Oracle8*i*™ Spatial

User's Guide and Reference

Release 8.1.5

February 1999

Part No. A67295-01

ORACLE®

Enabling the Information Age™

Oracle8*i* Spatial User's Guide and Reference

Part No. A67295-01

Release 8.1.5

Primary Author: Jeff Hebert

Contributing Author: Anna Logan

Contributors: Frank Wang, Siva Ravada, Ran Wei, Jayant Sharma, and Dan Geringer

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.

Contents

3 Loading and Indexing Spatial Object Types

4 Querying Spatial Data

5 Indexing Statements for Object Relational Model

6 Tuning Functions and Procedures for Object-Relational Model

7 Geometry Functions for Object-Relational Model

8 Migration Procedures

B Installation, Compatibility, and Migration Issues

C Partitioning Legacy Point Data

Glossary

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle8*i* Spatial User's Guide and Reference, Release 8.1.5

Part No. A67295-01

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 chapter, section, and page number (if available).

You can send comments to us in the following ways

- e-mail: nedc_doc@us.oracle.com
- FAX: 603.897.3316 Attn: Spatial Documentation
- postal service:
  Oracle Corporation
  Oracle8*i* Spatial Documentation
  One Oracle Drive
  Nashua, NH 03062
  USA

If you would like a reply, please include your name, address, and telephone number.

Preface

The *Oracle8i Spatial User's Guide and Reference* provides user and reference information for the Spatial product, and extensions to Oracle8*i* Enterprise Edition.

Spatial requires Oracle8*i* Enterprise Edition. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features. However, several advanced features, such as extended data types, are available only with the Enterprise Edition, and some of these features are optional. For example, to use Oracle8*i* table partitioning, you must have the Enterprise Edition and the Partitioning Option.

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

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 new 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 how to use partitioned point data.
Glossary	Provides definitions of terms used in this guide.

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 Error Messages* - Spatial messages are in the range of 13000 to 13499
- *Oracle8i Concepts*
- *Oracle8i Tuning*
- *Oracle8i Utilities*

For additional information about the Spatial option, including a demonstration program, several white papers, and other assorted collateral, visit the official Spatial Web site: `http://www.oracle.com/st/cartridges/spatial/`

Conventions

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

The following conventions are used in this guide:

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

1

Spatial Concepts

Oracle8*i* Spatial is an integrated set of functions and procedures that enables spatial data to be stored, accessed, and analyzed quickly and efficiently in an Oracle8*i* database.

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

1.1 What Is the Spatial Product?

Oracle8*i* Spatial, often referred to as Spatial, provides a standard SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle8*i* database. It consists of four components:

1. A schema that prescribes the storage, syntax, and semantics of supported geometric data types
2. A spatial indexing mechanism
3. A set of operators and functions for performing area-of-interest and spatial join queries
4. Administrative utilities

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

This release of Spatial supports two mechanisms for representing geometry. The first, an object-relational scheme, uses a table with single column of type MDSYS.SDO_GEOMETRY and a single row per geometry instance. The second, a relational scheme, uses a table with a predefined set of columns of type NUMBER and one or more rows for each geometry instance. These mechanisms roughly correspond to two alternatives described in the OpenGIS ODBC/SQL specification

for geospatial features. The first corresponds to a “SQL with Geometry Types” implementation of spatial feature tables, and the second 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. The remainder of this chapter describes Spatial concepts and features, without reference to their implementation wherever possible.

1.2 Introduction to Spatial Data

The Spatial option 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 the Spatial option 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.

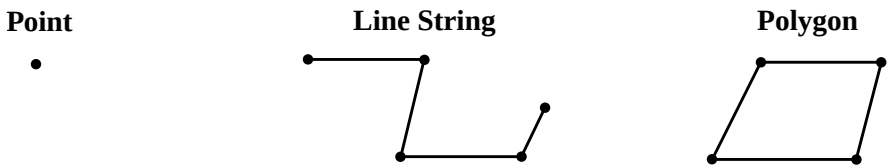
1.3 Geometric Types for Relational and Object-Relational Models

The relational model of the Spatial option supports three geometric primitive types and geometries composed of collections of these types. The three 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



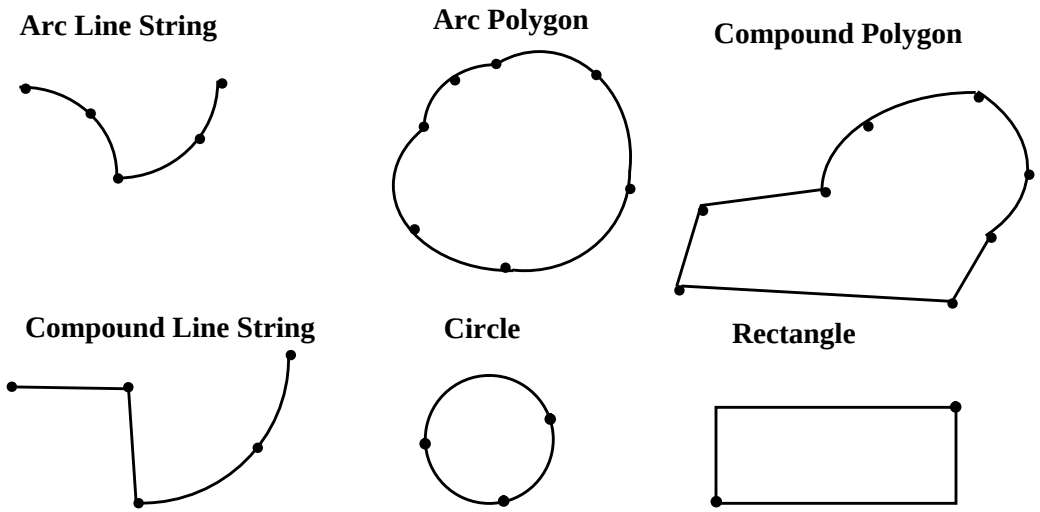
Self-crossing polygons are not supported although self-crossing line strings are. If a line string crosses itself, it does not become a polygon. A self-crossing line string does not have any implied interior.

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

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



1.4 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 be a road or flight path, and a polygon could be a state, city, zoning district, or city block.

1.4.1 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[1] 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 either a clockwise or counter-clockwise order around the polygon.

1.4.2 Geometry

A **geometry** is the representation of a user's 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] Point data can also be stored in a partitioned table. See Appendix C, "Partitioning Legacy Point Data" for details.

1.4.3 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 their associated spatial index are stored in the database in standard tables.

1.5 Query Model

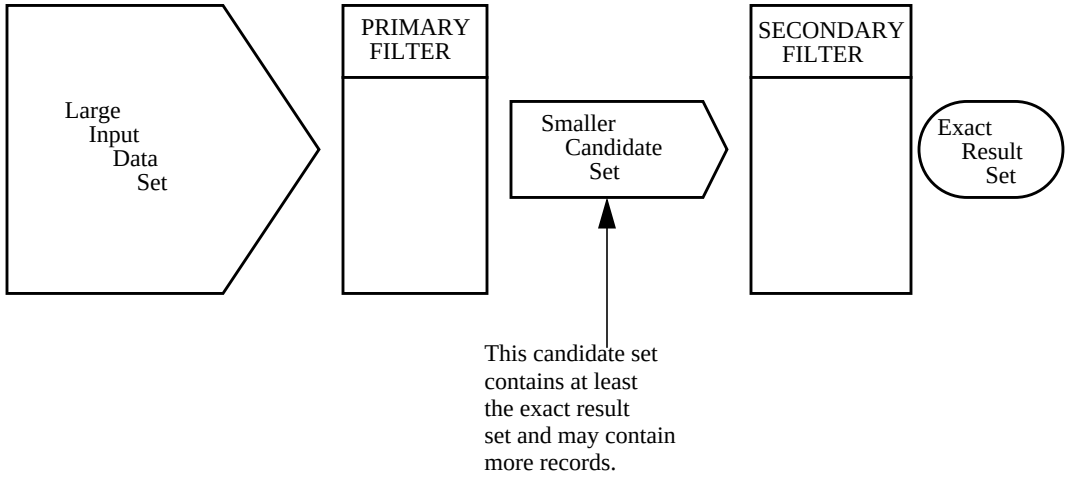
Spatial uses a *two-tier* query model to resolve spatial queries and spatial joins. The term is used to indicate that two distinct operations are performed in order to resolve queries. The output of both operations yields the exact result set.

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 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.

1.6 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 within tables (or data spaces) 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. Variable-sized tiling is controlled by the value supplied for the maximum number of tiles. 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. 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 fewer the number of tiles or the larger the tiles, the coarser the approximations.

Spatial supports two valid combinations of SDO_LEVEL and SDO_NUMTILES. The first, with a non-null SDO_LEVEL and a null SDO_NUMTILES value, results in fixed-sized tiles (called fixed indexing in this guide.) The second, with a non-null SDO_LEVEL and a non-null SDO_NUMTILES, results in hybrid indexing. Hybrid

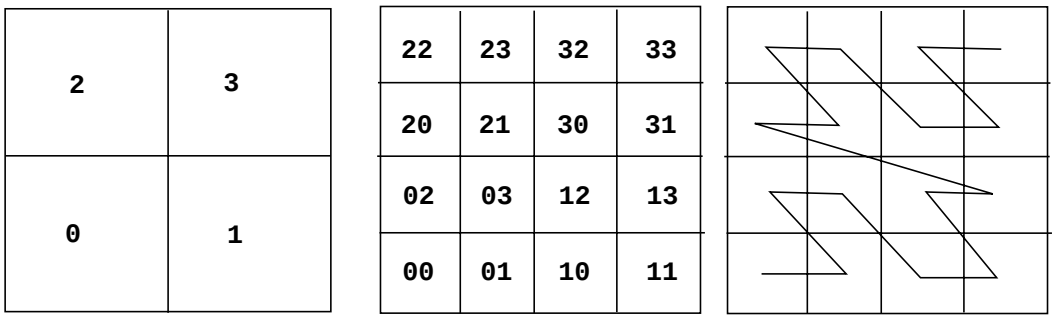
indexing generates two sets of tiles per geometry. One set contains fixed-size tiles and the other set contains variable-sized tiles.

1.6.1 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



1.6.2 Fixed Indexing

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

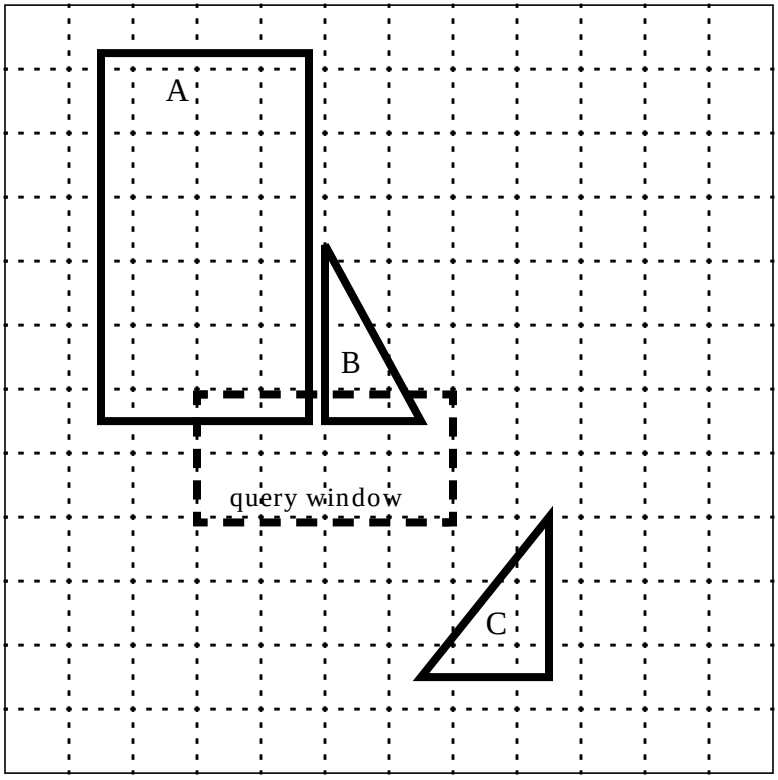
Two geometries are likely to interact, and hence pass the primary filter stage, if they share one or more tiles. The SQL statement for the primary filter stage is:

```
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.

Using 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



Using 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. A window query would likely 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 this figure.

The `SDO_TUNE` package has an `ESTIMATE_TILING_LEVEL()` function that helps determine an appropriate tiling level for your data set.

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

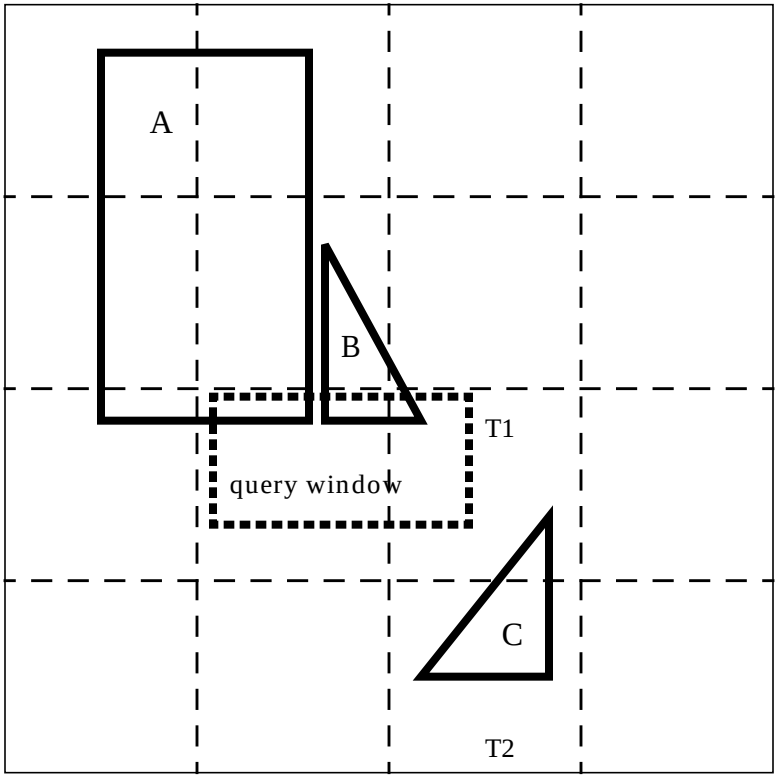
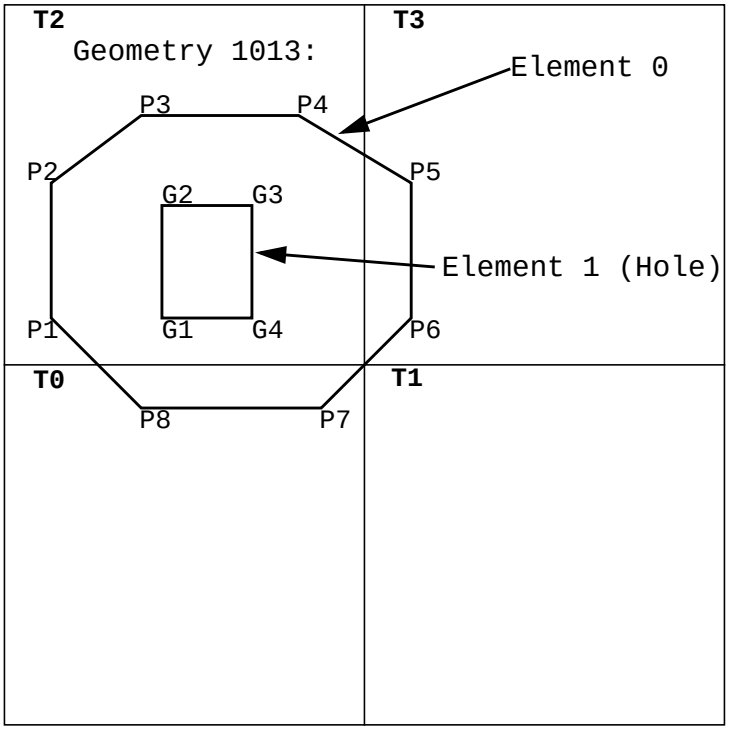


Figure 1–7 illustrates geometry 1013 tessellated to three fixed-sized tiles at level 1. The codes for these cover tiles are then stored in an SDOINDEX table.

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 have to 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

Table 1–1 SDOINDEX Table Using Fixed-Size Tiles(Cont.)

SDO_GID <number>	SDO_CODE <raw>
1013	T3

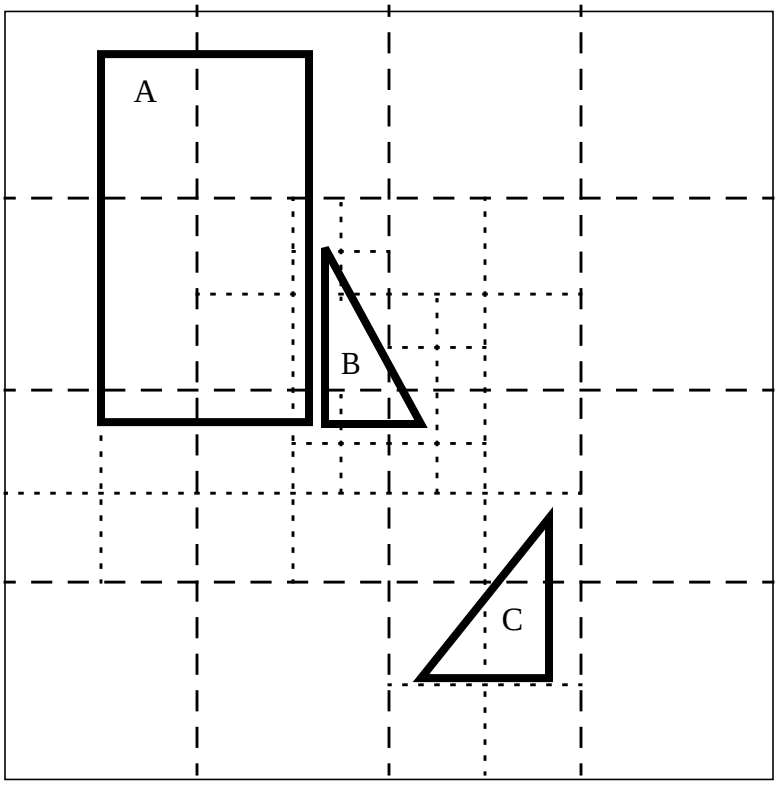
All elements in a geometry are tessellated. In a multielement geometry like 1013, Element 1 is already covered by tile T2 from the tessellation of Element 0. If, however, the specified tiling resolution were such that tile T2 were further subdivided and one of these smaller tiles were completely contained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

1.6.3 Hybrid Indexing

Hybrid indexing is the preferred method for indexing the object-relational model. Hybrid indexing uses a combination of fixed- 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 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.

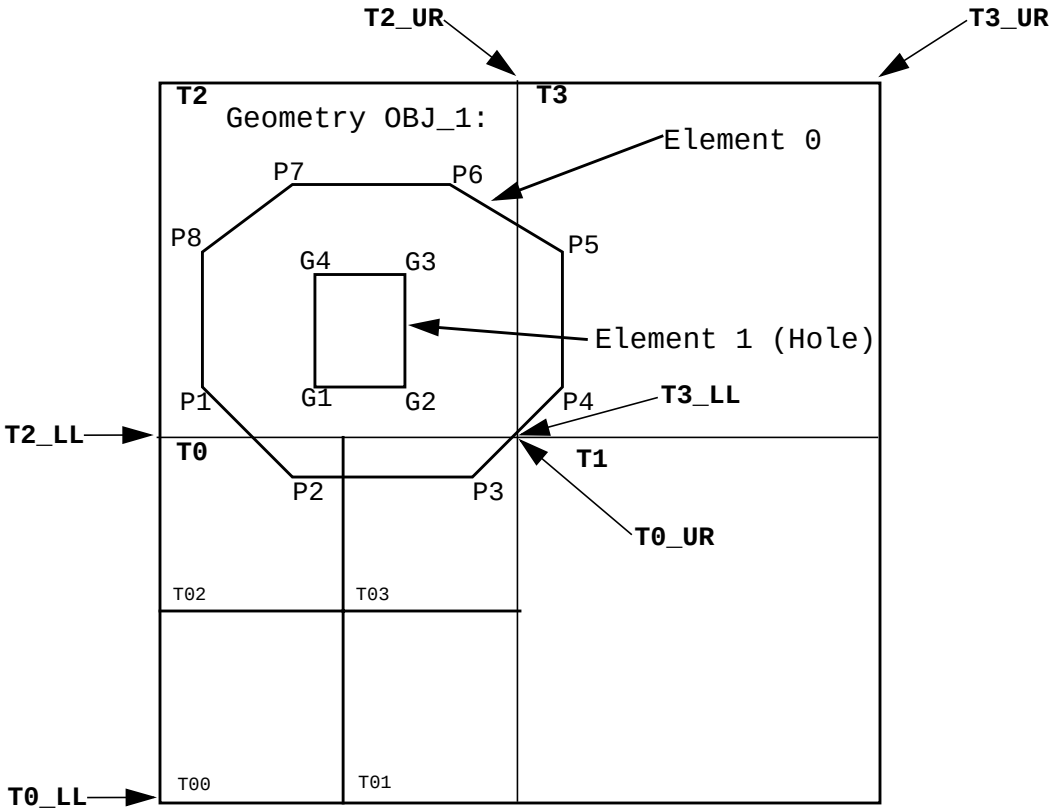
Figure 1–8 Variable-Sized Tile Spatial Indexing



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. Note that the tiles have been numbered for simplicity and do not reflect the format used in Spatial.

Figure 1–9 Decomposition of the Geometry



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 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 2.4.

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 is covered by a redundant tile from the tessellation of Element 0 but the tile, T2, 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.

1.7 Spatial Relations and Filtering

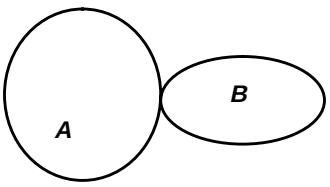
Spatial uses filter methods to determine the spatial relationship between entities in the database. The spatial relation is based on geometry locations. The most common spatial relations are based on topology and distance. For example, the boundary of an area consists of a set of curves that separate 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. Next, 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.

Spatial has two secondary filter methods. One method evaluates topological criteria and a second method determines if two spatial objects are within a Euclidean distance of each other. The secondary filter that evaluates topological criteria is called RELATE. The syntax is given in subsequent chapters that describe geometry functions and operators. RELATE 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 three components -- a boundary Ab, an interior Ai, and an exterior Ae, any pair of objects will have nine possible interactions between their components. Pairs of components will have an empty (0), or a non-empty (1) set intersection. The set of interactions between two 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 two polygons that are adjacent to one another. This matrix yields the following bit mask, generated in row-major form: “101001111”.

Figure 1–10 The 9-Intersection Model



A Touch B

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

The 9-intersection matrix

Some of the topological relationships identified in the seminal work by Dr. Egenhofer[1] 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

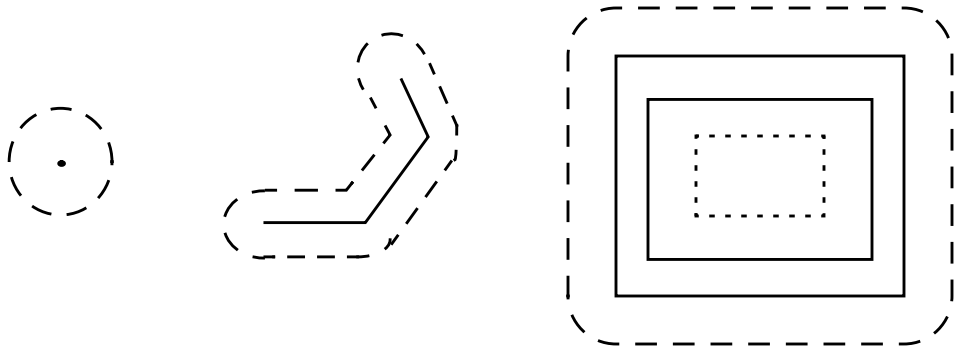
[1] Dr. Max Egenhofer, University of Maine, Orono.

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, WITHIN_DISTANCE, determines if two spatial objects, A and B, are within a Euclidean distance of one another. First it constructs a distance buffer, D_b, around the reference object B. It then checks that A and D_b are non-disjoint.The distance buffer of an object consists of all points within the given distance from that object. Figure 1–11 shows the distance buffers for point, line, and area objects. Notice how the buffer is rounded near the corners of the objects.

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



1.8 Partitioned Point Data

Point data, unlike line and polygon data, has the unique characteristic of always using only one tile per point. There are cases where this difference can be exploited.

Spatial has an enhanced spatial indexing mechanism capable of handling very large data sets consisting of complex geometries. For applications handling point data sets that are several tens of gigabytes or larger, further performance gains can be achieved by using Oracle8*i* table partitioning features.

Table partitioning is available only with the Partitioning Option of Oracle8*i* Enterprise Edition. If the Partitioning Option is available to you, the preferred method is to use Oracle8*i* table partitioning in conjunction with spatial indexing (using the relational model). See the *Oracle8i Concepts* guide for a description of Oracle8*i* Partitioning. See Section A.2.6.3 for a description of a sample script that uses table partitioning with point data.

A previous release of Spatial Data Option (from which the current Spatial product has evolved) utilized its own version of table partitioning instead of spatial indexing. Appendix C briefly describes the deprecated partitioning scheme for those customers with legacy point data sets. While this feature is still enabled in the current release, it may be removed in the future.

Part I

Object-Relational Model

Oracle8*i* Spatial supports two models for representing geometries: relational and object-relational. The two models are mutually exclusive. See Section A.1 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 new object-relational model.

This part of the User's Guide and Reference contains the following chapters, describing 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"

2

The Object-Relational Schema

The object-relational implementation of Oracle8*i* 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 Object-Relational Data Structures

In the Spatial object-relational model, a single SDO_GEOMETRY object replaces the rows and columns in a <layername>_SDOGEOM table of the 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. The table does not require the "_SDOGEOM" suffix anymore. Because the SDO_GEOMETRY type does not have an SDO_GID attribute, 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.

Oracle8*i* 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 attributes of the SDO_GEOMETRY object type have the following semantics:

- **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
0	UNKNOWN_ GEOMETRY	Spatial ignores this geometry.
1	POINT	Geometry contains one point.
2	LINESTRING	Geometry contains one line string.
3	POLYGON	Geometry contains one polygon with or without holes[1].
4	Collection	Geometry is a heterogeneous collection of elements.[2]
5	MULTIPOINT	Geometry has multiple points.
6	MULTILINESTRING	Geometry has multiple line strings.
7	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.

  Values 8-99 are reserved for future use. The enumeration of types shown in Table 2–1 is recommended, however, Spatial does not check or enforce all geometry consistency constraints. Spatial does check the following constraints:

  - For SDO_GTYPEs 1 and 5, any subelement not of ETYPE 1 is ignored.
  - For SDO_GTYPEs 2 and 6, any subelement not of ETYPE 2 or 4 is ignored.
  - For SDO_GTYPEs 3 and 7, any subelement not of ETYPE 3 or 5 is ignored.

  The `SDO_GEOM.VALIDATE_GEOMETRY( )` function may be used to evaluate the consistency of a single geometry object or all the instances of SDO_ GEOMETRY in a specified feature table.

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

- **SDO_POINT** - Is 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 points in the SDO_POINT attribute for optimal storage.
- **SDO_ELEM_INFO** - Is a varying length array of numbers. This attribute lets you know how to interpret the ordinates stored in the SDO_ORDINATES attribute.

  Each triplet set of numbers conveys information about one geometry element, and a geometry may contain many elements. If a geometry has one element, then the SDO_ELEM_INFO array has three numbers; if the geometry has two elements, then the array has six numbers, and so on. Each triplet set is interpreted as follows:

  1. 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).
  2. SDO_ETYPE - Indicates the type of the element. Valid values are 0 through 5.

     SDO_ETYPEs 1, 2, and 3, are considered simple elements. They are defined by a single triplet entry in the SDO_ELEMINFO array. SDO_ETYPEs 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.

     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.
  3. 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.

     A description of valid SDO_ETYPE and SDO_INTERPRETATION value pairs is 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_ETYPEs 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 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 is 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.

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

SDO_ETYPE	SDO_INTERPRETATION	Meaning
3	1	Simple polygon whose vertices are connected by straight line segments.
3	2	Polygon made up of a connected sequence of circular arcs that closes on itself. The end point of the last arc is the same as the start point of the first arc. Each circular arc is described using three coordinates: the arc's start point, any point on the arc, and the arc's end point. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a polygon made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc. The coordinates for points 1 and 5 must be the same, and point 3 is not repeated.
3	3	Rectangle type. A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it.
3	4	Circle type. Described by three points, all on the circumference of the circle.
4	N > 1	Line string with some vertices connected by straight line segments and some by circular arcs. The value, N, in the Interpretation column specifies the number of contiguous subelements that make up the line string. The next N triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The last point of a subelement is the first point of the next subelement, and must not be repeated. See Section 2.2 and Figure 2–2 for an example of a geometry using this type.

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

SDO_ ETYPE	SDO_ INTERPRETATION	Meaning
5	N > 1	Simple polygon with some vertices connected by straight line segments and some by circular arcs. The value, N, in the Interpretation column specifies the number of contiguous subelements that make up the polygon. The next N triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The end point of a subelement is the start point of the next subelement and must not be repeated. The start and end points of the polygon must be the same. See Section 2.2 and Figure 2-3 for an example of a geometry using this type.

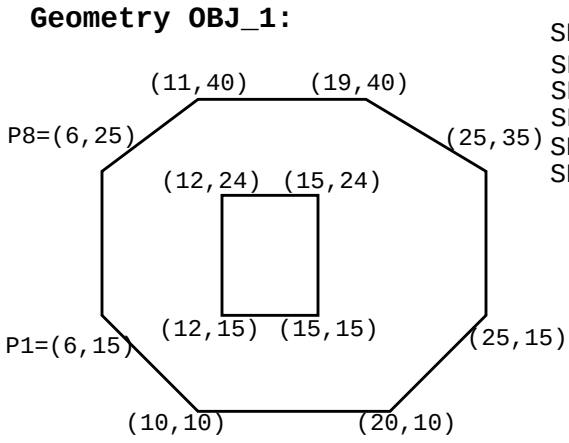
- **SDO_ORDINATES** - Is a varying length array (1048576) of NUMBER type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the SDO_ELEM_INFO varying length array. The values in the array are ordered by dimension. For example, 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 SDO_GEOM_METADATA table, described in Section 2.3.

  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 multi-element 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 Geometry Examples Using the Object-Relational Model

Figure 2–1 illustrates a geometry with two elements. The inner element in this example is treated as a void (a hole).

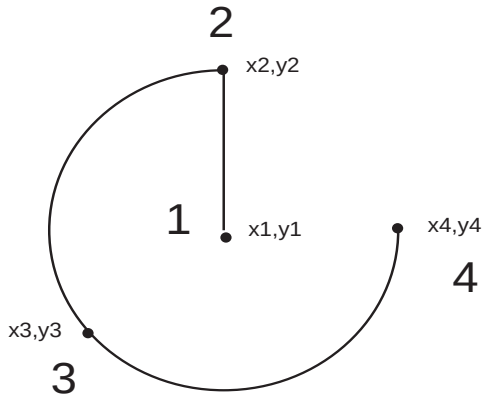
Figure 2–1 Geometry with a Hole




```
SDO_GEOMETRY Column = (
SDO_GTYPE = 3
SDO_SRID = NULL
SDO_POINT = NULL
SDO_ELEM_INFO = (1,3,1,19,3,1)
SDO_ORDINATES = (6,15,10,10,20,10,
                 25,15,25,35,19,40,
                 11,40,6,25,6,15,
                 12,15,15,15,15,24,
                 12,24,12,15) )
```

In Figure 2–2, consider 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 2–2 Compound Element



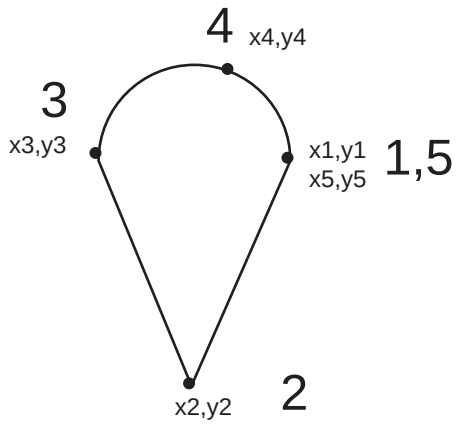
The first triplet indicates that this element is a compound line string made up of two line strings, which are described with the next two triplets.

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

The third triplet indicates that the second line string is made up of circular arcs with ordinates starting at offset 3. The end point of this line string is determined by the starting offset of the next element or the current length of the SDO_ ORDINATES array, if this is the last element.

In Figure 2-3, consider 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 {(1,5,2), (1,2,1), (5,2,2)}.

Figure 2–3 Compound Polygon



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.

2.3 Geometry Metadata Structure

The geometry metadata describing the dimensions, lower and upper bounds, and tolerance in each dimension must be stored as a single entry in a table named SDO_ GEOM_METADATA created in your schema and defined as follows:

```
Create Table SDO_GEOM_METADATA (
  TABLE_NAME   VARCHAR2(30),
  COLUMN_NAME  VARCHAR2(30),
  DIMINFO      MDSYS.SDO_DIM_ARRAY);
```

There should be one table with the previous name and layout created 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, then there must be three entries in the table herman.sdo_geom_metadata. The user, or application, is responsible for populating and maintaining the data in this table.

The SDO_GEOM_METADATA.TABLE_NAME column contains the name of a feature table, such as Roads or Parks, that has a column of type SDO_GEOMETRY. The name of this column, of type SDO_GEOMETRY, is stored in the feature table in the SDO_GEOM_METADATA.COLUMN_NAME column. For the tables Roads and Parks, this column is called theGeometry, and therefore the SDO_GEOM_METADATA table should contain rows (Roads, theGeometry, SomeDimInfo1) and (Parks, theGeometry, SomeDimInfo2). The SDO_GEOM_METADATA.DIMINFO row is a varying length array of an object type, ordered by dimension, and has one entry per dimension. The row is defined as follows:

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

where SDO_DIM_ELEMENT is defined as:

```
Create Type SDO_DIM_ELEMENT as OBJECT (
  SDO_DIMNAME VARCHAR2(64),
  SDO_LB NUMBER NOT NULL,
  SDO_UB NUMBER NOT NULL,
  SDO_TOLERANCE NUMBER NOT NULL);
```

The SDO_DIM_ARRAY instance is of size N if there are N-dimensions. That is, SDO_GEOM_METADATA.DIMINFO contains 2 SDO_DIM_ELEMENT instances for 2-dimensional geometries, or 3 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.

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 SDO_GEOM_METADATA table are populated with the data for those objects.

2.4 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 1.6. Both the spatial index data and metadata are stored in tables created and maintained by the spatial indexing routines. These tables are created in the same schema as the owner of the feature (underlying) table with a spatial index created on a column of type SDO_GEOMETRY.

2.4.1 Spatial Index Tables

There is one metadata view per schema (user), named SDO_INDEX_METADATA, and one index data table per spatially indexed column in that schema. Thus if user Herman has five feature tables, each with a spatial index on their respective SDO_GEOMETRY typed column, then Herman's schema has one SDO_INDEX_METADATA view and five tables containing spatial index data. The index data table names are not created by adding the " _SDOINDEX" suffix to the layer name. Instead, the 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 1.6) and the values of the relevant index parameters.

The SDO_INDEX_METADATA view contains the following columns whose type and purpose are shown in Table 2–3.

Table 2–3 Columns in an SDO_INDEX_METADATA View

Column Name	Data Type	Purpose
SDO_LEVEL	NUMBER	The fixed tiling level at which to tile all objects in the feature table.
SDO_NUMTILES	NUMBER	Suggested number of tiles per object that should be used to approximate the shape.
SDO_MAXLEVEL	NUMBER	The maximum level for any tile for any object. It will always be greater than the SDO_LEVEL value.
SDO_COMMIT_INTERVAL	NUMBER	The number of geometries (rows) to process, during index creation, before committing the insertion of spatial index entries into the SDOINDEX table. See Appendix A for a discussion of the use of this parameter.

Table 2–3 Columns in an SDO_INDEX_METADATA View (Cont.)

Column Name	Data Type	Purpose
SDO_INDEX_TABLE	VARCHAR2	Name of the SDOINDEX table.
SDO_TABLESPACE	VARCHAR2	Same as in the basic SQL CREATE TABLE statement. Tablespace in which to create the SDOINDEX table.
SDO_INITIAL_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_NEXT_EXTENT	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_PCTINCREASE	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_MIN_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_MAX_EXTENTS	NUMBER	Same as in SQL CREATE TABLE statement.
SDO_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_COORDSYS	VARCHAR2	Reserved for future use.
SDO_INDEX_PRIMARY	NUMBER	Indicates if this is a primary or secondary index. 1 = primary, 2 = secondary.
SDO_INDEX_OWNER	VARCHAR2	The owner of the index.
SDO_INDEX_NAME	VARCHAR2	The name of the index.
SDO_TSNAME	VARCHAR2	The schema name of the SDO_INDEX_TABLE.
SDO_COLUMN_NAME	VARCHAR2	The column name on which this index is built.

2.4.2 Spatial Index Data Dictionary View

The index data table will have some or all of the columns shown in Table 2–4.

Table 2–4 Columns in a Spatial Index Data Table

Column Name	Data Type	Purpose
SDO_ROWID	RAW	Row ID of a row in a feature table containing the indexed object.
SDO_CODE	RAW	Index entry for the object in the row identified by SDO_ROWID.
SDO_MAXCODE	RAW	Padded SDO_CODE column.
SDO_GROUPCODE	RAW	Index entry at level SDO_LEVEL.

Table 2–4 Columns in a Spatial Index Data Table (Cont.)

Column Name	Data Type	Purpose
SDO_META	RAW	Metadata portion of the SDO_CODE for a hybrid index.

The columns, SDO_ROWID and SDO_CODE, are always present. The remaining columns are present only when the selected index type is HYBRID, which is the recommended indexing for the object-relational model.

2.5 Usage Notes

Stored procedures, provided with Spatial release 8.1, assume the existence of the following schema objects: instances of SDO_GEOMETRY and SDO_DIM_ARRAY objects and the metadata table SDO_GEOM_METADATA in the user's schema. While specific instances of the SDO_GEOM_METADATA view may contain additional columns, they are required to contain the columns described in Table 2–3 with the *same* column names and data types.

3

Loading and Indexing Spatial Object Types

This chapter describes how to load spatial data into a database, including storing the data in a table with a column of type SDO_GEOMETRY and creating a spatial index for it.

The following steps will enable you to 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 SQL*Loader[1] 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.

[1] See *Oracle8i Utilities* for information on SQL*Loader.

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 *
INTO TABLE POLY_4PT
REPLACE
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(
gid    char(6),
geometry COLUMN OBJECT
( sdo_gtype INTEGER EXTERNAL,
  sdo_srid INTEGER EXTERNAL,
   isnull FILLER CHAR,
  SDO_POINT COLUMN OBJECT  NULLIF geometry.isnull="pt"
  ( X INTEGER EXTERNAL,
    Y INTEGER EXTERNAL,
    Z INTEGER EXTERNAL),
 SDO_ELEM_INFO VARRAY terminated by ';'
       (SDO_ORDINATES char(38)),
 SDO_ORDINATES VARRAY terminated by ':'
    (SDO_ORDINATES char(38))))
begindata
1,3,,pt,,,,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,,pt,,,,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,,pt,,,,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 *
INTO TABLE POINT
REPLACE
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(
gid    char(6),
geometry COLUMN OBJECT
( sdo_gtype INTEGER EXTERNAL,
  sdo_srid INTEGER EXTERNAL,
  SDO_POINT COLUMN OBJECT
  ( X INTEGER EXTERNAL,
    Y INTEGER EXTERNAL,
    Z INTEGER EXTERNAL),
    is_null1 FILLER CHAR,
 SDO_ELEM_INFO VARRAY terminated by ';' NULLIF geometry.is_null1="v1"
       (SDO_ORDINATES char(38)),
    is_null2 FILLER CHAR,
 SDO_ORDINATES VARRAY terminated by ':'
 NULLIF geometry.is_null2="v2"
    (SDO_ORDINATES char(38))))
begindata
1,1,,-122.4215,37.7862,,v1,;v2,:
2,1,,-122.4019,37.8052,,v1,;v2,:
3,1,,-122.426,37.803,,v1,;v2,:
4,1,,-122.4171,37.8034,,v1,;v2,:
5,1,,-122.416151,37.8027228,,v1,;v2,:
```

3.1.2 Transactional Insert Using SQL

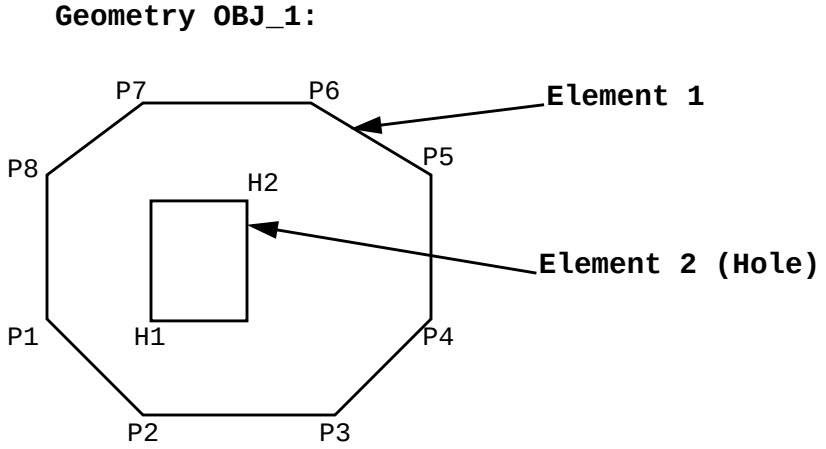
Oracle8*i* Spatial uses standard Oracle8*i* tables that can be accessed or loaded with standard SQL syntax. This section contains examples of transactional inserts into columns of type SDO_GEOMETRY. Note that SQL statements in Oracle8*i* have a limit of 999 arguments. Therefore, you cannot create variable-length arrays of more than 999 elements using transactional INSERT statements.

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



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(3, NULL,NULL,
                      MDSYS.SDO_ELEM_INFO_ARRAY(1,3,1, 19,3,3),
                      MDSYS.SDO_ORDINATE_ARRAY(6,15, 10,10, 20,10, 25,15, 25,35,
                         19,40, 11,40, 6,25, 6,15, 12,15, 15,24)));
```

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

```
INSERT INTO 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)));
```

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)]
```

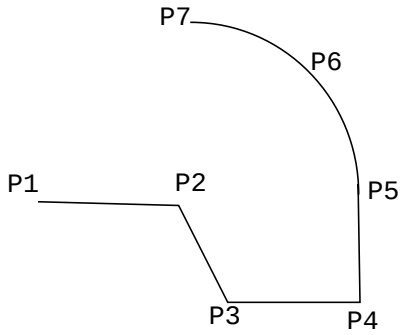
The SQL statement for inserting this compound line string in a feature table ROADS(GID Varchar2, Shape MDSYS.SDO_GEOMETRY) is:

```
INSERT INTO ROADS VALUES ('OBJ_2', MDSYS.SDO_GEOMETRY(2, 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 GTYPE, SDO_ELEM_INFO, and SDO_ORDINATES. The GTYPE is 2, the SDO_ELEM_INFO has nine values because there are two subelements for the compound line string. 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. Similarly, subelement 2 has a starting offset of 9. That is, the first ordinate value is SDO_ORDINATES(9), 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 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 SDO_GEOM_METADATA table is:

```
INSERT INTO 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)));
```

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:

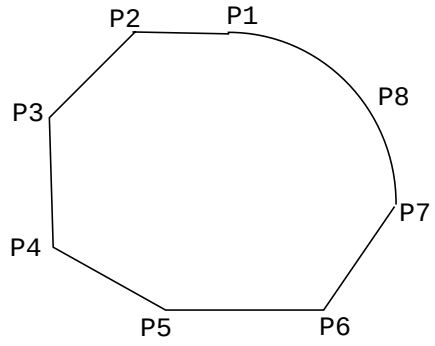
```
CREATE TABLE PARKS (GID VARCHAR2(32), SHAPE MSSYS.SDO_GEOMETRY);
```

The SQL statement for inserting this compound polygon is:

```
INSERT INTO PARKS VALUES ('OBJ_3', MDSYS.SDO_GEOMETRY(3, NULL,NULL
  MDSYS.SDO_ELEM_INFO_ARRAY(1,5,2, 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



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 SDO_GEOM_METADATA table is:

```
INSERT INTO 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)));
```

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 MSSYS.SDO_GEOMETRY);
```

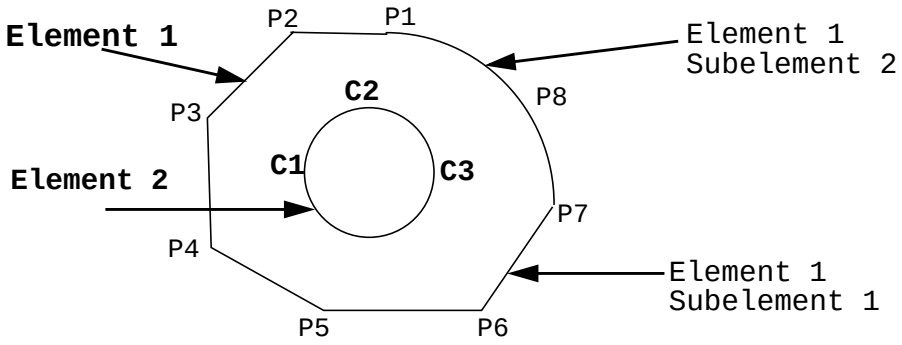
The SQL statement for inserting this compound polygon with holes is:

```
INSERT INTO Parks VALUES ('OBJ_4', MDSYS.SDO_GEOMETRY(3, NULL,NULL
  MDSYS.SDO_ELEM_INFO_ARRAY(1,5,2, 1,2,1, 13,2,2, 19,3,4),
  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 GTYPE is 3, the SDO_ELEM_INFO has four 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 SDO_GEOM_METADATA table is:

```
INSERT INTO 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)));
```

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(1,NULL,
                                             MDSYS.SDO_POINT(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 cover 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 and index maintenance routines on INSERT, or UPDATE, is determined by the SDO_LEVEL and SDO_NUMTILES values supplied by the user in the PARAMETERS clause of the CREATE INDEX statement.They are interpreted as follows:

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

The CREATE INDEX routine for spatial indexing has the same semantics as a standard SQL CREATE INDEX statement. An explicit commit is executed after the tessellation of all the geometries in a geometry column.

Because spatial index creation operates as a single transaction, it may require 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 dropped 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.

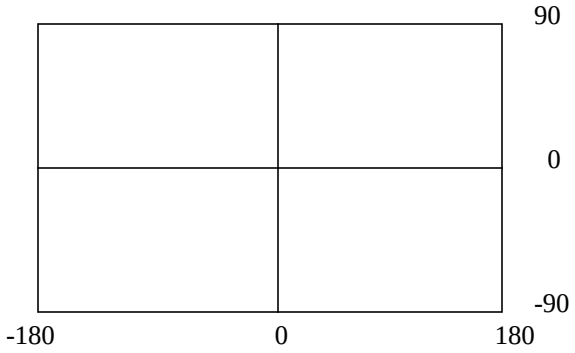
The domain used for indexing is defined by the upper and lower boundaries of each dimension stored in the DIMINFO column of the SDO_GEOM_METADATA table, which contains an entry for the table and geometry column to spatially index. A typical domain could be -180 to 180 degrees for longitude[1], and -90 to 90 degrees for latitude, as represented in Figure 3–5.

Figure 3–5 Sample Domain

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

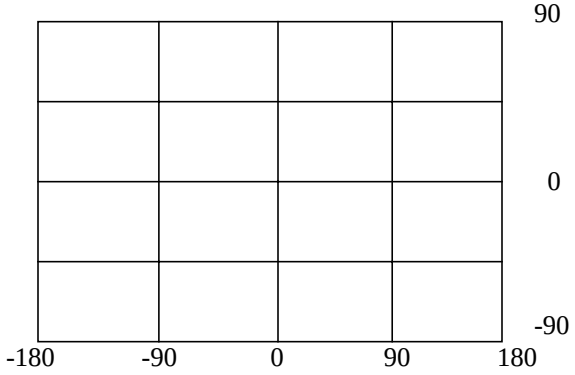
[1] The transference of the domain onto a sphere or Mercator projection is left up to an application. Spatial treats the domain as a flat Cartesian grid.

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:

length = (upper_bound - lower_bound) / 2 ^ sdo_level

The length refers to the length of the tile along the specified dimension. Applying this formula to the tiling shown in Figure 3–7 yields the following sizes:

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

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

At level 2 the tiles are 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 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 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 SDO_GEOM_METADATA has an entry for ROADS. GEOMETRY. Use the following SQL statement to create a fixed index named ROADS_FIXED on ROADS_ GEOMETRY.

Example 3–3 Create a Fixed Index

```
CREATE INDEX ROADS_FIXED ON ROADS(GEOMETRY) 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 cover the geometry, and a set of variable-sized tiles that fully cover 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 it would be 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 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".

As in Example 3–3, assume that the ROADS table has been loaded. Use the following statement to create the spatial index on ROADS.GEOMETRY:

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

4

Querying Spatial Data

This chapter describes how the structures of an object-relational model Spatial layer 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 model.

4.1 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. If both operations are performed, the exact result set is returned.

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

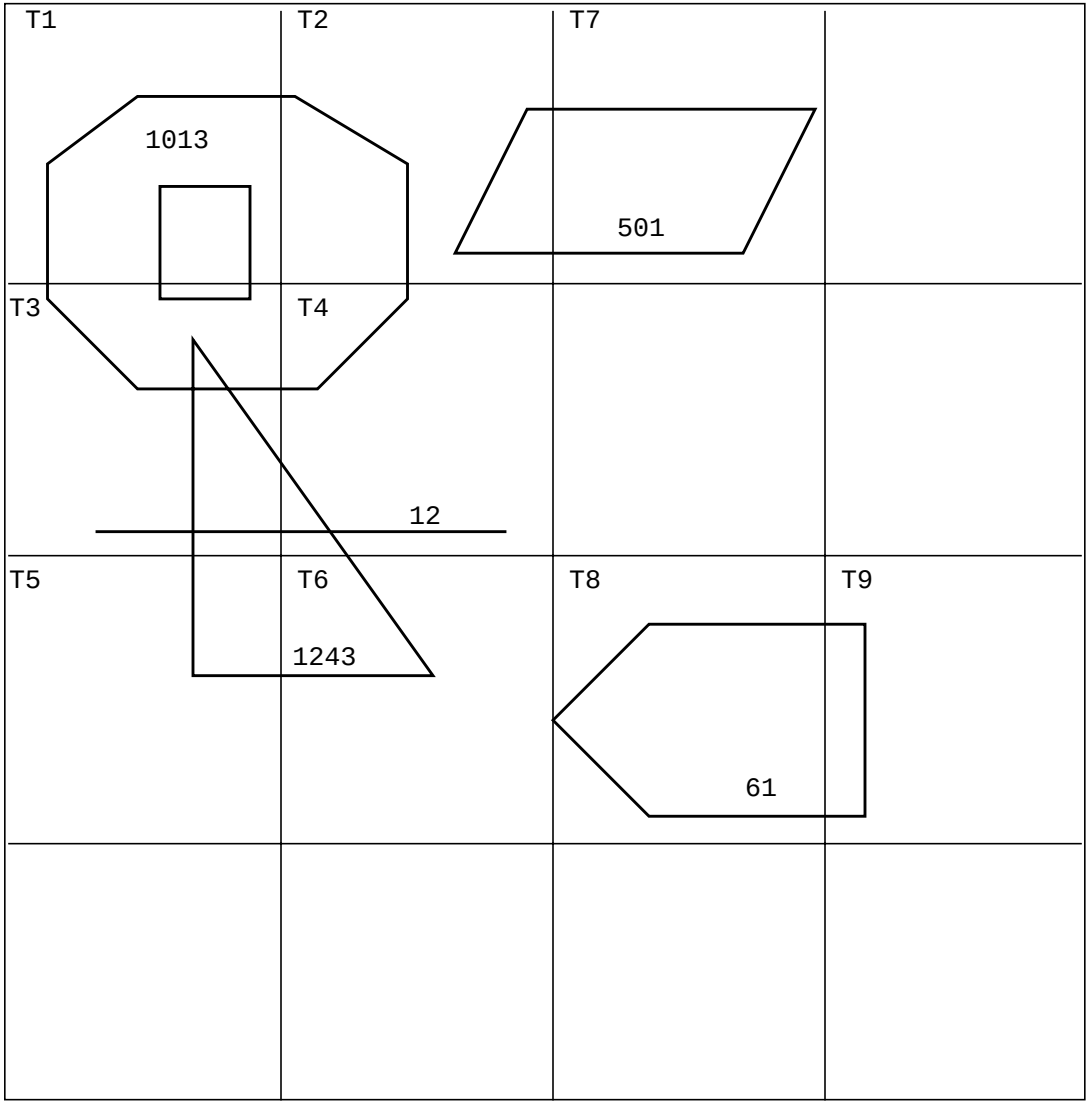
- The primary filter permits fast selection of candidate records to pass along to the secondary filter. The primary filter uses geometry approximations (or index tiles) to reduce computational complexity and is considered a lower cost filter.
- The secondary filter applies exact computational geometry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

4.2 Spatial Query

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

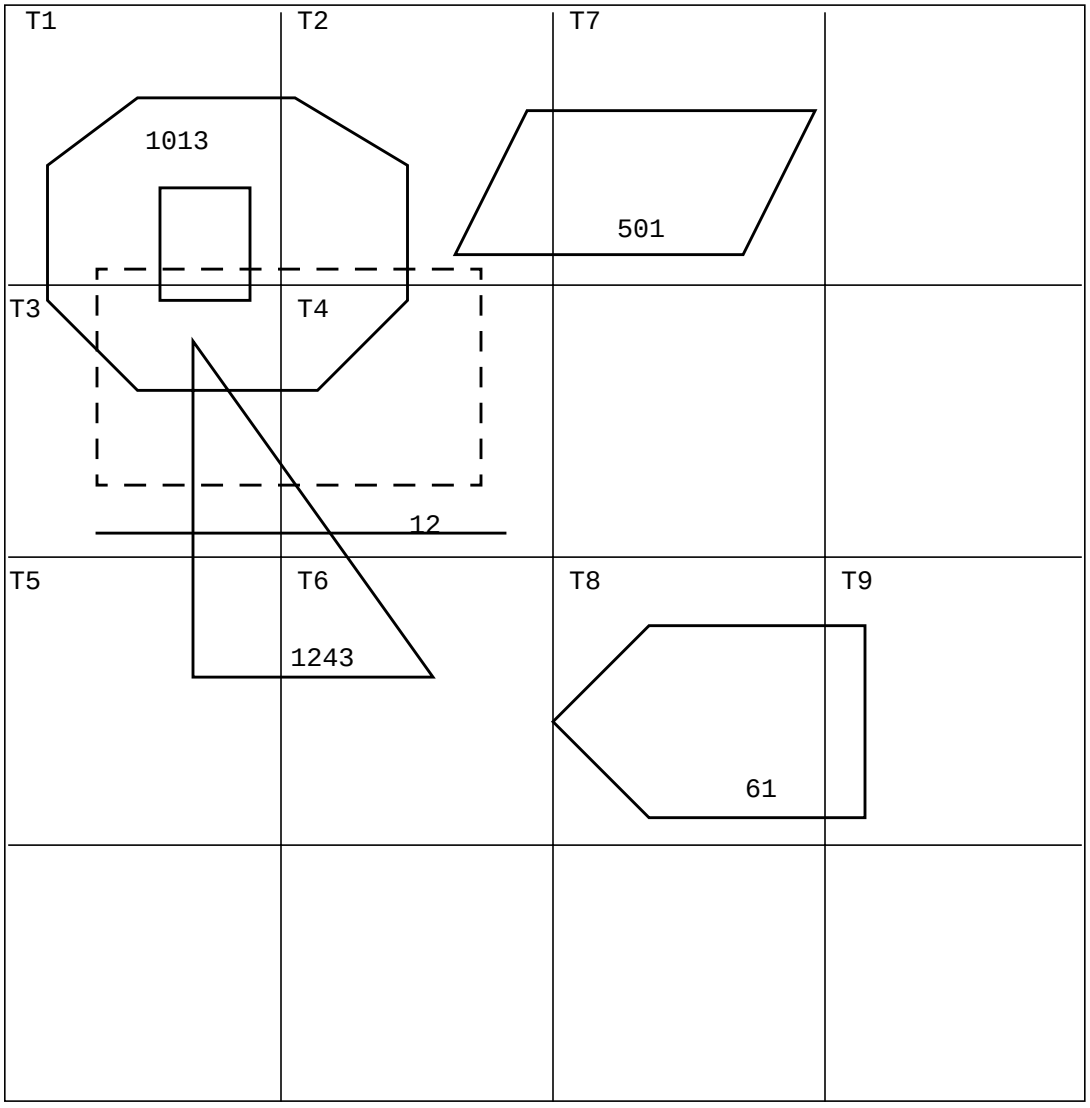
Consider the following layer containing several objects in Figure 4–1. Each object is labeled with its SDO_GID. The relevant tiles are labeled with 'T*n*'.

Figure 4–1 Tessellated Layer with Multiple Objects



A typical spatial query is to request all objects that lie within a defined fence or window. A **query window** is shown in Figure 4–2 by the dotted-line box. A dynamic query window refers to a fence that is not defined in the database, but that must be defined prior to using it.

Figure 4–2 Tessellated Layer with a Query Window



4.2.1 Primary Filter

Spatial release 8.1 provides a new 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 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, T2, 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 was 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 mdsys.sdo_filter(A.shape, mdsys.sdo_geometry(3,NULL,NULL,
                                          mdsys.sdo_elem_info(1,3,3),
                                       mdsys.sdo_ordinates(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 mdsys.sdo_filter(A.shape, :theWindow,'querytype=window') = 'TRUE';
```

Example 4–3[1] 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= 'WIN_1' AND
  mdsys.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 spatial 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 Filter

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:

[1] A limitation in SQLPLUS may result in an error if Example 4–3 is run in SQLPLUS. This error results when both tables are indexed, and it can occur with any Spatial operator (primary or secondary filter). The error does not occur with any other interface than SQLPLUS. The limitation will be fixed in a future release.

- `geometry1` is a column of type MDSYS.SDO_GEOMETRY in a feature table T1 and is spatially indexed.
- `geometry2` is a column of type MDSYS.SDO_GEOMETRY in a feature table T2. It may or may not have a spatial index built for it. T2 may also be the same table as T1.
- `params` is a quoted string of keyword value pairs that determines 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 mdsys.sdo_relate(A.shape, mdsys.sdo_geometry(3,NULL,NULL,
                                          mdsys.sdo_elem_info(1,3,3),
                                       mdsys.sdo_ordinates(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= 'WIN_1' AND
        mdsys.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 spatial 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 fence 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:

- `T1.Col` 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 MDSYS.SDO_WITHIN_DISTANCE( A.shape, :theWindow, 'distance=1.35') = 'TRUE';
```

The distance units are based on the geometry coordinate system in use. If your data consists of latitude and longitude pairs, then one distance unit corresponds to the length of one degree. For city, county, state, and even country-wide applications, this is probably acceptable. Unfortunately, one degree of longitude at the equator is much different than one degree at the poles. As mentioned previously, the Spatial product treats the coordinate space as a flat Cartesian grid. It currently does not take projections into account. Projections are left up to the application.

This 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 COASTLINE 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, 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 PARK GIDs and HIGHWAY GIDs 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 mdsys.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 relation 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 mdsys.sdo_relate(A.shape, B.shape,
           'mask = ANYINTERACT querytype = join');
```

5

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

This statement 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
add_index	Specifies the name of the new index table to add. Data type is VARCHAR2.
delete_index	Specifies the name of the index table to delete. You can only delete index tables that were created with the ALTER INDEX add_index statement. The primary index table cannot be deleted with this parameter. To delete the primary index table, use DROP INDEX. Data type is VARCHAR2.
sdo_level	Specifies the desired fixed-size tiling level. Data type is NUMBER.
sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object. Data type is NUMBER.
sdo_maxlevel	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be greater than the sdo_level minimum tiling level. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
sdo_commit_ interval	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	Determines the storage parameters used for altering the spatial index data table. A spatial index data table is a standard Oracle table with a prescribed format. Not all physical_storage_params that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

Keyword	Description
tablespace	Specifies the tablespace in which the index data table is created. This parameter is the same as TABLESPACE in the STORAGE clause of a CREATE TABLE statement.
initial	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
next	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
minextents	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
maxextents	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
btree_initial	Is the same as INITIAL in the STORAGE clause of a CREATE INDEX statement in the case of a standard btree index.
btree_next	Is the same as NEXT in the STORAGE clause of a CREATE INDEX statement in the case of a standard btree index.
btree_pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE INDEX statement in the case of a standard btree index.

Prerequisites

- You must have EXECUTE privileges on the index type and its implementation type.
- The spatial index to be altered is not marked in-progress.

Usage Notes

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

```
ALTER INDEX qtree PARAMETERS ('add_index=HYBRID_INDEX
                               sdo_numtiles=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

This function 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
rebuild_index	Specifies the name of the spatial index table to be rebuilt. Data type is VARCHAR2.
sdo_level	Specifies the desired fixed-size tiling level. Data type is NUMBER.
sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object. Data type is NUMBER.
sdo_maxlevel	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be greater than the sdo_level minimum tiling level. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
sdo_commit_interval	Specifies the number of underlying table rows that are processed between commit intervals for the index data. The default behavior is that a commit of the index data is done only after all rows in the underlying table have been processed. See the Usage Notes for further details. Data type is NUMBER.
layer_gtype	Specifies special processing for point data. If the layer you are indexing is all points, set the parameter to 'POINT' for optimal performance. Data type is VARCHAR2.

PHYSICAL_ STORAGE_ PARAMS	Determines the storage parameters used for rebuiding the spatial index data table. A spatial index data table is a regular Oracle table with a prescribed format. Not all physical_storage_params that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

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

Prerequisites

- You must have EXECUTE privileges on the index type and its implementation type.
- The spatial index to be altered is not marked in-progress.

Usage Notes

- sdo_commit_interval: An ALTER INDEX REBUILD 'rebuild_params' statement will rebuild 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 perform commits of index data after every N rows of the underlying table have been processed. This is done by specifying that sdo_commit_interval = N. The potential complication is that if there is an error during index rebuild and periodic commits 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 builds oldindex with an SDO_LEVEL = 12 and SDO_NUMTILES = NULL.

```
ALTER INDEX oldindex REBUILD PARAMETERS('sdo_level = 12');
```

Related Topics

- CREATE INDEX
- DROP INDEX

ALTER INDEX RENAME TO

Purpose

This statement 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 not marked in-progress.

Usage Notes

None.

Examples

The following example renames the index 'oldindex' to 'newindex'.

```
ALTER INDEX oldindex RENAME TO newindex ;
```

Related Topics

- CREATE INDEX
- DROP INDEX

CREATE INDEX

Purpose

This statement 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

INDEX_PARAMS Determine the type, fixed or hybrid, and characteristics of the spatial index.

Keyword	Description
sdo_level	Specifies the desired fixed-size tiling level. Data type is NUMBER.
sdo_numtiles	Specifies the number of variable-sized tiles to be used in tessellating an object Data type is NUMBER.
sdo_maxlevel	Specifies the maximum tiling level. This parameter determines the tiling resolution. It must be set greater than the sdo_level minimum tiling level. Modifying the default value is not recommended. Data type is NUMBER. Default is 32.
sdo_commit_ interval	Specifies the number of underlying table rows that are processed between commit intervals for the index data. The default behavior is that a commit of the index data is done only after all rows in the underlying table have been processed. See the Usage Notes for further details. Data type is NUMBER.
layer_gtype	Specifies special processing for point data. If the layer you are indexing is all points, set this parameter to 'POINT' for optimal performance. Data type is VARCHAR2.

PHYSICAL_ STORAGE_ PARAMS	Determines the storage parameters used for creating the spatial index data table. A spatial index data table is a regular Oracle table with a prescribed format. Not all physical_storage_params that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.

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

Prerequisites

- All the current SQL CREATE INDEX prerequisites apply.
- You must have EXECUTE privilege on the index type and its implementation type.
- The SDO_GEOM_METADATA table must contain an entry with the dimensions and coordinate boundary information for the table column to be spatially indexed.

Usage Notes

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

- The index_params string must contain valid values for either SDO_LEVEL or both SDO_LEVEL and SDO_NUMTILES.
- DEFAULT VALUES:
  - sdo_numtiles must be supplied with a value greater than or equal to one 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.
- SDO_LEVEL must be greater than zero and less than SDO_MAXLEVEL
- The value passed to SDO_NUMTILES is considered a recommendation. In some cases, this value may be overwritten by the indexing algorithm.
- sdo_commit_interval: 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 the rebuilding of the index, and the spatial index data was periodically committed, then the spatial index will be in an inconsistent state. The only recovery option is to use DROP INDEX 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 are shown in Table 5–2.

Table 5–2 SDO_LEVEL and SDO_NUMTILE 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_NUMTILE Combinations (Cont.)

SDO_LEVEL	SDO_NUMTILES	Type of Spatial Index
>= 1	>= 1	Hybrid indexing with fixed-size and variable-sized tiles. The SDO_LEVEL column defines the fixed tile size. The SDO_NUMTILES column defines the number of variable tiles to generate per geometry.
Not specified.	>= 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.
- 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.

Related Topics

- ALTER INDEX
- DROP INDEX

DROP INDEX

Purpose

This statement deletes a spatial index.

Syntax

DROP INDEX [schema.]index [FORCE]

Keywords and Parameters

FORCE Causes the spatial index to be deleted from the system tables even if the index is marked in-progress or some other error condition occurs.

Prerequisites

You must have EXECUTE privileges on the index type and its implementation type.

Usage Notes

Use DROP INDEX indexname FORCE to clean up after a failure in the CREATE INDEX statement.

Examples

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

Related Topics

- CREATE INDEX

6

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.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

This function calculates the average minimum bounding rectangle (MBR) for all geometries in a column of type SDO_GEOMETRY.

Syntax

SDO_TUNE.AVERAGE_MBR (*table_name, column_name, width, height*)

Keywords and Parameters

table_name	Specifies the name of the geometry table to examine. Data type is VARCHAR2.
column_name	Specifies the name of the geometry object column to examine. Data type is VARCHAR2.
width	Returns the width of the average MBR. Data type is OUT NUMBER.
height	Returns the height of the average MBR. Data type is OUT NUMBER.

Returns

The function returns the width and height of the average MBR for all geometries in a geometry table.

Data types for height and width are NUMBER.

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 geometry table.

SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE

Purpose

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

Syntax

SDO_TUNE.ESTIMATE_INDEX_PERFORMANCE (*table_name, column_name, sample_ratio, tiling_level, num_tiles, window_object, tiling_time, filter_time, query_time*)

Keywords and Parameters

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

Returns

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

The 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

This function estimates the appropriate sdo_level to use when indexing with hybrid or fixed-size tiles.

Syntax

MDSYS.SDO_TUNE.ESTIMATE_TILING_LEVEL *(table_name, column_name, maxtiles, type_of_estimate)*

Keywords and Parameters

table_name — Specifies the name of the geometry table.
Data type is VARCHAR2.

column_name — Specifies the name of the geometry column to examine.
Data type is VARCHAR2.

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

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

- LAYER_EXTENT -- Use the rectangle defined by your coordinate system.
- ALL_GID_EXTENT -- Use the minimum bounding rectangle that encompasses all the geometric objects in the column. This estimate is recommended for most applications with a maxtiles of 10,000.
- AVG_GID_EXTENT -- Use a rectangle representing the average size of the individual geometric objects within the column. This option performs the most analysis of the three types.

Data type is VARCHAR2.

Returns

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.

Usage Notes

None.

Related Topics

- MDSYS.SDO_TUNE.EXTENT_OF
- Section A.2.2, "Understanding the Tiling Level"
- Section A.2.4, "Visualizing the Spatial Index (Drawing Tiles)"

SDO_TUNE.ESTIMATE_TILING_TIME

Purpose

This function provides the estimated time to tessellate a column of type SDO_GEOMETRY.

Syntax

SDO_TUNE.ESTIMATE_TILING_TIME (*table_name, column_name, sample_ratio, tiling_level, num_tiles*)

Keywords and Parameters

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

Returns

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

Data type is NUMBER.

Usage Notes

None.

SDO_TUNE.EXTENT_OF

Purpose

This function determines the extent of all geometries in a column of type SDO_GEOMETRY.

Syntax

SDO_TUNE.EXTENT_OF *(table_name, column_name)*

Keywords and Parameters

table_name	Specifies the name of the geometry table. Data type is VARCHAR2.
column_name	Specifies the name of the geometry column to examine. Data type is VARCHAR2.

Returns

This function returns a geometry object representing the minimum bounding rectangle for all geometric data in a column. The function returns NULL if the data is inconsistent.

Usage Notes

None.

Related Topics

SDO_TUNE.ESTIMATE_TILING_LEVEL() function

SDO_TUNE.HISTOGRAM_ANALYSIS

Purpose

This procedure generates statistical histograms based on columns of type SDO_GEOMETRY.

Syntax

SDO_TUNE.HISTOGRAM_ANALYSIS (*table_name, column_name, result_table, type_of_histogram, max_value, intervals*)

Keywords and Parameters

table_name	Specifies the name of the geometry table to examine. Data type is VARCHAR2.
column_name	Specifies the name of the geometry column to examine. Data type is VARCHAR2.
result_table	Specifies the name of the result table where the histogram will be stored. Data type is VARCHAR2.
type_of_histogram	Specifies one of three types of histograms: • TILES_VS_LEVEL (default) • GEOMS_VS_AREA • GEOMS_VS_VERTICES Data type is VARCHAR2.
max_value	Specifies the upper limit of the histogram. Data type is NUMBER.
intervals	Specifies the number of intervals between 0 and max_value. Data type is INTEGER.

Returns

The procedure populates the result table with statistical histograms for a geometry table.

Usage Notes

- Prior to calling this procedure, create the result table as follows:

```
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. This histogram is used to evaluate the spatial index that is already built on the data set 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_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

This procedure provides the number of geometries of each type stored in a column of type SDO_GEOMETRY.

Syntax

SDO_TUNE.MIX_INFO (*table_name, column_name*)

Keywords and Parameters

table_name	Specifies the name of the geometry table to examine. Data type is VARCHAR2.
column_name	Specifies the name of the geometry column to examine. Data type is VARCHAR2.

Returns

The procedure calculates geometry type information for the table. It calculates the number of geometries of different types, as well as the percentages of points, line strings, polygons, and complex geometries.

Usage Notes

None.

7

Geometry Functions for Object-Relational Model

This chapter contains descriptions of the geometry functions shown in Table 7–1.

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

Function	Description
SDO_GEOM.AREA	Computes the area of a two-dimensional polygon.
SDO_GEOM.LENGTH	Computes the length or perimeter of a geometry.
SDO_GEOM.RELATE	Determines how two objects interact.
SDO_GEOM.SDO_BUFFER	Generates a buffer polygon around a geometry.
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 combination of two geometries.
SDO_GEOM.SDO_POLY_XOR	Generates a polygon representing the symmetric difference between two geometries.
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.

SDO_GEOM.AREA

Purpose

This function computes the area of a two-dimensional polygon.

Syntax

SDO_GEOM.AREA (*geometry, dim_array*)

Keywords and Parameters

geometry	Specifies the geometry object to analyze. Data type is MDSYS.SDO_GEOMETRY.
dim_array	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

This function returns the area of a polygon.

Data type is NUMBER.

Usage Notes

This function works with any polygon, including polygons with holes.

Related Topics

None.

SDO_GEOM.LENGTH

Purpose

This function computes the length or perimeter of a geometry.

Syntax

SDO_GEOM.LENGTH (*geometry, dim_array*)

Keywords and Parameters

geometry	Specifies the geometry object to analyze. Data type is MDSYS.SDO_GEOMETRY.
dim_array	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

This function returns the length or perimeter of an object.

Data type is NUMBER.

Usage Notes

If the input polygon contains one or more holes, this function calculates the perimeters of the exterior boundary and all of the holes. It returns the sum of all the perimeters.

Related Topics

None.

SDO_GEOM.RELATE

Purpose

This function examines two geometry objects to determine their spatial relationship.

Syntax

SDO_GEOM.RELATE (*geometry1, dim_array1, mask, geometry2, dim_array2*)

Keywords and Parameters

geometry1, *geometry2*	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, *dim_array2*	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.
mask	Specifies a list of relationships to check. See the list of keywords in the Usage Notes.

Returns

The `MDSYS.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 procedure returns FALSE.
2. If you pass the DETERMINE keyword in the 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.
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

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 a logical Boolean operator OR. For example, 'INSIDE + TOUCH' returns either 'INSIDE', 'TOUCH', or 'FALSE' depending on the outcome of the test.

Related Topics

None.

SDO_GEOM.SDO_BUFFER

Purpose

This function generates a buffer polygon around a geometry object.

Syntax

SDO_GEOM.SDO_BUFFER (*geometry, dim_array, distance*)

Keywords and Parameters

geometry	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.
distance	Specifies the Euclidean distance value. Data type is NUMBER.

Returns

This function returns a geometry object representing the buffer polygon.

Data type is MDSYS.SDO_GEOMETRY.

Usage Notes

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.

Related Topics

- SDO_TUNE.EXTENT_OF
- SDO_GEOM.SDO_POLY_DIFFERENCE
- SDO_GEOM.SDO_POLY_INTERSECTION
- SDO_GEOM.SDO_POLY_UNION
- SDO_GEOM.SDO_POLY_XOR

SDO_GEOM.SDO_POLY_DIFFERENCE

Purpose

This function computes the difference (A minus B) of two polygon objects.

Syntax

SDO_GEOM.SDO_POLY_DIFFERENCE (*geometry1, dim_array1, geometry2, dim_array2*)

Keywords and Parameters

geometry1, geometry2	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, dim_array2	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

This function returns a geometry object representing the difference of two polygon objects.

Data type is MDSYS.SDO_GEOMETRY.

Usage Notes

None.

Related Topics

- SDO_GEOM.SDO_POLY_BUFFER
- SDO_GEOM.SDO_POLY_INTERSECTION
- SDO_GEOM.SDO_POLY_UNION
- SDO_GEOM.SDO_POLY_XOR

SDO_GEOM.SDO_POLY_INTERSECTION

Purpose

This function computes the intersection of two polygon objects.

Syntax

SDO_GEOM.SDO_POLY_INTERSECTION (*geometry1, dim_array1, geometry2, dim_array2*)

Keywords and Parameters

geometry1, *geometry2*	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, *dim_array2*	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

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

Data type is MDSYS.SDO_GEOMETRY.

Usage Notes

None.

Related Topics

- SDO_GEOM.SDO_POLY_BUFFER
- SDO_GEOM.SDO_POLY_DIFFERENCE
- SDO_GEOM.SDO_POLY_UNION
- SDO_GEOM.SDO_POLY_XOR

SDO_GEOM.SDO_POLY_UNION

Purpose

This function computes the union of two polygon objects.

Syntax

SDO_GEOM.SDO_POLY_UNION (*geometry1, dim_array1, geometry2, dim_array2*)

Keywords and Parameters

geometry1, geometry2	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, dim_array2	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

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

Data type is MDSYS.SDO_GEOMETRY.

Usage Notes

None.

Related Topics

- SDO_GEOM.SDO_POLY_BUFFER
- SDO_GEOM.SDO_POLY_DIFFERENCE
- SDO_GEOM.SDO_POLY_INTERSECTION
- SDO_GEOM.SDO_POLY_XOR

SDO_GEOM.SDO_POLY_XOR

Purpose

This function computes the symmetric difference of two polygon objects.

Syntax

SDO_GEOM.SDO_POLY_XOR (*geometry1*, *dim_array1*, *geometry2*, *dim_array2*)

Keywords and Parameters

geometry1, *geometry2*	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, *dim_array2*	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

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

Data type is MDSYS.SDO_GEOMETRY.

Usage Notes

None.

Related Topics

- SDO_GEOM.SDO_POLY_BUFFER
- SDO_GEOM.SDO_POLY_DIFFERENCE
- SDO_GEOM.SDO_POLY_INTERSECTION
- SDO_GEOM.SDO_POLY_UNION

SDO_GEOM.VALIDATE_GEOMETRY

Purpose

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

Syntax

SDO_GEOM.VALIDATE_GEOMETRY *(geometry, dim_array*)

Keywords and Parameters

geometry	Specifies the geometry object to test. Data type is MDSYS.SDO_GEOMETRY.
dim_array	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.

Returns

This function returns:

- TRUE if the geometry is valid.
- FALSE if the geometry fails for some unknown reason.
- An Oracle error message number based on the specific reason the geometry in invalid.

Data type is VARCHAR2.

Usage Notes

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.

Example

The following example tests a geometry stored in a table my_geometry_table with one column called "geometry" and another column called "my_id," which is the primary key:

```
SELECT mdsys.sdo_geom.validate_geometry (a.geometry, b.diminfo)
  FROM abi_81_hy a,
       sdo_geom_metadata b
  WHERE a.gid = 1
    AND b.table_name = 'MY_GEOMETRY_TABLE'
    AND b.column_name = 'GEOMETRY';
```

Related Topics

None.

SDO_GEOM.VALIDATE_LAYER

Purpose

This function examines a geometry column to determine if the stored geometries follow the defined rules for geometry objects.

Syntax

SDO_GEOM.VALIDATE_LAYER *(table_name, column_name, pkey_column, result_table_name*)

Keywords and Parameters

table_name	Specifies the name of the geometry object table. Data type is VARCHAR2.
column_name	Specifies the name of the geometry object column to examine. Data type is VARCHAR2.
pkey_column	Specifies the primary key column. This can be the rowid, or any other primary key. Data type is VARCHAR2.
result_table_name	Specifies the name of the result table. Data type is VARCHAR2.

Returns

This function populates the result table with validation results.

Usage Notes

- An empty result table must be created prior to calling this function. The format of the result table is: (sdo_gid number, result varchar2(10)).
- This function returns TRUE in the result table if a geometry is valid. If it is invalid, the result column for that geometry (the pkey_column corresponds to the SDO_GID) contains an Oracle error message number. You can then look up this error message to determine the cause of the failure.
- 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.

Related Topics

None.

SDO_GEOM.WITHIN_DISTANCE

Purpose

This function determines if two spatial objects are within some specified Euclidean distance from each other.

Syntax

SDO_GEOM.WITHIN_DISTANCE *(geometry1, dim_array1, distance, geometry2, dim_array2*)

Keywords and Parameters

geometry1, *geometry2*	Specifies the geometry objects to compare. Data type is MDSYS.SDO_GEOMETRY.
dim_array1, *dim_array2*	Specifies the dimensional information array, usually selected from the SDO_GEOM_METADATA table. Data type is MDSYS.SDO_DIM_ARRAY.
distance	Specifies the Euclidean distance value. Data type is NUMBER.

Returns

This function returns TRUE for object pairs that are within the specified distance, and FALSE otherwise.

Usage Notes

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.

Related Topics

None.

8

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.TO_734	Migrate tables from Spatial Data Option 7.3.3 to Spatial Data Option 7.3.4 and Spatial Cartridge 8.0.4 format.
SDO_MIGRATE.TO_81X	Migrate tables from Spatial Data Option 7.3.4 or Spatial Cartridge 8.0.4 to Oracle8*i* Spatial.
SDO_MIGRATE.OGIS_METADATA_FROM	Generate a temporary table used when migrating OGIS metadata tables.
SDO_MIGRATE.OGIS_METADATA_TO	Read a temporary table used when migrating OGIS metadata tables.

SDO_MIGRATE.TO_734

Purpose

This procedure migrates data from a previous release of Spatial Data Option to release 7.3.4.

Syntax

SDO_MIGRATE.TO_734 (*schema_name, layername, tessellation, param*)

Keywords and Parameters

schema_name	Specifies the schema name of the owner of the layer. Data type is VARCHAR2.
layername	Specifies the name of the layer. Data type is VARCHAR2.
tessellation	Specifies the type of indexing used, either FIXED or VARIABLE. Data type is VARCHAR2.
param	Specifies the SDO_LEVEL or SDO_NUMTILES value. Data type is NUMBER.

Usage Notes

The value of the param argument is evaluated relative to the value of the tessellation argument. If tessellation is FIXED, then param is the SDO_LEVEL value. If tessellation is VARIABLE, then param is the SDO_NUMTILES value.

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

This procedure migrates data from a previous release of Spatial Cartridge or Spatial Data Option to Oracle8*i* Spatial (release 8.1.5.)

Syntax

SDO_MIGRATE.TO_81X (*old_layername, new_tablename geom_column, GID_column, layer_gtype, update_flag*)

Keywords and Parameters

old_layername	Specifies the name of the old layer. Data type is VARCHAR2.
new_tablename	Specifies the name of the new table to which you are migrating the data. Data type is VARCHAR2.
GID_column	Specifies the name of the column in which to store the GID from the old table. Data type is NUMBER.
geom_column	Specifies the column name in the new table where the geometry object will be inserted. Data type is SDO_GEOMETRY.
layer_gtype	If the layer you are migrating is composed soley of point data, set this parameter to 'POINT' for optimal performance. Data type is VARCHAR2. Default is 'NOTPOINT'.
update_flag	Specifies special processing for point data. If you are migrating the layer into an existing populated attribute table, set this parameter to 'UPDATE'. Data type is VARCHAR2. Default is '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.
- There is no implicit commit when using this procedure. You must commit the migration explicitly.
- If any of the migration steps fails, nothing is migrated for the layer.
- The old_layername is the underlying layername, without the _SDOGEOM suffix.
- An SDO_GEOM_METADATA table is required in the user's schema.
- The old SDO_GID is stored in GID_column.

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

This procedure is 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.

Syntax

SDO_MIGRATE.OGIS_METADATA_FROM

Keywords and 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

This procedure is 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.

Syntax

SDO_MIGRATE.OGIS_METADATA_TO

Keywords and 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.

9

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_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

This operator 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
querytype	Specifies valid query types: JOIN or WINDOW. This is a required parameter. WINDOW implies that geometry2 should be considered a dynamic (transient) area-of-interest. Use WINDOW when you want to compare a single geometry (geometry2) to all the geometries in a column (geometry1). JOIN implies that the second argument 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.
idxtab1	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for geometry1.
idxtab2	Not supported in this release. Specifies the name of the index table (if there are multiple spatial indexes) for geometry2. Only valid for 'querytype = JOIN.'
layer_gtype	Specifies special processing for point data. If the columns you are comparing are comprised soley of 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 the 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 is transient, it is indexed in memory.

  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.

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(3,NULL,NULL,
                                          mdsys.sdo_elem_info(1,3,3),
                                          mdsys.sdo_ordinates(x1,y1,x2,y2)),
                     'querytype=WINDOW') = 'TRUE';
```

Related Topics

SDO_RELATE

SDO_RELATE

Purpose

This operator uses the spatial index to identify either the set of 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

geometry1 Specifies a geometry column in a table. The column must be spatially indexed.
Data type is MDSYS.SDO_GEOMETRY.

geometry2 Specifies either a geometry from a table or a transient instance of a geometry. (Specified using a bind variable or SDO_GEOMETRY constructor.)
Data type is MDSYS.SDO_GEOMETRY.

PARAMS Determines the behavior of the operator.
Data type is VARCHAR2.

Keyword	Description
mask	Specifies the topological relation of interest. This is a required parameter. Valid values are one or more of {TOUCH, OVERLAP, EQUAL, INSIDE, COVEREDBY, CONTAINS, COVERS, ANYINTERACT} in the 9-intersection pattern. Multiple masks are combined with a the logical Boolean operator OR as follows: 'mask=(inside+touch)'. See Section 1.7 for an explanation of the 9-intersection relationship pattern.
querytype	Valid query types are: JOIN or WINDOW. This is a required parameter. WINDOW implies that geometry2 should be considered a dynamic (transient) area-of-interest. Use WINDOW when you want to compare a single geometry (geometry2) to all the geometries in a column (geometry1). JOIN implies that the second argument 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.

idxtab1	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for geometry1.
idxtab2	Not supported in this release. Specifies the name of the index, if there are multiple spatial indexes, for geometry2. Only valid for 'querytype = JOIN'.
layer_gtype	Specifies special processing for point data. If the columns you are comparing are composed soley of point data, set this parameter to 'POINT' for optimal performance. Data type is VARCHAR2. Default is 'NOTPOINT'.

Returns

The expression SDO_RELATE(geometry1,geometry2, 'mask = <some_mask_val> querytype = <some_querytype>') = 'TRUE' evaluates to TRUE for object pairs that have the topological relationship specified by <some_mask_val> and FALSE otherwise.

Usage Notes

- The operator must always be used in a WHERE clause, and the condition that includes the operator should be an expression of the form SDO_RELATE(arg1, arg2, 'mask = <some_mask_val> querytype = <some_querytype>') = 'TRUE'.
- If 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 is transient, it is indexed in memory.

  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 re-use its index. Performance will degrade if geometry2 is not indexed with the same sdo_level parameter as geometry1.
- Unlike 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(3,NULL,NULL,
                                          mdsys.sdo_elem_info(1,3,3),
                                          mdsys.sdo_ordinates(x1,y1,x2,y2)),
                   'mask=ANYINTERACT querytype=WINDOW') = 'TRUE';
```

Related Topics

- SDO_FILTER
- SDO_WITHIN_DISTANCE
- SDO_GEOM.RELATE( )

SDO_WITHIN_DISTANCE

Purpose

This operator 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 or point-of-interest.)

Syntax

SDO_WITHIN_DISTANCE(T.column, aGeom, params) ;

Keywords and Parameters

T.column	Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
aGeom	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
distance	Specifies the Euclidean distance value. This is a required parameter. Data type is NUMBER.
idxtab1	Not supported in this release. Specifies the name of the index if there are multiple spatial index tables for geometry1.
querytype	Set 'querytype=FILTER' to perform only a primary filter operation. If querytype is not specified, both primary and secondary filter operations are performed (default). Data type is VARCHAR2.
layer_gtype	Specifies special processing for point data. If the columns you are comparing are composed soley of point data, set this parameter to 'POINT' for optimal performance. 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(3,NULL,NULL,
                                    mdsys.sdo_elem_info(1,3,3),
                                    mdsys.sdo_ordinates(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_RELATE
- SDO_FILTER

Part II

Relational Model

Oracle8*i* Spatial supports two models for representing geometries: relational and object-relational. The two models are mutually exclusive. See Appendix A 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"

10

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 new 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 A.1, 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.3.2, "crlayer.sql Script" for details.

Table 10–1 through Table 10–4 describe the schema of a Spatial layer.

Table 10–1 <layername>_SDOLAYER

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

[1] SDO_MAXLEVEL is an optional column.

[2] SDO_COORDSYS is an optional column.

Table 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 [1]	SDO_GROUPCODE [2]	SDO_META
<number>	<raw>	<raw>	<raw>	<raw>

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

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

The columns of each table are defined as follows:

<layername>_SDOLAYER:

- **SDO_ORDCNT -** The SDO_ORDCNT column is the total number of ordinates per row in the <layername>_SDOGEOM table. That is, the total number of data value columns, and not the number of points or coordinates. SDO_ORDCNT should not be multiplied by the total number of dimensions per coordinate as it is already a total.
- **SDO_LEVEL** - The SDO_LEVEL column stores the number of times the tiles that interact with a geometry should be decomposed. It is the termination criteria 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 criteria for the variable component of hybrid tiling.
- **SDO_COORDSYS -** The SDO_COORDSYS column is optional; where 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 zeroes 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 **SEQ**uence number.
- **SDO_ETYPE** - The SDO_ETYPE column is the geometric primitive type of the element. For this release of Spatial, the valid values are SDO_GEOM.POINT_TYPE, SDO_GEOM.LINESTRING_TYPE, or SDO_GEOM.POLYGON_TYPE (ETYPE values 1, 2, and 3, respectively). The SDO_ETYPE values 4 and 5, supported in the object-relational schema, are not supported. Setting the ETYPE to zero indicates that this element should be ignored. See Section A.2.8 for information on ETYPE=0.
- **SDO_SEQ** - The SDO_SEQ column records the order (the **SEQ**uence number) of each row of data making up the element.
- **SDO_X1** - X value of the first coordinate.
- **SDO_Y1** - Y value of the first coordinate.
- **SDO_Xn** - X value of the Nth coordinate.

- **SDO_Yn** - Y value of the Nth coordinate.

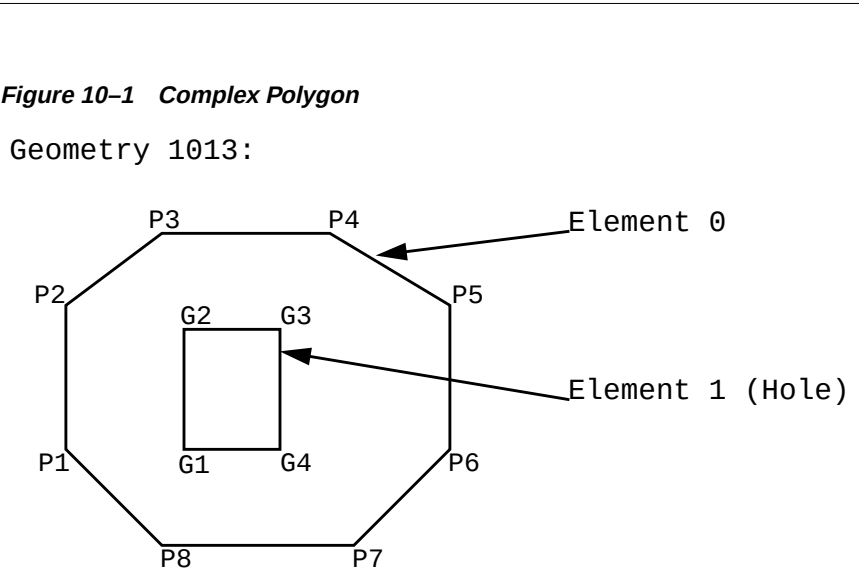
<layername>_SDOINDEX:

- **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.2.4 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



<layername>_SDOLAYER

SDO_ORDCNT (number)
4

<layername>_SDODIM

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

<layername>_SDOGEOM

SDO_GID (number)	SDO_ESEQ (number)	SDO_ETYPE (number)	SDO_SEQ (number)	SDO_X1 (number)	SDO_Y1 (number)	SDO_X2 (number)	SDO_Y2 (number)
1013	0	3	0	P1(X)	P1(Y)	P2(X)	P2(Y)
1013	0	3	1	P2(X)	P2(Y)	P3(X)	P3(Y)
1013	0	3	2	P3(X)	P3(Y)	P4(X)	P4(Y)
1013	0	3	3	P4(X)	P4(Y)	P5(X)	P5(Y)
1013	0	3	4	P5(X)	P5(Y)	P6(X)	P6(Y)

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

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

The data in the <layername>_SDOINDEX table is described in further detail Section 1.6, “Indexing Methods”. 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_CODEs. Each SDO_CODE value may be associated with one or more SDO_GIDs.

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

11

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[1].

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
```

[1] 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
```

Note that table ROADS_SDOGEOM exists in the schema before attempting the load.

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 Oracle8*i* 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 SDO_ ADMIN.UPDATE_INDEX( ) 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, "Administrative Functions and Procedures" 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 inTable 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.

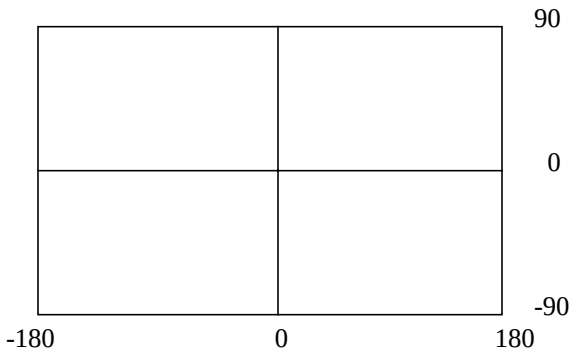
The domain used for indexing is defined by the upper and lower boundaries of each dimension stored in the <layername>_SDODIM table. A typical domain in a GIS application could be -90 to 90 degrees for latitude, and -180 to 180 degrees for longitude[1], as represented in Figure 11–1.

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

Figure 11–1 Sample GIS Domain

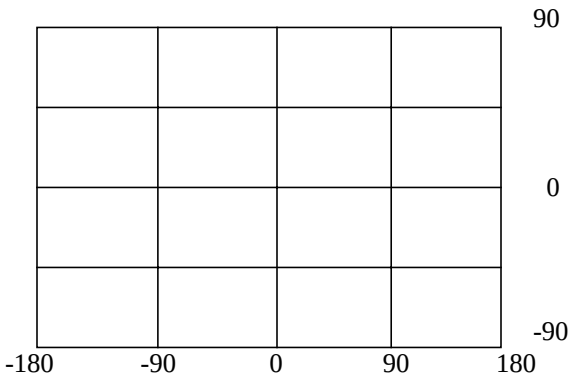
If the SDO_LEVEL column is set to 1, then the tiles created by the indexing mechanism are the same size as tiles at the first level of tessellation. Each tile would be 180 degrees by 90 degrees as shown in Figure 11–2.

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:

length = (upper_bound - lower_bound) / 2 ^ sdo_level

The length refers to the length of the tile along the specified dimension. Applying this formula to the tiling shown in Figure 11–3 yields the following sizes:

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

Thus, at level 2 the tiles are 90x45 degrees in size. As the number of levels increases, the tiles become smaller and smaller. Smaller tiles provide a more precise fit of the tiles over the geometry being indexed. However, because the number of tiles generated is unbounded, you must take into account the performance implications of using higher levels. The `SDO_TUNE.ESTIMATE_TILING_LEVEL()` function can be used to determine an appropriate level for indexing with fixed-size tiles. See Chapter 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. 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.3.1, "cr_spatial_index.sql Script" 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 the 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 <layername>_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. 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.3.1, "cr_spatial_index.sql Script" for more information.

12

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. 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.

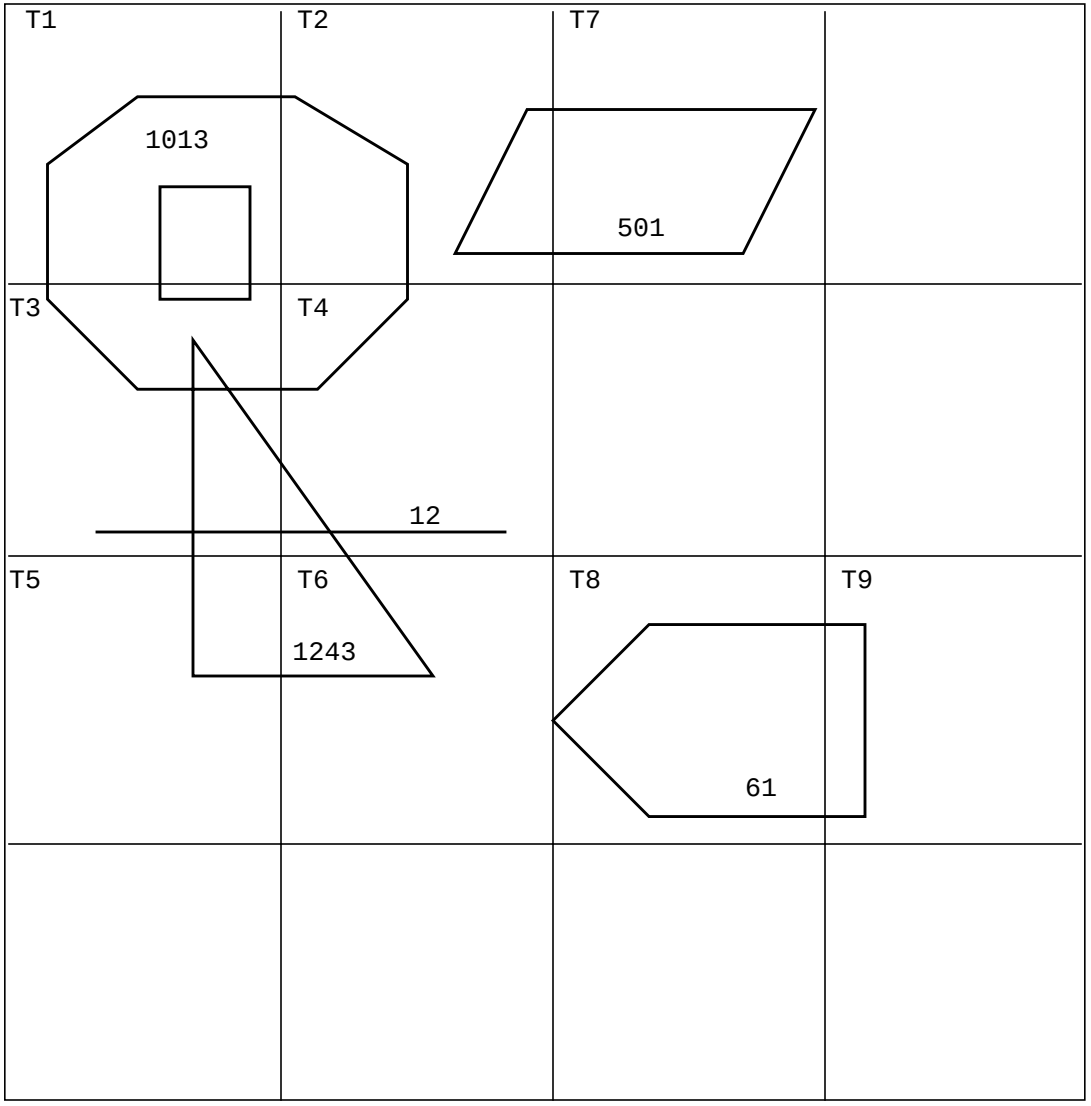
- The primary filter permits fast selection of a small number of candidate records to pass along to the secondary filter. The primary filter uses approximations in order to reduce computational complexity and is considered a lower cost filter.
- The secondary filter applies exact computational geometry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

SDO_ ORDCNT (number)	SDO_LEVEL (number)	SDO_NUMTILES (number)
4	2	NULL

Table 12–2 <layername>_SDOGEOM

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

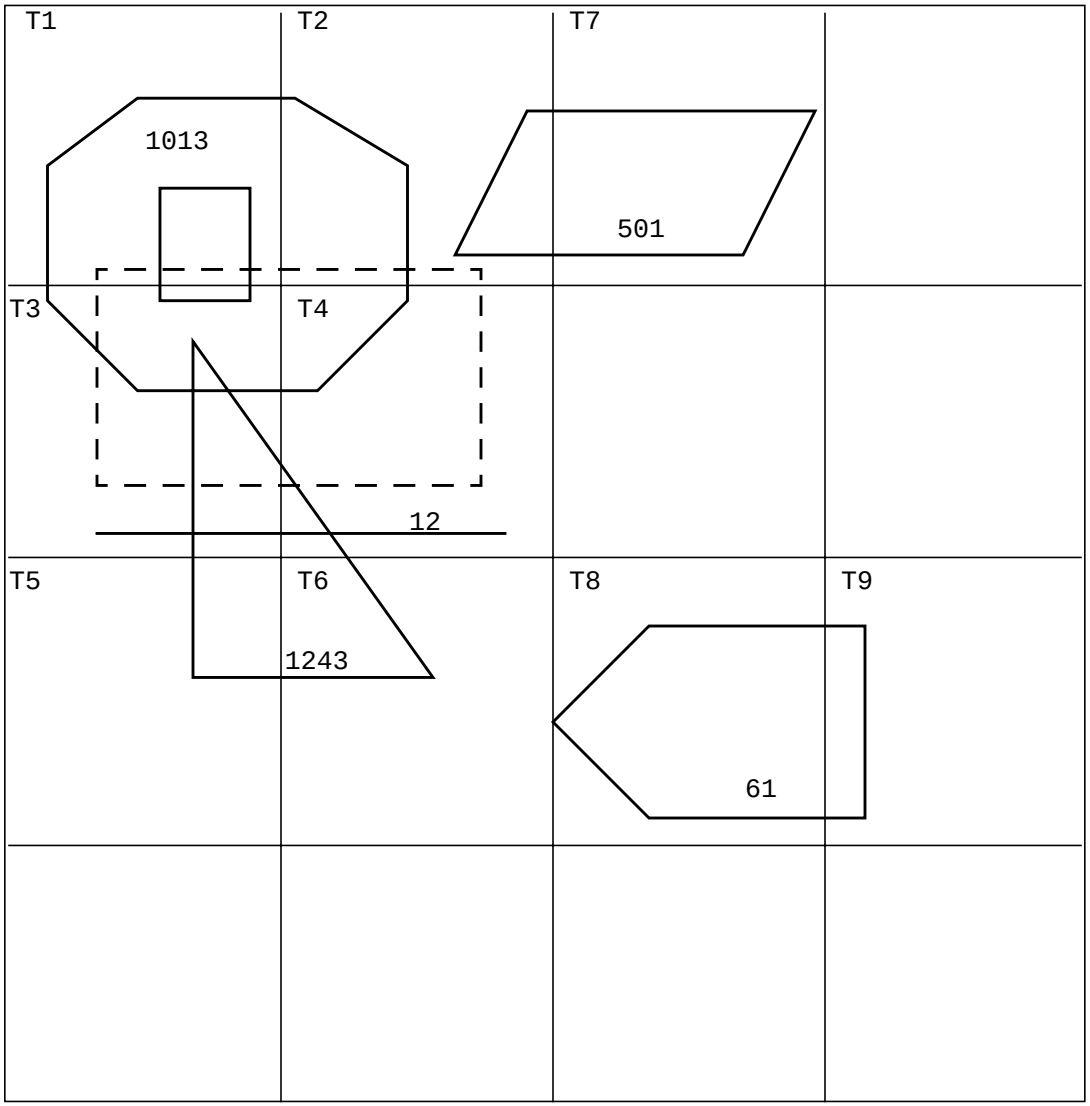
Table 12–3 <layername>_SDOINDEX

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

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 insert 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, "Geometry Functions and Procedures", for details on using this function.

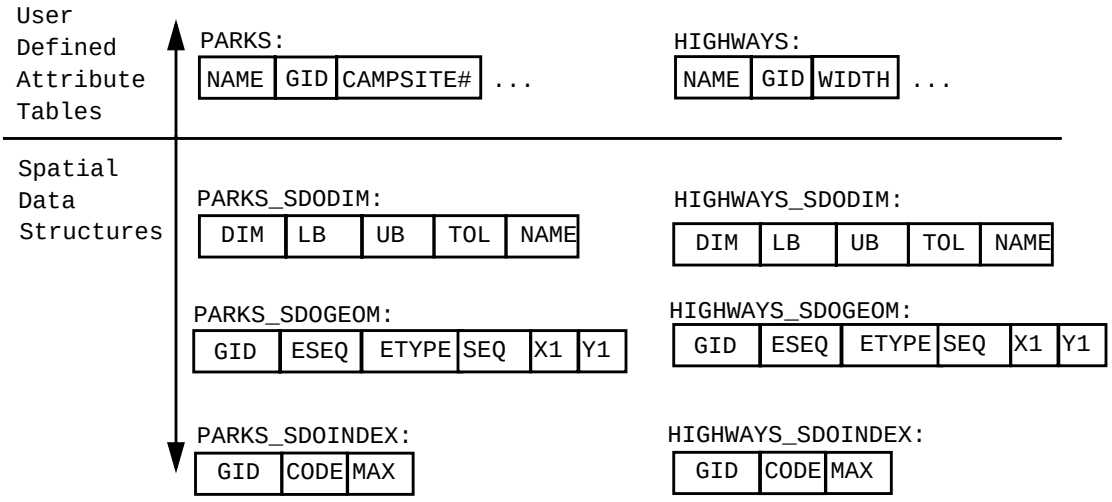
12.4 Spatial Join

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';
```

Primary Filter

Secondary Filter

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

13

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 the Spatial option.
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.)

Procedure or Function	Description
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

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

Syntax

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. 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 is executed after the procedure is called.
- `SDO_ADMIN.POPULATE_INDEX()` operates as a single transaction. To reduce the amount of rollback required to execute this procedure, you can write a routine that loops and calls `SDO_ADMIN.UPDATE_INDEX()` repeatedly. See Section A.3.1, "cr_spatial_index.sql Script" 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

This procedure is 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.

Syntax

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

Keywords and Parameters

layername	Specifies the name of the data set layer. The layer name is used to construct the name of the geometry and spatial index tables. Data type is VARCHAR2.
tile_size	Specifies the number of tessellations required to achieve the desired tile size (see the Usage Notes). Data type is INTEGER.
synch_flag	Specifies whether to tessellate every geometric object in the geometry table, or only those that do not have corresponding entries in the spatial index table. If TRUE, only those geometric objects in the geometry table that do not have any corresponding tiles in the spatial index table are tessellated. If FALSE, all the geometric objects in the geometry table are tessellated and new tiles are simply added to the spatial index table. Default value is TRUE. Data type is BOOLEAN.
sdo_tile_flag	For internal use only. Not supported in this release. 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, "Spatial Indexing with Fixed-Size Tiles".
- 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 is executed after the procedure is called.
- `SDO_ADMIN.POPULATE_INDEX_FIXED()` operates as a single transaction. To reduce the amount of rollback required to execute this procedure, you can write a routine that loops and calls `SDO_ADMIN.UPDATE_INDEX_FIXED()` repeatedly. See Section A.3.1, "cr_spatial_index.sql Script" 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

This procedure 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.

Syntax

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

Keywords and Parameters

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

sdo_tile_flag Specifies whether or not to generate the SDO_TILE column.
Default value is FALSE.
Data type is BOOLEAN.

commit_count Specifies how many points to index before updating and committing the data.
Default value is 50.
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

This function determines the size that the SDO_MAXCODE column should be in the <layername>_SDOINDEX table.

Syntax

SDO_ADMIN.SDO_CODE_SIZE (*layername*)

Keywords and Parameters

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

Returns

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

Usage Notes

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

Always declare the SDO_CODE column to raw(255).

Related Topics

None.

SDO_ADMIN.SDO_VERSION

Purpose

This function returns the current installed version of Spatial.

Syntax

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 by this procedure:

8.0.5.0.0
8.1.0.0.0
8.1.3.0.0
8.1.5.0.0

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

This procedure 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.

Syntax

SDO_ADMIN.UPDATE_INDEX (*layername, GID*)

Keywords and Parameters

layername Specifies the name of the data set layer. The layer name is used to construct the name of the geometry table.
Data type is VARCHAR2.

GID Specifies the geometric object identifier.
Data type is NUMBER.

Usage Notes

Considert 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 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

This procedure is 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.

Syntax

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

Keywords and Parameters

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

This procedure checks for the existence of the geometry and spatial index tables.

Syntax

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

Keywords and Parameters

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

maxtiles For internal use only. Not supported in this release.

Usage Notes

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

Example 13–6 verifies the LAYER1 data set layer.

Example 13–6 Verify a Layer

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

Related Topics

None.

14

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

This function calculates the average minimum bounding rectangle (MBR) for all geometries in a layer.

Syntax

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

Keywords and Parameters

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

Returns

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

Data types for height and width are NUMBER.

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

This function estimates the spatial index performance such as query selectivity and window query time for a layer.

Syntax

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

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

Returns

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

The 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

This function estimates the appropriate tiling level to use when indexing with fixed-size tiles.

Syntax

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

Keywords and Parameters

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

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

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

- LAYER_EXTENT -- Use the rectangle defined by your coordinate system.
- ALL_GID_EXTENT -- Use the minimum bounding rectangle that encompasses all the geometric objects within the layer. Recommended for most applications with a maxtiles of 10,000.
- AVG_GID_EXTENT -- Use a rectangle representing the average size of the individual geometries within the layer. This option performs the most extensive analysis of the three types.

Returns

The 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 Lat/Lon 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 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.2.2, "Understanding the Tiling Level"
- Section A.2.4, "Visualizing the Spatial Index (Drawing Tiles)"

SDO_TUNE.ESTIMATE_TILING_TIME

Purpose

This function returns the estimated time to tessellate a layer.

Syntax

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

Keywords and Parameters

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

Returns

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

Data type is NUMBER.

Usage Notes

None.

SDO_TUNE.EXTENT_OF

Purpose

This function determines the extent of all geometries in a layer.

Syntax

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

Keywords and Parameters

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

Returns

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

This procedure generates statistical histograms based on a layer.

Syntax

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

Keywords and Parameters

layername	Specifies the name of the spatial data set layer to examine. Data type is VARCHAR2.
result_table	Specifies the name of the result table where the histogram will be stored. Data type is VARCHAR2.
type_of_histogram	Specifies one of three types of histograms: • TILES_VS_LEVEL (default) • GEOMS_VS_AREA • GEOMS_VS_VERTICES Data type is VARCHAR2.
max_value	Specifies the upper limit of the histogram. Data type is NUMBER.
intervals	Specifies the number of intervals between 0 and max_value. Data type is INTEGER.

Returns

The 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. 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_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

This function provides the number of geometries of each type stored in the layer.

Syntax

SDO_TUNE.MIX_INFO (*layername*)

Keywords and Parameters

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

Returns

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

Usage Notes

None.

15

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
SDO_GEOM.RELATE	Determines how two objects interact.
SDO_GEOM.VALIDATE_GEOM	Determines if a geometry is valid.
SDO_GEOM.VALIDATE_LAYER	Determines if all geometries in a layer are valid.

SDO_GEOM.RELATE

Purpose

This function examines two geometry objects to determine their spatial relationship. It 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

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

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

Returns

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

1. If you pass a mask listing one or more relationships, the function returns the name of the relationship if it is true for the pair of geometries. If all 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. DETERMINE 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 either 'INSIDE', 'TOUCH', or 'FALSE' depending on the outcome of the test.

Related Topics

None.

SDO_GEOM.VALIDATE_GEOM

Purpose

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

Syntax

SDO_GEOM.VALIDATE_GEOM *(layername,SDO_GID*)

Keywords and Parameters

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

Returns

This function returns one of the following:

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

The data type is VARCHAR2.

Usage Notes

This function checks for the following:

- Polygons 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 is the first point on the next row

Related Topics

None.

SDO_GEOM.VALIDATE_LAYER

Purpose

This function 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

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

Returns

This function fills the result table with validation results.

Usage Notes

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

This function checks for the following:

- Polygons 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 is the first point on the next row

Related Topics

None.

16

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

This function 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

comp_name	Specifies the name of the user who compiled this package. This user must have appropriate privileges to read and write into the database. Data type is VARCHAR2.
layername	Specifies the name of the window layer into which the coordinates will be inserted. Data type is VARCHAR2.
SDO_ETYPE	Specifies the type of geometry element. Data type is INTEGER, corresponding to the following constants: 1 or SDO_GEOM.POINT_TYPE 2 or SDO_GEOM.LINESTRING_TYPE 3 or SDO_GEOM.POLYGON_TYPE
SDO_NUMTILES	Value must be NULL for Spatial release 8.0.4 and later. Data type is NUMBER.
X ordinateN, *Y ordinateN*	Specifies the X and Y values of a vertex (coordinate pair) in a geometry. Up to 125 pairs may be added in a single call. Data type is NUMBER.

Returns

This function returns the SDO_GID of the new geometry. 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() function

SDO_WINDOW.BUILD_WINDOW_FIXED

Purpose

This function 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

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

Returns

This function returns the SDO_GID of the new geometry. Data type is NUMBER.

Usage Notes

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

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

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

Related Topics

None.

SDO_WINDOW.CLEAN_WINDOW

Purpose

This procedure removes the four tables created in the layer for the query window.

Syntax

SDO_WINDOW.CLEAN_WINDOW (*layername*);

Keywords and Parameters

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

Usage Notes

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

Related Topics

SDO_WINDOW.CLEANUP_GID

SDO_WINDOW.CLEANUP_GID

Purpose

This procedure removes the query window from the layer tables.

Syntax

SDO_WINDOW.CLEANUP_GID (*layername, SDO_GID*);

Keywords and Parameters

layername	Specifies the name of the window layer associated with the query window. Data type is VARCHAR2.
SDO_GID	Specifies the geometric object identifier of the query window. Data type is NUMBER.

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

This procedure 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

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

Usage Notes

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

Related Topics

None.

A

Tuning Tips and Sample SQL Scripts

This appendix provides supplemental information to aid in 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.

A.1 Selecting a Spatial Model

This section describes how to 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.

A.1.1 Benefits of the Object-Relational Model

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

- Additional geometry types: arcs, circles, compound polygons, compound line strings, and optimized rectangles are supported.
- Index and query ease of use improved.
- Indexing is maintained by the Oracle database server.
- Geometries modeled in a single row, single column.
- Performance is improved.

A.1.2 Benefits of the Relational Model

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

- Database replication is supported.

- Distributed database is 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.

A.2 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.2.1 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.

A.2.2 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 if 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. Assume 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 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 to start 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 would give an indication of what is the best tiling level for the total data set.

A.2.3 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.
- Define the initial extent as small as possible when you create the SDO_GEOM_METADATA table in the object-relational model, or the <layername>_SDOLAYER and the <layername>_SDODIM tables in the relational model. These tables contain a few rows each and a small initial extent will reduce the amount of wasted space.
- Use the `SDO_GEOM.VALIDATE_GEOMETRY()` procedure 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()` functions.
- Visualizing the indexing tiles, as described in Section A.2.4, can lead to a greater understanding of the tuning process with respect to the size of the tiles.

The following guidelines refer to only the relational model:

- 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_GIDs 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.2.4 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.2.4.1 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. The functions are not recommended for general use, except for this visualization example. Use the following syntax for the internal functions:

```
hhcellbndry (sdo_code || sdo_meta, sdo_dimnum, sdo_lb, sdo_ub,
    hhlength(sdo_code || sdo_meta) {'MIN' | 'MAX'})
```

In the following examples, the dimension boundaries were assumed to be -180 to 180, and -90 and 90. 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 a <layername>_SDOINDEX 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_table_name = 'TEST');
```

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

```
SELECT HHCELLBNDRY(sdo_code || sdo_meta, 1,-180.0, 180.0,
                   HHLENGTH(sdo_code || sdo_meta), 'MIN') min_x,
       HHCELLBNDRY(sdo_code || sdo_meta, 1,-180.0, 180.0,
                   HHLENGTH(sdo_code || sdo_meta), 'MAX') max_x,
       HHCELLBNDRY(sdo_code || sdo_meta, 2, -90.0,  90.0,
                   HHLENGTH(sdo_code || sdo_meta), 'MIN') min_y,
```

```
        HHCELLBNDRY(sdo_code || sdo_meta, 2, -90.0,  90.0,
                    HHLENGTH(sdo_code || sdo_meta), '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 a <layername>_SDOINDEX 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_table_name = 'TEST'
               AND a.sdo_rowid = 'AAAA59AAFAADzAZAAA');
```

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

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

A.2.4.2 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,
    hhlength(sdo_code || sdo_meta) {'MIN' | 'MAX'})
```

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

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

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

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

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

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

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

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

See Section A.4.2.3 for another method of viewing tiles.

A.2.5 Performing Secondary Filter Queries and the Redo Log

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 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.

The RELATE( ) secondary filter issues 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.

A.2.6 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.2.6.1 Efficient Queries for Relational Point Data

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

A rectangle can be defined by its lower-left and upper-right coordinates (Xmin, Ymin and Xmax, Ymax). A point has a single set of coordinates (Px, Py). When

your area-of-interest is a rectangle, instead of using the SDO_GEOM.RELATE() function in the secondary filter, you can use simple SQL comparison operators as follows:

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

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

A.2.6.2 Efficient Schema for Relational Point Layers

Because a point is always referenced by only one tile in a spatial index, for additional 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 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.2.6.3 Script for Using Table Partitioning with Relational Point Data

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

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

A.2.7 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.2.7.1 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.2.7.2 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.2.8 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 your knowledge of the type. After storing the approximated element, create another element in that geometry with ETYPE=0. Spatial ignores elements of ETYPE=0. You can then write your own routines to handle your specialized geometry type.

A.2.9 Partitioning Spatial Data Using the Relational Model

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

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

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

A.2.10 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.2.9, you can achieve further performance gains by loading and indexing geometries in parallel. The partitioned tables can be loaded by selecting from non-partitioned source tables, or using the SQL*Loader utility. Parallel threads (one for each partition,) can be submitted to load the partitioned table. For information on parallel loading, see the description of the SQL*Loader utility in *Oracle8i Utilities*.

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

```
CREATE VIEW a_sdogeom AS SELECT * FROM a_sdogeom
  WHERE sdo_gid BETWEEN 1 and 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 non-partitioned, 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.3 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.3.1 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 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 amount of rollback 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.3.2 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.4 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.4.1 Oracle8*i* *inter*Media Locator

Oracle8*i* *inter*Media Locator is a related product that supports online internet-based geocoding facilities for locator applications and proximity queries.

A.4.1.1 Geocoding Support

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

The *inter*Media Locator option provides an interface to the online geocoding service provided by Qualitative Marketing Service, Inc. (QMS). You can use PL/SQL stored procedure functions to geocode an address, and record and fetch all the information into two predefined objects from the QMS Web site. The first object is of type SDO_GEOMETRY, and it contains the spatial longitude and latitude information stored as point data. The second object returned is GEOCODE_RESULT which contains text fields of a standardized address and other fields mentioned previously such as postal carrier route or block code.

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

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

For more information about *inter*Media Locator, see *Oracle8i interMedia Locator User's Guide and Reference*.

A.4.1.2 Compatibility with Spatial Objects

*inter*Media Locator is a subset of Oracle8*i* Spatial and, therefore, is completely compatible with Spatial objects. The index uses the same set of metadata tables, for instance. One difference is that *inter*Media Locator locates only points, while Spatial supports multiple geometry types.

The `LOCATOR_WITHIN_DISTANCE( )` operator is similar to the SDO_GEOM.WITHIN_DISTANCE( ) operator.

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

A.4.1.3 Sample Locator Code

Sample scripts are available in the following directory after you install Oracle8*i* *inter*Media Locator:

`$ORACLE_HOME/md/demo/geocoder`

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

A.4.2 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.4.2.1 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.4.2.2 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 will pop up, one where geometries are drawn, and a second with several buttons. Click the CHOOSE LAYER button 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 the ZOOM TO EXTENT button, 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.4.2.3 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 the PERFORM QUERY button:

- 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 the PRIM & SEC or the PRIMARY FILTER ONLY radio button.
2. Click either SELECT BOX, SELECT CIRCLE, or SELECT POLYGON, and draw the area of interest on the map.
3. Click the PERFORM QUERY button, and the geometries will display on the base map.

You can look at individual geometries by clicking the SHOW GIDS button. You can also click the SHOW ALL TILES button to look at index tiles. This can help you tune your spatial index. See Section A.2.4 for another method of drawing tiles.

A.4.3 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.4.3.1 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.4.3.2 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 Oracle8*i* Spatial geometry objects.
- All the files and directories under *ORACLE_HOME*/md/demos/NT are components of the Spatial Visualizer demonstration program. They should be used for demonstration purposes only.
- The workspaces are created with Visual C++ 6.0, and might not be compatible with previous versions.

- The ZIP file (DEMO_Visualizer.zip) contains all the contents under this directory. Due to system dependencies, copy the ZIP file only to a Windows NT system.

B

Installation, Compatibility, and Migration Issues

This appendix provides information concerning installation, compatibility, and migration between various Oracle Spatial product releases.

Beginning with Spatial Data Option 7.3.3, all interfaces are supported in each subsequent release. A spatial application built for and using the 7.3.3 spatial data option interfaces will work with an 8.0.4, 8.0.5, or 8.1.3 database server. The implementations of these interfaces have changed and therefore PL/SQL packages from older versions of the spatial cartridge will not work with newer versions of the Oracle database server. Therefore, you must upgrade both server and Spatial at the same time if you wish to use older spatial applications with new Oracle8*i* releases.

Spatial must always be synchronized with the Oracle database server on upgrade or downgrade. In both cases, Spatial must be re-installed.

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 features this option depends on are new in release 8.1 of the database server. Therefore, there are many compatibility and migration issues that need to 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:

- 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 only attempt to use 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.x.x.x, the DBA must shut down the database and restart with COMPATIBLE=8.1.x.x.x.

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.x.x.x from 8.1.x.x.x, 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 Oracle8*i* Spatial will be deleted. The relational implementation available in release 8.0.x.x.x will remain installed and accessible.
4. While connected as SYSTEM, enter the following:

```
ALTER DATABASE RESET COMPATIBILITY
SHUTDOWN
Change  the init.ora parameter COMPATIBLE=8.1.0.0.0
STARTUP
```

After running *ORACLE_HOME*/MD/c813d805.sql, resetting the database comptability to 8.1.x.x.x from 8.0.x.x.x requires running the script *ORACLE_HOME*/MD/c805u813.sql to re-install and enable the object-relational implementation of Spatial.

B.3 Compatibility Details

All releases of the Spatial product provide a set of predefined spatial data types, topological operators such as RELATE(), and a spatial indexing mechanism. The Oracle8*i* Spatial release differs from previous 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 Oracle8*i* are maintained, but the package bodies have been changed to use the above features. Thus for Oracle8*i*, the Spatial packages must be reinstalled to use these interfaces even if the compatibility parameter is set to 8.0.

No data migration is needed and the 7.3.4/8.0.4 spatial applications will work without modification. Any OCI-specific migration issues must be handled in the same manner as they would have to be for any OCI application.

The 7.3.4/8.0.4 to 8.1.3 upgrade requirements are the same. Upgrade both Oracle8*i* and Spatial. Perform all the necessary steps for an upgrade. Your spatial applications will continue to work as before.

Downgrading from 8.0.5 or earlier releases to a previous release of the server and Spatial requires no special steps specific to the Spatial implementation. However, this situation is different for Oracle8*i*. In Oracle8*i*, Spatial uses objects and extensible indexing. Therefore, it creates Oracle8*i* specific database objects that are not compatible with previous releases of the database server. When you downgrade the server and Spatial from Oracle8*i* to 8.0.5, a spatial-specific downgrade script must be executed to remove all the spatial geometry type, indexmethod 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 Oracle8*i* because Spatial now allows two storage mechanisms. If you want the features specific to Oracle8*i*, such as extensible indexing and spatial operators, 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 Oracle8*i* 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, used for indexing purposes, which encloses the spatial object. This did not exist in release 7.3.3.
- From 8.1.3 onward, Spatial supports arcs, circles, arc strings, and geometries made up of a mixture of arc and line segments.

You cannot store arcs and circles in any release earlier than Oracle8*i*. And you cannot use data from a 7.3.4 or later spatial layer in 7.3.3 if it contains instances of type UNSUPPORTED_GEOMETRY (etype=0).

C

Partitioning Legacy Point Data

Spatial has undergone an architectural change, beginning with the 7.3.3 release. The emphasis on partitioned tables has been replaced by the improved spatial indexing features.

Spatial provides the essential functions, procedures, and scripts for using and managing both spatially indexed data and partitioned point data. The information in this appendix is relevant only to users who have not yet migrated to the new data model. For all other users, spatial indexing is preferred and recommended.

Note: The functions described in this appendix will be removed in a future release.

C.1 Overview

Partitioning is a technique where data is loaded into tables that automatically subdivide when a predefined maximum size is reached. During subdivision, data is moved from the parent partition to the child partitions and the parent partition is deleted. Storage parameters for child partitions are inherited from the root partition and can be changed at any time.

A partitioned table has a partition key that is an HHCODE column created by encoding multidimensional point data using the `SDO_ENCODE()` function. In the partitioning process, at each subdivision, data is subdivided into 2^n partitions where *n* is the number of dimensions encoded in the HHCODE column. You can encode up to 32 dimensions using Spatial.

C.2 Partitioning Process

This guide does not attempt to provide the information necessary for fully utilizing table partitioning for point data. Existing users who need to use this method should continue to use their Spatial Data Option 7.3.2 documentation. The following is a high-level description of the partitioning process:

1. Start with an Oracle8*i* table containing multidimensional point data. For example, columns of X and Y coordinate data from a blueprint or map.
2. Create a table or view from the original Oracle8*i* table containing the columns you want, plus a new HHCODE column.

   An HHCODE column is a new data type used to encode multiple dimensions into a unique orderable value. HHCODE is not a point, but rather a bounded cell representing an object space in as many dimensions as have been defined. An HHCODE data type is defined as RAW(255).
3. Create the HHCODE data type by encoding multiple dimensions into a single value using the `SDO_ENCODE()` function. The HHCODE data type will be used as the partition key.
4. Register a partitioned table in the Spatial data dictionary using the `SDO_ADMIN.REGISTER_PARTITION_INFO()` procedure. This procedure takes the name of a table, the name of the partition key column, and the maximum number of records you want stored in a partition before it subdivides.
5. Call the `SDO_ADMIN.PARTITION()` procedure with the name of the table or view containing the partition key column and the tablespace in which the partitions should be created. In this step, the data is partitioned based on dimensions encoded in the HHCODE column.
6. If the underlying table has constraints, grants, or triggers, the owner needs to use the `SDO_ADMIN.PROPAGATE_GRANTS()` procedure to set those properties on the partitions.
7. To add more partitioned point data, load the data into a table, and call `SDO_ADMIN.PARTITION()` again. The dimensions encoded in the HHCODE column must have the same boundaries to be loaded into the existing partitioned table.
8. After you have added data multiple times, or after adding or deleting a large amount of data, there may be partitions that exceed the high-water mark or there may be partitions that can be merged. Call the `SDO_ADMIN.REPARTITION()` procedure to reorganize the partitioned table.

Repartitioning is a computation-intensive task that should be performed only when necessary.

C.3 Scripts for the Deprecated Partitioned Point Data Model

This section describes the following scripts:

- altpart.sql
- drppart.sql
- sdogrant.sql

Although the scripts described in this section are available, the recommended approach is to use Oracle8*i* partitioning and spatial indexing.

C.3.1 altpart.sql Script

The altpart.sql script file shows how to use dynamic SQL in a PL/SQL procedure to modify all partitions of a Spatial partitioned table.

The Spatial data dictionary view used in this SQL script requires that a registered Spatial partitioned table is specified. If the table is not registered, you can use the USER_TABLES view to select all the partitioned tables from the user's schema. To use the USER_TABLES view, enter the following syntax:

```
SQL> SELECT TABLENAME FROM user_tables WHERE TABLENAME LIKE
2> '%tablename_P%';
```

C.3.2 drppart.sql Script

The drppart.sql script file shows how to use dynamic SQL in a PL/SQL procedure to drop (remove) all partitions of a Spatial partitioned table. After running this procedure, you must run the `SDO_ADMIN.DROP_PARTITION_INFO()` procedure.

The Spatial data dictionary view used in this SQL script requires that a registered Spatial partitioned table is specified. If the table is not registered, you can use the USER_TABLES view to select all the partitioned tables from the user's schema. To use the USER_TABLES view, use the following syntax:

```
SQL> SELECT TABLENAME FROM user_tables WHERE TABLENAME LIKE
2> '%tablename_P%';
```

C.3.3 sdogrant.sql Script

The sdogrant.sql script file contains an administrative procedure, PROPAGATE_GRANTS(), which is used after calling the SDO_ADMIN.PARTITION() or SDO_ADMIN.REPARTITION() procedures.

This procedure must first be compiled by running the sdogrant.sql file. The PROPAGATE_GRANTS() procedure is callable only by the user who compiled it.

C.4 Administrative Functions for the Deprecated Model

Table C–1 lists the procedures that can be used with partitioned point data. These procedures are neither required nor compatible with the geometry-based data format.

Table C–1 Administrative Procedures for Partitioned Point Data

Procedure	Data Structure	Description
SDO_ADMIN.ALTER_HIGH_WATER_MARK	Partitioned points	Alters the high-water mark of a partitioned table.
SDO_ADMIN.DROP_PARTITION_INFO	Partitioned points	Removes a partitioned table.
SDO_ADMIN.PARTITION	Partitioned points	Places data into partitioned tables.
SDO_ADMIN.PROPAGATE_GRANTS	Partitioned points	Propagates the grants on the registered underlying table to the various partitions.
SDO_ADMIN.REGISTER_PARTITION_INFO	Partitioned points	Creates a partitioned spatial table.
SDO_ADMIN.REPARTITION	Partitioned points	Reorganizes a table based on the sorted values of the data contained within it.
SDO_ADMIN.VERIFY_PARTITIONS	Partitioned points	Checks for the existence of a table.

SDO_ADMIN.ALTER_HIGH_WATER_MARK

Purpose

This procedure alters the high-water mark of a partitioned spatial table. The high-water mark defines how many records can be stored in a partition before it subdivides. The table must exist and be registered in the Spatial data dictionary.

This procedure is for use only with partitioned point data.

Syntax

SDO_ADMIN.ALTER_HIGH_WATER_MARK (*tablename, high_water_mark*)

Keywords and Parameters

tablename	Specifies the name of the partitioned table. Data type is VARCHAR2.
high_water_mark	Specifies the new high-water mark for the table. Data type is INTEGER.

Usage Notes

None.

The following example changes the high-water mark to 5000 records for the table1 partitioned spatial table:

```
SQL> EXECUTE SDO_ADMIN.ALTER_HIGH_WATER_MARK('table1', 5000);
```

Related Topics

- SDO_ADMIN.REPARTITION() procedure
- altpart.sql sample SQL script file

SDO_ADMIN.DROP_PARTITION_INFO

Purpose

This procedure removes a partitioned spatial table from the Spatial data dictionary. The table must exist and must be registered in the Spatial data dictionary.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.DROP_PARTITION_INFO (*tablename*)

Keywords and Parameters

tablename Specifies the name of the partitioned table. Data type is VARCHAR2.

Usage Notes

This procedure does not remove the spatial table and its associated partition tables from the user's schema. For a description of how to remove a partitioned spatial table from the user's schema, see the drppart.sql sample SQL script file described in Section C.3.2.

The following example removes the table1 table from the Spatial data dictionary:

```
SQL> EXECUTE SDO_ADMIN.DROP_PARTITION_INFO('table1');
```

Related Topics

- drppart.sql sample SQL script file

SDO_ADMIN.PARTITION

Purpose

This procedure places data into partitioned tables based on the sorted order of encoded dimensional values.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.PARTITION (*owner.source_table, tablename, parallel, guess, plummet_flag [,tablespace]*)

Keywords and Parameters

owner.source_table	Specifies the Oracle8*i* table or view of the table containing the partition key column. Data type is VARCHAR2.
tablename	Specifies the name of the table to partition. Data type is VARCHAR2.
parallel	Specifies the degree of parallelism for an operation on a single instance. Data type is INTEGER.
guess	Specifies the estimated largest common level of all the potential partitions to be created from data in the source_table. The common level of a partition is the number of levels of resolution of the common HHCODE for the partition. Data type is INTEGER.
plummet_flag	Specifies if the common HHCODE for all the potential partitions to be created from data in the source_table contains the maximum possible common level. If TRUE, the common HHCODE for each potential partition contains the maximum possible common level. If FALSE, the common HHCODE for each potential partition contains the minimum possible common level. Default value is FALSE. Data type is BOOLEAN.
tablespace	Specifies the tablespace in which the partitions should be created. Default is the tablespace of the underlying table.

Usage Notes

Consider the following when using this procedure:

- The maximum size of the partitioned tables is determined by the high-water mark of the partitioned spatial table.
- To perform this procedure, first load the original data into an Oracle8*i* table using a utility such as SQL*Loader. After the data is loaded, encode the data using the appropriate combination of Spatial data conversion functions. The encoded data is used as the partition key column. The partition key column is provided as either a column in the Oracle8*i* table or as a view of that table.
- For more information on specifying the degree of parallelism, see the *Oracle8i Tuning* manual.

The following example partitions the table1 partitioned spatial table with data contained in the source1 table:

```
SQL> EXECUTE SDO_ADMIN.PARTITION('source1','table1',1,10,FALSE);
```

Related Topics

- SDO_ADMIN.REGISTER_PARTITION_INFO( ) procedure

SDO_ADMIN.PROPAGATE_GRANTS

Purpose

This procedure is used to propagate the grants on the underlying table to the partitions.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.PROPAGATE_GRANTS (*tablename)*

Keywords and Parameters

tablename	Specifies the name of the partitioned table. Data type is VARCHAR2.

Usage Notes

This procedure is used after calls to `SDO_ADMIN.PARTITION()` or `SDO_ADMIN.REPARTITION()`. It must be called by the owner of the partition.

This procedure must be compiled prior to use. See Section C.3.3.

The following example propagates grants from the TABLE1 partitioned spatial table:

```
SQL>  EXECUTE SDO_ADMIN.PROPAGATE_GRANTS('TABLE1');
```

Related Topics

- SDO_ADMIN.PARTITION( ) procedure
- SDO_ADMIN.REPARTITION( ) procedure

SDO_ADMIN.REGISTER_PARTITION_INFO

Purpose

This procedure creates a partitioned spatial table entry in the Spatial data dictionary, and defines the partition key column and the high-water mark for the table.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.REGISTER_PARTITION_INFO (*tablename, column, high_water_mark*)

Keywords and Parameters

tablename	Specifies the name of the partitioned table. Data type is VARCHAR2.
column	Specifies the name of the partition key column for the table. Data type is VARCHAR2.
high_water_mark	Specifies the number of records to store in a partition before the partition subdivides. Data type is INTEGER.

Usage Notes

The SQL CREATE TABLE statement is used to create the partitioned spatial table, with the partition key column defined as RAW(255), prior to calling this procedure.

The following example registers the TABLE1 partitioned spatial table:

```
SQL> EXECUTE SDO_ADMIN.REGISTER_PARTITION_INFO('table1',
2> 'hhcolumn', 1000);
```

Related Topics

- SDO_ADMIN.PARTITION( ) procedure

SDO_ADMIN.REPARTITION

Purpose

This procedure reorganizes a partitioned spatial table based on the sorted order of encoded dimensional values already contained in it. The table must exist and must be registered in the Spatial data dictionary.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.REPARTITION (*tablename, parallel, [tablespace]*)

Keywords and Parameters

tablename	Specifies the name of the partitioned table. Data type is VARCHAR2.
parallel	Specifies the degree of parallelism for an operation on a single instance. Data type is INTEGER.
tablespace	Specifies the name of the tablespace in which to create the partition. Data type is VARCHAR2.

Usage Notes

Consider the following when using this procedure:

- The tablespace variable is optional. If you do not supply a tablespace name, the partitions are created in the same tablespace as the registered partition table.
- The maximum size of the reorganized partition tables is determined by the high-water mark of the partitioned spatial table.

The following example repartitions the table1 partitioned spatial table:

```
SQL> EXECUTE SDO_ADMIN.REPARTITION('table1', 1);
```

Related Topics

- SDO_ADMIN.ALTER_HIGH_WATER_MARK( ) procedure

SDO_ADMIN.VERIFY_PARTITIONS

Purpose

This procedure checks if the partitioned spatial table exists, if it is registered in the Spatial data dictionary, and if the partition key column exists as defined in the Spatial data dictionary.

This procedure is used only with partitioned point data.

Syntax

SDO_ADMIN.VERIFY_PARTITIONS (*tablename*)

Keywords and Parameters

tablename Specifies the name of the table.
Data type is VARCHAR2.

Usage Notes

This procedure can generate the following errors depending on the results of the verification:

- SDO 13113 (Oracle table does not exist)
- SDO 13108 (spatial table not found)
- SDO 13111 (spatial table has no partition key defined)
- SDO 13129 (HHCODE column not found)

The following example verifies the table1 partitioned spatial table:

```
SQL> EXECUTE SDO_ADMIN.VERIFY_PARTITIONS('table1');
```

Related Topics

- SDO_ADMIN.REGISTER_PARTITION_INFO( ) procedure

C.5 Data Functions

The functions described in this section are not required for creating or maintaining a spatial database, however, they are provided for convenience in working with legacy data in partitioned point data tables. They are used with SQL SELECT, INSERT, UPDATE, and DELETE statements to perform the following:

- Generate dimensions from bounded, hierarchical, or date data values
- Encode and decode dimensions
- Retrieve bounded, hierarchical, or date data values from dimensions

When using these functions in basic SQL statements, use the form: SDO_<function>. When using the functions inside a PL/SQL block, use a period (.) instead of the underscore (_).

This section contains descriptions of the spatial functions listed in Table C–2.

Table C–2 Partitioned Point Data Functions

Function	Purpose
SDO_BVALUETODIM	Creates a dimension from bounded data values.
SDO_COMPARE	Evaluates the relationship between two objects described by HHCODEs.
SDO_DATETODIM	Creates a dimension from an Oracle DATE data type.
SDO_DECODE	Extracts a single dimension from an HHCODE.
SDO_ENCODE	Creates an HHCODE by combining dimensions to describe an area or point.
SDO_TO_BVALUE	Extracts a bounded data value from a dimension.
SDO_TO_DATE	Extracts an Oracle DATE data type from a dimension.

SDO_BVALUETODIM

Purpose

This function creates a dimension from a bounded value, which is a value contained in a set of values expressed as a lower boundary and an upper boundary.

Syntax

SDO_BVALUETODIM (*value, lower_boundary, upper_boundary, decimal_scale*)

Keywords and Parameters

value	Specifies the value for the particular dimension. Data type is NUMBER.
lower_boundary	Specifies the lower boundary of the dimension range. Data type is NUMBER.
upper_boundary	Specifies the upper boundary of the dimension range. Data type is NUMBER.
decimal_scale	Specifies the number of digits to the right of the decimal point. Data type is NUMBER.

Returns

This function returns a dimension. The data type is RAW.

Usage Notes

The following example shows the SDO_BVALUETODIM() function:

```
SQL> INSERT INTO sourcetable1(SAMPLENAME,DATA_PT)
2> VALUES ('SAMPLE1',SDO_ENCODE(SDO_BVALUETODIM(10,-100,100,7),
3> SDO_BVALUETODIM(20,-100,100,7));
```

Related Topics

- SDO_ENCODE( ) function
- SDO_TO_BVALUE( ) function

SDO_COMPARE

Purpose

This function evaluates the relationship between an area or point described by an HHCODE and another HHCODE, or a range of HHCODEs expressed as an upper bound and lower bound.

Syntax

SDO_COMPARE (*hhcode_expression*, *{hhcode_expression | lower_bound_HHCODE,upper_bound_HHCODE}*)

Keywords and Parameters

hhcode_expression	Specifies an expression that evaluates to an HHCODE. Data type is RAW.
lower_bound_HHCODE	Specifies the lower bound HHCODE expression. Data type is RAW.
upper_bound_HHCODE	Specifies the upper bound HHCODE expression. Data type is RAW.

Returns

This function returns one of the following keywords:

- ENCLOSES
- EQUAL
- INSIDE
- OUTSIDE
- OVERLAP

The data type is VARCHAR2.

Usage Notes

The following example selects all points that fall within the given multidimensional range:

```
SQL> SELECT SDO_GID FROM layer1_SDOINDEX WHERE
2> SDO_COMPARE(SDO_MAXCODE,
```

```
3> SDO_ENCODE(5,5),
4> SDO_ENCODE(25,25))='INSIDE';
```

The following example selects GIDs based on interaction between their spatial index tiles:

```
SQL> SELECT SDO_GID FROM layer1_SDOINDEX A, layer2_SDOINDEX B
2> WHERE SDO_COMPARE(A.SDO_CODE,B.SDO_CODE) != 'OUTSIDE';
```

Related Topics

- SDO_GEOM.RELATE( ) function

SDO_DATETODIM

Purpose

This function creates a dimension from an Oracle DATE data type. The component number determines the level of resolution of the date in the dimension.

Syntax

SDO_DATETODIM (*date* [, *component*])

Keywords and Parameters

date — Specifies the calendar date.
Data type is DATE.

component — Specifies the level of resolution. The component number values are defined as follows:

1 accurate to year
2 accurate to month
3 accurate to day
4 accurate to hour
5 accurate to minute
6 accurate to second

The default value is 6.
Data type is INTEGER.

Returns

This function returns a dimension. The data type is RAW.

Usage Notes

You must use a valid Oracle date format string.

The following example shows the SDO_DATETODIM( ) function:

```
SQL> INSERT INTO sourcetable1(SAMPLENAME,DATA_PT)
2> VAUES('SAMPLE1',SDO_ENCODE(SDO_DATETODIM(TO_DATE('19-Jul-96'),
3> SDO_BVALUETODIM(100,-1000,1000,7)));
```

Related Topics

- SDO_ENCODE( ) function
- SDO_TO_DATE( ) function

SDO_DECODE

Purpose

This function extracts a single dimension from an HHCODE.

Syntax

SDO_DECODE (*hhcode_expression*, *dimension_number*)

Keywords and Parameters

hhcode_expression	Specifies an expression that evaluates to an HHCODE. Data type is RAW.
dimension_number	Specifies the dimension number to extract. Data type is INTEGER.

Returns

This function returns a dimension. The data type is RAW.

Usage Notes

The SDO_DECODE( ) function is called once for each dimension to be decoded.

The following example shows the SDO_DECODE( ) function:

```
SQL> SELECT
2> SDO_TO_BVALUE(SDO_DECODE(DATA_PT,1),1,6),
3> SDO_TO_BVALUE(SDO_DECODE(DATA_PT,2),-100,100),
4> SDO_TO_DATE(SDO_DECODE(DATA_PT,3))
5> FROM sourcetable1 WHERE SAMPLENAME='SAMPLE1';
```

Related Topics

- SDO_TO_BVALUE( ) function
- SDO_TO_DATE( ) function

SDO_ENCODE

Purpose

This function combines dimensions to create the HHCODE that describes an area or point.

Syntax

SDO_ENCODE (*dimension1*[,*dimension2* ...])

Keywords and Parameters

dimension Specifies an expression created by the SDO_BVALUETODIM or SDO_DATETODIM functions.
Data type is RAW.

Returns

This function returns an HHCODE. The data type is RAW.

Usage Notes

Consider the following when using this function:

- When encoding dimensions, the order of the dimensions in the parameter list must be consistent for all rows within the table.
- This function can encode up to 32 dimensions.

The following example shows the SDO_ENCODE( ) function:

```
SQL> INSERT INTO sourcetable1(SAMPLENAME,DATA_PT)
2> VALUES ('SAMPLE1',SDO_ENCODE(SDO_BVALUETODIM(50,-100, 100, 10),
3> SDO_BVALUETODIM(30,-100,100,10),
4> SDO_DATETODIM(TO_DATE('05-Jul-96'),3)));
```

Related Topics

- SDO_BVALUETODIM( ) function
- SDO_DATETODIM( ) function

SDO_TO_BVALUE

Purpose

This function returns the original bounded data value of a dimension.

Syntax

SDO_TO_BVALUE (*dimension*, *lower_boundary, upper_boundary*)

Keywords and Parameters

dimension	Specifies the dimension. Data type is RAW.
lower_boundary	Specifies the lower boundary of the dimension range. Data type is NUMBER.
upper_boundary	Specifies the upper boundary of the dimension range. Data type is NUMBER.

Returns

This function returns a bounded data value. The data type is NUMBER.

Usage Notes

This function returns a number that is the value for a dimension within the specified range. This is not necessarily the range for which the dimension was originally created.

The following example shows the SDO_TO_BVALUE( ) function:

```
SQL> SELECT (SDO_TO_BVALUE(SDO_DECODE(DATA_PT,2),-100,100)
2> FROM sourcetable1 WHERE SAMPLENAME='SAMPLE1';
```

Related Topics

- SDO_DECODE( ) function
- SDO_BVALUETODIM( ) function

SDO_TO_DATE

Purpose

This function returns the original date value of a dimension.

Syntax

SDO_TO_DATE (*dimension*)

Keywords and Parameters

dimension Specifies the dimension.
Data type is RAW.

Returns

This function returns an Oracle DATE data type.

Usage Notes

The following example shows the SDO_TO_DATE( ) function:

```
SQL> SELECT SDO_TO_DATE(SDO_DECODE(DATA_PT,3))
2> FROM sourcetable1 WHERE SAMPLENAME='SAMPLE1';
```

Related Topics

- SDO_DATETODIM( ) function
- SDO_DECODE( ) function

C.6 Data Dictionary

The Spatial data dictionary is a set of tables owned by the database user mdsys. An extension to the Oracle8*i* data dictionary, it automatically maintains information about spatial tables, columns, and partitions. The Spatial data dictionary is created during the installation process. All nonspatial attribute information is maintained in the Oracle8*i* data dictionary.

The Spatial data dictionary has public views that provide extensive information about spatial tables. This section contains descriptions of the views that are available.

Note: Only the partitioned point routines use the Spatial data dictionary.

The following views are publicly available:

- ALL_MD_COLUMNS
- ALL_MD_DIMENSIONS
- ALL_MD_EXCEPTIONS
- ALL_MD_LOADER_ERRORS
- ALL_MD_PARTITIONS
- ALL_MD_TABLES
- ALL_MD_TABLESPACES
- DBA_MD_COLUMNS
- DBA_MD_DIMENSIONS
- DBA_MD_EXCEPTIONS
- DBA_MD_LOADER_ERRORS
- DBA_MD_PARTITIONS
- DBA_MD_TABLES
- DBA_MD_TABLESPACES
- USER_MD_COLUMNS

- USER_MD_DIMENSIONS
- USER_MD_EXCEPTIONS
- USER_MD_LOADER_ERRORS
- USER_MD_PARTITIONS
- USER_MD_TABLES
- USER_MD_TABLESPACES

WARNING: Do not delete or modify any of the tables in the mdsys account. This corrupts the Spatial data dictionary.

ALL_MD_COLUMNS

Returns a list of all columns that are part of spatial tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the column
DATA_TYPE	data type of the column
DATA_LENGTH	length of the column in bytes
DATA_PRECISION	scale for NUMBER data type, binary precision for FLOAT data type, and NULL for all other data types
DATA_SCALE	digits to right of decimal point in an HHCODE column or a number
NDIM	number of dimensions in the HHCODE column (It is NULL for all other data types.)
MAX_LEVEL	maximum number of levels in the column
NULLABLE	indicates if column allows NULL values
PARTITION_KEY	indicates if column is the partition key column; only one is allowed per partitioned table
COLUMN_ID	sequence number of the column as created
DEFAULT_LENGTH	length of the default value for the column

Column	Description
NUM_DISTINCT	number of distinct values in each column of the table
LOW_VALUE	lowest value for tables with three or fewer rows (It is the second-lowest value in the column for tables with more than three rows.)
HIGH_VALUE	highest value for tables with three or fewer rows (It is the second-highest value in the column for tables with more than three rows.)

ALL_MD_DIMENSIONS

Returns a list of all dimensions that are part of HHCODE columns.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the column
DIMENSION_NAME	name of the dimension
DIMENSION_NUMBER	dimension number
LOWER_BOUND	lower boundary of the dimension range
UPPER_BOUND	upper boundary of the dimension range
SCALE	scale of the dimension
RECURSION_LEVEL	number of levels encoded in the HHCODE column

ALL_MD_EXCEPTIONS

Contains information about spatial tables that should be removed (dropped) as a result of some failed operation, such as a failed load.

Column	Description
OWNER	owner of the object
NAME	object name
OPERATION	operation during which the failure occurred
CCHH	common code HHCODE

ALL_MD_LOADER_ERRORS

Contains the current status of a file that was loaded into a table using SD*Loader.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	spatial table name
FILENAME	SLF file name
ROWS_LOADED	number of rows loaded before failure

ALL_MD_PARTITIONS

Returns a list of all the partitioned tables that are part of a user-accessible spatial table.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
PARTITION_TABLE_NAME	name of the partitioned table
CLASS	class of partition: NODE or LEAF
COMMON_LEVEL	number of levels of resolution of the common HHCODE column for the partition
COMMON_HHCODE	common HHCODE substring for the partition
OFFLINE_STATUS	status of partition: ONLINE or OFFLINE
ARCHIVE_DATE	date of last archive

ALL_MD_TABLES

Returns a list of all the user-accessible spatial tables.

Column	Description
OWNER	owner of the table
MD_TABLE_NAME	name of the spatial table
CLASS	class of table: PARTITIONED or NON-PARTITIONED
PTAB_SEQ	number of last partitioned table created

Column	Description
HIGH_WATER_MARK	maximum number of rows that can be inserted into a partitioned table
OFFLINE_PATH	complete path name to directory where the table is archived
COUNT_MODE	count mode for estimating number of rows in a partition: ESTIMATE or EXACT

ALL_MD_TABLESPACES

Returns a list of all tablespaces used by spatial tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
TABLESPACE_NAME	name of tablespace
SEQUENCE	sequence number
STATUS	status of tablespace: ACTIVE or INACTIVE

DBA_MD_COLUMNS

Returns a list of all columns that are part of Spatial tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the column
DATA_TYPE	data type of the column
DATA_LENGTH	length of the column in bytes
DATA_PRECISION	scale for NUMBER data type, binary precision for FLOAT data type, and NULL for all other data types
DATA_SCALE	digits to right of decimal point in an HHCODE column or a number
NDIM	number of dimensions in the HHCODE column (It is NULL for all other data types.)
MAX_LEVEL	maximum number of levels in the column

Column	Description
NULLABLE	indicates if column allows NULL values
PARTITION_KEY	indicates if column is the partition key column; only one is allowed per partitioned table
COLUMN_ID	sequence number of the column as created
DEFAULT_LENGTH	length of the default value for the column
NUM_DISTINCT	number of distinct values in each column of the table
LOW_VALUE	lowest value for tables with three or fewer rows (It is the second-lowest value in the column for tables with more than three rows.)
HIGH_VALUE	highest value for tables with three or fewer rows (It is the second-highest value in the column for tables with more than three rows.)

DBA_MD_DIMENSIONS

Returns a list of all dimensions that are a part of spatial tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the column
DIMENSION_NAME	name of the dimension
DIMENSION_NUMBER	dimension number
LOWER_BOUND	lower boundary of the dimension range
UPPER_BOUND	upper boundary of the dimension range
SCALE	scale of the dimension
RECURSION_LEVEL	number of levels encoded in the HHCODE column

DBA_MD_EXCEPTIONS

Contains information about spatial tables that should be removed (dropped) as a result of some failed operation, such as a failed load.

Column	Description
OWNER	owner of the object
NAME	object name
OPERATION	operation during which the failure occurred
CCHH	common code HHCODE

DBA_MD_LOADER_ERRORS

Contains the current status of a file that was loaded into a table using SD*Loader.

Column	Description
OWNER	owner of the table where the error occurred
MD_TABLE_NAME	spatial table name
FILENAME	SLF file name
ROWS_LOADED	number of rows loaded before failure

DBA_MD_PARTITIONS

Returns a list of all the partitioned tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
PARTITION_TABLE_NAME	name of the partitioned table
CLASS	class of partition: NODE or LEAF
COMMON_LEVEL	number of levels of resolution of the common HHCODE column for the partition
COMMON_HHCODE	common HHCODE substring for the partition
OFFLINE_STATUS	status of partition: ONLINE or OFFLINE
ARCHIVE_DATE	date of last archive

DBA_MD_TABLES

Returns a list of all the spatial tables.

Column	Description
OWNER	owner of the table
MD_TABLE_NAME	name of the spatial table
CLASS	class of table: PARTITIONED or NON-PARTITIONED
PTAB_SEQ	number of last partitioned table created
HIGH_WATER_MARK	maximum number of rows that can be inserted into a partitioned table
OFFLINE_PATH	complete path name to directory where the table is archived
COUNT_MODE	count mode for estimating number of rows in a partition: ESTIMATE or EXACT

DBA_MD_TABLESPACES

Returns a list of all tablespaces used by spatial tables.

Column	Description
OWNER	owner of the object
MD_TABLE_NAME	name of the spatial table
TABLESPACE_NAME	name of tablespace
SEQUENCE	sequence number
STATUS	status of tablespace: ACTIVE or INACTIVE

USER_MD_COLUMNS

Returns a list of all the HHCODE columns that are part of tables owned by the user.

Column	Description
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the spatial table
DATA_TYPE	data type of the column
DATA_LENGTH	length of the column in bytes

Column	Description
DATA_PRECISION	scale for NUMBER data type, binary precision for FLOAT data type, and NULL for all other data types
DATA_SCALE	digits to right of the decimal point in an HHCODE column or a number
NDIM	number of dimensions in the HHCODE column (It is NULL for all other data types.)
MAX_LEVEL	maximum number of levels in the column
NULLABLE	indicates if column allows NULL values
PARTITION_KEY	indicates if column is the partition key column; only one allowed per partitioned table
COLUMN_ID	sequence number of the column as created
DEFAULT_LENGTH	length of the default value for the column
NUM_DISTINCT	number of distinct values in each column of the table
LOW_VALUE	lowest value for tables with three or fewer rows (It is the second-lowest value in the column for tables with more than three rows.)
HIGH_VALUE	highest value for tables with three or fewer rows (It is the second-highest value in the column for tables with more than three rows.)

USER_MD_DIMENSIONS

Returns a list of all dimensions that are part of HHCODE columns owned by the user.

Column	Description
MD_TABLE_NAME	name of the spatial table
COLUMN_NAME	name of the column
DIMENSION_NAME	name of the dimension
DIMENSION_NUMBER	dimension number
LOWER_BOUND	lower boundary of dimension range
UPPER_BOUND	upper boundary of dimension range
SCALE	scale of the dimension

Column	Description
RECURSION_LEVEL	number of levels encoded in the HHCODE column

USER_MD_EXCEPTIONS

Contains information about spatial tables that should be removed (dropped) as a result of some failed operation, such as a failed load.

Column	Description
NAME	object name
OPERATION	operation during which the failure occurred
CCHH	common code HHCODE

USER_MD_LOADER_ERRORS

Contains the current status of a file that was loaded into a table using SD*Loader.

Column	Description
MD_TABLE_NAME	spatial table name
FILENAME	SLF file name
ROWS_LOADED	number of rows loaded before failure

USER_MD_PARTITIONS

Returns a list of all the partitioned tables that are part of spatial tables owned by the user.

Column	Description
MD_TABLE_NAME	name of the spatial table
PARTITION_TABLE_NAME	name of the partition
CLASS	class of partition: NODE or LEAF
COMMON_LEVEL	number of levels of resolution of the common HHCODE column for the partition
COMMON_HHCODE	common HHCODE substring for the partition
OFFLINE_STATUS	status of partition: ONLINE or OFFLINE
ARCHIVE_DATE	date of last archive

USER_MD_TABLES

Returns a list of all the spatial tables owned by the user.

Column	Description
MD_TABLE_NAME	name of the spatial table
CLASS	class of table: PARTITIONED or NON-PARTITIONED
PTAB_SEQ	number of last sequence created
HIGH_WATER_MARK	maximum number of rows that can be inserted into a partitioned table
OFFLINE_PATH	complete path name to directory where the table is archived
COUNT_MODE	count mode for estimating number of rows in a partition: ESTIMATE or EXACT

USER_MD_TABLESPACES

Returns a list of all tablespaces used by spatial tables.

Column	Description
MD_TABLE_NAME	name of the spatial table
TABLESPACE_NAME	name of tablespace
SEQUENCE	sequence number
STATUS	status of the tablespace: ACTIVE or INACTIVE

C.7 Messages and Codes

MDSQL-00001: partition is OFFLINE

Cause: An MDSQL operation was attempted on a partition that is OFFLINE.

Action: Restore the partition and try the operation again.

MDSQL-00002: PK is out of bounds

Cause: The partition key for the record being inserted belongs in another partition.

Action: Insert the record into the correct partition. The correct partition can be identified using the GET_PARTITION_NAME( ) function.

MDSQL-00003: updates that move the PK are not supported

Cause: The update of the partition key would result in the record belonging to another partition.

Action: Use the MD_DML.MOVE_RECORD( ) procedure to update the partition key and move the record to the correct partition.

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.

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

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 a 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. Oracle8*i* is an object-relational database management system. Other types of database systems are called hierarchical or network database systems.

resolution

The number of subdivision levels of data.

scale

1. The number of digits to the right of the decimal point in a number representing the level of resolution of an HHCODE.
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 Oracle8*i* data dictionary. It keeps track of the number of partitions created in a spatial table. The Spatial data dictionary is owned by user mdsys. The data dictionary is used only by the deprecated partitioned point routines.

spatial data structures

A class of data structures designed to store spatial information and facilitate its manipulation.

Spatial 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

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

V

W