

Oracle[®] Spatial

User's Guide and Reference

Release 8.1.6

December 1999

Part No. A77132-01

ORACLE

Oracle Spatial User's Guide and Reference

Part No. A77132-01

Release 8.1.6

Copyright © 1997, 1999, Oracle Corporation. All rights reserved.

Primary Authors: Jeff Hebert, Chuck Murray

Contributing Author: Anna Logan

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

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

This Program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright, patent and other intellectual property law. Reverse engineering of the software 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. No part of this document 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 this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs 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 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle, SQL*Loader, SQL*Net, and SQL*Plus are registered trademarks, and Oracle7 and Oracle8i 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	xv
Preface	xvii
Intended Audience	xvii
Structure	xvii
New Features for Release 8.1.6	xviii
Features Released Separately.....	xix
Related Documents.....	xx
Conventions.....	xx
1 Spatial Concepts	
1.1 What Is the Spatial Product?.....	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 Methods	1-9

1.7.1	Tessellation of a Layer During Indexing.....	1-11
1.7.2	Fixed Indexing	1-12
1.7.3	Hybrid Indexing.....	1-16
1.8	Spatial Relations and Filtering.....	1-19
1.9	Partitioned Point Data	1-22

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.....	2-12
2.3	Geometry Examples Using the Object-Relational Model.....	2-12
2.3.1	Rectangle.....	2-13
2.3.2	Polygon with a Hole	2-13
2.3.3	Compound Element	2-15
2.3.4	Compound Polygon.....	2-16
2.4	Geometry Metadata Structure	2-17
2.5	Spatial Index-Related Structure.....	2-19
2.5.1	Spatial Index Views and Tables	2-19
2.5.2	Spatial Index Table Definition	2-21

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.....	3-1
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.....	3-5
3.1.2.3	Compound Polygon	3-6
3.1.2.4	Compound Polygon with Holes.....	3-8
3.1.2.5	Transactional Insert of Point-Only Data.....	3-9
3.2	Index Creation	3-9
3.2.1	Determining Index Creation Behavior.....	3-10
3.2.2	Spatial Indexing with Fixed-Size Tiles.....	3-10
3.2.3	Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles	3-13
3.2.4	Cross-Schema Index Creation	3-14

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
4.3	Spatial Join.....	4-8
4.4	Cross-Schema Operator Invocation.....	4-9

5 Indexing Statements for Object-Relational Model

ALTER INDEX.....	5-2
ALTER INDEX REBUILD	5-5
ALTER INDEX RENAME TO	5-8
CREATE INDEX.....	5-9
DROP INDEX.....	5-13

6 Tuning Functions and Procedures for Object-Relational Model

SDO_TUNE.AVERAGE_MBR.....	6-2
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE.....	6-4
SDO_TUNE.ESTIMATE_TILING_LEVEL	6-7
SDO_TUNE.ESTIMATE_TILING_TIME	6-9
SDO_TUNE.ESTIMATE_TOTAL_NUMTILES	6-11
SDO_TUNE.EXTENT_OF	6-14

SDO_TUNE.HISTOGRAM_ANALYSIS	6-16
SDO_TUNE.MIX_INFO.....	6-18

7 Geometry Functions for Object-Relational Model

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_POLY_DIFFERENCE	7-27
SDO_GEOM.SDO_POLY_INTERSECTION	7-29
SDO_GEOM.SDO_POLY_UNION	7-31
SDO_GEOM.SDO_POLY_XOR.....	7-33
SDO_GEOM.SDO_UNION.....	7-35
SDO_GEOM.SDO_XOR	7-38
SDO_GEOM.VALIDATE_GEOMETRY.....	7-41
SDO_GEOM.VALIDATE_LAYER.....	7-43
SDO_GEOM.WITHIN_DISTANCE.....	7-45

8 Migration Procedures

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

9 Spatial Operators

SDO_FILTER.....	9-2
SDO_NN.....	9-4
SDO_RELATE.....	9-6
SDO_WITHIN_DISTANCE.....	9-9

Part II Relational Model

10 The Relational Schema

10.1 Database Structures for the Relational Implementation.....	10-1
---	------

11 Loading Spatial Data

11.1 Load Model.....	11-1
11.2 Load Process.....	11-2
11.2.1 Bulk Loading.....	11-2
11.2.2 Transactional Insert Using SQL.....	11-4
11.3 Index Creation.....	11-6
11.3.1 Choosing a Tessellation Algorithm.....	11-6
11.3.2 Spatial Indexing with Fixed-Size Tiles.....	11-7
11.3.3 Hybrid Spatial Indexing with Fixed-Size and Variable-Sized Tiles.....	11-10

12 Querying Spatial Data

12.1 Query Model.....	12-1
12.2 Spatial Index Data Structures.....	12-1
12.3 Spatial Query.....	12-4
12.3.1 Dynamic Query Window.....	12-5
12.3.2 Primary Filter Query.....	12-6
12.3.3 Secondary Filter Query.....	12-7
12.4 Spatial Join.....	12-8

13 Administrative Functions and Procedures

SDO_ADMIN.POPULATE_INDEX.....	13-3
SDO_ADMIN.POPULATE_INDEX_FIXED.....	13-5

SDO_ADMIN.POPULATE_INDEX_FIXED_POINTS	13-8
SDO_ADMIN.SDO_CODE_SIZE.....	13-10
SDO_ADMIN.SDO_VERSION.....	13-11
SDO_ADMIN.UPDATE_INDEX.....	13-12
SDO_ADMIN.UPDATE_INDEX_FIXED.....	13-14
SDO_ADMIN.VERIFY_LAYER	13-16

14 Tuning Functions and Procedures

SDO_TUNE.AVERAGE_MBR.....	14-2
SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE	14-3
SDO_TUNE.ESTIMATE_TILING_LEVEL.....	14-5
SDO_TUNE.ESTIMATE_TILING_TIME	14-8
SDO_TUNE.EXTENT_OF	14-9
SDO_TUNE.HISTOGRAM_ANALYSIS	14-10
SDO_TUNE.MIX_INFO.....	14-12

15 Geometry Functions and Procedures

SDO_GEOM.RELATE.....	15-2
SDO_GEOM.VALIDATE_GEOMETRY.....	15-5
SDO_GEOM.VALIDATE_LAYER	15-6

16 Window Functions and Procedures

SDO_WINDOW.BUILD_WINDOW	16-2
SDO_WINDOW.BUILD_WINDOW_FIXED.....	16-4
SDO_WINDOW.CLEAN_WINDOW.....	16-6
SDO_WINDOW.CLEANUP_GID.....	16-7
SDO_WINDOW.CREATE_WINDOW_LAYER.....	16-8

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-8
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-13
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-15
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	Oracle8i <i>interMedia</i> 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 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-19
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 to 8.1.6.....	B-5
B.5.1	Data Migration.....	B-6
B.5.2	Compatibility Between Releases 8.1.5 and 8.1.6	B-6

C Generic Geocoding Interface

C.1	<i>interMedia</i> Locator Implementation: Benefits and Limitations	C-2
C.2	Generic Geocoding Client	C-2
C.3	Geocoder Metadata	C-4
C.3.1	Server Properties	C-5
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

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	Create a Fixed Index	3-13
3-4	Create a Hybrid Index	3-14
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
11-1	Raw Data Format	11-2
11-2	Control File to Load Data into the Geometry Table	11-3
11-3	Raw Data Format	11-3
11-4	Control File to Load from a Single Flat File	11-4
11-5	Transactional Insert	11-4
11-6	Transactional Insert for a Large Geometry	11-5
13-1	Populate an Index	13-4
13-2	Populate an Index with Fixed-Size Tiles	13-7
13-3	Populate an Index with Fixed-Size Tiles Based on Point Data	13-9
13-4	Update an Index	13-13
13-5	Update an Index with Fixed-Size Tiles	13-15
13-6	Verify a Layer	13-16
14-1	Recommended Tile Level for One-Degree Latitude/Longitude Cells	14-6
14-2	Recommended Tile Level Based on the GIDs of All Geometries	14-6
14-3	Recommended Tile Level Based on Average Extent of All Geometries	14-7
A-1	View Fixed-Size Tiles for All Geometries	A-5
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-6

List of Figures

1-1	Geometric Primitive Types	1-4
1-2	Additional Geometry Types Using the Object-Relational Model	1-6
1-3	Query Model	1-9
1-4	Quadtree Decomposition and Morton Codes	1-11
1-5	Fixed-Size Tiling with Many Small Tiles	1-13
1-6	Fixed-Size Tiling with Fewer Large Tiles.....	1-14
1-7	Tessellated Figure.....	1-15
1-8	Variable-Sized Tile Spatial Indexing.....	1-17
1-9	Decomposition of the Geometry	1-18
1-10	The 9-Intersection Model.....	1-20
1-11	Distance Buffers for Points, Lines, and Polygons	1-22
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	2-15
2-5	Compound Polygon.....	2-16
3-1	Example Geometry OBJ_1	3-4
3-2	Line String Consisting of Arcs and Straight Line Segments	3-6
3-3	Compound Polygon.....	3-7
3-4	Compound Polygon with a Hole	3-9
3-5	Sample Domain	3-11
3-6	Fixed-Size Tiling at Level 1	3-12
3-7	Fixed-Size Tiling at Level 2	3-12
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.....	7-36
7-4	SDO_GEOM.SDO_XOR	7-39
10-1	Complex Polygon	10-5
11-1	Sample GIS Domain	11-8
11-2	Fixed-Size Tiling at Level 1	11-8
11-3	Fixed-Size Tiling at Level 2	11-9
12-1	Tessellated Layer with Multiple Objects.....	12-2
12-2	Tessellated Layer with a Query Window	12-5
12-3	Spatial Join of Two Layers.....	12-8
C-1	Oracle Geocoding Framework.....	C-3

List of Tables

1-1	SDOINDEX Table Using Fixed-Size Tiles.....	1-16
1-2	Section of the SDOINDEX Table.....	1-19
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.....	2-20
2-4	Columns in a Spatial Index Data Table	2-21
5-1	Spatial Index Creation and Usage Statements	5-1
5-2	SDO_LEVEL and SDO_NUMTILES Combinations.....	5-11
6-1	Tuning Functions and Procedures	6-1
7-1	Geometric Functions for the Object-Relational Model	7-1
8-1	Migration Procedures	8-1
9-1	Spatial Usage Operators	9-1
10-1	<layername>_SDOLAYER Table	10-1
10-2	<layername>_SDODIM Table or View	10-1
10-3	<layername>_SDOGEOM Table or View.....	10-2
10-4	<layername>_SDOINDEX Table	10-2
11-1	<layername>_SDOLAYER Table	11-1
11-2	<layername>_SDODIM Table or View	11-1
11-3	<layername>_SDOGEOM Table or View.....	11-2
11-4	<layername>_SDOINDEX Table	11-2
11-5	Choosing a Tessellation Algorithm	11-7
12-1	<layername>_SDOLAYER Table	12-3
12-2	<layername>_SDOGEOM Table or View.....	12-3
12-3	<layername>_SDOINDEX Table	12-4
13-1	Administrative Procedures for Spatially Indexed Data.....	13-1
14-1	Tuning Functions and Procedures	14-1
15-1	Geometric Functions and Procedures	15-1
16-1	Window Functions and Procedures	16-1

Send Us Your Comments

Oracle Spatial User's Guide and Reference, Release 8.1.6

Part No. A77132-01

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 book title and (if possible) the chapter, section, and page number. 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 Oracle8i Enterprise Edition.

Spatial requires Oracle8i Enterprise Edition. Oracle8i and Oracle8i 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 Oracle8i table partitioning, you must have the Enterprise Edition and the Partitioning Option.

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

Intended Audience

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

Structure

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 book:

- | | |
|---------------------------|--|
| 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 the tuning functions and procedures.
Chapter 7	Provides the syntax and semantics for the geometric functions and procedures.
Chapter 8	Provides the syntax and semantics for the migration functions.
Chapter 9	Provides the syntax and semantics for operators used with the spatial object data type.
Part II	The following chapters describe the relational spatial model:
Chapter 10	Explains the relational schema.
Chapter 11	Explains spatial data loading.
Chapter 12	Explains methods for querying a spatial database.
Chapter 13	Provides the syntax and semantics for the administrative functions and procedures.
Chapter 14	Provides the syntax and semantics for the tuning functions and procedures.
Chapter 15	Provides the syntax and semantics for the geometric functions and procedures.
Chapter 16	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.
Glossary	Provides definitions of terms used in this guide.

New Features for Release 8.1.6

Oracle Spatial includes the following enhancements for release 8.1.6:

- Improved metadata management through system-level views and metadata tables.
- New types (geometry and element) allow for better representation of the geometries in the database. These types also enable better management of spatial geometries in a client-side application.
- More features in the Validate Geometry function ([SDO_GEOM.VALIDATE_GEOMETRY](#)). These new features can help validate complex geometries and provide better feedback on invalid geometries.
- Improved performance for all spatial functions, especially for [SDO_GEOM.WITHIN_DISTANCE](#) and [SDO_GEOM.SDO_BUFFER](#) operations.
- Performance of all index creation operations is improved.
- Enhancement of Boolean operations to operate at the geometry level rather than at the polygon level.

(This is not an exhaustive list of new and changed features for release 8.1.6.)

Because of changes in release 8.1.6, spatial indexes created with previous releases of Spatial must be re-created, and the xxx_SDO_GEOM_METADATA views must be updated. For information about compatibility and migration, see [Appendix B](#), especially [Section B.5](#) on migrating from release 8.1.5 to 8.1.6.

Features Released Separately

The following features or capabilities are being released separately from release 8.1.6, and will be available through the Oracle Technology Network (<http://technet.oracle.com/>):

- Linear Referencing System (LRS)
- R-tree indexes
- GeoImage
- Coordinate Systems
- Version Management (long transactions)

These features are not documented in this manual. Instead, each is documented in a separate manual or other publication, which will be available through the Oracle Technology Network when the associated software is available.

Related Documents

For more information, see the following manuals:

- *Oracle8i 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
- *Oracle8i Concepts*
- *Oracle8i Tuning*
- *Oracle8i Utilities*

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

<http://www.oracle.com/database/options/spatial/>

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

Convention	Meaning
[]	Brackets enclose optional clauses from which you can choose one or none.
%	The percent sign represents the system prompt on a UNIX system.

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 Oracle8i 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 the Spatial Product?

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 Oracle8i 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 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 geometry:

- The **object-relational** model uses a table with 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. Other sections in this chapter describe Spatial concepts and features, without reference to their implementation wherever possible.

You should select the best model to fit your 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:* Oracle8i does not 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 Oracle8i 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 Oracle 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.

1.3 Introduction to Spatial Data

Oracle Spatial is designed to make spatial data management easier and more natural to users or 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.

Other types of spatial data that can be stored using Spatial besides 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.

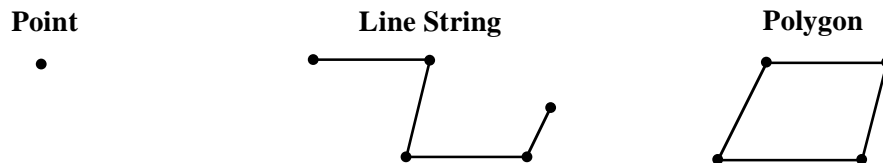
1.4 Geometric Types for Relational and Object-Relational Models

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. If a line string crosses itself, it does not become a polygon. A self-crossing line string does not have any implied interior.

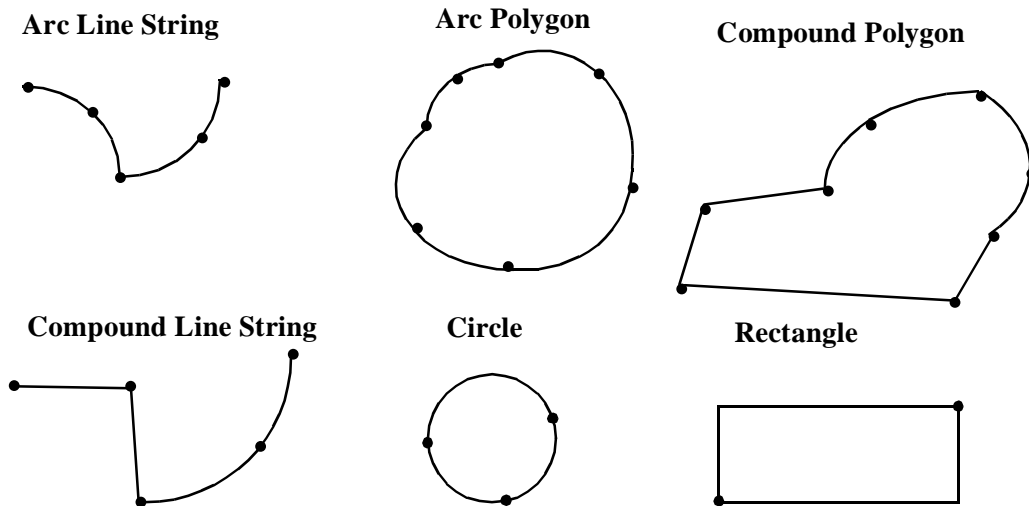
Thus, the object-relational implementation supports the types listed in [Figure 1-1](#), as well as the types shown in [Figure 1-2](#).

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 (clockwise for an exterior polygon ring, counter-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

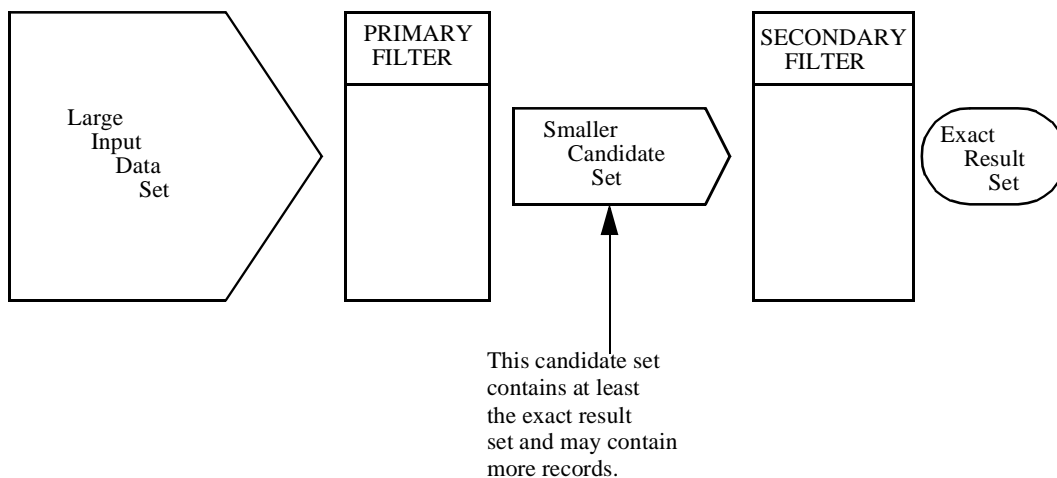
Spatial uses a *two-tier* query model to resolve spatial queries and spatial joins. The term is used to indicate that two distinct operations are performed in order to resolve queries. The output of both operations yields the exact result set.

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



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 Methods

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 required 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 spatially interact 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. Spatial uses a linear quadtree-based indexing scheme, also known as z-ordering, which works as described in the following paragraphs.

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

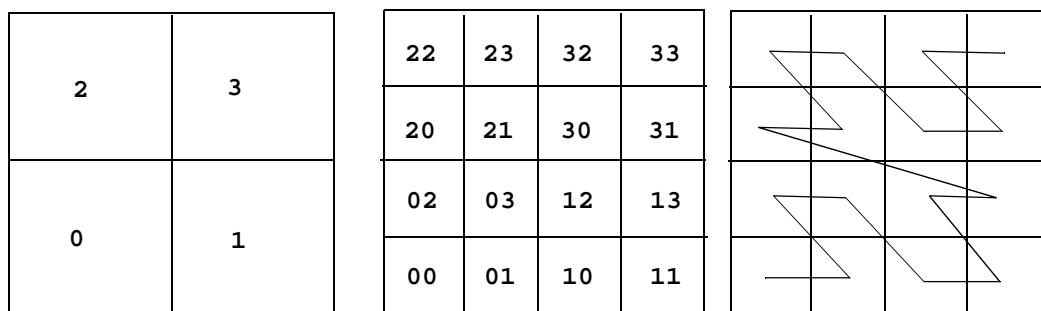
- **Fixed indexing:** a non-null SDO_LEVEL value and a null SDO_NUMTILES value, resulting in fixed-sized tiles. Fixed indexing is described in [Section 1.7.2](#).
- **Hybrid indexing:** non-null values for SDO_LEVEL and SDO_NUMTILES, resulting in two sets of tiles per geometry. One set contains fixed-size tiles and the other set contains variable-sized tiles. Hybrid indexing is described in [Section 1.7.3](#).

1.7.1 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 equal-sized covering tiles. 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 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.

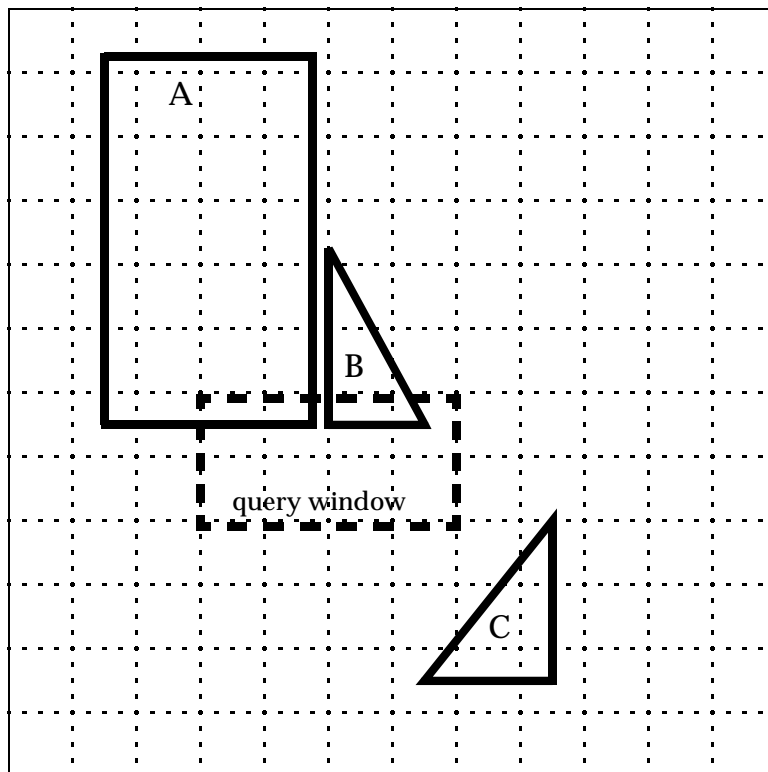
Two geometries are likely to interact, and hence pass the primary filter stage, if they share one or more tiles. The SQL statement for the primary filter stage is:

```
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.

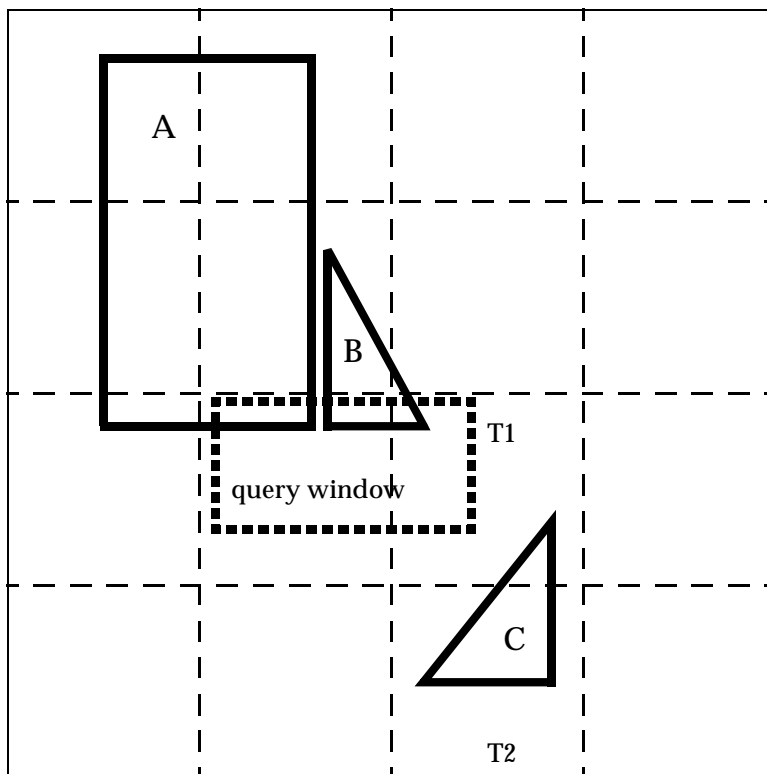
With a small fixed-size tile as shown in [Figure 1-5](#), selectivity is good, but a large number of tiles is needed to cover large geometries. A window query would easily identify geometries A and B, but would reject C.

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](#).

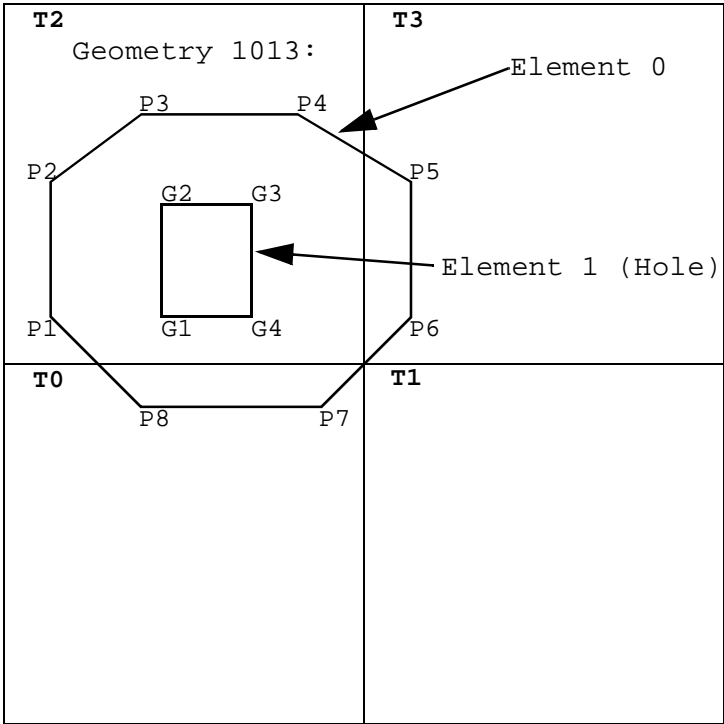
Figure 1–6 Fixed-Size Tiling with Fewer Large Tiles



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 Figure



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

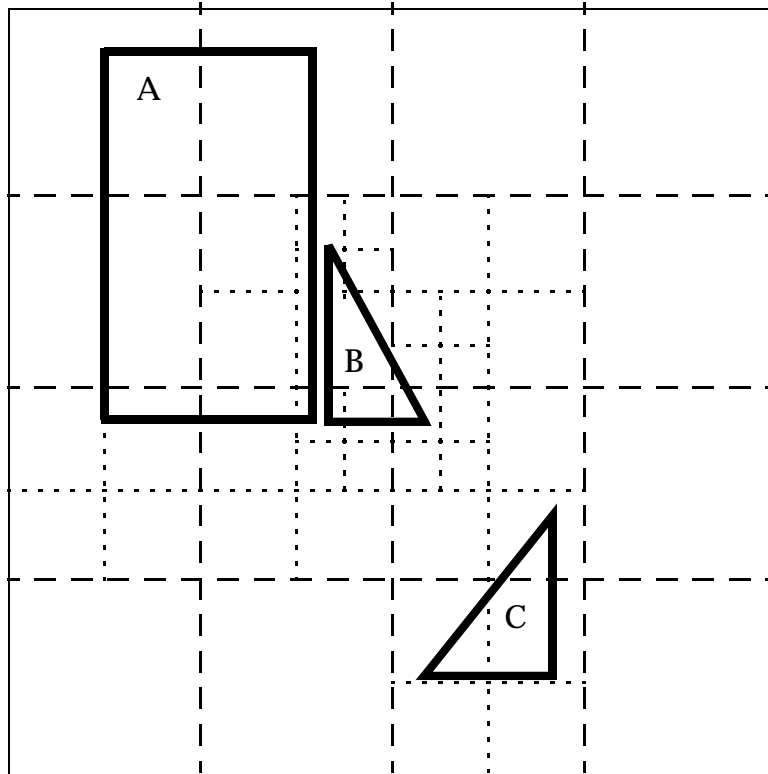
All elements in a geometry are tessellated. In a multielement geometry like 1013, Element 1 is already covered by tile T2 from the tessellation of Element 0. If, however, the specified tiling resolution were such that tile T2 were further subdivided and one of these smaller tiles were completely contained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

1.7.3 Hybrid Indexing

Hybrid indexing is the preferred method for indexing the object-relational model. 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.

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.

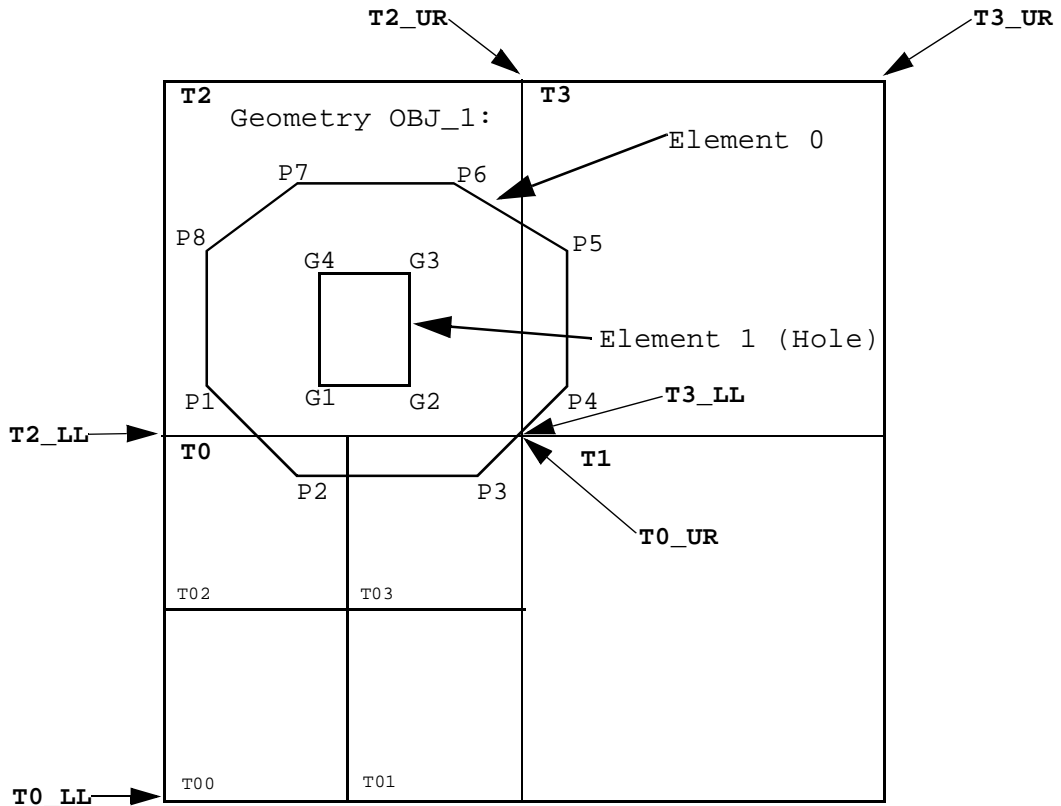
Figure 1–8 Variable-Sized Tile Spatial Indexing



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` parameter. A user may modify this parameter, but it is not recommended.

[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

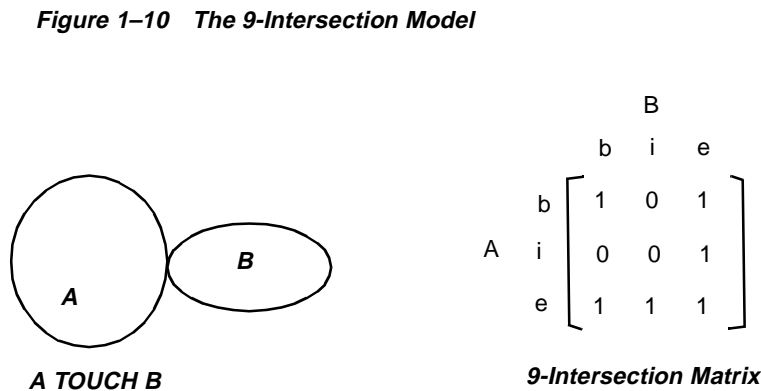
Spatial uses filter methods to determine the spatial relationship between entities in the database. The spatial relation is based on geometry locations. The most common spatial relations are based on topology and distance. For example, the *boundary* of an area consists of a set of curves that separates 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 no 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 two secondary filter methods. One method ([SDO_RELATE](#) operator) evaluates topological criteria, and a second method ([SDO_WITHIN_DISTANCE](#) operator) determines if two spatial objects are within a Euclidean distance of each other. The syntax of both operators is given in [Chapter 9](#).

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 A_b , an interior A_i , and an exterior A_e), 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”.



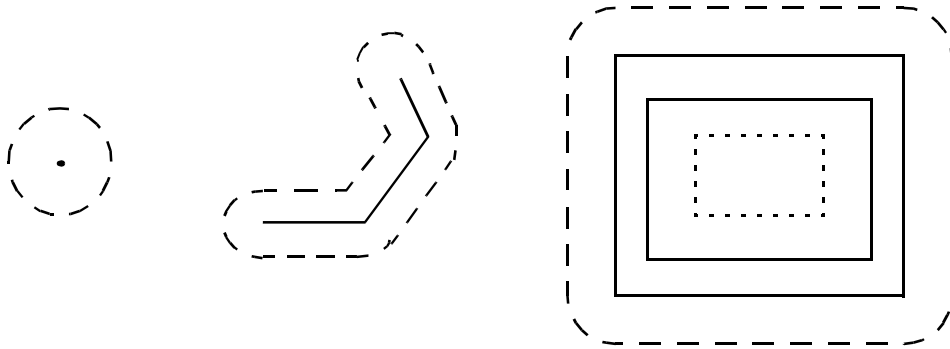
Some of the topological relationships identified in the seminal work by Dr. Egenhofer¹ and colleagues have names associated with them. Spatial uses the following names:

- DISJOINT -- The boundaries and interiors do not intersect.
- TOUCH -- The boundaries intersect but the interiors do not.
- 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.
- COVERS -- The interior of one object is completely contained in the interior of the other 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 other secondary filter method ([SDO_WITHIN_DISTANCE](#) operator) determines if two spatial objects, A and B, are within a Euclidean distance of one another. It 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.

¹ Dr. Max Egenhofer, University of Maine, Orono.

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



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 Oracle8i table partitioning features.

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

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 for Object-Relational Model"](#)
- [Chapter 6, "Tuning Functions and Procedures for Object-Relational Model"](#)
- [Chapter 7, "Geometry Functions for Object-Relational Model"](#)
- [Chapter 8, "Migration Procedures"](#)
- [Chapter 9, "Spatial Operators"](#)

The Object-Relational Schema

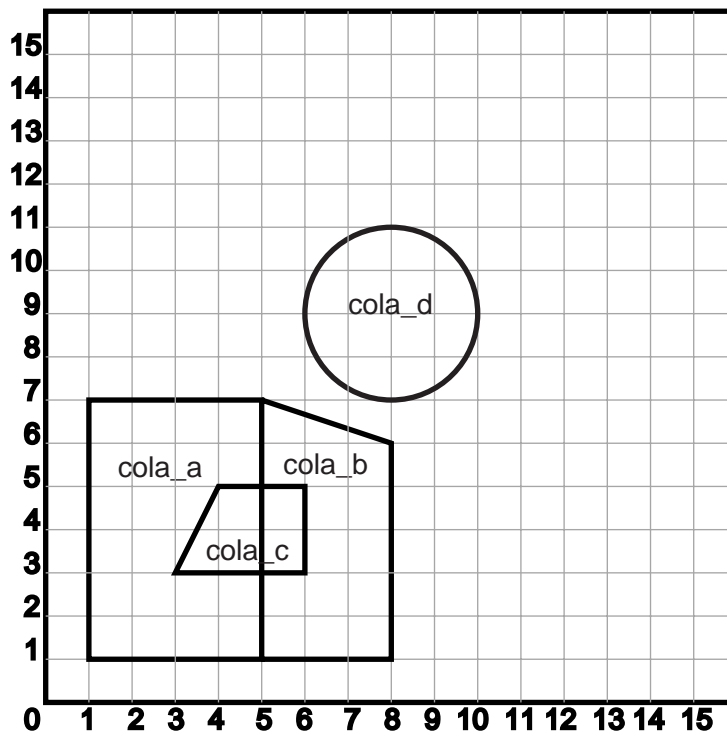
The object-relational implementation of Oracle Spatial consists of a set of object data types, an index method type, and operators on these types. A geometry is stored as an object, in a single row, in a column of type `SDO_GEOMETRY`. Spatial index creation and maintenance is done using basic DDL (`CREATE`, `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 (reserved for future Spatial releases)
);

-----
-- 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
d000	UNKNOWN_GEOMETRY	Spatial ignores this geometry.
d001	POINT	Geometry contains one point.
d002	LINestring	Geometry contains one line string.
d003	POLYGON	Geometry contains one polygon with or without holes ¹ .
d004	COLLECTION	Geometry is a heterogeneous collection of elements. ²
d005	MULTIPOINT	Geometry has multiple points.
d006	MULTILINestring	Geometry has multiple line strings.
d007	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 coordinates used to represent each vertex (for example, X, Y for 2-dimensional objects). Points and lines are considered 2-dimensional objects.

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 d008-d099 are reserved for future use.

2.2.2 SDO_SRID

SDO_SRID is reserved for future use. This attribute is intended to be a foreign key in a spatial reference system definition table.

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.

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 counter-clockwise 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 counter-clockwise 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-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.
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.

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

SDO_ETYPE	SDO_INTERPRETATION	Meaning
3	2	<p>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.</p> <p>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.</p>
3	3	<p>Rectangle type. A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it.</p>
3	4	<p>Circle type. Described by three points, all on the circumference of the circle.</p>
4	$n > 1$	<p>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.</p> <p>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.</p> <p>See Section 2.3 and Figure 2–4 for an example of a geometry using this type.</p>
5	$n > 1$	<p>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.</p> <p>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.</p> <p>See Section 2.3.4 and Figure 2–5 for an example of a geometry using this type.</p>

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 *d001* and *d005*, any subelement not of SDO_ETYPE 1 is ignored.
- For SDO_GTYPE values *d002* and *d006*, any subelement not of SDO_ETYPE 2 or 4 is ignored.
- For SDO_GTYPE values *d003* and *d007*, 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.

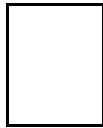
2.3.1 Rectangle

Figure 2-2 illustrates a rectangle.

Figure 2-2 Rectangle

Geometry RECT_1:

(12,24) (15,24)



(12,15) (15,15)

```
SDO_GEOMETRY Column = (
SDO_GTYPE = 2003
SDO_SRID = NULL
SDO_POINT = NULL
SDO_ELEM_INFO = (1,1003,3)
SDO_ORDINATES = (12,15,15,24) )
```

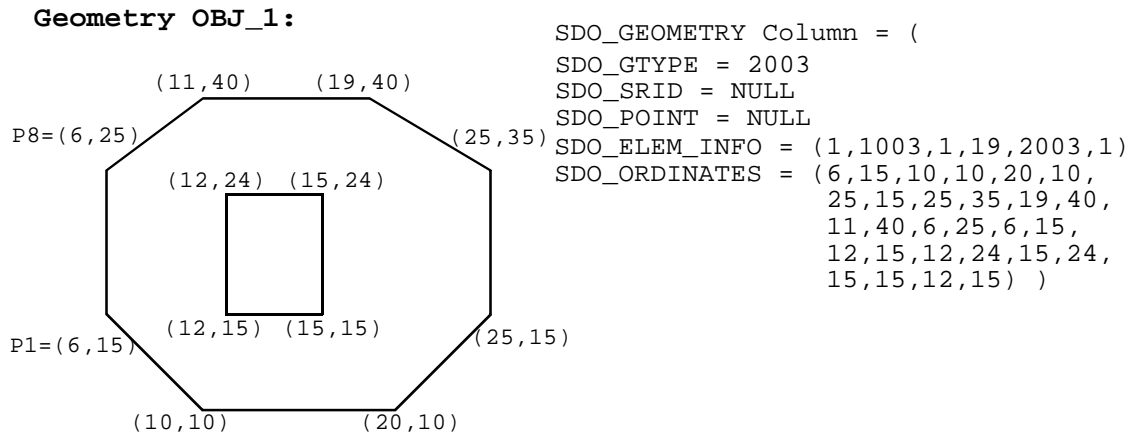
Notes on Figure 2-2:

- In SDO_GTYPE, the 2 indicates two-dimensional and the 3 indicates a polygon.
- In SDO_ELEM_INFO, the final 3 in 1,1003,3 indicates that this is a rectangle. Because it is a rectangle, only two ordinates are specified (lower-left 12,15 and upper-right 15,24).

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



Notes on Figure 2-3:

- In SDO_ELEM_INFO, 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.
- In SDO_ELEM_INFO, 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).
- 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 **multi-polygon** (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

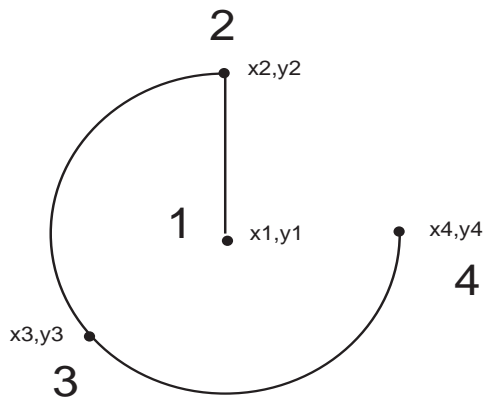
The elements in SDO_ELEM_INFO and SDO_ORDINATES must be in one of the following orders (depending on whether you specify Polygon A or Polygon B first):

- 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



NU-3746A-AI

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.

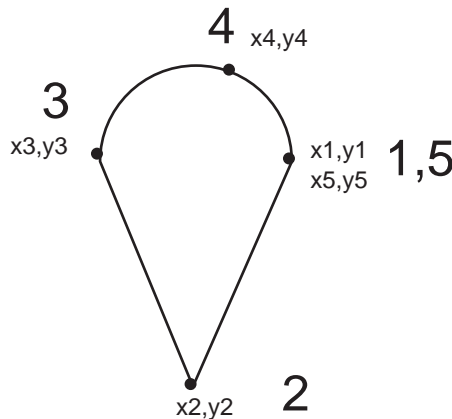
Assuming the vertices are 2-dimensional, the coordinates for the end point of the first line string are at ordinates 3 and 4.

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. 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.

These metadata views are available in the schema of each user having tables with a column, or columns, of type `SDO_GEOMETRY`. For example, if user Herman has tables named *Roads*, *Parks*, and *Rivers*, each with a column named *theGeometry* of type `SDO_GEOMETRY`, the following metadata views are available in the *Herman* schema: `USER_SDO_GEOM_METADATA`, `ALL_SDO_GEOM_METADATA`, and `DBA_SDO_GEOM_METADATA`.

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 are new for release 8.1.6. If you are migrating from a previous release of Spatial, see [Appendix B](#), especially [Section B.5](#) on migrating from release 8.1.5 to 8.1.6.

Each metadata view has the following definition:

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

The TABLE_NAME column contains the name of a feature table, such as *Roads* or *Parks*, that has a column of type SDO_GEOMETRY.

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*).

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

The SRID column is reserved for future use, and should be specified as null.

For the DIMINFO column, 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 so on. 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.

2.5 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 and Tables

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 are new for release 8.1.6. If you are migrating from a previous release of Spatial, see [Appendix B](#), especially [Section B.5](#) on migrating from release 8.1.5 to 8.1.6.

There is also one index data table per spatially indexed column in that schema.

Thus, for example, if user Herman has five feature tables, each with a spatial index on their respective SDO_GEOMETRY type column, then Herman's schema has one USER_SDO_INDEX_METADATA view, one ALL_SDO_INDEX_METADATA view, perhaps one DBA_SDO_INDEX_METADATA view, and five tables containing spatial index data. Each index data table is named using the user-specified index name and a suffix that indicates the spatial index type (fixed or hybrid, as defined in [Section 1.7](#)) and the values of the relevant index parameters.

The USER_SDO_INDEX_METADATA, ALL_SDO_INDEX_METADATA, and DBA_SDO_INDEX_METADATA views contain the same columns, as shown [Table 2-3](#).

Table 2-3 Columns in the xxx_SDO_INDEX_METADATA Views

Column Name	Data Type	Purpose
SDO_COLUMN_NAME	VARCHAR2	The column name on which this index is built.
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_METADATA	RAW	If applicable, this column contains the metadata portion of the SDO_GROUPCODE or SDO_CODE for a fixed-level index.
SDO_INDEX_NAME	VARCHAR2	The name of the index.
SDO_INDEX_OWNER	VARCHAR2	The owner of the index.
SDO_INDEX_PRIMARY	NUMBER	Indicates if this is a primary or secondary index. 1 = primary, 2 = secondary.
SDO_INDEX_TABLE	VARCHAR2	Name of the spatial index table (described in Section 2.5.2).
SDO_INDEX_TYPE	VARCHAR2	Reserved for future use.
SDO_INITIAL_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_LEVEL	NUMBER	The fixed tiling level at which to tile all objects in the feature table.
SDO_MAX_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_MAXLEVEL	NUMBER	The maximum level for any tile for any object. It will always be greater than the SDO_LEVEL value.
SDO_MIN_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.

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

Column Name	Data Type	Purpose
SDO_NEXT_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_NUMTILES	NUMBER	Suggested number of tiles per object that should be used to approximate the shape.
SDO_PCTINCREASE	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_RTREE_DIMENSIONALITY	NUMBER	Reserved for future use.
SDO_RTREE_FANOUT	NUMBER	Reserved for future use.
SDO_RTREE_HEIGHT	NUMBER	Reserved for future use.
SDO_RTREE_NUM_NODES	NUMBER	Reserved for future use.
SDO_RTREE_ROOT	VARCHAR2	Reserved for future use.
SDO_RTREE_SEQ_NAME	VARCHAR2	Reserved for future use.
SDO_TABLESPACE	VARCHAR2	Same as in the basic SQL CREATE TABLE statement. Tablespace in which to create the SDOINDEX table.
SDO_TSNAME	VARCHAR2	The schema name of the SDO_INDEX_TABLE.

2.5.2 Spatial Index Table Definition

Each spatial index data 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>I</i> if the tile is inside the geometry, or <i>B</i> 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`, which is the recommended indexing for the object-relational model.

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 efficiently query spatial data:

1. Load data into 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 the *Oracle8i Utilities* manual.)

3.1.1.1 Bulk Loading the `SDO_GEOMETRY` Object

The following example assumes a table called `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|3|1|3|1|/
  #+
  #-122.4215|37.7862|-122.422|37.7869|-122.421|37.789|-122.42|37.7866|
  #-122.4215|37.7862|/
  2|3|1|3|1|/
  #+
  #-122.4019|37.8052|-122.4027|37.8055|-122.4031|37.806|-122.4012|37.8052|
  #-122.4019|37.8052|/
  3|3|1|3|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| 1| -122.4215| 37.7862|
2| 1| -122.4019| 37.8052|
3| 1| -122.426| 37.803|
4| 1| -122.4171| 37.8034|
5| 1| -122.416151| 37.8027228|

```

3.1.2 Transactional Insert Using SQL

Oracle Spatial uses standard Oracle8i 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 SQL statements in Oracle8i have 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.)

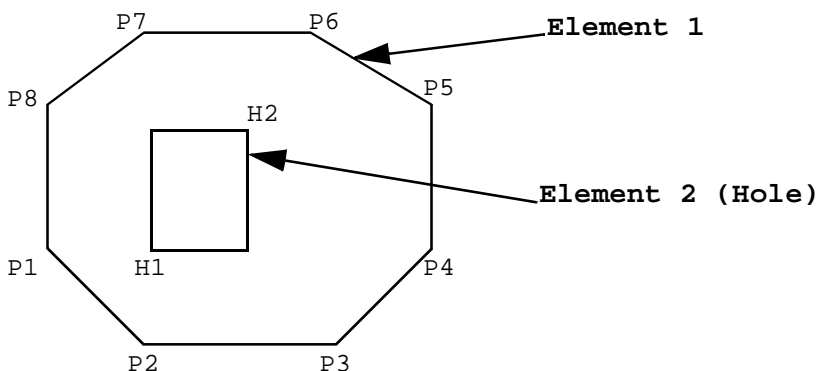
3.1.2.1 Polygon with Hole

The geometry to be stored is a polygon with a hole, as shown in [Figure 3-1](#). The coordinate values for elements 1 and 2 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)]
```

Figure 3-1 Example Geometry OBJ_1

Geometry OBJ_1:



This example assumes that table *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),
                                     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

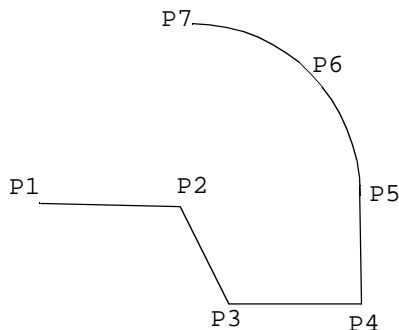
A compound line string is a connected sequence of straight line segments and circular arcs. [Figure 3-2](#) is an example of a compound line string. The coordinate values for points P1..P7 that describe the line string OBJ_2 are:

```
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.

Figure 3–2 Line String Consisting of Arcs and Straight Line Segments**Geometry OBJ_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. 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)]
```

This example assumes the *Parks* table was created as follows:

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

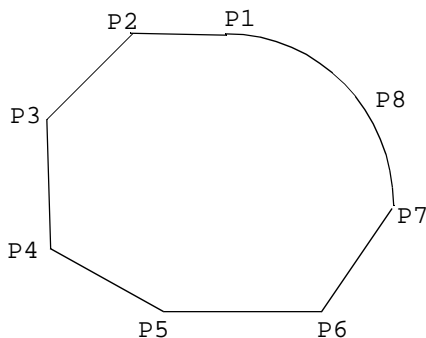
The SQL statement for inserting this compound polygon is:

```
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 3, the `SDO_ELEM_INFO` has 3 triplet values. The first triplet (1,5,2) identifies the element as a compound polygon (ETYPE 5) 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.

Figure 3–3 Compound Polygon

Geometry OBJ_3:



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). 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)]
```

This example assumes the table *Parks* has been created as follows:

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

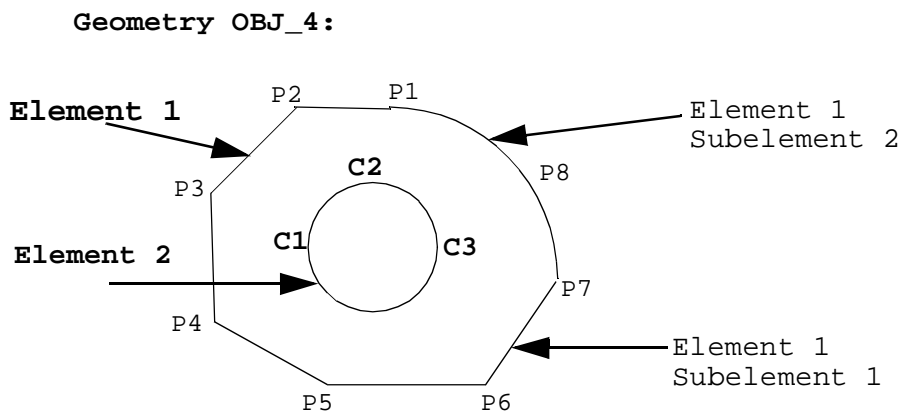
The SQL statement for inserting this compound polygon with holes is:

```
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 (1,5,2) 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 (19,3,4) represents element 2. Element 2 starts at offset 19, is of ETYPE 3, 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.

Figure 3–4 Compound Polygon with a Hole



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. This is done by approximating geometries with tiles. For each geometry, you will have a set of tiles that fully covers the geometry.

3.2.1 Determining Index Creation Behavior

Spatial provides two methods for spatial indexing, fixed and hybrid. Hybrid indexing is recommended for the Spatial object-relational model. If specified correctly, it will provide better selectivity and spatial join performance for most data sets and application scenarios.

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:

<code>SDO_LEVEL</code>	<code>SDO_NUMTILES</code>	Action
Not specified.	Not specified.	Error.
<code>>= 1</code>	Not specified.	Fixed indexing, (indexing with fixed-size tiles).
<code>>= 1</code>	<code>>= 1</code>	Hybrid indexing with fixed-size and variable-sized tiles. The <code>SDO_LEVEL</code> column defines the fixed tile size. The <code>SDO_NUMTILES</code> column defines the number of variable tiles to generate per geometry.
Not specified.	<code>>= 1</code>	Not supported.

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

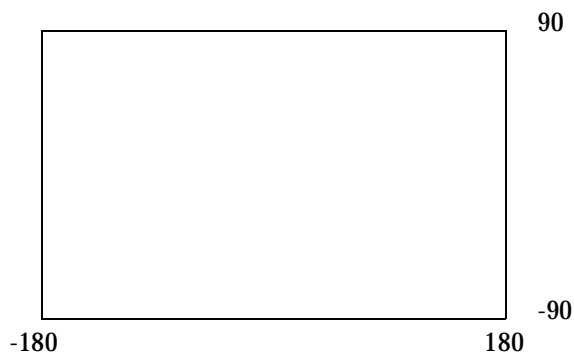
Oracle recommends using hybrid indexing when indexing a geometry using the object-relational model. Because fixed indexing is an integral part of hybrid indexing, it is important to understand the information in this section. Hybrid indexing is discussed in [Section 3.2.3](#).

While not the preferred method, you can use fixed-size tiles to index the object-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, 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.

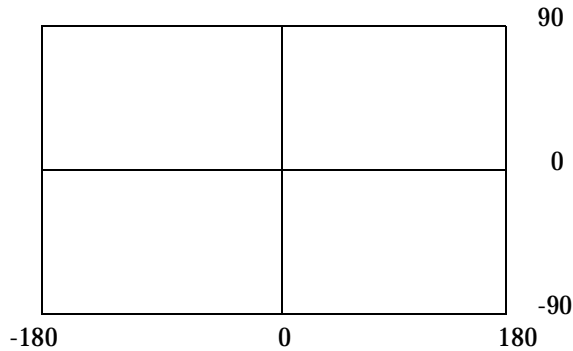
The domain used for indexing is defined by the upper and lower boundaries of each dimension stored in the `DIMINFO` column of the `USER_SDO_GEOM_METADATA` view, which contains an entry for the table and geometry column to spatially index. A typical domain could be -180 to 180 degrees for longitude¹, and -90 to 90 degrees for latitude, as represented in [Figure 3-5](#).

Figure 3-5 Sample Domain

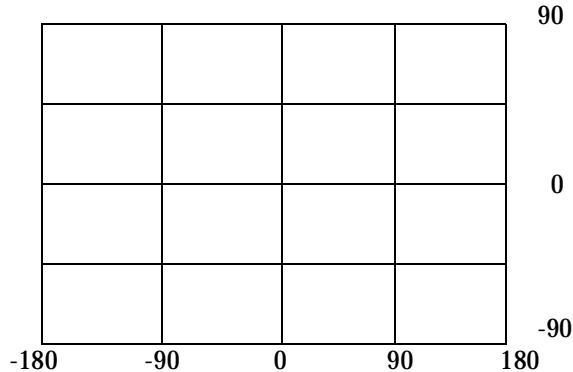


If the `SDO_LEVEL` column is set to 1, then the tiles created by the indexing mechanism are the same size as tiles at the first level of tessellation. Each tile would be 180 degrees by 90 degrees as shown in [Figure 3-6](#).

¹ The transference of the domain onto a sphere or Mercator projection is left up to an application. Spatial treats the domain as a flat Cartesian grid.

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 necessarily 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:

$$\text{length} = (\text{upper_bound} - \text{lower_bound}) / 2^{\text{sdo_level}}$$

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:

$$\begin{aligned} \text{length for dimension X} &= (180 - (-180)) / 2^2 \\ &= (360) / 4 \end{aligned}$$

$$\begin{aligned}
 &= 90 \\
 \text{length for dimension Y} &= (90 - (-90)) / 2^2 \\
 &= (180) / 4 \\
 &= 45
 \end{aligned}$$

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. The [SDO_TUNE.ESTIMATE_TILING_LEVEL](#) function can be used to determine an appropriate level for indexing with fixed-size tiles. See [Chapter 14](#) for a description of this function.

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 is an important part of any database design, and is essential in a spatial database where the data often represents actual physical locations.

In [Example 3-3](#), 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 3-3 Create a Fixed Index

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

The value in `SDO_LEVEL` is used while tessellating objects. Increasing the level results in smaller tiles and better geometry approximations. See the description of the `ESTIMATE_TILING_LEVEL` function in [Chapter 14](#) for information on estimating the tiling level in several different ways.

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.

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` parameter, the better the tiles will approximate the geometry being covered. This improves the selectivity of the primary filter, but 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, and other techniques, are described in [Appendix A, "Tuning Tips and Sample SQL Scripts"](#).

In [Example 3–4](#), 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 3–4 Create 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 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');
```

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.

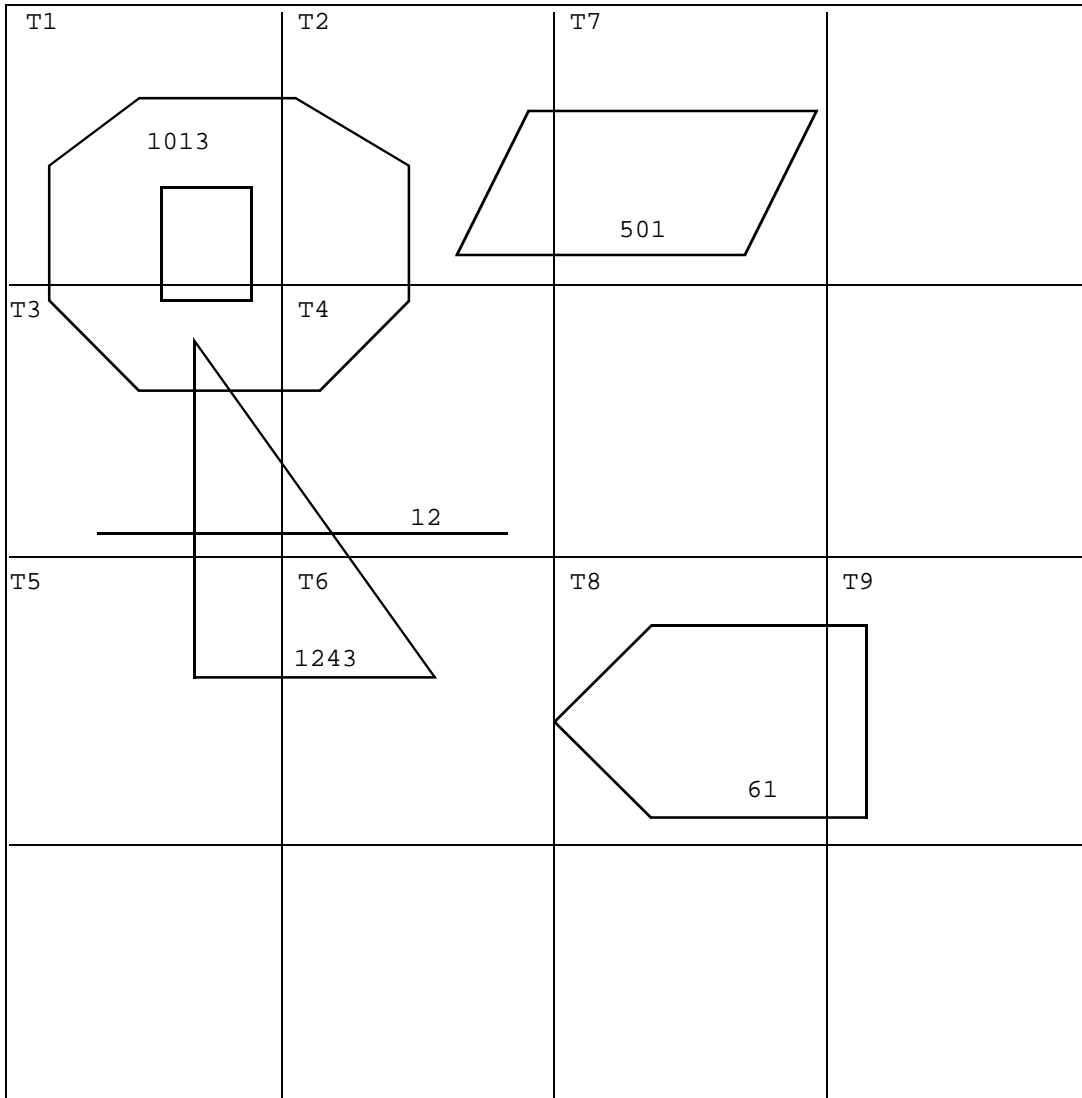
- The **primary filter** permits fast selection of candidate records to pass along to the secondary filter. The primary filter uses geometry approximations (or index tiles) to reduce computational complexity and is considered a lower-cost filter.
- The **secondary filter** applies exact computational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

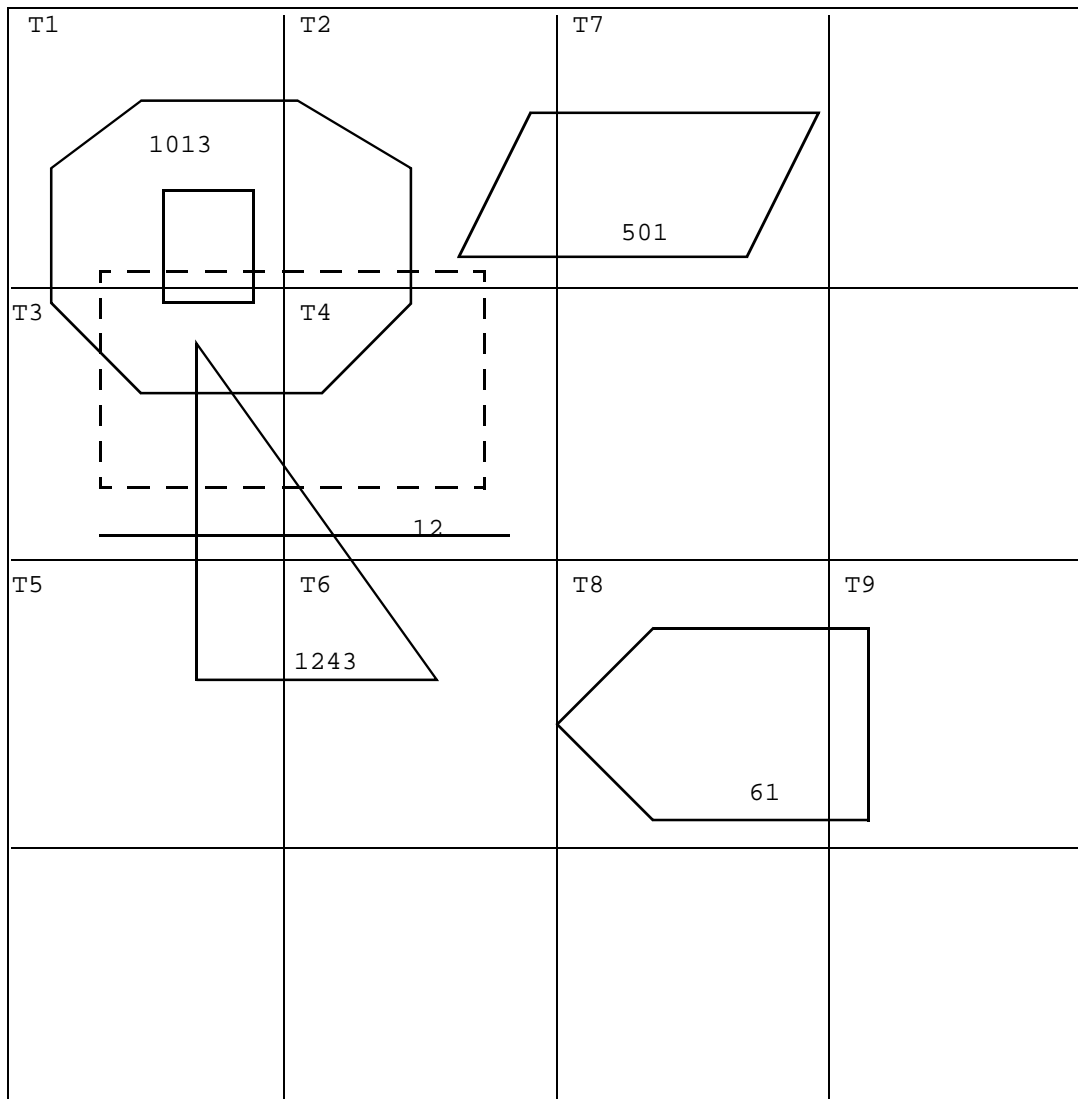
Consider the following layer containing several objects in [Figure 4-1](#). Each object is labeled with its SDO_GID. The relevant tiles are labeled with ‘Tn’.

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 prior to using it.

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 9](#) 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.

As mentioned previously, these examples are performed with fixed-size tiles, which is not the recommended indexing method for the object model. If hybrid indexing were used, the selectivity would improve.

[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 reuses the existing index, but performance degrades.

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 a quoted string of keyword value pairs that determine the behavior of the operator. See the [SDO_RELATE](#) operator in [Chapter 9](#) 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_RELATE](#) operator reuses the existing index, but performance degrades.

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 a quoted string of keyword value pairs that determines the behavior of the operator. See the `SDO_WITHIN_DISTANCE` operator in [Chapter 9](#) for a list of parameters.

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

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

The distance units are based on the geometry coordinate system in use. The Spatial product 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 like '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' ;
```

4.3 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.

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 because we need 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, highwaysB
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 A.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, b.gid
FROM A.T1 a, B.T2 b
WHERE sdo_filter(a.geometry, b.geometry, 'querytype=JOIN') = 'TRUE';
```

Indexing Statements for Object-Relational Model

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	Alter a spatial index on a column of type MDSYS.SDO_GEOMETRY.
ALTER INDEX REBUILD	Rebuild a spatial index on a column of type MDSYS.SDO_GEOMETRY.
ALTER INDEX RENAME TO	Change the name of a spatial index on a column of type MDSYS.SDO_GEOMETRY.
CREATE INDEX	Create a spatial index on a column of type MDSYS.SDO_GEOMETRY.
DROP INDEX	Delete 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 type, (fixed or hybrid), and characteristics of the spatial index.

Keyword	Description
<i>add_index</i>	Specifies the name of the new index table to add. Data type is VARCHAR2.
<i>delete_index</i>	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 DROP INDEX. Data type is VARCHAR2.
<i>sdo_level</i>	Specifies the desired fixed-size tiling level. Data type is NUMBER.
<i>sdo_numtiles</i>	Specifies the number of variable-sized tiles to be used in tessellating an object. Data type is NUMBER.
<i>sdo_maxlevel</i>	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be greater than the <i>sdo_level</i> value. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
<i>sdo_commit_interval</i>	Specifies the number of underlying table rows that are processed between commit intervals for the index data. 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.

PHYSICAL_STORAGE_PARAMS Determine 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_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
<i>tablespace</i>	Specifies the tablespace in which the index data table is created. This parameter is the same as <code>TABLESPACE</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>initial</i>	Is the same as <code>INITIAL</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>next</i>	Is the same as <code>NEXT</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>minextents</i>	Is the same as <code>MINEXTENTS</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>maxextents</i>	Is the same as <code>MAXEXTENTS</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>pctincrease</i>	Is the same as <code>PCTINCREASE</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>btree_initial</i>	Is the same as <code>INITIAL</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.
<i>btree_next</i>	Is the same as <code>NEXT</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.
<i>btree_pctincrease</i>	Is the same as <code>PCTINCREASE</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.

Prerequisites

- You must have `EXECUTE` privileges on the index type and its implementation type.
- The spatial index to be altered is 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.

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](#)

ALTER INDEX REBUILD

Purpose

Rebuilds a spatial index.

Syntax

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

Keywords and Parameters

REBUILD_ PARAMS	Specifies in a command string the index parameters to use in rebuilding the spatial index.
Keyword	Description
<i>rebuild_index</i>	Specifies the name of the spatial index table to be rebuilt. Data type is VARCHAR2.
<i>sdo_level</i>	Specifies the desired fixed-size tiling level. Data type is NUMBER.
<i>sdo_numtiles</i>	Specifies the number of variable-sized tiles to be used in tessellating an object. Data type is NUMBER.
<i>sdo_maxlevel</i>	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be greater than the <i>sdo_level</i> value. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
<i>sdo_commit_interval</i>	Specifies the number of underlying table rows that are processed between commit intervals for the index data. 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.
<i>layer_gtype</i>	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.

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_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
<i>tablespace</i>	Specifies the tablespace in which the index data table is created. Same as 'TABLESPACE' in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>initial</i>	Is the same as <code>INITIAL</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>next</i>	Is the same as <code>NEXT</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>minextents</i>	Is the same as <code>MINEXTENTS</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>maxextents</i>	Is the same as <code>MAXEXTENTS</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>pctincrease</i>	Is the same as <code>PCTINCREASE</code> in the <code>STORAGE</code> clause of a <code>CREATE TABLE</code> statement.
<i>btree_initial</i>	Is the same as <code>INITIAL</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.
<i>btree_next</i>	Is the same as <code>NEXT</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.
<i>btree_pctincrease</i>	Is the same as <code>PCTINCREASE</code> in the <code>STORAGE</code> clause of a <code>CREATE INDEX</code> statement in the case of a standard B-tree index.

Prerequisites

- You must have `EXECUTE` privileges on the index type and its implementation type.
- The spatial index to be altered is 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 command does not remember any previous index parameters. All parameters should be specified for the index you want to rebuild.

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

Purpose

Alters the name of a spatial index.

Syntax

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

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 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

Purpose

Creates a spatial index on a column of type MDSYS.SDO_GEOMETRY.

Syntax

```
CREATE INDEX [schema.]<index_name> ON [schema.]<tableName> (column)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  [PARAMETERS 'index_params [physical_storage_params]');
```

Keywords and Parameters

<i>INDEX_PARAMS</i>	Determine the type, fixed or hybrid, and characteristics of the spatial index.
Keyword	Description
<i>sdo_level</i>	Specifies the desired fixed-size tiling level. Data type is NUMBER.
<i>sdo_numtiles</i>	Specifies the number of variable-sized tiles to be used in tessellating an object. Data type is NUMBER.
<i>sdo_maxlevel</i>	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be greater than the <i>sdo_level</i> value. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
<i>sdo_commit_interval</i>	Specifies the number of underlying table rows that are processed between commit intervals for the index data. 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.
<i>layer_gtype</i>	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.

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
<i>tablespace</i>	Specifies the tablespace in which the index data table is created. Same as 'TABLESPACE' in the STORAGE clause of a CREATE TABLE statement.
<i>initial</i>	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
<i>next</i>	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
<i>minextents</i>	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
<i>maxextents</i>	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
<i>pctincrease</i>	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
<i>btree_initial</i>	Is the same as INITIAL in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index.
<i>btree_next</i>	Is the same as NEXT in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index.
<i>btree_pctincrease</i>	Is the same as PCTINCREASE in the STORAGE clause of a CREATE INDEX statement in the case of a standard B-tree index.

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

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.

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:

- *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.
- *sdo_maxlevel* equals 32. Modification is not recommended.

The *sdo_level* value must be greater than zero and less than the *sdo_maxlevel* value.

The *sdo_numtiles* value is considered a recommendation. In some cases, this value may be overwritten 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 is shown in [Table 5-2](#).

Table 5-2 SDO_LEVEL and SDO_NUMTILES Combinations

SDO_LEVEL	SDO_NUMTILES	Type of Spatial Index
Not specified.	Not specified.	Error.
>= 1	Not specified.	Fixed indexing, (indexing with fixed-size tiles).

Table 5–2 SDO_LEVEL and SDO_NUMTILES Combinations (Cont.)

SDO_LEVEL	SDO_NUMTILES	Type of Spatial Index
>= 1	>= 1	Hybrid indexing with fixed-size and variable-sized tiles. The <i>sdo_level</i> value defines the fixed tile size. The <i>sdo_numtiles</i> value defines the number of variable tiles to generate per geometry.
Not specified.	>= 1	Not supported.

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.

Example

The following example creates an index named *qtree*.

```
CREATE INDEX qtree ON POLY_4PT(geometry)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX
  PARAMETERS('sdo_numtiles=4 sdo_level=6 sdo_maxlevel=36
    sdo_commit_interval=500 tablespace=system initial=10K
    next=10K pctincrease=10 minextents=10 maxextents=20');
```

Related Topics

- [ALTER INDEX](#)
- [DROP INDEX](#)

DROP INDEX

Purpose

Deletes a spatial index.

Syntax

```
DROP INDEX [schema.]index [FORCE]
```

Keywords and Parameters

<i>FORCE</i>	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

1. `DROP INDEX oldindex`
2. `DROP INDEX oldindex FORCE`

Related Topics

- [CREATE INDEX](#)

Tuning Functions and Procedures for Object-Relational Model

This chapter contains descriptions of the tuning functions and procedures shown in Table 6-1.

Table 6-1 *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

Purpose

Calculates the average minimum bounding rectangle (MBR) for a geometry object column.

Format

```
SDO_TUNE.AVERAGE_MBR(  
    table_name  IN VARCHAR2,  
    column_name IN VARCHAR2,  
    width       OUT NUMBER,  
    height      OUT NUMBER);
```

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.)

Example

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

Purpose

Estimates the spatial index performance such as query selectivity and window query time for a column of type SDO_GEOMETRY.

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;
```

Parameters

table_name

Geometry table.

column_name

Geometry column for which the tiling time is to be estimated.

sample_ratio

Approximate size ratio between the original layer and the sample layer (to be generated in order to perform the estimate). The default is 20: that is, the sample layer will be approximately 1/20 (0.05) the size of 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).

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.

Example

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

Purpose

Estimates the appropriate SDO_LEVEL value to use when indexing with hybrid or fixed-size tiles.

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;
```

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 -- Use the rectangle defined by your coordinate system.
- ALL_GID_EXTENT -- Use the minimum bounding rectangle that encompasses all the geometric objects in the column. This estimate is recommended for most applications.
- AVG_GID_EXTENT -- Use 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 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.

Example

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, 'AVG_GID_EXTENT')
FROM DUAL;

SDO_TUNE.ESTIMATE_TILING_LEVEL('COLA_MARKETS', 'SHAPE', 10000, 'AVG_GID_EXTENT')
```

8

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

Purpose

Returns the estimated time (in seconds) to tessellate a column of type SDO_GEOMETRY.

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;
```

Parameters

table_name

Geometry table.

column_name

Geometry column for which the tiling time is to be estimated.

sample_ratio

Approximate size ratio between the original layer and the sample layer (to be generated in order to perform the estimate). The default is 20: that is, the sample layer will be approximately 1/20 (0.05) the size of 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).

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.

Example

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

Purpose

Estimates the total number of spatial tiles for a layer.

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_targetiles OUT INTEGER  
) RETURN INTEGER;
```

Parameters

table_name

Geometry table.

column_name

Geometry column for which the total number of spatial tiles is to be estimated.

sample_ratio

Approximate size ratio between the original layer and the sample layer (to be generated in order to perform the estimate). The default is 20: that is, the sample layer will be approximately 1/20 (0.05) the size of 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).

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_targetiles

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_targetiles* 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.

Example

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_targetiles  INTEGER;
  ret_integer     INTEGER;
BEGIN
  ret_integer := SDO_TUNE.ESTIMATE_TOTAL_NUMTILES(
    table_name,
    column_name,
    sample_ratio,
    tiling_level,
    num_tiles,
    num_targetiles
  );
```



```
END;  
/
```

SDO_TUNE.EXTENT_OF

Purpose

Returns the minimum bounding rectangle of all geometries in a column of type SDO_GEOMETRY.

Format

```
SDO_TUNE.EXTENT_OF(  
    table_name  IN VARCHAR2,  
    column_name IN VARCHAR2  
    ) RETURN MDSYS.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.

Example

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

Purpose

Generates statistical histograms based on columns of type SDO_GEOMETRY.

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);
```

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 to 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

Purpose

Provides information about each geometry type stored in a column of type SDO_GEOMETRY.

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]);
```

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.

Example

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%)
.
```

Geometry Functions for Object-Relational Model

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

This group also includes the following deprecated functions: SDO_POLY_DIFFERENCE, SDO_POLY_INTERSECTION, SDO_POLY_UNION, SDO_POLY_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.

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

Function	Description
SDO_GEOM.SDO_DIFFERENCE	Returns a geometry object that is the topological difference (MINUS operation) of two geometry objects.
SDO_GEOM.SDO_DISTANCE	Computes the distance between two geometry objects.
SDO_GEOM.SDO_INTERSECTION	Returns a geometry object that is the topological intersection (AND operation) of two geometry objects.
SDO_GEOM.SDO_LENGTH	Computes the length or perimeter of a geometry.
SDO_GEOM.SDO_POINTONSURFACE	Returns a point that is guaranteed to be on the surface of a polygon.
SDO_GEOM.SDO_POLY_DIFFERENCE	Generates a polygon representing the difference between two geometries.
SDO_GEOM.SDO_POLY_INTERSECTION	Generates a polygon representing the intersection of two geometries.
SDO_GEOM.SDO_POLY_UNION	Generates a polygon representing the union of two geometries.
SDO_GEOM.SDO_POLY_XOR	Generates a polygon representing the symmetric difference between two geometries.
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 are deprecated and will not be supported in future versions of Oracle Spatial. 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

Purpose

Examines two geometry objects to determine their spatial relationship.

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;
```

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 of 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 a 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](#)).

Example

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

Purpose

Returns the area of a two-dimensional polygon.

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;
```

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 *interMedia* Locator, which is a component of the Oracle *interMedia* 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](#)).

Example

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

Purpose

Generates a buffer polygon around a geometry object.

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;
```

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 *interMedia* Locator, which is a component of the Oracle *interMedia* product.

Example

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

Purpose

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

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;
```

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](#)).

Example

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

Purpose

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

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;
```

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](#)).

Example

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

Purpose

Returns a geometry object that is the topological difference (*MINUS* operation) of two geometry objects.

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;
```

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](#)).

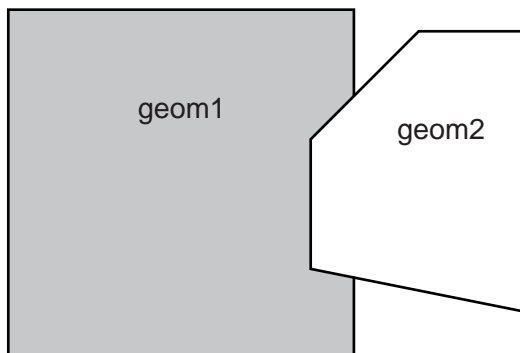
tolerance

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.

Example

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

Purpose

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.

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;
```

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 *interMedia* Locator, which is a component of the Oracle *interMedia* 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](#)).

Example

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

Purpose

Returns a geometry object that is the topological intersection (*AND* operation) of two geometry objects.

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;
```

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](#)).

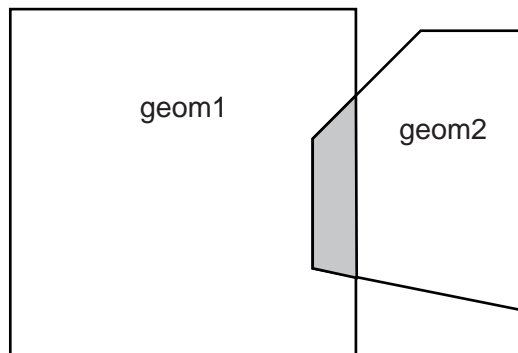
tolerance

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.

Example

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

Purpose

Returns the length or perimeter of a geometry object.

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;
```

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 *interMedia* Locator, which is a component of the Oracle *interMedia* product.

Example

The following example returns the perimeters of geometry objects stored in the *cola_markets* table. The first statement returns the perimeters 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 area 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

Purpose

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

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;
```

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 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](#)).

Example

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_POLY_DIFFERENCE

Purpose

Returns a geometry object representing the difference (A *minus* B) of two polygon objects.

Format

```
SDO_GEOM.SDO_POLY_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;
```

Parameters

geom1

Polygon 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

Polygon geometry object.

dim2

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

Usage Notes

The SDO_POLY_DIFFERENCE function is deprecated and will not be supported in future releases of Oracle Spatial. You should use instead the [SDO_GEOM.SDO_DIFFERENCE](#) function.

This function returns an empty geometry if any of the following occurs:

- *geom1* or *geom2*, or both, are not polygons.
- *geom1* and *geom2* are tangent polygons. (Tangent polygons have only a common line or point, but do not otherwise intersect.)
- *geom2* is INSIDE *geom1* or EQUAL to *geom1*.

Related Topics

- [SDO_GEOM.SDO_BUFFER](#)
- [SDO_GEOM.SDO_POLY_INTERSECTION](#)
- [SDO_GEOM.SDO_POLY_UNION](#)
- [SDO_GEOM.SDO_POLY_XOR](#)

SDO_GEOM.SDO_POLY_INTERSECTION

Purpose

Returns a geometry object representing the intersection (*A and B*) of two polygon objects.

Format

```
SDO_GEOM.SDO_POLY_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;
```

Parameters

geom1

Polygon 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

Polygon geometry object.

dim2

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

Usage Notes

The SDO_POLY_INTERSECTION function is deprecated and will not be supported in future releases of Oracle Spatial. You should use instead the [SDO_GEOM.SDO_INTERSECTION](#) function.

This function returns an empty geometry if any of the following occurs:

- *geom1* or *geom2*, or both, are not polygons.
- *geom1* and *geom2* are tangent polygons. (Tangent polygons have only a common line or point, but do not otherwise intersect.)
- *geom1* or *geom2* are disjoint.

Related Topics

- [SDO_GEOM.SDO_BUFFER](#)
- [SDO_GEOM.SDO_UNION](#)
- [SDO_GEOM.SDO_POLY_UNION](#)
- [SDO_GEOM.SDO_POLY_XOR](#)

SDO_GEOM.SDO_POLY_UNION

Purpose

Returns a geometry object representing the union (A *or* B) of two polygon objects.

Format

```
SDO_GEOM.SDO_POLY_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;
```

Parameters

geom1

Polygon 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

Polygon geometry object.

dim2

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

Usage Notes

The SDO_POLY_UNION function is deprecated and will not be supported in future releases of Oracle Spatial. You should use instead the [SDO_GEOM.SDO_UNION](#) function.

This function returns an empty geometry if any of the following occurs:

- *geom1* or *geom2*, or both, are not polygons.
- *geom1* and *geom2* are tangent polygons. (Tangent polygons have only a common line or point, but do not otherwise intersect.)

Related Topics

- [SDO_GEOM.SDO_BUFFER](#)
- [SDO_GEOM.SDO_UNION](#)
- [SDO_GEOM.SDO_POLY_INTERSECTION](#)
- [SDO_GEOM.SDO_POLY_XOR](#)

SDO_GEOM.SDO_POLY_XOR

Purpose

Returns a geometry object representing the symmetric difference (A *xor* B) of two polygon objects.

Format

```
SDO_GEOM.SDO_POLY_XOR(  
    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;
```

Parameters

geom1

Polygon 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

Polygon geometry object.

dim2

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

Usage Notes

The SDO_POLY_XOR function is deprecated and will not be supported in future releases of Oracle Spatial. You should use instead the [SDO_GEOM.SDO_XOR](#) function.

This function returns an empty geometry if any of the following occurs:

- *geom1* or *geom2*, or both, are not polygons.
- *geom1* and *geom2* are tangent polygons. (Tangent polygons have only a common line or point, but do not otherwise intersect.)

Related Topics

- [SDO_GEOM.SDO_BUFFER](#)
- [SDO_GEOM.SDO_UNION](#)
- [SDO_GEOM.SDO_POLY_INTERSECTION](#)
- [SDO_GEOM.SDO_POLY_UNION](#)

SDO_GEOM.SDO_UNION

Purpose

Returns a geometry object that is the topological union (*OR* operation) of two geometry objects.

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;
```

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](#)).

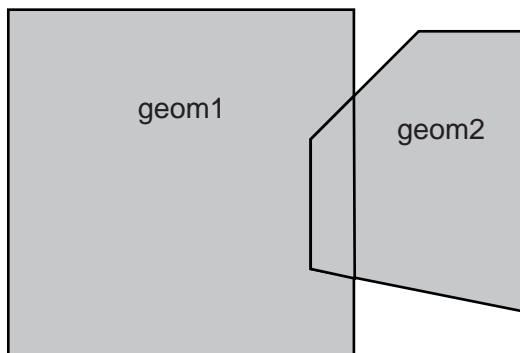
tolerance

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.

Example

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

Purpose

Returns a geometry object that is the topological symmetric difference (*XOR* operation) of two geometry objects.

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;
```

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](#)).

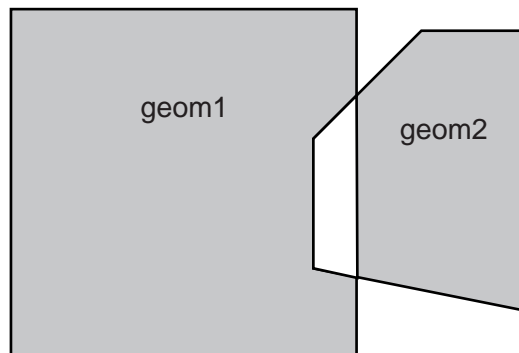
tolerance

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.

Example

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

Purpose

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

Format

```
SDO_GEOM.VALIDATE_GEOMETRY(  
    geom IN MDSYS.SDO_GEOMETRY,  
    dim  IN MDSYS.SDO_DIM_ARRAY  
    ) RETURN VARCHAR2;  
  
or  
  
SDO_GEOM.VALIDATE_GEOMETRY(  
    geom  IN MDSYS.SDO_GEOMETRY,  
    tolerance IN NUMBER  
    ) RETURN VARCHAR2;
```

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 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 the following:

- Polygons have at least four points, which includes the point that closes the polygon. (The last point is the same as the first.)
- Line strings have at least two points.
- In a heterogeneous collection (SDO_GTYPE 4) or multipolygon (SDO_GTYPE 7), all polygons are disjoint.
- 1-digit and 4-digit SDO_ETYPE values are not mixed (that is, both used) in defining polygon ring elements.

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](#)).

Example

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

Purpose

Examines a geometry column to determine if the stored geometries follow the defined rules for geometry objects.

Format

```
SDO_GEOM.VALIDATE_LAYER(  
    geom_table    IN VARCHAR2,  
    geom_column  IN VARCHAR2,  
    pkey_column   IN VARCHAR2,  
    result_table  IN VARCHAR2);
```

Parameters

geom_table

Geometry table.

geom_column

Geometry object column to be examined.

pkey_column

The primary key column. This can be the rowid or any other single-column primary key; however, the specified column must be numeric (NUMBER data type).

result_table

Result table to hold the validation results.

Usage Notes

This procedure populates the result table with validation results. (VALIDATE_LAYER is a procedure, not a function.)

An empty result table (*result_table* parameter) must be created prior to calling this procedure. The format of the result table is: (pkey_column NUMBER, result VARCHAR2(10)).

In the result table, for each *pkey_column* the *result* column contains TRUE if the geometry is valid. If it is invalid, the *result* column for that geometry contains an Oracle error message number. You can then look up this error message to determine the cause of the failure.

This procedure checks for the following:

- Polygons have at least four points, which includes the point that closes the polygon. (The last point is the same as the first.)
- Line strings have at least two points.
- In a heterogeneous collection (SDO_GTYPE *d004*) or multipolygon (SDO_GTYPE *d007*), all polygons are disjoint. For an explanation of SDO_GTYPE values, see [Section 2.2](#).

Example

The following example validates the geometry objects stored in the *shape* column of the *cola_markets* table. The example includes the creation of the results table. (The example uses the definitions and data from [Section 2.1](#).)

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

PL/SQL procedure successfully completed.

```
SELECT * from val_results;
```

```
      MKT_ID RESULT
-----
          1 TRUE
          2 TRUE
          3 TRUE
          4 TRUE
```

Related Topics

- [SDO_GEOM.VALIDATE_GEOMETRY](#)

SDO_GEOM.WITHIN_DISTANCE

Purpose

Determines if two spatial objects are within some specified Euclidean distance from each other.

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;
```

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](#)).

Example

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](#)

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 8-1.

Table 8-1 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 metadata tables.
SDO_MIGRATE.OGIS_METADATA_TO	Reads a temporary table used when migrating OGIS metadata tables.

SDO_MIGRATE.FROM_815_TO_81X

Purpose

Migrates data from Spatial release 8.1.5 to the current release.

Format

```
SDO_MIGRATE.FROM_815_TO_81X(  
    tablename IN VARCHAR2);
```

Parameters

tablename
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 *tablename* 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('ROADS');
```


SDO_MIGRATE.TO_734

Purpose

Migrates data from a previous release of Spatial Data Option to release 7.3.4.

Format

```
SDO_MIGRATE.TO_734(  
    sn      IN VARCHAR2,  
    layer   IN VARCHAR2,  
    tess_type IN VARCHAR2,  
    param   IN INTEGER);
```

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

Purpose

Migrates data from a previous release of Spatial Cartridge or Spatial Data Option to Oracle Spatial release 8.1.5.

Format

```
SDO_MIGRATE.TO_81X(  
    layer          IN VARCHAR2,  
    newtabname     IN VARCHAR2,  
    gidcolumn      IN VARCHAR2,  
    geocolname     IN VARCHAR2,  
    layer_gtype    IN VARCHAR2,  
    updateflag     IN VARCHAR2);
```

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 prior to 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 (*d00n*) shown in [Table 2-1](#) in [Section 2.2.1](#).
- This procedure assigns SDO_GTYPE values of *d001*, *d002*, or *d003*. (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 *d005*, *d006*, or *d007*, 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

Purpose

Called at the source database when migrating from one 8.1.5 database to another 8.1.5 database. The procedure migrates OGIS metadata entries from schemas owned by MDSYS.

Format

SDO_MIGRATE.OGIS_METADATA_FROM

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

Purpose

Used at the destination database when migrating from one 8.1.5 database to another 8.1.5 database. The procedure migrates OGIS metadata entries from schemas owned by MDSYS.

Format

SDO_MIGRATE.OGIS_METADATA_TO

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](#).

Spatial Operators

This chapter describes the operators used when working with the spatial object data type. The operators are listed in [Table 9-1](#).

Table 9-1 *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

Purpose

Uses the spatial index to identify either the set of spatial objects that may spatially interact with a given object (such as an area-of-interest), or pairs of spatial objects that might spatially interact. Objects spatially interact if they are not disjoint. This operator performs only a primary filter operation.

Syntax

```
SDO_FILTER(geometry1, geometry2, params) ;
```

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
<i>querytype</i>	Specifies valid query types: JOIN or WINDOW. This is a required parameter. WINDOW implies that <i>geometry2</i> should be considered a dynamic (transient) area-of-interest. Use WINDOW when you want to compare a single geometry (<i>geometry2</i>) to all the geometries in a column (<i>geometry1</i>). JOIN implies that <i>geometry2</i> refers to a table column that must have a spatial index built on it. Use JOIN when you want to compare all the geometries of a column to all the geometries of another column.
<i>idxTAB1</i>	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for <i>geometry1</i> .
<i>idxTAB2</i>	Not supported in this release. Specifies the name of the index table, if there are multiple spatial indexes, for <i>geometry2</i> . Only valid for 'querytype = JOIN.'
<i>layer_gtype</i>	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 is '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 *geometry2* comes from a table column that is not spatially indexed, *geometry2* is indexed in memory.
- If *geometry2* comes from a table column that is spatially indexed, *geometry2* will reuse its index. Performance will degrade if *geometry2* is not indexed with the same *sdo_level* parameter as *geometry1*.
- If *geometry2* is a transient `SDO_GEOMETRY` object, it is indexed in memory.

Examples

1.

```
SELECT A.gid
FROM Polygons A, query_polys B
WHERE B.gid = 1
AND SDO_FILTER(A.Geometry, B.Geometry, 'querytype = WINDOW') = 'TRUE';
```
2.

```
SELECT A.gid
FROM Polygons A, query_polys B
WHERE SDO_FILTER(A.Geometry, B.Geometry, 'querytype = JOIN') = 'TRUE';
```
3.

```
Select A.Gid
FROM Polygons A
WHERE SDO_FILTER(A.Geometry, :aGeom, 'querytype=WINDOW') = 'TRUE';
```
4.

```
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';
```

Related Topics

- [SDO_RELATE](#)

SDO_NN

Purpose

Uses the spatial index to identify the nearest neighbors for a geometry.

Syntax

```
SDO_NN(geometry1, geometry2, param) ;
```

Keywords and Parameters

<i>geometry1</i>	Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
<i>geometry2</i>	Specifies either a geometry from a table or a transient instance of a geometry. The nearest neighbor or neighbors to <i>geometry2</i> will be returned from <i>geometry1</i> . (<i>geometry2</i> 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
<i>sdo_num_res</i>	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.

Example

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 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

Purpose

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.

Syntax

```
SDO_RELATE(geometry1, geometry2, params) ;
```

Keywords and Parameters

<i>geometry1</i>	Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
<i>geometry2</i>	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
<i>mask</i>	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)'. See Section 1.8 for an explanation of the 9-intersection relationship pattern.

<i>querytype</i>	<p>Valid query types are: JOIN or WINDOW. This is a required parameter if <i>geometry2</i> is from another table, but it is not a required parameter if <i>geometry2</i> is a literal or a host variable.</p> <p>WINDOW implies that <i>geometry2</i> should be considered a dynamic (transient) area-of-interest. Use WINDOW when you want to compare a single geometry (<i>geometry2</i>) to all the geometries in a column (<i>geometry1</i>).</p> <p>JOIN implies that <i>geometry2</i> refers to a table column that must have a spatial index built on it. Use JOIN when you want to compare all the geometries of a column to all the geometries of another column.</p>
<i>idxtab1</i>	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for <i>geometry1</i> .
<i>idxtab2</i>	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for <i>geometry2</i> . Only valid for 'querytype = JOIN'.
<i>layer_gtype</i>	<p>Specifies special processing for point data.</p> <p>If the columns you are comparing have only point data, set this parameter to 'POINT' for optimal performance. Data type is VARCHAR2. Default is 'NOTPOINT'.</p>

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 the query type 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 *geometry2* comes from a table column that is not spatially indexed, *geometry2* is indexed in memory.
- If *geometry2* comes from a table column that is spatially indexed, *geometry2* will reuse its index. Performance will degrade if *geometry2* is not indexed with the same *sdo_level* parameter as *geometry1*.
- If *geometry2* is a transient SDO_GEOMETRY object, it is indexed in memory.

Unlike with the [SDO_GEOM.RELATE](#) function, DISJOINT and DETERMINE masks are not allowed in the relationship mask. 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.

Examples

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';
```
2.

```
SELECT A.gid
FROM Polygons A, query_polys B
WHERE SDO_RELATE(A.Geometry, B.Geometry,
               'mask=ANYINTERACT querytype = JOIN') = 'TRUE';
```
3.

```
Select A.Gid
FROM Polygons A
WHERE SDO_RELATE(A.Geometry, :aGeom, 'mask=ANYINTERACT querytype=WINDOW')
= 'TRUE';
```
4.

```
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';
```

Related Topics

- [SDO_FILTER](#)
- [SDO_WITHIN_DISTANCE](#)
- [SDO_GEOM.RELATE](#) function

SDO_WITHIN_DISTANCE

Purpose

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).

Syntax

```
SDO_WITHIN_DISTANCE(T.column, aGeom, params) ;
```

Keywords and Parameters

<i>T.column</i>	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 (<i>aGeom</i>). The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
<i>aGeom</i>	Specifies the object to be checked for distance against the geometry objects in <i>T.column</i> . 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
<i>distance</i>	Specifies the Euclidean distance value. This is a required parameter. Data type is NUMBER.
<i>idxtab1</i>	Not supported in this release. Specifies the name of the index if there are multiple spatial index tables for <i>T.column</i> .
<i>querytype</i>	Set 'querytype=FILTER' to perform only a primary filter operation. If <i>querytype</i> is not specified, both primary and secondary filter operations are performed (default). Data type is VARCHAR2.
<i>layer_gtype</i>	Allows special processing for point data. If the objects in <i>T.column</i> have only point data, set this parameter to 'POINT' for optimal performance. Do not set this parameter to 'POINT' if <i>T.column</i> contains any <i>n</i> on-point objects. Data type is VARCHAR2. Default is '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

- ```
1. SELECT A.GID
 FROM POLYGONS A
 WHERE
 SDO_WITHIN_DISTANCE(A.Geometry, :aGeom, 'distance = 10') = 'TRUE' ;
```
- ```
2. 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' ;
```
- ```
3. 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](#)

# 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.

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

- [Chapter 10, "The Relational Schema"](#)
- [Chapter 11, "Loading Spatial Data"](#)
- [Chapter 12, "Querying Spatial Data"](#)
- [Chapter 13, "Administrative Functions and Procedures"](#)
- [Chapter 14, "Tuning Functions and Procedures"](#)
- [Chapter 15, "Geometry Functions and Procedures"](#)
- [Chapter 16, "Window Functions and Procedures"](#)



## The Relational Schema

Prior to 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 Open GIS 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.

### 10.1 Database Structures for the Relational Implementation

The four tables, used to store and index geometry, 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 10-1](#) through [Table 10-4](#) describe the schema of a Spatial layer.

**Table 10-1** <layername>\_SDOLAYER Table

| SDO_ORDCNT | SDO_LEVEL | SDO_NUMTILES | SDO_MAXLEVEL <sup>1</sup> | SDO_COORDSYS <sup>2</sup> |
|------------|-----------|--------------|---------------------------|---------------------------|
| <number>   | <number>  | <number>     | <number>                  | <varchar>                 |

<sup>1</sup> SDO\_MAXLEVEL is an optional column.

<sup>2</sup> SDO\_COORDSYS is an optional column.

**Table 10-2** <layername>\_SDODIM Table or View

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

**Table 10–3** *<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 10–4** *<layername>\_SDOINDEX Table*

| SDO_GID  | SDO_CODE | SDO_MAXCODE <sup>1</sup> | SDO_GROUPCODE <sup>2</sup> | SDO_META |
|----------|----------|--------------------------|----------------------------|----------|
| <number> | <raw>    | <raw>                    | <raw>                      | <raw>    |

<sup>1</sup> SDO\_MAXCODE is not required for the recommended fixed-size tile indexing algorithm.

<sup>2</sup> 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 your 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 **S**Equence 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 **S**Equence 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 *n*th 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.

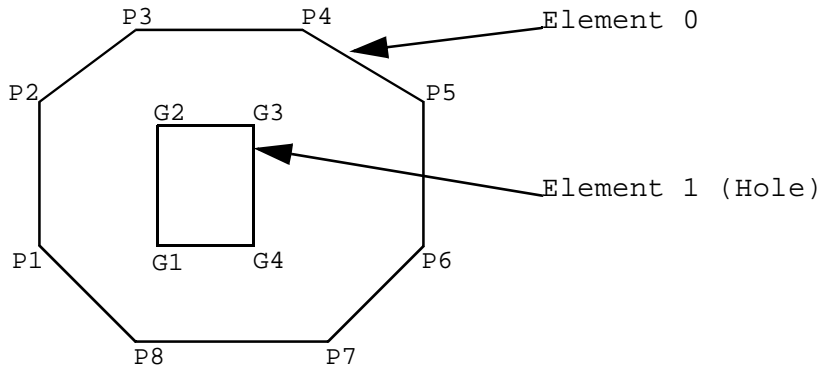
Spatial provides stored procedures that assume the existence of the layer schema as described in this section. While layer tables may contain additional columns, they are required to contain at least the columns described in this section with the same column names and data types.

[Figure 10-1](#) 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 10-1 Complex Polygon**

Geometry 1013:



**<layername>\_SDOLAYER**

| SDO_ORDCNT<br>(number) |
|------------------------|
| 4                      |

**<layername>\_SDODIM**

| SDO_DIMNUM<br>(number) | SDO_LB<br>(number) | SDO_UB<br>(number) | SDO_TOLERANCE<br>(number) | SDO_DIMNAME<br>(varchar) |
|------------------------|--------------------|--------------------|---------------------------|--------------------------|
| 1                      | 0                  | 100                | .05                       | X axis                   |
| 2                      | 0                  | 100                | .05                       | Y axis                   |

**<layername>\_SDOGEOM**

| SDO_GID<br>(number) | SDO_ESEQ<br>(number) | SDO_ETYPE<br>(number) | SDO_SEQ<br>(number) | SDO_X1<br>(number) | SDO_Y1<br>(number) | SDO_X2<br>(number) | SDO_Y2<br>(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)              |

| SDO_GID<br>(number) | SDO_ESEQ<br>(number) | SDO_ETYPE<br>(number) | SDO_SEQ<br>(number) | SDO_X1<br>(number) | SDO_Y1<br>(number) | SDO_X2<br>(number) | SDO_Y2<br>(number) |
|---------------------|----------------------|-----------------------|---------------------|--------------------|--------------------|--------------------|--------------------|
| 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<sup>1</sup> 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.

<sup>1</sup> 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.

---



---

## Loading Spatial Data

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.

### 11.1 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 11-1](#) through [Table 11-4](#) show the format of the tables or views needed to store and index spatial data. Note that these tables show the relational schema.

**Table 11-1** *<layername>\_SDOLAYER Table*

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

**Table 11-2** *<layername>\_SDODIM Table or View*

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

**Table 11–3** *<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 11–4** *<layername>\_SDOINDEX Table*

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

## 11.2 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.

### 11.2.1 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<sup>1</sup>.

[Example 11–1](#) 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 11–1](#):

**Example 11–1 Raw Data Format**

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

<sup>1</sup> See the *Oracle8i Utilities User's Guide* for information on SQL\*Loader.

The coordinates in the geometry rows represent the end points of line segments, which taken together, represent a polygon. [Example 11-2](#) shows the control file for loading the data into the geometry table.

**Example 11-2 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 11-3](#), 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 11-3 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 11-4](#)

**Example 11-4 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
```

## 11.2.2 Transactional Insert Using SQL

Spatial uses standard Oracle8i tables that can be accessed or loaded with standard SQL syntax. [Example 11-5](#) 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 11-5 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 11-5](#), 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 in order 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 11-6](#).

**Example 11-6 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);
```

## 11.3 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 13](#) for a complete description of the SDO\_ADMIN.POPULATE\_INDEX and SDO\_ADMIN.UPDATE\_INDEX procedures.

### 11.3.1 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 11-5](#).

**Table 11-5 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.                                                                                                                                                                            |

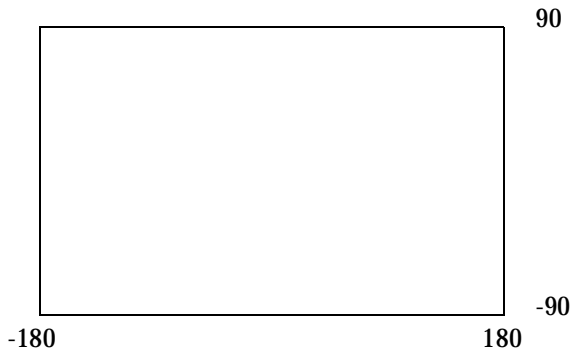
### 11.3.2 Spatial Indexing with Fixed-Size Tiles

Oracle recommends using fixed-size cover tiles for indexing a geometry stored using the relational model.

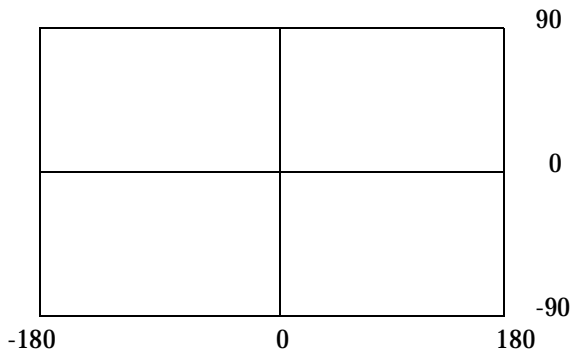
The fixed-size tile algorithm is expressed as a level referring to the number of tessellations performed. To use fixed-size tile indexing, set the SDO\_NUMTILES column in the <layername>\_SDOLAYER table to NULL and the SDO\_LEVEL column to the desired tiling level. The relationship between the tiling 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<sup>1</sup>, as represented in [Figure 11-1](#).

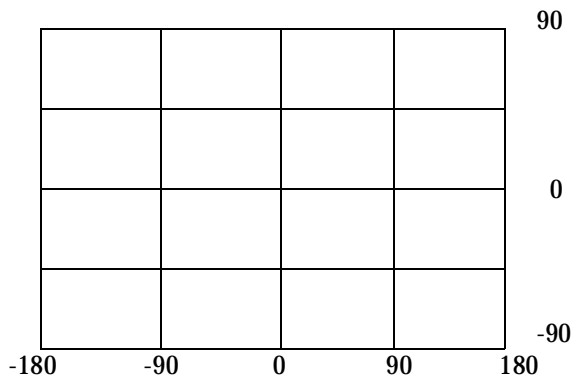
<sup>1</sup> The transference of the domain onto a sphere or Mercator projection is left to GIS (or other) application programmers. Spatial treats the domain as a conventional X by Y rectangle.

**Figure 11-1 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 11-2](#).

**Figure 11-2 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 11-3](#) shows fixed-size tiling at level 2. In this figure, each tile is 90 degrees by 45 degrees.

**Figure 11-3 Fixed-Size Tiling at Level 2**

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

$$\text{length} = (\text{upper\_bound} - \text{lower\_bound}) / 2^{\text{sdo\_level}}$$

The length refers to the length of the tile along the specified dimension. Applying this formula to the tiling shown in [Figure 11-3](#) yields the following sizes:

$$\begin{aligned} \text{length for dimension X} &= (180 - (-180)) / 2^2 \\ &= (360) / 4 \\ &= 90 \\ \text{length for dimension Y} &= (90 - (-90)) / 2^2 \\ &= (180) / 4 \\ &= 45 \end{aligned}$$

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 14](#) 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 is an important part any database design, and 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 14](#) 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.

### 11.3.3 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 12.3.2](#) and [Section 12.3.3](#) 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.

---

## Querying Spatial Data

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.

### 12.1 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 in order 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.

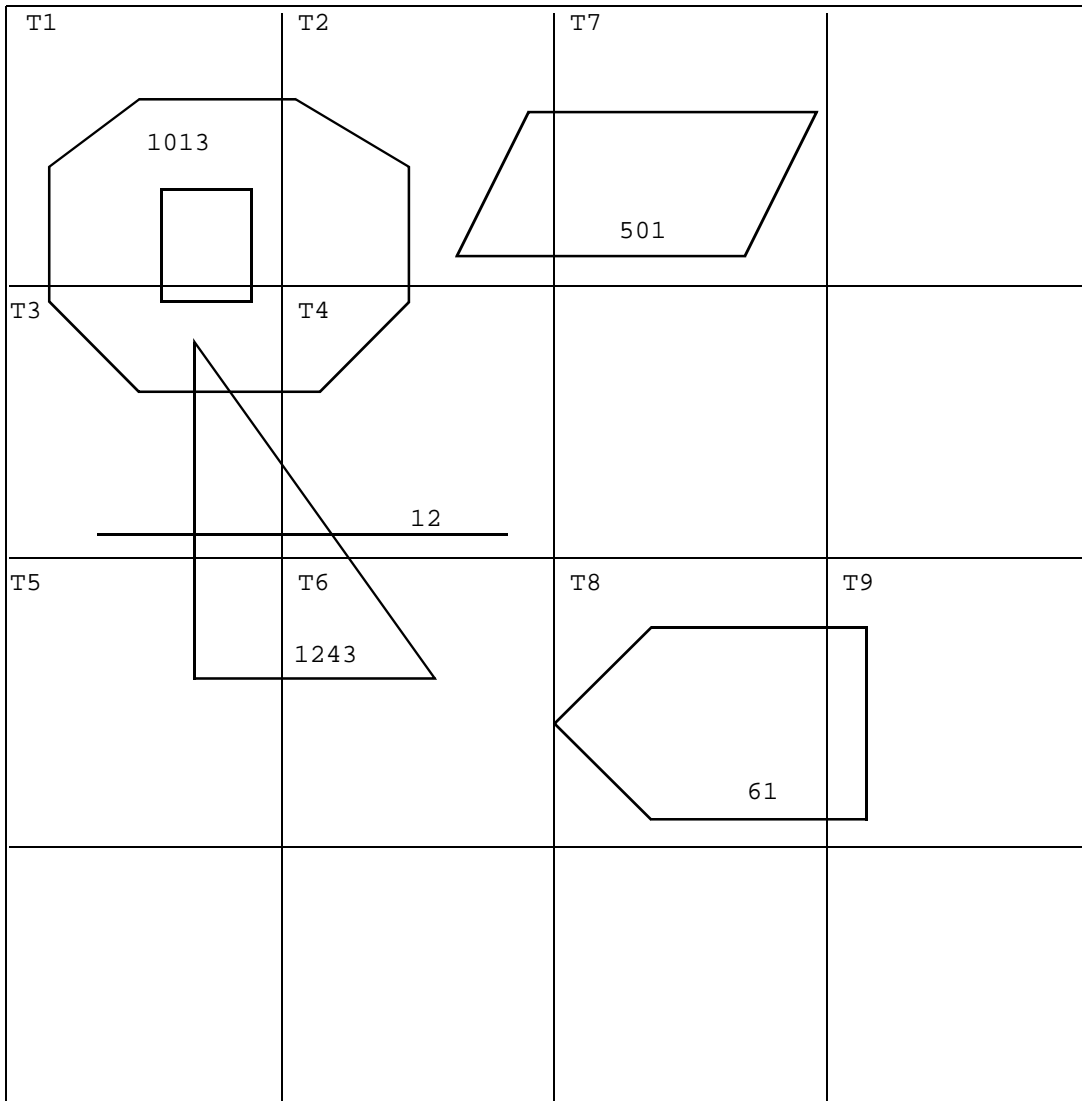
- The **primary filter** permits fast selection of a small number of candidate records to pass along to the secondary filter. The primary filter uses approximations in order to reduce computational complexity and is considered a lower-cost filter.
- The **secondary filter** applies exact computational 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.

### 12.2 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 12-1](#). Each object is labeled with its SDO\_GID. The relevant tiles are labeled with 'Tn'.

**Figure 12-1 Tessellated Layer with Multiple Objects**



The Spatial layer tables would have the following information stored in them for these geometries as shown in [Table 12-1](#), [Table 12-2](#), and [Table 12-3](#).



Table 12-1 &lt;layername&gt;\_SDOLAYER Table

| SDO_ORDCNT<br>(number) | SDO_LEVEL<br>(number) | SDO_NUMTILES<br>(number) |
|------------------------|-----------------------|--------------------------|
| 4                      | 2                     | NULL                     |

Table 12-2 &lt;layername&gt;\_SDOGEOM Table or View

| SDO_GID<br>(number) | SDO_ESEQ<br>(number) | SDO_ETYPE<br>(number) | SDO_SEQ<br>(number) | SDO_X1<br>(number) | SDO_Y1<br>(number) | SDO_X2<br>(number) | SDO_Y2<br>(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)              |

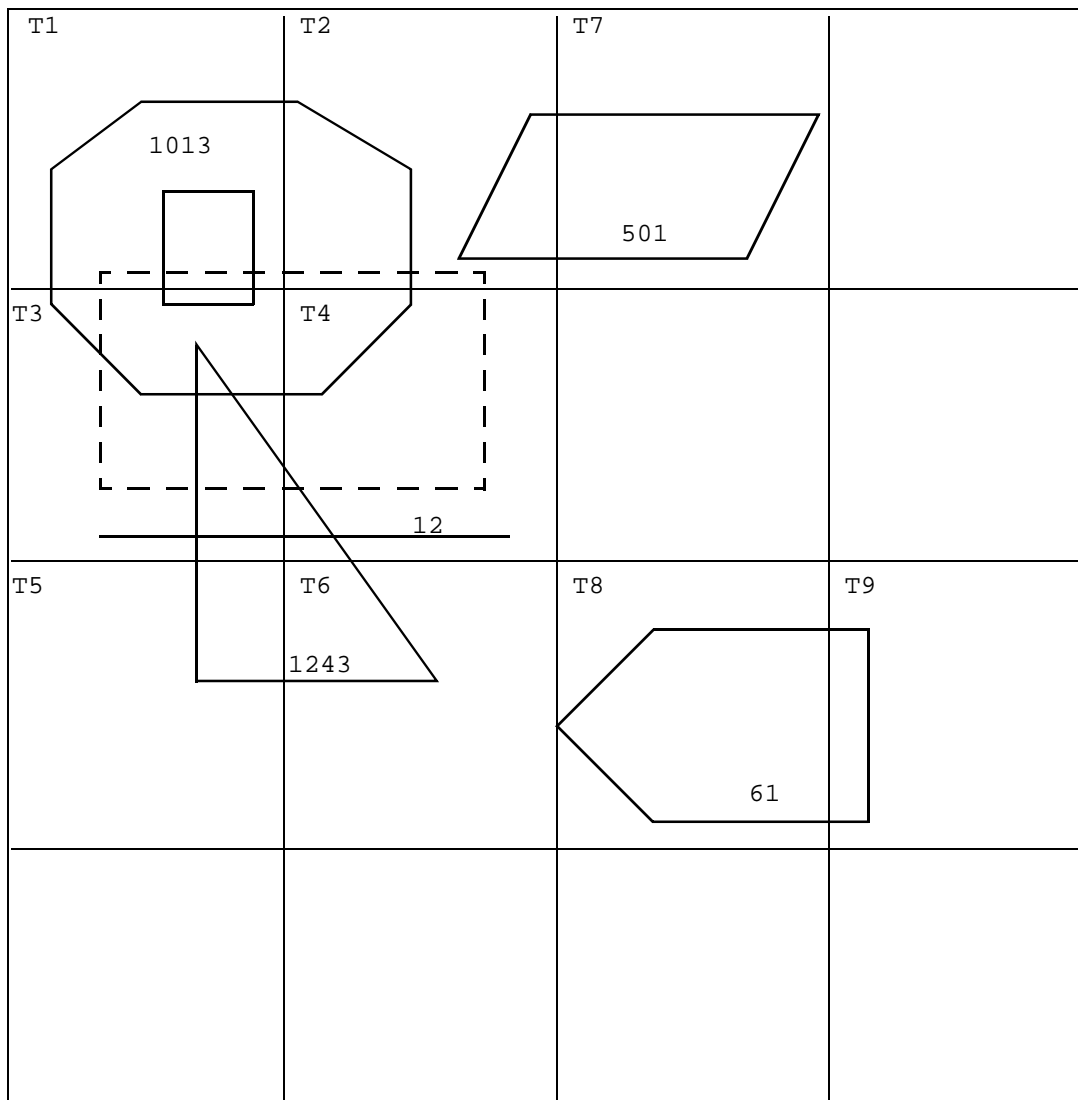
**Table 12–3** <layername>\_SDOINDEX Table

| <b>SDO_GID<br/>(number)</b> | <b>SDO_CODE<br/>(raw)</b> |
|-----------------------------|---------------------------|
| 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                        |

## 12.3 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 12–2](#) 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 prior to using it.

**Figure 12-2 Tessellated Layer with a Query Window**



### 12.3.1 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 16, "Window Functions and Procedures"](#), 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.

## 12.3.2 Primary Filter Query

To resolve the window query shown in [Figure 12–2](#), 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.PUTLINE(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 }
```

### 12.3.3 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 15](#) for details on using this function.

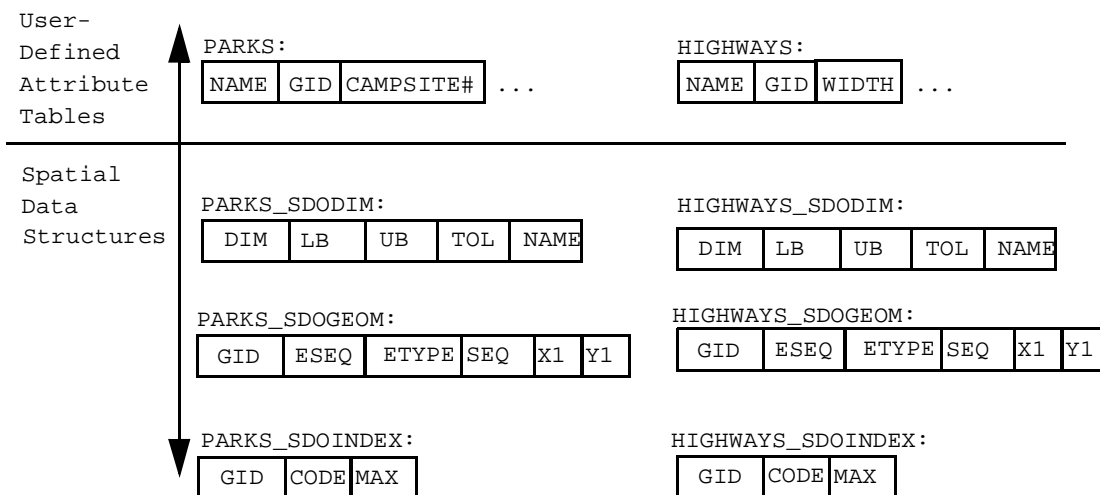
## 12.4 Spatial Join

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 12-3](#) illustrates how the join would be accomplished for this example using the OGIS V1 schema model.

**Figure 12-3 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';

```

The diagram illustrates the flow of data in the SQL query. The Primary Filter is the subquery in parentheses, which filters for parks and highways with the same SDO\_CODE. The Secondary Filter is the WHERE clause, which filters for parks and highways that do not intersect. Arrows indicate the flow of data from the Primary Filter to the 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.





---



---

## Administrative Functions and Procedures

The SDO\_ADMIN procedures create and maintain spatial structures in the database, and 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 13–1 lists the administrative functions and procedures for working with spatially indexed geometry-based data.

**Table 13–1 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        | Generate 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.                                            |
| SDO_ADMIN.UPDATE_INDEX                | Updates the spatial index based on changes to the geometry table.                                          |

---

**Table 13–1 Administrative Procedures for Spatially Indexed Data (Cont.)**

| <b>Procedure or Function</b> | <b>Description</b>                                                             |
|------------------------------|--------------------------------------------------------------------------------|
| 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

### Purpose

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.

### Format

SDO\_ADMIN.POPULATE\_INDEX (*layername*)

### 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 prior to 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.<br><br>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 13–1** tessellates all the geometric objects in the LAYER1\_SDOGEOM table and adds the generated tiles to the LAYER1\_SDOINDEX table.

**Example 13–1** *Populate an Index*

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

## Related Topics

- SDO\_ADMIN.UPDATE\_INDEX procedure

---

## SDO\_ADMIN.POPULATE\_INDEX\_FIXED

### Purpose

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 10.1](#).

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.

### Format

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

### Keywords and Parameters

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i>     | Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables.<br>Data type is VARCHAR2.                                                                                                                                                                                                                                                                                                                                              |
| <i>tile_size</i>     | Specifies the number of tessellations required to achieve the desired tile size (see the Usage Notes).<br>Data type is INTEGER.                                                                                                                                                                                                                                                                                                                                                                         |
| <i>synch_flag</i>    | 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 simply added to the spatial index table.<br>Default value is TRUE.<br>Data type is BOOLEAN. |
| <i>sdo_tile_flag</i> | For internal use only. Not supported in this release.<br>Default value is 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. Default value is TRUE. Data type is BOOLEAN.

## 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, prior to 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 11.3.2](#).
- 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 13-2** 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 13–2 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 procedure
- SDO\_TUNE.ESTIMATE\_TILING\_LEVEL function

## SDO\_ADMIN.POPULATE\_INDEX\_FIXED\_POINTS

### Purpose

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.

### Format

SDO\_ADMIN.POPULATE\_INDEX\_FIXED\_POINTS (*layername*, *sdo\_tile\_flag*, *commit\_count*)

### Keywords and Parameters

|                      |                                                                                                                             |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i>     | Specifies the name of the data set layer.<br>Data type is VARCHAR2.                                                         |
| <i>sdo_tile_flag</i> | Specifies whether or not to generate the SDO_TILE column.<br>Default value is FALSE.<br>Data type is BOOLEAN.               |
| <i>commit_count</i>  | Specifies how many points to index before updating and committing the data.<br>Default value is 50.<br>Data type is NUMBER. |

### Usage Notes

Consider the following when using this procedure:

- The <layername>\_SDOLAYER, <layername>\_SDOGEOM, and <layername>\_SDODIM tables must be populated prior to calling this procedure.
- The <layername>\_SDOINDEX table must be created prior to 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 13-3** 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 13-3** *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 procedure

## SDO\_ADMIN.SDO\_CODE\_SIZE

---

### Purpose

Determines the size that the SDO\_MAXCODE column should be in the <layername>\_SDOINDEX table.

### Format

SDO\_ADMIN.SDO\_CODE\_SIZE (*layername*)

### Keywords and Parameters

|                  |                                                                     |
|------------------|---------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the data set layer.<br>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

### Purpose

Returns the current installed version of Spatial.

### Format

SDO\_ADMIN.SDO\_VERSION

### 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 in when migrating data between systems, or when upgrading. See [Appendix B](#) for more information about migration.

### Related Topics

None.

---

## SDO\_ADMIN.UPDATE\_INDEX

### Purpose

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.

### Format

SDO\_ADMIN.UPDATE\_INDEX (*layername*, *GID*)

### Keywords and Parameters

|                  |                                                                                                                                         |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the data set layer. The layer name is used to construct the name of the geometry table.<br>Data type is VARCHAR2. |
| <i>GID</i>       | Specifies the geometric object identifier.<br>Data type is NUMBER.                                                                      |

### Usage Notes

Consider the following when using this procedure:

- The <layername>\_SDOINDEX table must exist prior to 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 and therefore the transaction can be rolled back.

[Example 13–4](#) tessellates the polygon for geometry 25 and adds the generated tiles to the LAYER1\_SDOINDEX table.

**Example 13–4 Update an Index**

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

## Related Topics

- SDO\_ADMIN.POPULATE\_INDEX procedure

## SDO\_ADMIN.UPDATE\_INDEX\_FIXED

---

### Purpose

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 10.1](#).

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.

### Format

SDO\_ADMIN.UPDATE\_INDEX\_FIXED (*layername*, *GID*, *tile\_size*, [*replace\_flag*,] [*sdo\_tile\_flag*] [*sdo\_maxcode\_flag*])

### Keywords and Parameters

|                      |                                                                                                                                                                                                                                                                                          |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i>     | Specifies the name of the data set layer. The layer name is used to construct the name of the geometry table.<br>Data type is VARCHAR2.                                                                                                                                                  |
| <i>GID</i>           | Specifies the geometric object identifier.<br>Data type is NUMBER.                                                                                                                                                                                                                       |
| <i>tile_size</i>     | 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.<br>Data type is INTEGER.                                                                                 |
| <i>replace_flag</i>  | Specifies whether or not to delete tiles for the GID before adding new ones. If TRUE, tiles are deleted prior to inserting new entries into the spatial index table. If FALSE, new tiles are simply added to the spatial index table.<br>Default value is TRUE.<br>Data type is BOOLEAN. |
| <i>sdo_tile_flag</i> | For internal use only. Not supported in this release.<br>Default value is FALSE.<br>Data type is BOOLEAN.                                                                                                                                                                                |

*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.  
Default value is TRUE.  
Data type is BOOLEAN.

## 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 and therefore this transaction can be rolled back.

**Example 13–5** tessellates the polygon for geometry 25 and adds the generated tiles to the LAYER1\_SDOINDEX table.

**Example 13–5 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 procedure
- SDO\_TUNE.ESTIMATE\_TILING\_LEVEL function

## SDO\_ADMIN.VERIFY\_LAYER

### Purpose

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

### Format

SDO\_ADMIN.VERIFY\_LAYER (*layername*,*[maxtiles]*)

### Keywords and Parameters

- |                  |                                                                                                                                                         |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i> | 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. |
| <i>maxtiles</i>  | 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 13-6](#) verifies the LAYER1 data set layer.

#### **Example 13-6 Verify a Layer**

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

### Related Topics

None.



---

---

## Tuning Functions and Procedures

This chapter contains descriptions of the tuning functions and procedures shown in Table 14–1. This chapter refers to the relational Spatial model only.

**Table 14–1** *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

### Purpose

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

### Format

SDO\_TUNE.AVERAGE\_MBR (*layername*, *width*, *height*)

### Keywords and Parameters

|                  |                                                                                |
|------------------|--------------------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the data set layer to examine.<br>Data type is VARCHAR2. |
| <i>width</i>     | Returns the width of the average MBR.<br>Data type is OUT NUMBER.              |
| <i>height</i>    | Returns the height of the average MBR.<br>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

### Purpose

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

### Format

SDO\_TUNE.ESTIMATE\_INDEX\_PERFORMANCE (*layername*, *sample\_ratio*, *tiling\_level*, *num\_tiles*, *window\_layer*, *window\_gid*, *tiling\_time*, *filter\_time*, *query\_time*)

### Keywords and Parameters

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

### Returns

This function returns a number between 0.0 and 1.0 representing estimated spatial index selectivity. Data type is NUMBER.

This function 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

### Purpose

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

### Format

SDO\_TUNE.ESTIMATE\_TILING\_LEVEL (*layername*, *maxtiles*, *type\_of\_estimate*)

### Keywords and Parameters

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i>        | Specifies the name of the data set layer to examine.<br>Data type is VARCHAR2.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <i>maxtiles</i>         | Specifies the maximum number of tiles that can be used to index the rectangle defined by the <i>type_of_estimate</i> parameter.<br>Data type is INTEGER.                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>type_of_estimate</i> | Indicates by keyword one of three different models. Specify the type of estimate with one of the following keywords: <ul style="list-style-type: none"><li>• LAYER_EXTENT -- Use the rectangle defined by your coordinate system.</li><li>• ALL_GID_EXTENT -- Use the minimum bounding rectangle that encompasses all the geometric objects within the layer. Recommended for most applications.</li><li>• 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.</li></ul> |

### 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 prior to 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 14-1](#) 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 14-1 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 14-2](#), 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 14-2 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 14–3](#), eight tiles on average are used to cover any block group in San Francisco.

**Example 14–3 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

### Purpose

Returns the estimated time to tessellate a layer.

### Format

SDO\_TUNE.ESTIMATE\_TILING\_TIME (*layername*, *sample\_ratio*, *tiling\_level*, *num\_tiles*)

### Keywords and Parameters

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

### Purpose

Determines the extent of all geometries in a layer.

### Format

SDO\_TUNE.EXTENT\_OF (*layername*, *min\_X*, *max\_X*, *min\_Y*, *max\_Y*)

### Keywords and Parameters

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

### Returns

This function returns the coordinates of the minimum bounding rectangle for all geometric data in a layer. The data type is NUMBER for the four return values.

### Usage Notes

None.

### Related Topics

- SDO\_TUNE.ESTIMATE\_TILING\_LEVEL function

## SDO\_TUNE.HISTOGRAM\_ANALYSIS

### Purpose

Generates statistical histograms based on a layer.

### Format

SDO\_TUNE.HISTOGRAM\_ANALYSIS (*layername*, *result\_table*, *type\_of\_histogram*,  
*max\_value*, *intervals*)

### Keywords and Parameters

|                          |                                                                                                                                                                                                                                    |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i>         | Specifies the name of the spatial data set layer to examine.<br>Data type is VARCHAR2.                                                                                                                                             |
| <i>result_table</i>      | Specifies the name of the result table where the histogram will be stored.<br>Data type is VARCHAR2.                                                                                                                               |
| <i>type_of_histogram</i> | Specifies one of the following types of histograms: <ul style="list-style-type: none"><li>• TILES_VS_LEVEL (default)</li><li>• GEOMS_VS_AREA</li><li>• GEOMS_VS_TILES</li><li>• GEOMS_VS_VERTICES</li></ul> Data type is VARCHAR2. |
| <i>max_value</i>         | Specifies the upper limit of the histogram.<br>Data type is NUMBER.                                                                                                                                                                |
| <i>intervals</i>         | Specifies the number of intervals between 0 and <i>max_value</i> .<br>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 prior to 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

---

### Purpose

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

### Format

SDO\_TUNE.MIX\_INFO (*layername*)

### Keywords and Parameters

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the spatial data set layer to examine.<br>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.

---

---

## Geometry Functions and Procedures

This chapter contains descriptions of the geometric functions and procedures shown in Table 15–1. This chapter refers to the relational Spatial model only.

**Table 15–1 Geometric Functions and Procedures**

| Function/Procedure                         | Description                                        |
|--------------------------------------------|----------------------------------------------------|
| <a href="#">SDO_GEOM.RELATE</a>            | Determines how two objects interact.               |
| <a href="#">SDO_GEOM.VALIDATE_GEOMETRY</a> | Determines if a geometry is valid.                 |
| <a href="#">SDO_GEOM.VALIDATE_LAYER</a>    | Determines if all geometries in a layer are valid. |

## SDO\_GEOM.RELATE

### Purpose

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

### Syntax

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*]])

### Keywords and Parameters

|                                            |                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername1</i> ,<br><i>layername2</i>   | Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables.<br>Data type is VARCHAR2.                                                                                                                                                                        |
| <i>SDO_GID1</i> ,<br><i>SDO_GID2</i>       | Specifies the geometry object identifier.<br>Data type is NUMBER.                                                                                                                                                                                                                                                                 |
| <i>mask</i>                                | Specifies a list of relationships to check. See the list of keywords in the Usage Notes.                                                                                                                                                                                                                                          |
| <i>X_tolerance</i> ,<br><i>Y_tolerance</i> | 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.<br>Data type is NUMBER. |
| <i>SDO_ETYPE</i>                           | Specifies the type of geometry element.<br>Data type is INTEGER, corresponding to the following constants:<br><ol style="list-style-type: none"><li>1 SDO_GEOM.POINT_TYPE</li><li>2 SDO_GEOM.LINestring_TYPE</li><li>3 SDO_GEOM.POLYGON_TYPE</li></ol>                                                                            |

---

|                                            |                                                                                                   |
|--------------------------------------------|---------------------------------------------------------------------------------------------------|
| <i>num_ordinates</i>                       | Specifies the number of ordinates for this element. Data type is NUMBER.                          |
| <i>X_ordinateN</i> ,<br><i>Y_ordinateN</i> | Specifies the X and Y values of a vertex (coordinate pair) in a geometry.<br>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 of 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

### Purpose

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

### Syntax

SDO\_GEOM.VALIDATE\_GEOMETRY (*layername*,*SDO\_GID*)

### Keywords and Parameters

|                  |                                                                                                                                                            |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables.<br>Data type is VARCHAR2. |
| <i>SDO_GID</i>   | Specifies the geometric object identifier.<br>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

### Purpose

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

### Syntax

SDO\_GEOM.VALIDATE\_LAYER (*layername*, *result\_table*)

### Keywords and Parameters

|                     |                                                                       |
|---------------------|-----------------------------------------------------------------------|
| <i>layername</i>    | Specifies the name of the layer to examine.<br>Data type is VARCHAR2. |
| <i>result_table</i> | Specifies the name of the result table.<br>Data type is VARCHAR2.     |

### Returns

This function fills the result table with validation results.

### Usage Notes

Create an empty result table prior to 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.

---



---

## Window Functions and Procedures

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 16–1](#). This chapter refers to the relational Spatial model only.

**Table 16–1 Window Functions and Procedures**

| Function/Procedures            | Description                                           |
|--------------------------------|-------------------------------------------------------|
| SDO_WINDOW.BUILD_WINDOW        | Builds a query window geometric object.               |
| 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. |
| SDO_WINDOW.CREATE_WINDOW_LAYER | Creates the tables needed for a query window layer.   |

## SDO\_WINDOW.BUILD\_WINDOW

---

### Purpose

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.

### Syntax

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

### Keywords and Parameters

|                                            |                                                                                                                                                                                                    |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>comp_name</i>                           | 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.                                   |
| <i>layername</i>                           | Specifies the name of the window layer into which the coordinates will be inserted. Data type is VARCHAR2.                                                                                         |
| <i>SDO_ETYPE</i>                           | Specifies the type of geometry objects. Data type is INTEGER, corresponding to the following constants:<br>1 or SDO_GEOM.POINT_TYPE<br>2 or SDO_GEOM.LINestring_TYPE<br>3 or SDO_GEOM.POLYGON_TYPE |
| <i>SDO_NUMTILES</i>                        | Value must be NULL for Spatial release 8.0.4 and later. Data type is NUMBER.                                                                                                                       |
| <i>X ordinateN</i> ,<br><i>Y ordinateN</i> | 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. The 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

---

### Purpose

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

### Syntax

SDO\_WINDOW.BUILD\_WINDOW\_FIXED (*comp\_name*, *layername*, *SDO\_ETYPE*, *SDO\_TILESIZE*, *X1*, *Y1*, [...*Xn*, *Yn*])

### Keywords and Parameters

|                                            |                                                                                                                                                                                                    |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>comp_name</i>                           | 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.                                   |
| <i>layername</i>                           | Specifies the name of the window layer into which the coordinates will be inserted. Data type is VARCHAR2.                                                                                         |
| <i>SDO_ETYPE</i>                           | Specifies the type of geometry element. Data type is INTEGER, corresponding to the following constants:<br>1 or SDO_GEOM.POINT_TYPE<br>2 or SDO_GEOM.LINestring_TYPE<br>3 or SDO_GEOM.POLYGON_TYPE |
| <i>SDO_TILESIZE</i>                        | Specifies the number of tessellations required to achieve the desired fixed-size tiles. Data type is NUMBER.                                                                                       |
| <i>X ordinateN</i> ,<br><i>Y ordinateN</i> | 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. The 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

### Purpose

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

### Syntax

```
SDO_WINDOW.CLEAN_WINDOW (layername);
```

### Keywords and Parameters

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| <i>layername</i> | Specifies the name of the window layer that must be removed.<br>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

### Purpose

Removes the query window from the layer tables.

### Syntax

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

### Keywords and Parameters

|                  |                                                                                                                                               |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <i>gid</i>       | Specifies the geometric object identifier of the query window.<br>Data type is NUMBER.                                                        |
| <i>layername</i> | Specifies the name of the window layer associated with the query window.<br>Data type is VARCHAR2.                                            |
| <i>do_commit</i> | Specifies whether a commit operation is performed (TRUE) or is not performed (FALSE, the default) after the cleanup.<br>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

### Purpose

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

### Syntax

```
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)
```

### Keywords and Parameters

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



---

---

# Tuning Tips and Sample SQL Scripts

This appendix provides supplemental information to aid in the setup, maintenance, and tuning of 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 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* layer and the *Highway* layer 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.

Oracle recommends running the `SDO_TUNE.ESTIMATE_TILING_LEVEL` function 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 also 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)

When using the object-relational model of Spatial, Oracle recommends building indexes using hybrid indexing. 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 `SDO_LEVEL` is less than the value specified by `SDO_NUMTILES`, 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.

Hybrid indexing is recommended for point data as well as for other geometry types. However, 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` 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 in order 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,
 hhlengh(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 = 'AAA59AAFAADzAZAAA');

```

**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 in order 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,
 hhlenght(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,
 hhlenght (sdo_code || sdo_meta), 'MIN') min_x,
 hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
 hhlenght (sdo_code || sdo_meta), 'MAX') max_x,
 hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
 hhlenght (sdo_code || sdo_meta), 'MIN') min_y,
 hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
 hhlenght (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,
 hhlenght (sdo_code || sdo_meta), 'MIN') min_x,
 hhcellbndry (sdo_code || sdo_meta, 1, -180.000000000, 180.000000000,
 hhlenght (sdo_code || sdo_meta), 'MAX') max_x,
 hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
 hhlenght (sdo_code || sdo_meta), 'MIN') min_y,
 hhcellbndry (sdo_code || sdo_meta, 2, -90.000000000, 90.000000000,
 hhlenght (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 Oracle database server processes SQL statements that require sorting, such as statements containing an ORDER BY or DISTINCT clause, the Oracle 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_sdgeom,
 (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;
 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 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 Oracle8i 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 Oracle8i 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_sdoggeom AS SELECT * FROM a_sdoggeom
 WHERE sdo_gid BETWEEN 1 and 100;
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 INDEX 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 `SDO_ADMIN.POPULATE_INDEX` commands 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 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 Oracle8i *interMedia* Locator

Oracle8i *interMedia* Locator is a related product that supports online internet-based geocoding facilities for locator 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 *interMedia* 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 *interMedia Locator*, see *Oracle8i interMedia Locator User's Guide and Reference*.

### A.3.1.2 Compatibility with Spatial Objects

*interMedia 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 *interMedia 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 *interMedia 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 Locator Code

Sample scripts are available in the following directory after you install Oracle8i *interMedia 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.



---

---

## 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 an 8.0.4, 8.0.5, or 8.1.6 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 Oracle database server. Therefore, you must upgrade both the 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 Oracle 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 has been 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 the product uses extensible indexing and object types in 8.1, and therefore if an 8.1 database instance is downgraded to 8.0.5, then the spatial objects must be deleted and re-created. In this case, the data must be exported and imported into 8.0.5. This, and other requirements, result in

application incompatibility. An 8.1 Spatial application will likely use the new spatial operators and therefore will not work with an 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 an 8.0.5 server is upgraded to 8.1, Spatial also has to be upgraded. The reason has to do with using dynamic SQL in PL/SQL, and Invoker's Rights in 8.1. Similarly, if an 8.1 server is downgraded, Spatial must be downgraded too. Lastly, if an 8.1 server is running in 8.0 compatibility mode, then Spatial will experience various failures unless it is reconfigured for 8.0.5. You can reconfigure the product by running the downgrade script: `c813d805.sql`.

In summary:

- The Spatial release and the Oracle database server release must match.
- Upgrade and downgrade scripts must be run when upgrading or downgrading between 8.0.5 and 8.1.
- Spatial will work in 8.0 compatibility mode for an 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 tablespace.

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, then 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. This Oracle Spatial release differs from pre-Oracle8i ones in that it:

- Uses object types (a varray-based type called SDO\_GEOMETRY to store ordinates)
- Supports new spatial data types, namely arcs and circles
- Has new spatial operators (SDO\_WITHIN\_DISTANCE) and functions, SDO\_POLY\_UNION, SDO\_BUFFER, SDO\_POLY\_INTERSECT, and SDO\_POLY\_XOR
- Utilizes dynamic SQL in PL/SQL
- Allows invoker's rights
- Tessellates a geometry as a whole rather than an element at a time

All interfaces preceding Oracle8i are maintained, but the package bodies have been changed to use the preceding features. Thus for Oracle8i, 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 7.3.4/8.0.4 to 8.1.5 upgrade requirements are the same. Upgrade both Oracle8i 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 server and Spatial requires no special steps specific to the Spatial implementation. However, this situation is different for Oracle8i. In Oracle8i, Spatial uses objects and extensible indexing. Therefore, it creates database objects specific to Oracle8i that are not compatible with previous releases of the database server. When you downgrade the server and Spatial from Oracle8i to 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 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 Oracle8i because Spatial now allows two storage mechanisms. If you want the features specific to Oracle8i, such as extensible indexing and spatial operators, then you must migrate your spatial data from the 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 an 8.0.5 spatial schema for comparable data. The demo also addresses issues related to the changes in the way metadata is stored in Oracle8i compared to previous releases. The complexity arises from the following:

- From 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 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 Oracle8i, and you cannot use data from a 7.3.4 or later spatial layer in 7.3.3 if it contains instances of type UNSUPPORTED\_GEOMETRY (etype=0).

## B.5 Migrating from Spatial Release 8.1.5 to 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 drop all the spatial indexes built in 8.1.5 database and re-create them in an 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 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 drops 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](#).)
5. Manually re-create all the spatial indexes that are required.

## B.5.1 Data Migration

Spatial release 8.1.6 introduces 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.5.2 Compatibility Between Releases 8.1.5 and 8.1.6

Spatial release 8.1.6 changes 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.



---

---

## 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 locational information as instances of predefined object types. This interface is part of the geocoding framework in Oracle Spatial and *interMedia* 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 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 processing and gathering functions. Results of analyses or queries can be presented as maps, in addition to tabular formats, using third-party software integrated with Oracle Spatial.

This appendix 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 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 *interMedia* Locator Implementation: Benefits and Limitations

Oracle *interMedia* Locator contains a set of interface (API) functions that allows the integration of Oracle Spatial to third-party geocoding products and Web-based geocoding service. 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 what fields to extract and where to store them. However, to use the full range of features of Oracle Spatial or *interMedia* 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, it is preferable to 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 Oracle8i 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 Oracle8i object types.

Therefore, any client geocoder component written in Java can be embedded in the Oracle8i database as a JSP. This JSP interface can perform either one-record-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.

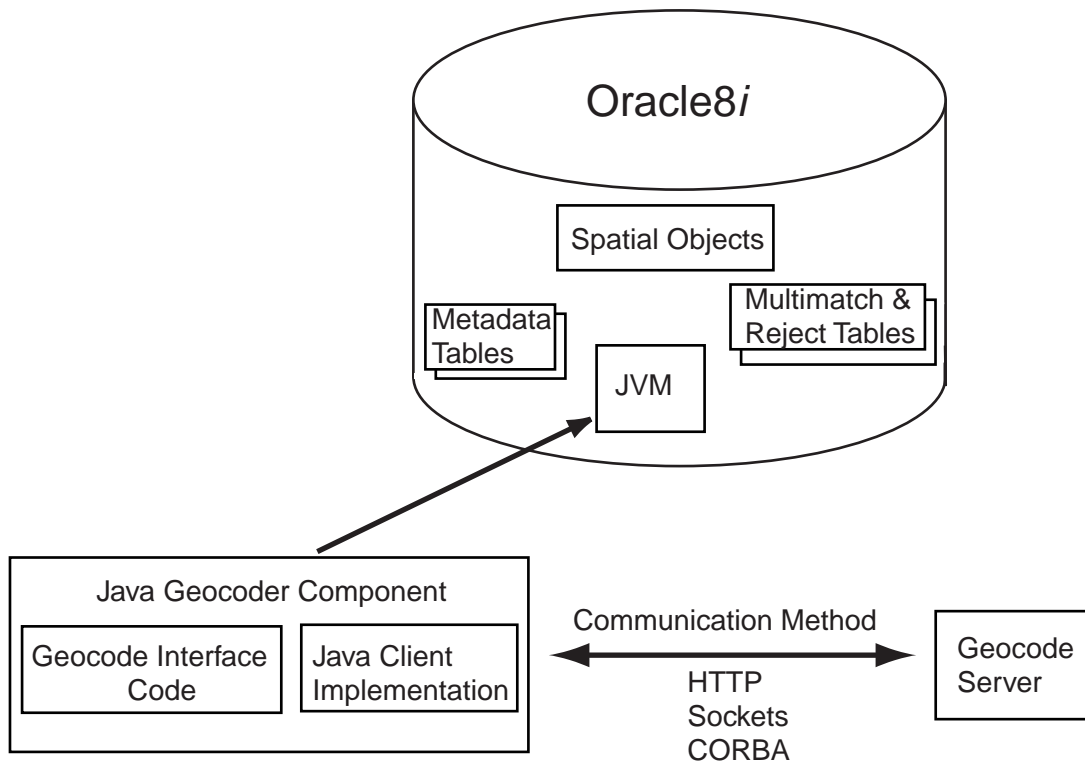
The stored procedures have an interface, `oracle.spatial.geocoder`, that must be implemented by each vendor whose geocoder is integrated with Oracle Spatial and *interMedia* Locator. The procedures also require certain object types to be defined

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, the geocoding metadata must exist in each user's schema. Each user of the geocoder service must have tables that implement the metadata schema.

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 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** allows the user to 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, geocoder 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 names and values arrays allow customization for unique geocoder processing options. They are not intended for name/password, because a local geocoder 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 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 geocoding 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 geocoding 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 of the *out\_table\_cols* fields can be null. *out\_table\_cols* and *geocode\_result* contain similar information, that is, the standardized 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 varchar2(16),
 lotnum varchar2(16)
);
```

- *in\_primary\_key* and *out\_foreign\_key*: 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, since 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 whether to 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. *reject\_table* will be created with a primary key column type in the input table, a match code column, and an optional error message column. *multi\_match\_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 a 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 `geocode10` 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 parameter, and get back an array of matches. The Java interface takes a metadata structure (see the `GEOCODE_SERVER_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. In 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 in order to integrate their products with Oracle Spatial and *interMedia* 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 msg)
 {
 super(msg);
 }
}

```

Further details, including some of the actual implementation, will be provided to developers by Oracle.

## 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 geocoding 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 their server and can provide the required services.

To load a client into the database, invoke the Oracle8i `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. Appropriate EXECUTE privileges should then be granted to PUBLIC or some set of users of the service.



---

---

# 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.

## **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.

## **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.

**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.

**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 counter-clockwise).

**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.

**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.

**georeferenced data**

*See* spatiotemporal data.

**GIS**

*See* geographical information system.

**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 comprising 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.

**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.

**multidimensional data**

*See* hyperspatial data.

**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.

**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.

**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. Oracle8i 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.

**SLF**

See Spatial Load Format.

**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 Oracle8i 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 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.

**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**

To describe a geometric relationship where two objects share a common point on their boundaries, but their interiors do not intersect.



---

---

# Index

## Numerics

---

### 8.1.6

- migrating to, 8-2
- new features, xviii

## A

---

- administrative procedures, 13-1
- ALL\_SDO\_GEOM\_METADATA view, 2-17
- ALL\_SDO\_INDEX\_METADATA view, 2-19
- ALTER INDEX, 5-2
  - REBUILD, 5-5
  - RENAME TO, 5-8
- ANYINTERACT, 7-5, 15-3
- arcs, A-13
- area, 7-7
- average minimum bounding rectangle, 6-2
- AVERAGE\_MBR, 6-2, 14-2

## B

---

- batch geocoding, C-4
- bounding rectangle
  - minimum, 6-14
- buffer area, 7-9
- BUILD\_WINDOW, 16-2
- BUILD\_WINDOW\_FIXED, 16-4
- bulk loading, 3-1, 11-2

## C

---

- center of gravity (centroid), 7-11
- centroid, 7-11

- circle, 2-11, A-13
- CLEAN\_WINDOW, 16-6
- CLEANUP\_GID, 16-7
- compound element, 2-9
- consistency check, 7-41, 15-5
- CONTAINS, 7-5, 15-3
- control file, 11-2
- convex hull, 7-13
- Coordinate Systems (separately released feature), xix
- COVEREDBY, 7-5, 15-3
- COVERS, 7-5, 15-4
- cr\_spatial\_index.sql, A-16
- CREATE INDEX, 5-9
- CREATE\_WINDOW\_LAYER, 16-8
- creating layer tables, A-17
- crlayer.sql, A-17
- customized geometry types, A-13

## D

---

- data model, 1-6, A-1
- DBA\_SDO\_GEOM\_METADATA view, 2-17
- DBA\_SDO\_INDEX\_METADATA view, 2-19
- difference, 7-27
- dimension (in SDO\_GTYPE), 2-7
- disjoint, 7-5, 15-4
- displaying geometries, A-18, A-20
- distance, 7-45
- DROP INDEX, 5-13
- dynamic query window, 4-3, 12-4

## E

---

ELEM\_INFO (SDO\_ELEM\_INFO), 2-8  
element, 1-6  
EQUAL, 7-5, 15-4  
error messages, xx  
ESTIMATE\_INDEX\_PERFORMANCE, 6-4, 14-3  
ESTIMATE\_TILING\_LEVEL, 6-7, 14-5  
ESTIMATE\_TILING\_TIME, 6-9, 14-8  
ESTIMATE\_TOTAL\_NUMTILES, 6-11  
ETYPE (SDO\_ETYPE), 2-8  
example  
    creating, indexing, and querying spatial  
        data, 2-1  
extent, 14-9  
EXTENT\_OF, 6-14, 14-9  
exterior polygon rings, 2-7, 2-9, 2-13, 2-14

## F

---

filter, 12-6  
fixed indexing, 1-12  
fixed-size tiles, 3-10, 11-7, 13-5, 13-14  
FROM\_815\_TO\_81x, 8-2

## G

---

generic geocoding interface, C-1  
GEOCODE\_SCHEMA\_PROPERTY\_TYPE, C-6  
GEOCODE\_SERVER\_PROPERTY\_TYPE, C-5  
GEOCODE\_TABLE\_COLUMN\_TYPE, C-6  
GEOCODE\_TASK\_METADATA, C-4  
GEOCODER\_HTTP package, C-10  
geocoding  
    generic interface, C-1  
geocoding support  
    *interMedia* Locator, A-17, C-2  
GeoImage feature, xix  
geometric primitive, 1-4  
geometry types, 1-4  
    custom, A-13  
    object-relational, 2-6  
    relational, 10-3  
GIS, 1-3  
GTYPE (SDO\_GTYPE), 2-6

## H

---

HISTOGRAM\_ANALYSIS, 6-16, 14-10  
hybrid indexing, 1-16

## I

---

index, 13-3, 13-5, 13-12, 13-14  
index creation, 3-9, 11-6  
    cross-schema, 3-14  
    in parallel, A-14  
inserting spatial data, 11-4  
INSIDE, 7-6, 15-4  
INTEPRETATION (SDO\_INTERPRETATION), 2-9  
interaction, 7-5, 15-3  
interior polygon rings, 2-7, 2-9, 2-13, 2-14  
*interMedia* Locator, A-17, C-2  
intersection, 7-29

## J

---

Java Virtual Machine, C-2  
JVM (Java Virtual Machine), C-2

## L

---

layer, 1-7, A-17  
    validating, 7-43  
length (SDO\_LENGTH), 7-23  
line, 2-11  
    length, 7-23  
line data, 1-7  
Linear Referencing System capability, xix  
loading process, 3-1, 11-2  
    in parallel, A-14  
location, 1-3  
Locator (*interMedia*), A-17, C-2  
long transactions (Version Management), xix  
LRS (Linear Referencing System), xix

## M

---

MDSYS schema, 1-1  
migration  
    OGIS, 8-8, 8-9  
    to Oracle 7.3.4, 8-3



- to Oracle 8.1.5, 8-5
- to Oracle 8.1.6, 8-2
- minimum bounding rectangle, 6-2, 6-14, 14-2, 14-9
  - average, 6-2
- MIX\_INFO, 6-18, 14-12
- multimatch table, C-9
- multiple matches, C-9
- multi-polygon, 2-14

## N

---

- nearest neighbor (SDO\_NN), 9-4
- new features for release 8.1.6, xviii

## O

---

- object-relational model
  - schema, 2-1
- OGIS\_METADATA\_FROM, 8-8
- OGIS\_METADATA\_TO, 8-9
- operators
  - cross-schema invocation, 4-9
  - SDO\_FILTER, 9-2
  - SDO\_NN, 9-4
  - SDO\_RELATE, 9-6
  - SDO\_WITHIN\_DISTANCE, 9-9
- Oracle Technology Network (OTN), xix
- oracle.spatial.geocoder.Metadata, C-9
- OVERLAPBDYDISJOINT, 7-6, 15-4
- OVERLAPBDYINTERSECT, 7-6, 15-4

## P

---

- parallel load, A-14
- partitioned tables, 1-22, A-14
- plotting tiles, A-5
- point data, 1-7, 13-8, 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, 2-13, 2-14
  - point on surface, 7-25
- polygon collection, 2-14
- polygon data, 1-7

- POPULATE\_INDEX, 13-3
- POPULATE\_INDEX\_FIXED, 13-5
- POPULATE\_INDEX\_FIXED\_POINTS, 13-8
- primary filter, 12-6
- primitive, 1-4

## Q

---

- query, 1-8
- query window, 4-3, 12-4

## R

---

- rectangle, 2-11
  - minimum bounding, 6-14
- rejected records, C-9
- RELATE, 7-4, 15-2
- relational model
  - schema, 10-1
- R-tree indexes, xix

## S

---

- sample program, A-18, A-20
- schema, 10-1
  - creating index on table in another schema, 3-14
  - invoking operators on table in another
    - schema, 4-9
  - object-relational model, 2-1
  - relational model, 10-1
- SDO\_AREA, 7-7
- SDO\_BUFFER, 7-9
- SDO\_CENTROID, 7-11
- SDO\_CODE, 2-21
- SDO\_CODE\_SIZE, 13-10
- SDO\_CONVEXHULL, 7-13
- SDO\_DIFFERENCE, 7-15
- SDO\_DISTANCE, 7-18
- SDO\_ELEM\_INFO, 2-8
- SDO\_ETYPE, 2-8
- SDO\_FILTER operator, 9-2
- SDO\_GEOMETRY object type, 2-6
- SDO\_GROUPCODE, 2-21
- SDO\_GTYPE, 2-6
- SDO\_INDEX\_TABLE, 2-21

- SDO\_INTERPRETATION, 2-9
- SDO\_INTERSECTION, 7-20
- SDO\_LENGTH, 7-23
- SDO\_NN operator, 9-4
- SDO\_ORDINATES, 2-12
- SDO\_POINT, 2-8
- SDO\_POINTONSURFACE, 7-25
- SDO\_POLY\_DIFFERENCE, 7-27
- SDO\_POLY\_INTERSECTION, 7-29
- SDO\_POLY\_UNION, 7-31
- SDO\_POLY\_XOR, 7-33
- SDO\_RELATE operator, 9-6
- SDO\_ROWID, 2-21
- SDO\_SRID, 2-8
- SDO\_STARTING\_OFFSET, 2-8
- SDO\_STATUS, 2-21
- SDO\_UNION, 7-35
- SDO\_VERSION, 13-11
- SDO\_WITHIN\_DISTANCE operator, 9-9
- SDO\_XOR, 7-38
- secondary filter, 12-7
- simple element, 2-8
- spatial data structures
  - object-relational model, 2-1
  - relational model, 10-1
- spatial database
  - sizing, A-4
- spatial index, 3-9, 11-6, 13-12, 13-14
  - performance, 6-4, 14-3
- spatial indexing
  - fixed, 1-12
  - hybrid, 1-16
- spatial join, 4-8, 12-8, A-11
- spatial query, 4-3, 12-4
- SQL script, A-15
- SQL\*Loader, 3-1, 11-2
- SRID (SDO\_SRID), 2-8

## T

---

- table partitioning, 1-22
- tessellation, 1-11, 11-6
- tile, 1-10, 4-1, 12-1
- tiling, 6-7, 13-14, 14-5, A-2
- TO\_734, 8-3

- TO\_81x, 8-5
- tolerance, 1-7
- TOUCH, 7-6, 15-4
- transactional insert, 3-3, 11-4
- two-tier query, 1-8, 4-1, 12-1

## U

---

- union, 7-31
- UPDATE\_INDEX, 13-12
- UPDATE\_INDEX\_FIXED, 13-14
- USER\_SDO\_GEOM\_METADATA view, 2-17
- USER\_SDO\_INDEX\_METADATA view, 2-19

## V

---

- VALIDATE\_GEOMETRY, 7-41, 15-5
- VALIDATE\_LAYER, 7-43
- VERIFY\_LAYER, 13-16
- Version Management, xix
- visualizing geometries, A-18, A-20
- visualizing tiles, A-5, A-20

## W

---

- WITHIN\_DISTANCE, 7-45