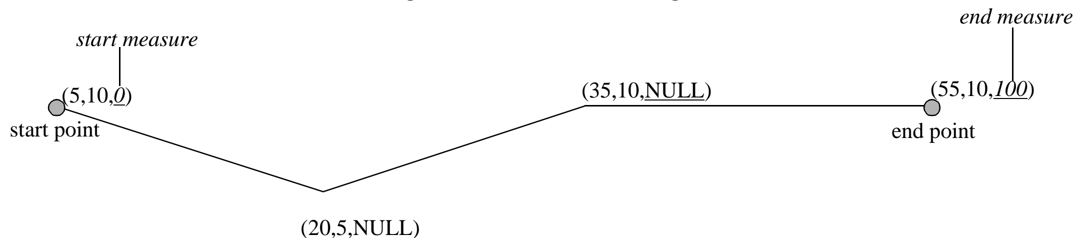
There are two ways to create a geometric segment with measure information:

- 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 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.

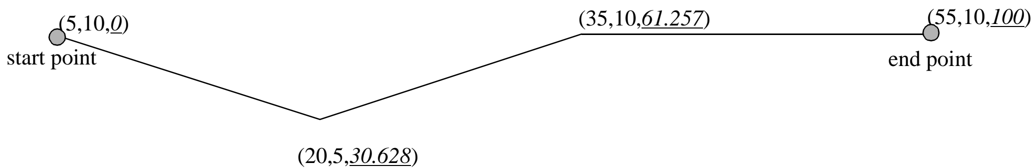
Figure E–8 shows different ways of defining a geometric segment.

Figure E–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

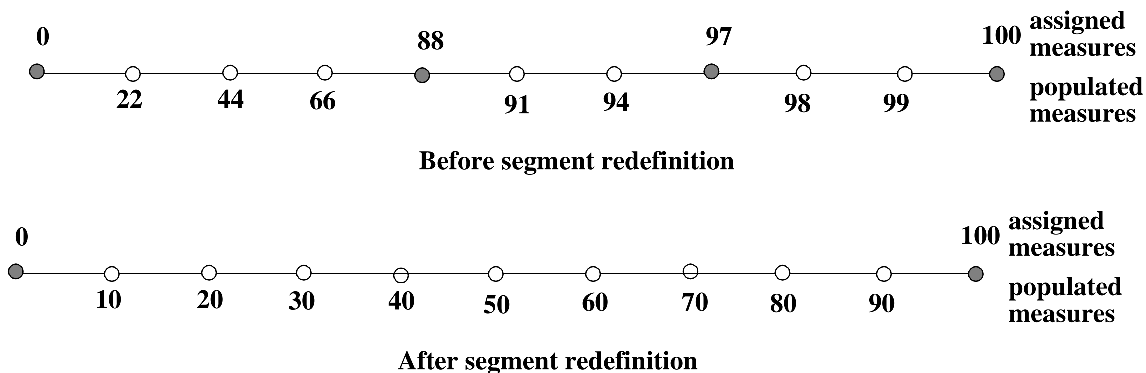
An LRS segment must be defined before any LRS operations can proceed. That is, the start, end, and any other assigned measures must be present to derive the location from a specified measure. The measure information of intermediate shape points will automatically be populated if they are not assigned.

E.4.2 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 E–9 shows the redefinition of a segment where the existing (before) assigned measure values are not proportional and reflect widely varying gaps.

Figure E–9 Redefining a Geometric Segment

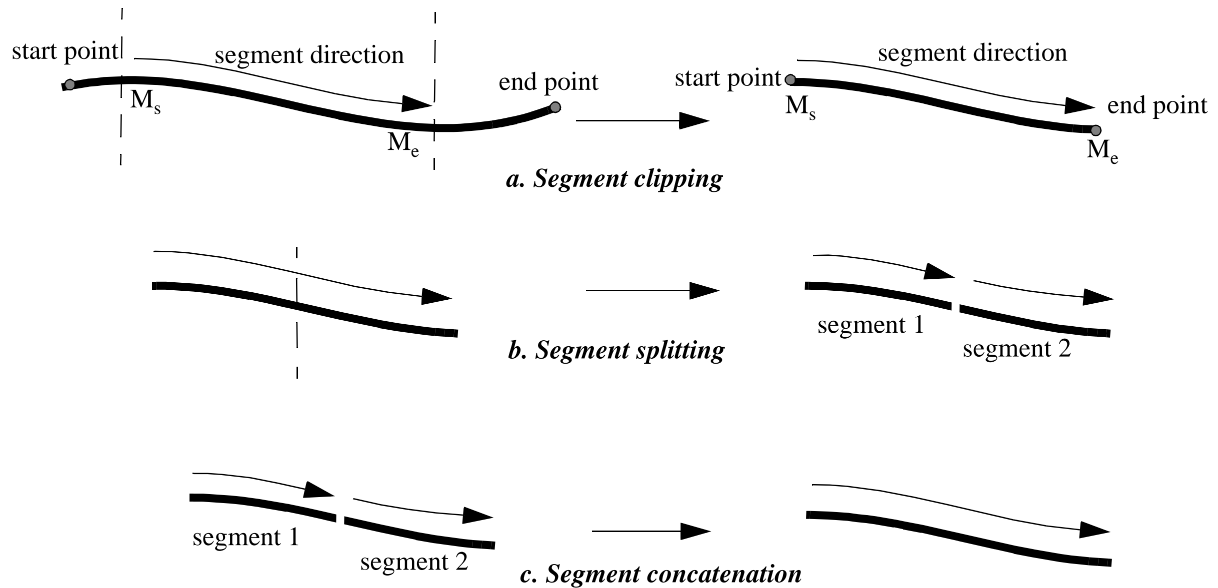


After the segment redefinition in Figure E–9, the populated measures reflect proportional distances along the segment.

E.4.3 Clipping a Geometric Segment

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

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



E.4.4 Splitting a Geometric Segment

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

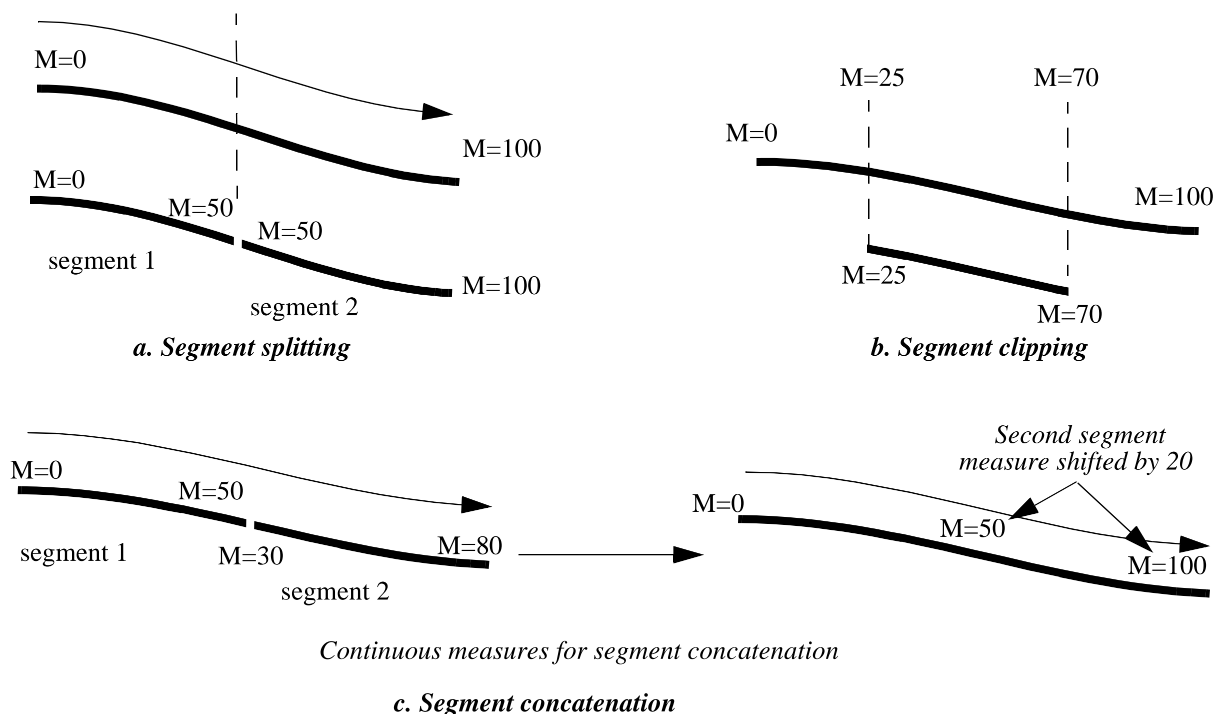
Note: In Figure E–10 and several that follow, small gaps between segments are used in illustrations of segment splitting and concatenation. Each gap simply reinforces the fact that two different segments are involved. However, the two segments (such as segment 1 and segment 2 in Figure E–10, parts b and c) are actually connected. The tolerance (see Section 1.5.4) is considered in determining whether or not segments are connected.

E.4.5 Concatenating Two Connected Geometric Segments

You can create a new geometric segment by concatenating two geometric segments (Figure E–10, part c). Note that the geometric segments must be spatially connected. 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.

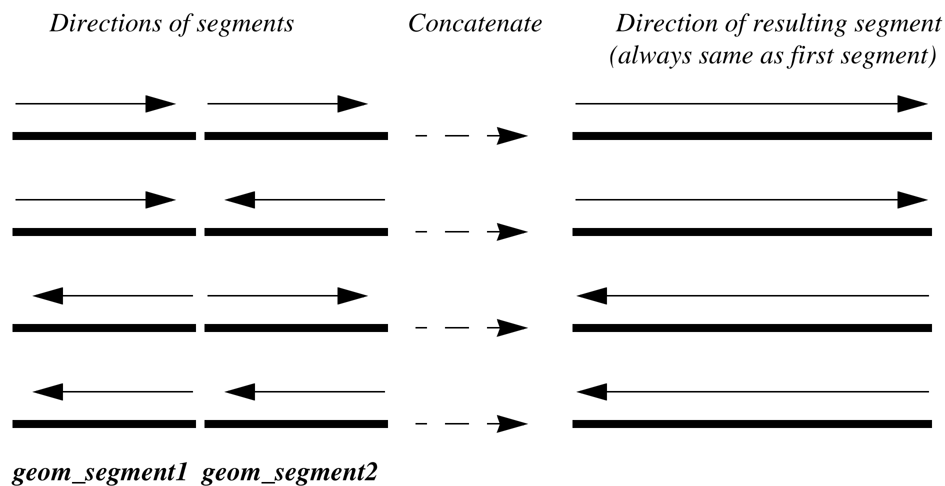
Measure assignments for the clipping, splitting, and concatenating operations in Figure E–10 are shown in Figure E–11. Measure information and segment direction are preserved in a consistent manner. The assignment is done automatically when the operations have completed.

Figure E–11 Measure Assignment in Geometric Segment Operations



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 E–12.

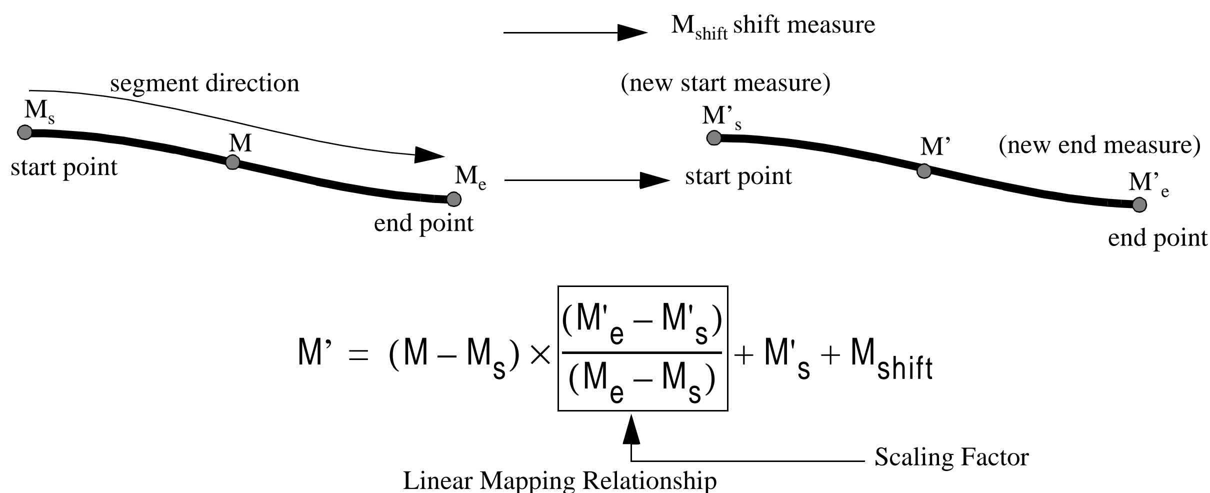
Figure E–12 Segment Direction with Concatenation



E.4.6 Scaling a Geometric Segment

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

Figure E–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:

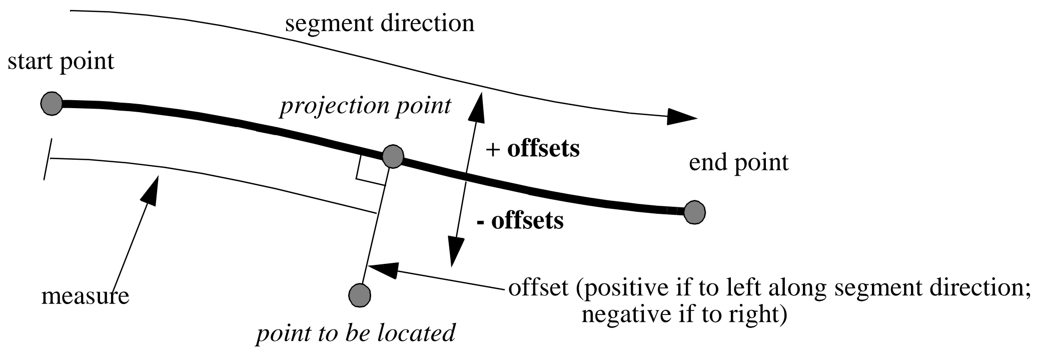
- Translating (shifting) measure information. (For example, add the same value to Ms and Me to get M's and M'e.)
- Reversing measure information. (Let M's = Me, M'e = Ms, and Mshift = 0.)
- Performing simple scaling of measure information. (Let Mshift = 0.)

For examples of these operations, see usage notes and examples for the SDO_LRS.SCALE_GEOM_SEGMENT function in Chapter 9.

E.4.7 Locating a Point on a Geometric Segment

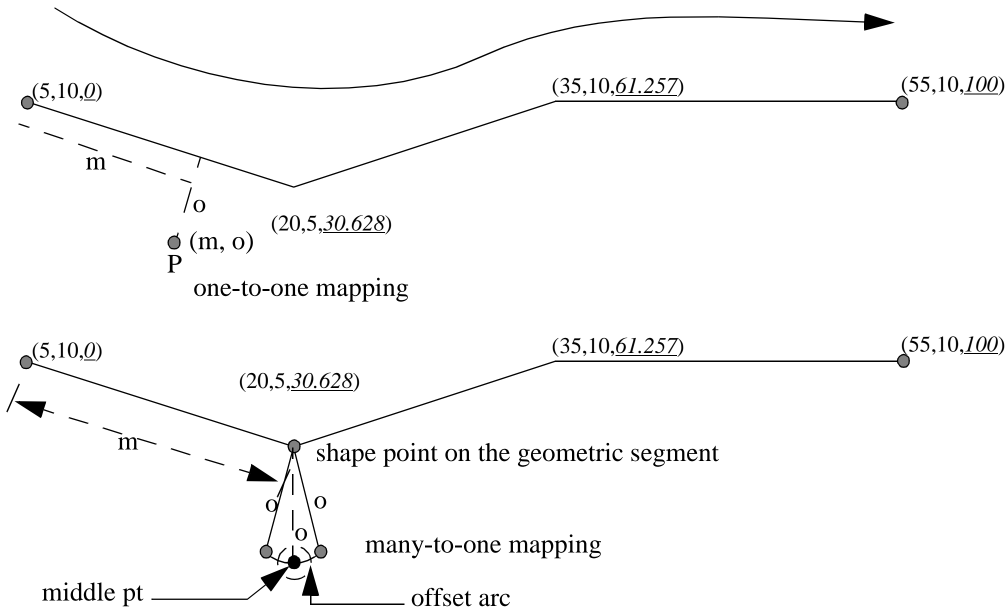
You can find the position of a point described by a measure and an offset on a geometric segment (see Figure E–14).

Figure E–14 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 E–15).

Figure E–15 Ambiguity in Location Referencing with Offsets



As shown in Figure E–15, 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

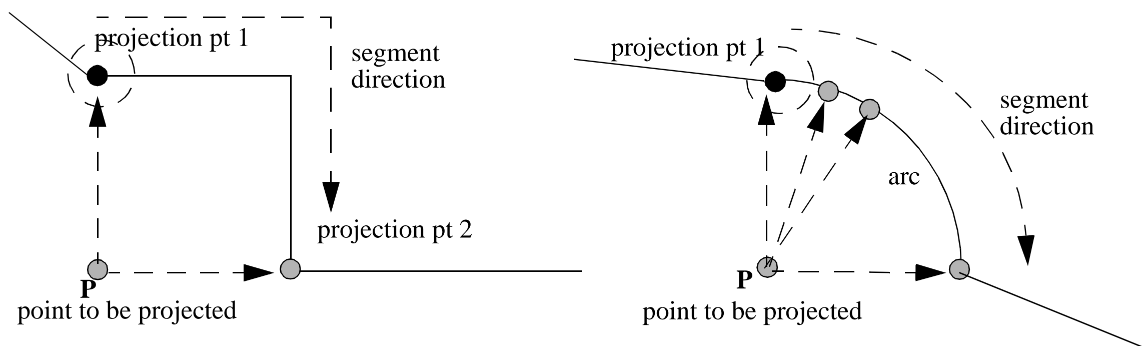
result, all points on the offset arc are represented by the same (measure, offset) pair. To resolve this one-to-many mapping problem, the middle point on the offset arc is returned.

E.4.8 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 E–14. 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 E–15). 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 E–16).

Figure E–16 Multiple Projection Points



E.4.9 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)

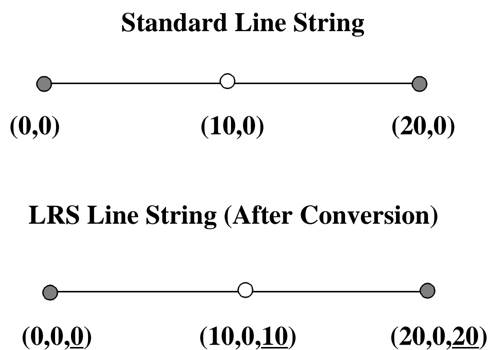
  For conversion from standard format to LRS format, a measure dimension (named *M* by default) is added, but no measure information is provided for each point. For conversion from LRS format to standard format, the measure dimension and information are removed. In both cases, the dimensional information (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 E–17 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 E–17.

Figure E–17 Conversion from Standard to LRS Line String



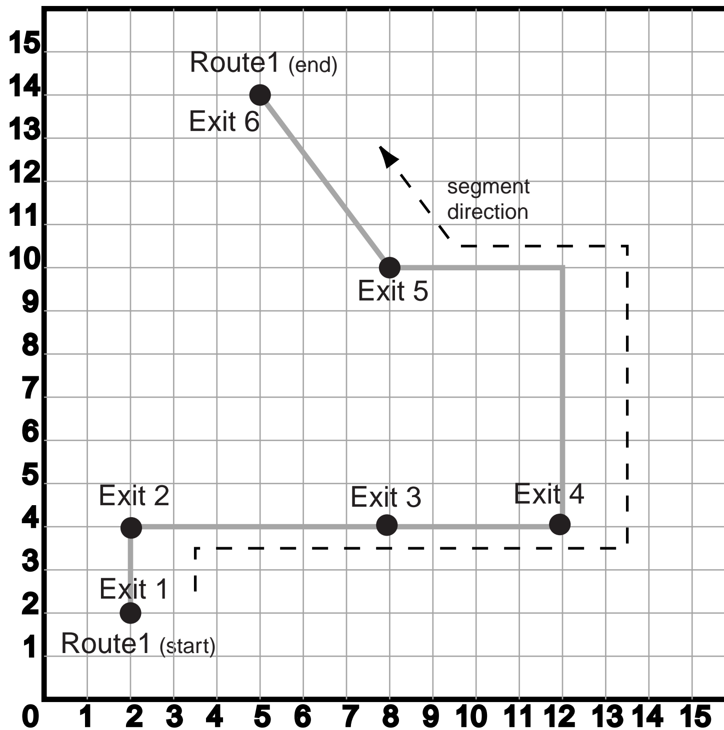
The conversion functions are listed in Table 9–3 in Chapter 9. See also the reference information in Chapter 9 about each conversion function.

E.5 Example

This section presents a simplified example that uses LRS functions. It refers to concepts that were explained in this chapter and uses functions documented in Chapter 9.

This example uses the road that is illustrated in Figure E–18.

Figure E–18 Simplified LRS Example: Highway



In Figure E–18, 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 E–1 lists an actual highway-related feature and the LRS feature that corresponds to it or that can be used to represent it.

Table E–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 E–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

Example E–2 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 E–3 includes the output of the SELECT statements in Example E–2.

Example E–2 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(
    3002,  -- 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 will be 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 (reserved for future Spatial releases)
);

-- 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';

-- 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;
/

-- 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, m.diminfo,
                                        b.route_geometry, m.diminfo)
  FROM lrs_routes a, lrs_routes b, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 11 AND b.route_id = 12;

-- Is the Route1 segment valid?
SELECT  SDO_LRS.VALID_GEOM_SEGMENT(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
```

```
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Is 50 a valid measure on Route1? (Should return FALSE; highest Route1 measure
is 27.)
SELECT  SDO_LRS.VALID_MEASURE(a.route_geometry, m.diminfo, 50)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Is the Route1 segment defined?
SELECT  SDO_LRS.IS_GEOM_SEGMENT_DEFINED(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- How long is Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- What is the start measure of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- What is the end measure of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_END_MEASURE(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- What is the start point of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- What is the end point of Route1?
SELECT  SDO_LRS.GEOM_SEGMENT_END_PT(a.route_geometry, m.diminfo)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.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;

-- Reverse direction (for example, to concatenate with another road)
SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo,27, 0, 0)
```

```
  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(a.route_geometry, m.diminfo,
                                       0, 43.443, 0)
    FROM lrs_routes a, user_sdo_geom_metadata m
    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Clip a piece of Route1.
SELECT  SDO_LRS.CLIP_GEOM_SEGMENT(a.route_geometry, m.diminfo, 5, 10)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

-- Point (9,3,NULL) is off the road; should return (9,4,9).
SELECT  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)) )
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.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(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 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(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 a.route_id = 1;

-- Locate the point on Route1 at measure 9, offset 0.
SELECT  SDO_LRS.LOCATE_PT(a.route_geometry, m.diminfo, 9, 0)
  FROM lrs_routes a, user_sdo_geom_metadata m
  WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;
```

Example E–3 shows the output of the SELECT statements in Example E–2.

Example E–3 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(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))

        11 result_geom_1
SDO_GEOMETRY(3002, 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(3002, 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(3002, 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, m.diminfo,
  2         b.route_geometry, m.diminfo)
  3    FROM lrs_routes a, lrs_routes b, user_sdo_geom_metadata m
  4    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 11 AND b.route_id =
12;

SDO_LRS.CONNECTED_GEOM_SEGMENTS(A.ROUTE_GEOMETRY,M.DIMINFO,B.ROUTE_GEOMETRY,M.DI
--------------------------------------------------------------------------------
```

```
TRUE

SQL>
SQL> -- Is the Route1 segment valid?
SQL> SELECT  SDO_LRS.VALID_GEOM_SEGMENT(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.VALID_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE

SQL>
SQL> -- Is 50 a valid measure on Route1? (Should return FALSE; highest Route1
measure is 27.)
SQL> SELECT  SDO_LRS.VALID_MEASURE(a.route_geometry, m.diminfo, 50)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.VALID_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO,50)
--------------------------------------------------------------------------------
FALSE

SQL>
SQL> -- Is the Route1 segment defined?
SQL> SELECT  SDO_LRS.IS_GEOM_SEGMENT_DEFINED(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.IS_GEOM_SEGMENT_DEFINED(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------------------------
TRUE

SQL>
SQL> -- How long is Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_LENGTH(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.GEOM_SEGMENT_LENGTH(A.ROUTE_GEOMETRY,M.DIMINFO)
-------------------------------------------------------
                                                     27

SQL>
SQL> -- What is the start measure of Route1?
```

```
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_START_MEASURE(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.GEOM_SEGMENT_START_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO)
--------------------------------------------------------------
                                                             0

SQL>
SQL> -- What is the end measure of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_END_MEASURE(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.GEOM_SEGMENT_END_MEASURE(A.ROUTE_GEOMETRY,M.DIMINFO)
------------------------------------------------------------
                                                          27

SQL>
SQL> -- What is the start point of Route1?
SQL> SELECT  SDO_LRS.GEOM_SEGMENT_START_PT(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.GEOM_SEGMENT_START_PT(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, S
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 0, 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(a.route_geometry, m.diminfo)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.GEOM_SEGMENT_END_PT(A.ROUTE_GEOMETRY,M.DIMINFO)(SDO_GTYPE, SDO_SRID, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 0, 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 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))


SQL>
SQL> -- Reverse direction (for example, to concatenate with another road)
SQL> SELECT  SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo,27, 0, 0)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,27,0,0)(SDO_GTYPE, SDO_SRI
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 2, 1), SDO_ORDINATE_ARRAY(
5, 14, 0, 8, 10, 5, 12, 10, 9, 12, 4, 15, 8, 4, 19, 2, 4, 25, 2, 2, 27))


SQL> -- "Convert" mile measures to kilometers (27 * 1.609 = 43.443)
SQL> SELECT     SDO_LRS.SCALE_GEOM_SEGMENT(a.route_geometry, m.diminfo, 0, 43.44
3, 0)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.SCALE_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,0,43.443,0)(SDO_GTYPE, SDO
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, 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(a.route_geometry, m.diminfo, 5, 10)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.CLIP_GEOM_SEGMENT(A.ROUTE_GEOMETRY,M.DIMINFO,5,10)(SDO_GTYPE, SDO_SRID,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3002, 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(a.route_geometry, m.diminfo,
  2    MDSYS.SDO_GEOMETRY(3001, NULL, NULL,
  3    MDSYS.SDO_ELEM_INFO_ARRAY(1, 1, 1),
  4    MDSYS.SDO_ORDINATE_ARRAY(9, 3, NULL)) )
  5    FROM lrs_routes a, user_sdo_geom_metadata m
  6    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.PROJECT_PT(A.ROUTE_GEOMETRY,M.DIMINFO,MDSYS.SDO_GEOMETRY(3001,NULL,NULL,
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, 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(3001, 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 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(3001, 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 a.route_id = 1;

SDO_LRS.VALID_LRS_PT(MDSYS.SDO_GEOMETRY(3001,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(a.route_geometry, m.diminfo, 9, 0)
  2    FROM lrs_routes a, user_sdo_geom_metadata m
  3    WHERE m.table_name = 'LRS_ROUTES' AND a.route_id = 1;

SDO_LRS.LOCATE_PT(A.ROUTE_GEOMETRY,M.DIMINFO,9,0)(SDO_GTYPE, SDO_SRID, SDO_POINT
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1), SDO_ORDINATE_ARRAY(
9, 4, 9))
```

E.6 Error Messages for Linear Referencing System

This section lists the LRS error messages, including the cause and recommended user action for each.

ORA-13331 invalid LRS segment

Cause: The given LRS segment was not a valid line string.

Action: A valid LRS geometric segment is a line string geometry in Oracle Spatial. It could be a simple or compound line string (made of lines or arcs, or both). The dimension information must include the measure dimension as the last element in Oracle Spatial metadata. Currently, the number of dimensions for an LRS segment must be greater than 2 (x/y or latitude/longitude plus measure).

ORA-13332 invalid LRS point

Cause: The given LRS point was not a valid LRS point.

Action: A valid LRS point is a point geometry in Oracle Spatial with additional measure dimension. The dimension information must include the measure dimension as the last element in the Spatial metadata. Currently, the number of dimensions for an LRS segment must be greater than 2 (x/y or latitude/longitude plus measure).

ORA-13333 invalid LRS measure

Cause: The given measure for linear referencing was out of linear measure range.

Action: The start and end measures of a geometric segment define the measure range of the segment. Any valid measures for a geometric segment must be within its measure range. The measures are always in an ascending order from the start to the end point.

ORA-13334 LRS segments not connected

Cause: The given geometric segments were not connected.

Action: For concatenating two geometric segments, the two segments must be spatially connected. That is, the end point of the first segment must be the same as the start point of the second segment.

ORA-13335 LRS segment is not defined

Cause: The given start or end measures are not defined, or some assigned measures in between are not in an ascending order.

Action: An LRS geometric segment is defined if its start and end measure are assigned (non-null). Any other measures assigned on the segment must be in an ascending order.

ORA-13336 LRS conversion failure

Cause: The conversion of an LRS geometry or layer was not successful.

Action: Check the following to see if they are valid: the geometry type for a geometry conversion, or the geometry type and dimensional information (dim-info) for a layer conversion. For example, polygon geometries are invalid as input to LRS functions.

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.

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.

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.

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.

HHCODE

A data type representing the intersection point of multiple dimensions. It encodes these multiple dimensions into a unique, linear value. The HHCODE data types were used for both spatial indexing and partitioned point data in previous releases of Spatial.

high-water mark

Expressed in number of records and associated with the deprecated Spatial partitioned table structure, it defines the maximum number of records to store in a table before decomposing another level. The high-water mark determines the maximum size of a partition within the Spatial table. Partitioned tables were an alternative to spatial indexing.

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.

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.

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.

partition

1. The spatial table that contains data only for a unique bounded *n*-dimensional space.
2. The process of grouping data into partitions that maintain the dimensional organization of the data.

partition key column

The primary HHCODE column that is used to dimensionally partition the data. One HHCODE data type column must be identified as the partition key for the table to be registered as partitionable in the Spatial data dictionary. There can be only one partition key per spatial table. Note that this is only used for the deprecated partitioned point data model, and not for spatially indexed data.

partitioned table

The spatial logical table structure that contains one or more partitions. Use partitioned tables only if you are dealing with a very large amount of legacy point data (over 50 gigabytes).

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.

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. Oracle8*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

1. The number of digits to the right of the decimal point in a number representing the level of resolution of an HHCODE instance.
2. The ratio of the distance on a map, photograph, or image to the corresponding image on the ground, all expressed in the same units.

SD*Converter

A utility used with previous releases of Spatial Data Option to prepare data for loading into spatial tables. Loading is now accomplished through SQL*Loader.

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.

SLF

See Spatial Load Format (SLF).

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 Oracle8*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 Load Format (SLF)

The format used to load data into spatial tables in a previous release of Spatial Data Option. Loading is now accomplished with the standard SQL*Loader.

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.

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

Numerics

8.1.7
  changes to this guide for release 8.1.7, xxiii
  migrating to release 8.1.7, 10-5
  migrating to release 8.1.7 from release 8.1.5, 10-2

A

administrative procedures, 15-1
ALL_SDO_GEOM_METADATA view, 2-18
ALL_SDO_INDEX_METADATA view, 2-20
ALTER INDEX, 5-2
  REBUILD, 5-5
  RENAME TO, 5-8
ANYINTERACT, 7-5, 17-3
arcs, A-14
area, 7-7
authalic sphere, D-3
average minimum bounding rectangle, 11-2
AVERAGE_MBR, 11-2, 16-2

B

batch geocoding, C-4
bounding rectangle
  minimum, 11-14
buffer area, 7-9
BUILD_WINDOW, 18-3
BUILD_WINDOW_FIXED, 18-5
bulk loading, 3-1, 13-2

C

C language examples (using OCI), 1-26
Cartesian coordinates, D-2
center of gravity (centroid), 7-11
centroid, 7-11
circle, 2-11, A-14
CLEAN_WINDOW, 18-7
CLEANUP_GID, 18-8
clip, E-11
CLIP_GEOM_SEGMENT, 9-5
COLUMN_NAME (in USER_SDO_GEOM_METADATA), 2-19
compound element, 2-9
CONCATENATE_GEOM_SEGMENT, 9-59
concatenating geometric segments, E-12
CONNECTED_GEOM_SEGMENTS, 9-10
consistency check, 7-33, 17-5
CONTAINS, 7-5, 17-3
control file, 13-2
CONVERT_TO_LRS_DIM_ARRAY, 9-12
CONVERT_TO_LRS_GEOM, 9-14
CONVERT_TO_LRS_LAYER, 9-16
CONVERT_TO_STD_DIM_ARRAY, 9-18
CONVERT_TO_STD_GEOM, 9-20
CONVERT_TO_STD_LAYER, 9-22
converting geometric segments
  functions for, 9-3
  overview, E-17
convex hull, 7-13
coordinate systems
  conceptual and usage information, D-1
  error messages, D-14
  example, D-8

coordinates
Cartesian, D-2
geodetic, D-2
geographic, D-2
projected, D-2
COVEREDBY, 7-5, 17-3
COVERS, 7-5, 17-4
cr_spatial_index.sql, A-16
CREATE INDEX, 5-9
CREATE_WINDOW_LAYER, 18-9
creating geometric segments
functions for, 9-1
creating layer tables, A-17
crlayer.sql, A-17
CS_SRS table, D-3
customized geometry types, A-14

D

data model, 1-6, A-1
LRS, E-7
datum
geodetic, D-2
transformation, D-3
DBA_SDO_GEOM_METADATA view, 2-18
DBA_SDO_INDEX_METADATA view, 2-20
DEFINE_GEOM_SEGMENT, 9-24
difference, 7-15
dimension (in SDO_GTYPE), 2-7
DIMINFO (in USER_SDO_GEOM_METADATA), 2-19
direction of geometric segment, E-3
concatenation result, E-13
disjoint, 7-5, 17-4
displaying geometries, A-18, A-20
distance, 7-39
DROP INDEX, 5-14
dynamic query window, 4-3, 14-4
DYNAMIC_SEGMENT, 9-27

E

editing geometric segments
functions for, 9-1
ELEM_INFO (SDO_ELEM_INFO), 2-8
element, 1-6
ellipsoids
list of supported, D-6
enabling third-party geocoders, C-11
EQUAL, 7-5, 17-4
error messages, xxiv
coordinate systems, D-14
linear referencing system, E-31
ESTIMATE_INDEX_PERFORMANCE, 11-4, 16-3
ESTIMATE_TILING_LEVEL, 11-7, 16-5
ESTIMATE_TILING_TIME, 11-9, 16-8
ESTIMATE_TOTAL_NUMTILES, 11-11
ETYPE (SDO_ETYPE), 2-9
examples
C, 1-26
coordinate systems, D-8
creating, indexing, and querying spatial data, 2-1
directory, 1-25
Linear Referencing System (LRS), E-19
OCI (Oracle Call Interface), 1-26
PL/SQL, 1-26
SQL, 1-26
extent, 16-9
EXTENT_OF, 11-14, 16-9
exterior polygon rings, 2-7, 2-9, 2-14, 2-15

F

features
linear, E-6
filter, 14-6
FIND_MEASURE, 9-29
fixed indexing, 1-13
fixed-size tiles, 3-12, 13-7, 15-5, 15-14
FROM_815_TO_81x, 10-2
functions
CLIP_GEOM_SEGMENT, 9-5
CONCATENATE_GEOM_SEGMENT, 9-59
CONNECTED_GEOM_SEGMENTS, 9-10
CONVERT_TO_LRS_DIM_ARRAY, 9-12
CONVERT_TO_LRS_GEOM, 9-14
CONVERT_TO_LRS_LAYER, 9-16
CONVERT_TO_STD_DIM_ARRAY, 9-18
CONVERT_TO_STD_GEOM, 9-20

CONVERT_TO_STD_LAYER, 9-22
DEFINE_GEOM_SEGMENT procedure, 9-24
DYNAMIC_SEGMENT, 9-27
FIND_MEASURE, 9-29
GEOM_SEGMENT_END_MEASURE, 9-41
GEOM_SEGMENT_END_PT, 9-33
GEOM_SEGMENT_LENGTH, 9-35
GEOM_SEGMENT_START_MEASURE, 9-37
GEOM_SEGMENT_START_PT, 9-39
GET_MEASURE, 9-41
IS_GEOM_SEGMENT_DEFINED, 9-69
LOCATE_PT, 9-45
MEASURE_RANGE, 9-47
MEASURE_TO_PERCENTAGE, 9-49
PERCENTAGE_TO_MEASURE, 9-51
PROJECT_PT, 9-53
REDEFINE_GEOM_SEGMENT, 9-55
REVERSE_MEASURE, 9-57
SCALE_GEOM_SEGMENT, 9-59
SPLIT_GEOM_SEGMENT procedure, 9-62
TRANSLATE_MEASURE, 9-65
VALID_GEOM_SEGMENT, 9-67
VALID_LRS_PT, 9-69
VALID_MEASURE, 9-71

G

generic geocoding interface, C-1
GEOCODE_SCHEMA_PROPERTY_TYPE, C-5
GEOCODE_SERVER_PROPERTY_TYPE, C-4
GEOCODE_TABLE_COLUMN_TYPE, C-5
GEOCODE_TASK_METADATA, C-4
geocoder metadata, C-3
GEOCODER_HTTP package, C-10
geocoding
generic interface, C-1
geocoding support
*inter*Media Locator, A-17
geodetic coordinates, D-2
geodetic datum, D-2
geographic coordinates, D-2
GeoImage feature, xxiv
GEOM_SEGMENT_END_MEASURE, 9-41
GEOM_SEGMENT_END_PT, 9-33
GEOM_SEGMENT_LENGTH, 9-35
GEOM_SEGMENT_START_MEASURE, 9-37
GEOM_SEGMENT_START_PT, 9-39
geometric primitive, 1-4
geometric segment
clipping, E-11
concatenating, E-12
converting (functions for), 9-3
converting (overview), E-17
creating (functions for), 9-1
direction, E-3
direction with concatenation, E-13
editing (functions for), 9-1
locating point on, E-15
projecting point onto, E-17
querying (functions for), 9-2
scaling, E-14
splitting, E-12
geometric segments, E-2
geometry types, 1-4
custom, A-14
object-relational, 2-6
relational, 12-3
GET_MEASURE, 9-41
GIS, 1-3
GTYPE (SDO_GTYPE), 2-6

H

HISTOGRAM_ANALYSIS, 11-16, 16-10
hybrid indexing, 1-18

I

index
creation, 3-11
creation (cross-schema), 3-16
creation in parallel, A-14
description of Spatial indexing, 1-9
performance, 11-4, 16-3
quadtree, 1-12
R-tree, 1-10
R-tree (requirements before creating), 1-11
inserting spatial data, 13-4
INSIDE, 7-6, 17-4
INTEPRETATION (SDO_INTERPRETATION), 2-9

interaction, 7-5, 17-3
interior polygon rings, 2-7, 2-9, 2-14, 2-15
*inter*Media Locator, A-17
intersection, 7-20
IS_GEOM_SEGMENT_DEFINED, 9-69

J

Java Virtual Machine, C-2
JVM (Java Virtual Machine), C-2

L

layer, 1-7, A-17
  transforming, 8-5
  validating, 7-36
length (SDO_LENGTH), 7-23
line, 2-12
  length, 7-23
line data, 1-6
linear features, E-6
linear measure, E-3
Linear Referencing System (LRS)
  conceptual and usage information, E-1
  data model, E-7
  error messages, E-31
  example, E-19
  function reference information, 9-1
  limiting indexing to X and Y dimensions, E-8
  LRS point, E-6
  segments, E-2
loading process, 3-1, 13-2
  in parallel, A-14
LOCATE_PT, 9-45
location, 1-3
Locator (*inter*Media), A-17
long transactions (Workspace Management), xxiv
LRS
  See Linear Referencing System (LRS)
LRS point, E-6

M

map projections
  list of supported, D-5
MDSYS schema, 1-1
MDSYS.CS_SRS table, D-3
measure, E-3
measure populating, E-4
measure range, E-6
MEASURE_RANGE, 9-47
MEASURE_TO_PERCENTAGE, 9-49
metadata for geocoding, C-3
migration
  OGIS, 10-8, 10-9
  to current Spatial release, 10-5
  to current Spatial release from 8.1.5, 10-2
  to release 7.3.4, 10-3
minimum bounding rectangle, 11-2, 11-14, 16-2, 16-9
  average, 11-2
MIX_INFO, 11-18, 16-12
multimatch table, C-9
multiple matches, C-9
multipolygon, 2-15

N

nearest neighbor (SDO_NN), 6-6

O

object-relational model
  schema, 2-1
OCI (Oracle Call Interface) examples, 1-26
offset, E-3
OGIS_METADATA_FROM, 10-8
OGIS_METADATA_TO, 10-9
operators
  cross-schema invocation, 4-9
  SDO_FILTER, 6-2
  SDO_NN, 6-6
  SDO_RELATE, 6-8
  SDO_WITHIN_DISTANCE, 6-13
Oracle Call Interface (OCI) examples, 1-26
Oracle Technology Network (OTN), xxiv
oracle.spatial.geocoder.Metadata, C-9
OVERLAPBDYDISJOINT, 7-6, 17-4
OVERLAPBDYINTERSECT, 7-6, 17-4

P

parallel load, A-14
partitioned tables, 1-25, A-14
PERCENTAGE_TO_MEASURE, 9-51
plotting tiles, A-5
PL/SQL and SQL examples, 1-26
point
  locating on geometric segment, E-15
  LRS, E-6
  shape, E-2
point data, 1-6, 15-8, A-9
point on surface of polygon, 7-25
polygon
  area of, 7-7
  centroid, 7-11
  exterior and interior rings, 2-7, 2-9, 2-14, 2-15
  point on surface, 7-25
polygon collection, 2-15
polygon data, 1-6
POPULATE_INDEX, 15-3
POPULATE_INDEX_FIXED, 15-5
POPULATE_INDEX_FIXED_POINTS, 15-8
populating
  measure, E-4
primary filter, 14-6
primitive, 1-4
problems in current release, D-7
procedures
  DEFINE_GEOM_SEGMENT, 9-24
  SPLIT_GEOM_SEGMENT, 9-62
PROJECT_PT, 9-53
projected coordinates, D-2
projection, E-6
  point onto geometric segment, E-17
  PROJECT_PT function, 9-53

Q

quadtree indexes, 1-12
query, 1-8
query window, 4-3, 14-4
querying geometric segments
  functions for, 9-2

R

range
  measure, E-6
rectangle, 2-11
  minimum bounding, 11-14
REDEFINE_GEOM_SEGMENT, 9-55
rejected records, C-9
RELATE, 7-4, 17-2
relational model
  schema, 12-1
release 8.1.7
  changes to this guide, xxiii
  migrating to release 8.1.7, 10-5
  migrating to release 8.1.7 from release 8.1.5, 10-2
restrictions in current release, D-7
REVERSE_MEASURE, 9-57
rollback segment
  R-tree index creation, 1-11
R-tree indexes, 1-10
  before creating, 1-11
  sequence object, 2-23

S

sample program, A-18, A-20
SCALE_GEOM_SEGMENT, 9-59
scaling a geometric segment, E-14
schema, 12-1
  creating index on table in another schema, 3-16
  invoking operators on table in another schema, 4-9
  object-relational model, 2-1
  relational model, 12-1
SDO_AREA, 7-7
SDO_BUFFER, 7-9
SDO_CENTROID, 7-11
SDO_CODE, 2-22
SDO_CODE_SIZE, 15-10
SDO_CONVEXHULL, 7-13
SDO_DIFFERENCE, 7-15
SDO_DISTANCE, 7-18
SDO_ELEM_INFO, 2-8
SDO_ETYPE, 2-9

SDO_FILTER operator, 6-2
SDO_GEOMETRY object type, 2-6
SDO_GROUPCODE, 2-22
SDO_GTYPE, 2-6
SDO_INDEX_TABLE, 2-22
SDO_INDX_DIMS, E-8
SDO_INTERPRETATION, 2-9
SDO_INTERSECTION, 7-20
SDO_LENGTH, 7-23
SDO_LEVEL, 1-12
SDO_NN
  optimizer hint, 6-7
SDO_NN operator, 6-6
SDO_NUMTILES, 1-12
SDO_ORDINATES, 2-12
SDO_POINT, 2-8
SDO_POINTONSURFACE, 7-25
SDO_POLY_xxx functions (deprecated and removed), 7-2
SDO_RELATE operator, 6-8
SDO_ROWID, 2-22
SDO_RTREE_SEQ_NAME, 2-23
SDO_SRID, 2-8
SDO_STARTING_OFFSET, 2-8
SDO_STATUS, 2-22
SDO_UNION, 7-27
SDO_VERSION, 15-11
SDO_WITHIN_DISTANCE operator, 6-13
SDO_XOR, 7-30
secondary filter, 14-7
segments
  geometric, E-2
sequence object for R-tree index, 2-23
shape point, E-2
simple element, 2-9
SORT_AREA_SIZE parameter
  R-tree index creation, 1-11
spatial data structures
  object-relational model, 2-1
  relational model, 12-1
spatial database
  sizing, A-4
spatial index
  See index
Spatial Index Advisor
  using to determine best tiling level, 3-14
spatial indexing
  fixed, 1-13
  hybrid, 1-18
spatial join, 4-9, 14-8, A-11
spatial query, 4-3, 14-4
spatial reference systems
  conceptual and usage information, D-1
  example, D-8
sphere
  authalic, D-3
SPLIT_GEOM_SEGMENT, 9-62
splitting a geometric segment, E-12
SQL and PL/SQL examples, 1-26
SQL script, A-16
SQL*Loader, 3-1, 13-2
SRID
  in USER_SDO_GEOM_METADATA, 2-20
  SDO_SRID in SDO_GEOMETRY, 2-8

T

table partitioning, 1-25
TABLE_NAME (in USER_SDO_GEOM_METADATA), 2-19
tessellation, 1-13, 13-6
tile, 1-12, 4-1, 14-1
tiling, 11-7, 15-14, 16-5, A-2
TO_734, 10-3
TO_81x, 10-5
tolerance, 1-7
TOUCH, 7-6, 17-4
transactional insert, 3-3, 13-4
TRANSFORM (function), 8-2
TRANSFORM_LAYER (procedure), 8-5
Transform_Layer (procedure)
  table for transformed layer, 8-6
transformation, D-3
TRANSLATE_MEASURE, 9-65
two-tier query, 1-8, 4-1, 14-1

U

union, 7-27
UPDATE_INDEX, 15-12

UPDATE_INDEX_FIXED, 15-14
USER_SDO_GEOM_METADATA view, 2-18
USER_SDO_INDEX_METADATA view, 2-20

V

VALID_GEOM_SEGMENT, 9-67
VALID_LRS_PT, 9-69
VALID_MEASURE, 9-71
VALIDATE_GEOMETRY, 7-33, 17-5
VALIDATE_LAYER, 7-36
VERIFY_LAYER, 15-16
visualizing geometries, A-18, A-20
visualizing tiles, A-5, A-20

W

well-known text (WKTEXT), D-4
WITHIN_DISTANCE, 7-39
WKTEXT, D-4
Workspace Management, xxiv

X

XOR, 7-30