Oracle7 Spatial Data Option™ Reference and Administrator's Guide

Version 7.3.2

April 1996
Part No. A43694–1

ORACLE®

New Dimensions in Transparent Data Sharing

Oracle7 Spatial Data Option Reference and Administrator's Guide, Release 7.3.2

Part No. A43694–1

Authors: Gwendolyn Gall, Robin Inglis–Arkell
Contributors: Nipun Agarwal, John Barranco, Frank Bishop, Mark Bradley, Jesse Driver, Linda Feng, Mike Galluchon, Meg Hennington, Edric Keighan, Cindee Kibbe, Nancy Raabe, Jim Rawlings, Tim Robertson, Glenn Stowe, Herman Varma, Jon Veach, Peter Vretanos

Preface

The *Oracle7 Spatial Data Option Reference and Administrator's Guide* provides reference information for the Spatial Data Option.

The following topics are included in this preface:

- audience
- how this guide is organized
- related publications
- document conventions
- customer support
- documentation sales and client relations
- reader comments

How This Guide is Organized

The chapters in this guide are divided into three parts, organized as follows:

Part I: Concepts

Part I provides a conceptual overview of the Spatial Data Option.

Chapter 1, ”Oracle7 Spatial Data Option Components”

This chapter describes how the Spatial Data Option documentation set is organized; and defines, describes, and explains basic Spatial Data Option concepts.

Chapter 2, ”Converting and Loading”

This chapter describes, in general terms, how to convert and load source data into the Spatial Data Option format.

Chapter 3, ”Data Manipulation and Query”

This chapter describes basic activities you must perform for data manipulation and query.

Chapter 4, ”Administration”

This chapter describes activities you must perform to administer and maintain the Spatial Data Option components.

Part II: Reference

Part II contains all Spatial Data Option utilities, kernel functions, and SD*SQL packages. The components in each chapter are listed in alphabetical order.

Chapter 5, ”Utilities”

This chapter describes and provides syntax for the Spatial Data Option utilities.

Chapter 6, ”SD*SQL Kernel Functions”

This chapter describes and provides syntax for the Spatial Data Option SD*SQL kernel functions.

Chapter 7, ”SD*SQL Packages”

This chapter describes and provides syntax for the SD*SQL packages.

Chapter 8, ”User–Developed SLF Converter”

This chapter contains information on how to create a user–developed SLF converter, and describes the SLF library and user SLF header file.

Part III: Appendices

Part III contains the following additional reference material.

Appendix A, ”Spatial Data Option Data Dictionary Reference”

This appendix contains an alphabetical list of the Spatial Data Option data dictionary views.

Appendix B, ”Quick Reference”

This appendix provides an alphabetical list of the syntax for Spatial Data Option utilities, kernel functions, and SD*SQL packages used in the Spatial Data Option.

Appendix C, ”Messages and Codes”

This appendix contains specific Spatial Data Option messages and codes that you might encounter when working with the product.

Glossary

Index

Related Publications

Oracle7 Server Documentation

In addition to this guide, general information about the Oracle7 Server for all operating systems is provided in the Oracle7 Server documentation set, which consists of the following manuals:

Oracle7 Server Concepts

This book describes all features of the Oracle7 Server. It provides a conceptual foundation for the practical information contained in the other Oracle7 Server documentation.

Oracle7 Server Administrator's Guide

This book describes how to manage the Oracle7 Server.

Oracle7 Server Tuning

This book describes how to enhance Oracle7 Server. database performance by adjusting database applications, the database itself, and the operating system.

Oracle7 Server Application Developer's Guide

This book describes features of the Oracle7 Server, and how to develop applications for Oracle7.

Oracle7 Server Reference

This book provides reference information about the Oracle7 Server, including initialization parameters, data dictionary views, database limits, and SQL scripts.

Oracle7 Server SQL Reference

This book provides a complete description of SQL, which is used to manage information in an Oracle7 database.

Oracle7 Server Utilities

This book describes how to use Oracle7 Server utilities for data transfer, maintenance, and database administration.

Oracle7 Server Messages

This book provides complementary information about messages generated by the Oracle7 Server and its integral parts such as PL/SQL, precompilers, and SQL*Loader.

PL/SQL User's Guide and Reference

This book explains how to use PL/SQL, Oracle Corporation's procedural language extension to SQL.

Oracle7 Spatial Data Option Documentation

In addition to this guide, information about the Spatial Data Option for all operating systems is provided in the following manuals:

Oracle7 Spatial Data Option Overview

This guide provides a brief overview of the Spatial Data Option.

Oracle7 Spatial Data Option Application Developer's Guide

This guide lists the tasks and procedures a user must perform to design and use a spatial database.

Document Conventions

Conventions used in this guide differ somewhat from those used in other Oracle documentation, including the generic Oracle7 Server publications listed previously. The following conventions are observed.

Special Conventions

Because many operating systems are case-sensitive, enter commands exactly as shown.

Bold — Bold type is used to denote directories and filenames, as in **init.ora**. Portions of the filename that may vary appear in italics, as in **sgadef*x*.dbf.**

System privilege also appear in bold.

`italics` — Italicized words in Courier font represent a variable. Substitute an appropriate value.

<u>underlining</u> — Underlining is used to define syntax defaults.

UPPERCASE WORDS — Uppercase text is used to call attention to Oracle command keywords and statements.

[UPPERCASE] — Key names are represented by uppercase letters enclosed in square brackets, as in [RETURN].

Command Syntax

`Courier` — Courier font is used for text that must be entered exactly as shown, as in the following example:

```
ls –l
```

Vertical lines | — Vertical lines are used for alternative choices. The set of alternative choices is enclosed by curly braces if one of the items is required, or by square brackets if the item is an optional alternative.

Curly braces { } — Curly braces are used for required items. Users must choose one of the alternatives as in the following example:

```
.DEFINE { macro1 | macro2 }
```

Brackets [ ] — Square brackets are used for optional items, as in the following example:

```
cvtcrt termname [outfile]
```

Ellipses — Ellipses are used for an arbitrary number of similar items, as in the following example:

```
CHKVAL fieldname value1 value2 ... valueN
```

The following symbols should always be entered as they appear in the command format:

• period	.
• comma	,
• hyphen	–
• semicolon	;
• colon	:
• equal sign	=
• backlash	\
• single quote	’
• double quote	”
• parentheses	()

Special Icon

The following special icon is provided to alert you to particular information within the body of this guide.

Warning: The warning symbol highlights text that warns you of actions that could be particularly damaging or fatal to your system.

Other Conventions

Note that ”Oracle7” and ”Oracle” refer to the relational database server product from Oracle Corporation. The term **“oracle**” refers to an executable or account by that name.

Unless otherwise stated, examples use the C shell (**csh**(1)) syntax.

All references made throughout this book to specific chapters refer to chapters in this guide except where noted.

Customer Support

Oracle Corporation's Worldwide Support technical support answer line can be reached 24 hours a day. If you have followed the instructions in the documentation and you need further assistance, please call:

in the USA: **1.415.506.1500**

in Europe: **+ 44 1344 860160**

You will be asked a series of questions to help navigate you to the correct Oracle product support group. Be prepared to supply the following information:

- your CSI number, which helps us track problems recorded for each customer and is necessary to identify you as a supported customer
- version numbers of the Oracle7 Server and associated products
- operating system name and version number
- details of error numbers and descriptions (write down the exact errors—it will help Worldwide Support track down the problem more quickly)
- a description of the problem

Documentation Sales and Client Relations

To order hard copy documentation, call Documentation Sales at the following number:

1.800.252.0303

For shipping inquiries, product exchanges or returns, call Client Relations at the following number:

1.415.506.1500

Your Comments Are Welcome

We value your comments as a user of the Spatial Data Option. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. Please use the Reader's Comment Form at the back of this manual to tell us what you like and dislike about this manual. Alternatively, you can contact us at the following address:

Documentation Manager Government Products Division
Oracle Corporation
500 Oracle Parkway Box 659204
Redwood Shores, California 94065
Phone: 1.415.506.2503 FAX: 1.415.506.7408

Contents

PART

I

Concepts

CHAPTER

1

Oracle7 Spatial Data Option Components

This chapter describes the concepts and components of a Spatial Data Option database. It provides information on the following topics:

- understanding the Oracle7 Spatial Data Option
- overview
- utilities and tools
- spatial table architecture
- HHCODE column
- attribute column
- spatial table types
- partitioning
- Spatial Data Option data dictionary

Understanding the Oracle7 Spatial Data Option

This guide is part of the following set of documents that describe the Spatial Data Option and tell you how to use it with the Oracle7 database:

- *Oracle7 Spatial Data Option Overview – Advances in Relational Database Technology for Spatial Data Management*

  This document presents a general overview of the Spatial Data Option, lists scenarios in which Spatial Data Option can be used effectively, and explains in general terms how to use the product.

- *Oracle7 Spatial Data Option Reference and Administrator's Guide*

  This guide is divided into the following sections:

  – Part I , "Concepts," provides definitions for the Spatial Data Option concepts, utilities, and procedures.

  – Part II , "Reference," is an alphabetized reference for all Spatial Data Option utilities, functions, packages, and libraries.

  – Part III , "Appendices," has an appendix listing all Spatial Data Option data dictionary tables and views and a quick reference appendix that lists the syntax for all utilities, functions, and packages.

- *Oracle7 Spatial Data Option Application Developer's Guide*

  This guide provides a step–by–step account of how to use the Spatial Data Option with Oracle7 to design and set up a database, and query and manipulate spatial data. It also includes a description of the demos available with the Spatial Data Option.

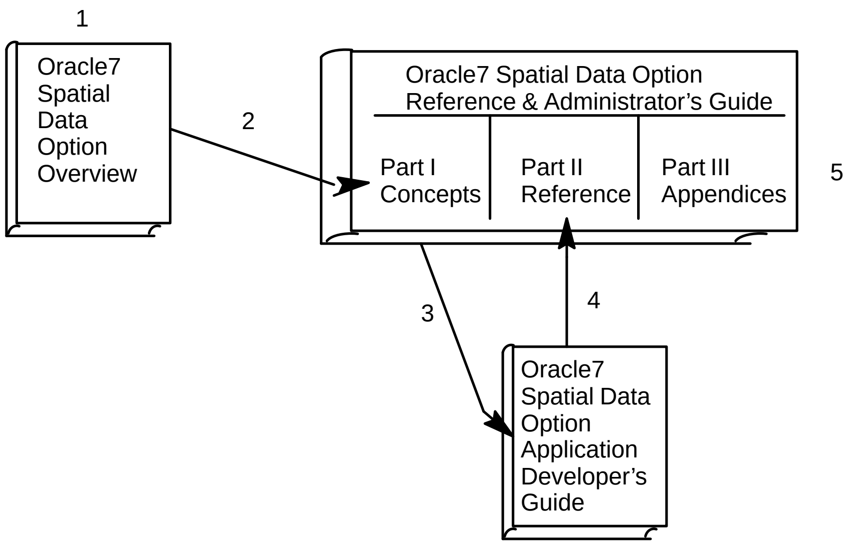

Figure 1 – 1 How to Read Spatial Data Option Documentation

To understand the Spatial Data Option and use it effectively, you should review the documentation in the following order:

1. Read *Oracle7 Spatial Data Option Overview – Advances in Relational Database Technology for Spatial Data Management* for a general explanation of the Spatial Data Option.
2. Read Part I, "Concepts." This section contains basic information about the Spatial Data Option. You should be familiar with this information before you install, use, or administer the product.
3. Read the chapters in the *Oracle7 Spatial Data Option Application Developer's Guide* that pertain to the tasks you want to perform.
4. Refer to pertinent sections of Part II, "Reference," for detailed information on syntax or to review examples of the utilities and procedures you want to perform.
5. After you are familiar with the Spatial Data Option, use Appendix B, "Quick Reference," to quickly review the syntax for Spatial Data Option utilities, functions, and procedures.

Overview

The Spatial Data Option is an integrated set of functions, applications, and utilities that enables spatial data to be stored, accessed, and analyzed quickly and efficiently in the Oracle7 database. Figure 1 – 2 illustrates how Spatial Data Option product modules relate to each other, to the Oracle7 Server, and to the stored data:

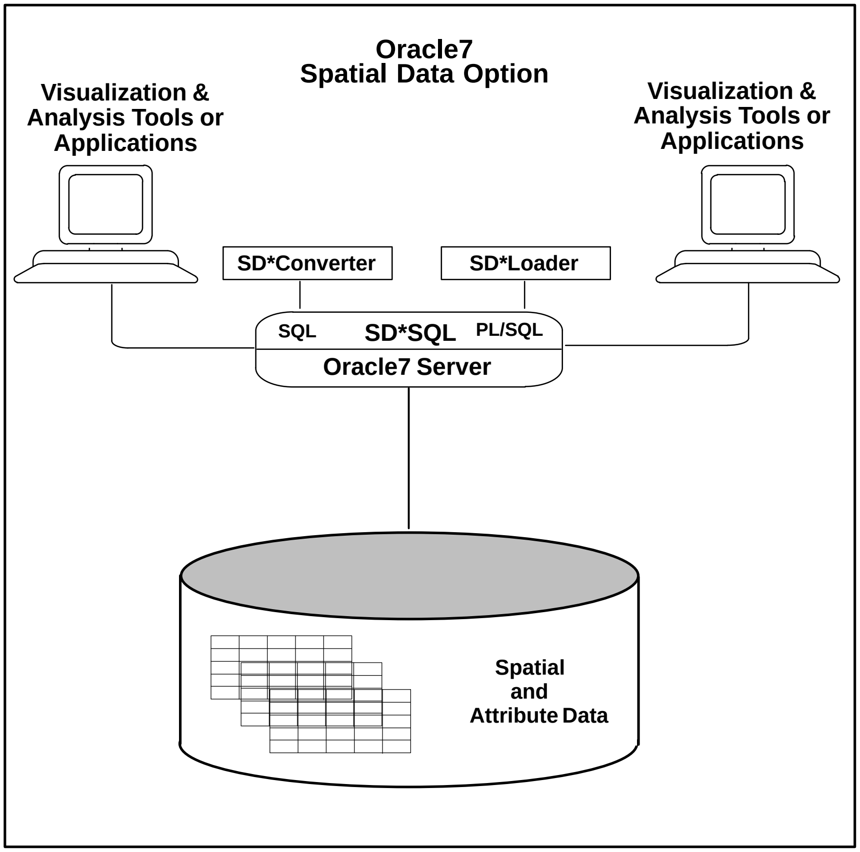

Figure 1 – 2 Oracle7 Spatial Data Option Product Architecture

Spatial Data Option product modules consist of the Spatial Data Option extensions to the Oracle7 Server, a set of utilities and tools for managing spatial data, and the Spatial Data Option data dictionary.

Utilities and Tools

The Spatial Data Option utilities and tools briefly described in this section are used to convert and load data into spatial tables, and manage and manipulate data in spatial tables. In addition, the tools assist users in creating their own applications for converting or displaying data in a variety of formats.

SD*Converter

This utility reads source data files and converts them into spatial load format (SLF), the format required to load data into spatial tables using SD*Loader. Initial data format can come from Oracle7 tables, ASCII flat files, or binary files.

SD*Loader

This utility reads in SLF files and loads the data into spatial tables in an Oracle7 database with the Spatial Data Option.

SD*SQL

This is a set of procedures and functions used to query, manage, and manipulate spatial data and spatial data objects stored in an Oracle7 database.

SLF Library

This is a library of C language functions that provides application developers with the tools to build programs for converting complex or proprietary data formats into SLF files.

The Spatial Data Option utilities, procedures, and functions are described in detail in Part II, "Reference."

Spatial Data Option Data Dictionary

This set of tables and views is an extension to the Oracle7 data dictionary. The Spatial Data Option data dictionary maintains information about spatial tables, columns, and partitions.

Spatial Table Architecture

A spatial table is a table that contains spatial data; or any set of data that is defined by more than one interrelated dimension. Some categories of information that can be defined as spatial include the following:

- geographic data: latitude, longitude, elevation
- demographic data: age, income, sex, marital status
- physical data: height, weight, width, depth
- time: year, month, day, hour, second
- manufacturing: valve, pressure, flow rate
- stock trading: price, block size, date, time
- consumer marketing: customer type, past purchase record, promotion

Spatial data can be organized by multiple criteria, and queried at different abstraction levels.

An Oracle7 database with the Spatial Data Option can contain both spatial tables and standard Oracle7 tables. Whether you define tables as spatial tables or standard Oracle7 tables depends on how you intend to store, analyze, and retrieve the data.

Advantages

One advantage of storing data in spatial tables is that related datasets maintain their spatial integrity. Data elements in a partition representing closely related objects are grouped proximally in tables. Because the spatial organization is maintained, the speed of data retrieval depends on the size of the retrieved dataset and the number of partitions where that data is stored, not the size of the database.

Characteristics

A spatial table has the following characteristics:

- It is registered in the Spatial Data Option data dictionary.
- It contains at least one spatial data column that is referred to as an HHCODE column. A spatial table can contain multiple HHCODE columns.
- It can contain one or more attribute columns, which are columns providing additional descriptive information.

Restrictions

A spatial table is subject to the following restrictions:

- The maximum length for a spatial tablename, trigger, or index is 19 characters.
- All spatial tables must reside in the same database as the Spatial Data Option data dictionary that defines them.
- The spatial data column must be defined as datatype RAW.
- LONG and LONG RAW columns are not supported in this release of the Spatial Data Option.
- You cannot execute a query that accesses a view whose text length is greater than 64 Kb.

  For information on how to extract large amounts of data, see Chapter 3, "Data Manipulation and Query."
- Distributed database capability is not supported in this release of the Spatial Data Option. For example, you cannot have part of a partitioned table on a remote database. Client/server configurations are, however, fully supported.

HHCODE Column

The Spatial Data Option features a new datatype, HHCODE. HHCODE encodes multiple dimensions into a unique orderable value that is stored in a single column in a spatial table. The HHCODE datatype functions as an orderly breakdown of object space represented as a linear string.

HHCODE is not a point, but rather a bounded cell representing an object space, in as many dimensions as have been defined, whose sides are not necessarily equal. In two dimensions, an HHCODE can be visualized as a variable–sized, interlocking polygon. In three dimensions, an HHCODE is a six–faced solid figure, or hexahedron. In *n*–dimensions, an HHCODE can be visualized as a multidimensional cell.

A visualization of an HHCODE in two–dimensional and three–dimensional space is illustrated in Figure 1 – 3:

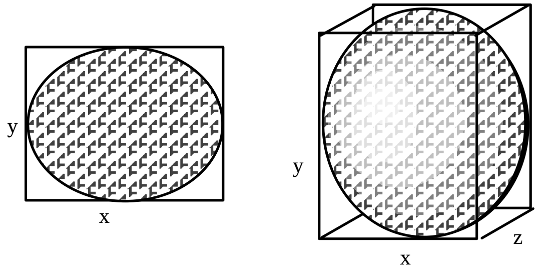

Figure 1 – 3 HHCODEs in Two– and Three–Dimensional Space

Advantages

When spatial data is stored in HHCODE format the spatial organization, which is based on all encoded values of the data, is embodied in the HHCODE value. Records can be grouped together based on their HHCODE value, and can therefore be accessed rapidly. HHCODE data is maintained to its full original precision within the encoding.

When the HHCODE data is sorted, the data clusters dimensionally, and the values representing all the dimensions match to a substring level. Therefore HHCODEs can be used to represent variable–sized object spaces, depending on specific requirements for data precision, analysis, and visual display. HHCODEs can be aggregated dynamically to vary resolution. The hierarchical organization of the HHCODE allows resolution to vary with the complexity of the object being represented.

Characteristics

The HHCODE datatype has the following characteristics:

- It represents the intersection of all defined dimensions.
- It stores a single, unique, linear, encoded value in a single column in a spatial table.
- It returns all the original values when the HHCODE is decoded.
- It holds all original values to their full defined precision.
- It contains the spatial data as a binary stream, which is stored as a raw datatype field.
- It maintains the spatial organization of the data in the HHCODE itself.
- Each dimension of an HHCODE is a single continuous range.
- To add dimensions, remove dimensions, or change the definition of an HHCODE column, the table must be dropped and re–created using the new HHCODE definition.

Restrictions

The HHCODE column is subject to the following restrictions:

- The dimensions of an HHCODE must be numeric.
- The HHCODE column must be defined as datatype RAW(255).
- A maximum of 32 dimensions can be defined.

Creating HHCODE Columns

Operations for creating and decoding HHCODE data and performing dimensional operations are supplied by SQL and PL/SQL extensions. When creating HHCODE, the following characteristics must be specified:

- dimensions
- dimension range
- scale

Dimensions

The dimensions of an HHCODE must be defined as numeric values. The dimensions can be coordinate values such as x, y, and z, or any other numeric data you define. For each dimension included in the HHCODE, you must specify the dimension range and scale.

Dimension Range

The dimension range is the extent defined by a minimum and maximum value for each dimension. The range for each dimension should be closely calculated to handle all current and future data values. If the range exceeds the number of data values that fall within its range, space is wasted and the table is not efficiently managed.

All HHCODE values within a spatial table column must share the same dimension definition. If you have spatial data with differing dimensions or dimension definitions, the data must be placed in separate columns. For example, an HHCODE column that includes a time element must use a the same unit of measure for each row; you cannot measure time in minutes for some rows, and hours for others.

Scale

The scale is the number of significant digits to the right of the decimal at which you store spatial data. The scale can be the same as the original data, or it can be lower. A scale of zero (0) can also be defined for use with integer values. All data for the same dimension should share the same precision.

Defining an HHCODE Column

Use the MD_DDL.ADD_HHCODE_COLUMN procedure to define an HHCODE column in a spatial table. For information on the MD_DDL.ADD_HHCODE_COLUMN procedure see Chapter 7, "SD*SQL Packages."

Attribute Column

An attribute column is a column in a spatial table that describes or is associated with the HHCODE data, but is not part of the HHCODE column.

Data is usually defined as attribute data under the following conditions:

- if it is used as selection criteria in a WHERE clause
- if it is used to perform computations in a SELECT statement
- if it supplies extra information

If queries on a table are frequently based on a characteristic of the data, such as year or latitude, the characteristic should be part of the HHCODE column. If, however, the characteristic is simply descriptive information, it should be classified as an attribute column.

Characteristics

Attribute column definitions can be modified without re–creating the spatial table.

Spatial Table Types

The Spatial Data Option uses the following spatial table types:

- partitioned table
- non–partitioned table

The spatial table type you choose depends on the amount of data you expect to store in the table and on how you intend to access the data. Create a partitioned table to store all or a majority of the data for the database, or if you want to use the advantages offered by table partitioning. Create a non–partitioned table to store supporting data that may need to be cross–reference with data in a partitioned table.

For a description of partitioning with the Spatial Data Option see the ”Partitioning” section in this chapter.

Characteristics

Both spatial table types share the following characteristics:

- They contain at least one HHCODE column.
- They are registered in the Spatial Data Option data dictionary.
- Standard SQL can be used to perform all DML commands.
- Constraints can be defined.
- The SQL ALTER TABLE command cannot be used to alter the definition of the HHCODE column in a spatial table.

  To add dimensions, remove dimensions, or change the definition of an HHCODE column, the table must be dropped and re–created using the new HHCODE definition.

Restrictions

Both spatial table types share the restriction that LONG and LONG RAW columns are not supported by the Spatial Data Option utilities, but can be defined for the spatial table.

Partitioned Table

A partitioned table is a set of Oracle7 tables comprised of a spatial table and a set of tables known as partitions that are treated conceptually as a single table. Each partition is an Oracle7 table and contains data that is spatially related. Each child partition is identical in structure to the spatial table and contains the same number of columns, column datatype and size specifications, and HHCODE dimension definitions.

Partitioned Table Parameters

When creating a partitioned table, the following parameters are defined:

- partition key
- high water mark
- compute mode

Partition Key: The partition key is the HHCODE column used to partition the data. In every partitioned table, one HHCODE column must be identified as the partition key.

The following considerations apply when defining the partition key:

- Only one partition key column is allowed for each partitioned table.
- Once data is loaded into the table, you cannot choose another partition key without re–creating the table and reloading the data.
- To optimize partitioning, differentiate the partition key as much as possible.

  For example, if the maximum number of records allowed in a partition is set to 10,000, but 15,000 identical partition key values exist, partitioning is not optimized. By adding a time dimension or another differentiator you can insure that each partition key value is unique.

High Water Mark: The high water mark defines the maximum number of records to be stored in a partition of a partitioned table before it is subdivided by SQL*Loader. The high water mark is defined when the spatial table is registered in the Spatial Data Option data dictionary. It can be modified after a spatial table is created.

Note: The partition subdivides into a new set of partitions automatically if you load data using the SD*Loader utility. You can also subdivide a partition manually.

For information on the SD*Loader utility, see Chapter 5, "Utilities."

The following considerations apply when defining the high water mark:

- If you do not specify a high water mark, the spatial table is registered in the Spatial Data Option data dictionary as non–partitioned.

- If the high water mark is exceeded while loading data using SD*Loader, the table automatically partitions and subdivides to another level.
- If the last partitioning level is reached and the high water mark is exceeded while loading data, the last partition grows beyond the high water mark up to the maximum size specified by the MAXEXTENTS parameter.

Compute Mode: The compute mode is the method used by SD*Loader to determine the number of rows in a partition when data is loaded into the partition. The compute mode is used to determine if the high water mark has been reached. The compute mode is defined as EXACT or ESTIMATE when the partitioned table is registered in the Spatial Data Option data dictionary.

The following considerations apply when defining the compute mode:

- When ESTIMATE is specified, the ANALYZE command is executed in the following format to determine the approximate number of rows:

  ```
  SQL> ANALYZE partition_name ESTIMATE STATISTICS;
  ```

- When EXACT is specified, a SELECT statement is executed in the following format to determine the exact number of rows:

  ```
  SQL> SELECT COUNT(*) FROM partition_name;
  ```

  Note: In general, a compute mode defined as ESTIMATE loads data faster, and is sufficiently accurate.

Characteristics

The following characteristics apply to a partitioned table:

- It must have at least one HHCODE column used to partition the data, defined as the partition key.
- It must have a predefined maximum number of records for a partition, defined as the high water mark.
- It subdivides when the number of records in a partition exceeds the high water mark when data is loaded using the SD*Loader utility.

Restrictions

Partitioned tables have the following restrictions:

- Additional steps are required when exporting or importing entire tables composed of multiple partitions.
- Individual records are inserted, updated, or deleted using SD*SQL packages.

- Triggers must be executed using dynamic SQL because a user must check the Spatial Data Option data dictionary using the MD_PART.GET_PARTITION_NAME function to determine the actual table or partition name on which to execute the statement, and then generate and dynamically execute the command.

  For information on the MD_PART.GET_PARTITION_NAME function, see Chapter 7, "SD*SQL Packages."

Non–Partitioned Table

A non–partitioned table is a single table in an Oracle7 database. Non–partitioned tables never subdivide, but continue to grow to the maximum size defined when the table is created, in the same fashion as standard Oracle7 tables. Future growth should be taken into consideration when defining a spatial table as a non–partitioned table.

Characteristics

The following characteristics apply to a non–partitioned table:

- Referential and data integrity constraints are supported.
- Triggers are supported in the same way as for standard Oracle7 tables.

Creating a Spatial Table

The syntax for the command to create a standard Oracle7 table with HHCODE and attribute columns is as follows:

```
CREATE TABLE tablename(columnname datatype) ...
   [,columnname datatype])
```

tablename	is the name of the table to be created. The maximum length is 19.
columnname	is the name of the column.
datatype	is the datatype of the column. HHCODE columns must be defined as datatype RAW(255).

See the *Oracle7 Server SQL Reference* for details on the CREATE TABLE command.

Usage Notes

Consider the following points when using this command:

- Define all attribute columns in the same way columns are defined in a standard Oracle7 table.
- When defining partitioned tables, set the following STORAGE clause parameters:
  - Set the INITIAL parameter to the smallest possible value; The minimum value is the size of 2 Oracle data blocks.
  - Set the MINEXTENTS parameter to 1.
- Create the table in the tablespace where you want the spatial data to reside.

Registering a Spatial Table

Use the MD_DDL.REGISTER_MD_TABLE procedure to register a spatial table. A table must be registered to be listed in the Spatial Data Option data dictionary. For information on the MD_DDL.REGISTER_MD_T ABLE procedure see Chapter 7, ”SD*SQL Packages.”

Partitioning

Partitioning is a technique that takes advantage of the HHCODE structure to sort and store spatial data in multiple tables called partitions. Table partitioning is based on the same principle of recursive decomposition of space that is used in HHCODE creation.

The initial partition or table is referred to as the root partition. The root partition never contains data and never subdivides. Parent partitions subdivide dynamically and automatically when you insert data using SD*Loader, according to the predefined maximum value, the high water mark. At each level of division there is a parent partition and associated child partitions. During subdivision, data is moved from the parent partition to the child partitions, and the parent partition is dropped. Storage parameters for child partitions are inherited from the root partition, and can be changed at any time.

Partitions that do not contain data are not created, but are inferred; these partitions exist only logically, not physically. If data that falls within the data space represented by an inferred partition is loaded later, the partition is created at that time.

Partitioning Process

In the partitioning process, at each subdivision data is subdivided into up to 2^n partitions, where n is equal to the number of dimensions encoded in the HHCODE. The partitioning process continues until all the data is loaded; the number of partitions that ultimately exist depends on the quantity and density of data. The optimal maximum size of each partition is determined according to the grouping of the data that belongs in the spatial table.

Subdivision occurs on one of the following ways:

- automatically using SD*Loader
- manually using the SD*SQL procedure MD_PART.SUBDIVIDE_PARTITION

For information about the SD*Loader utility, see Chapter 5, "Utilities."

For information about the MD_PART.SUBDIVIDE_PARTITION procedure, see Chapter 7, "SD*SQL Packages."

Advantages

Partitioning provides the following advantages:

- It limits the number of tables a query must consider when searching for the requested data.
- It minimizes the amount of data to be scanned once a table has been identified as containing query data.

- Related data is stored in proximity for optimal retrieval performance.
- The database designer does not need to predict exactly how much the volume of data will grow in the future.
- It optimizes the volume of data in a partition based on data density.

Figure 1 – 4 illustrates the partitioning process.

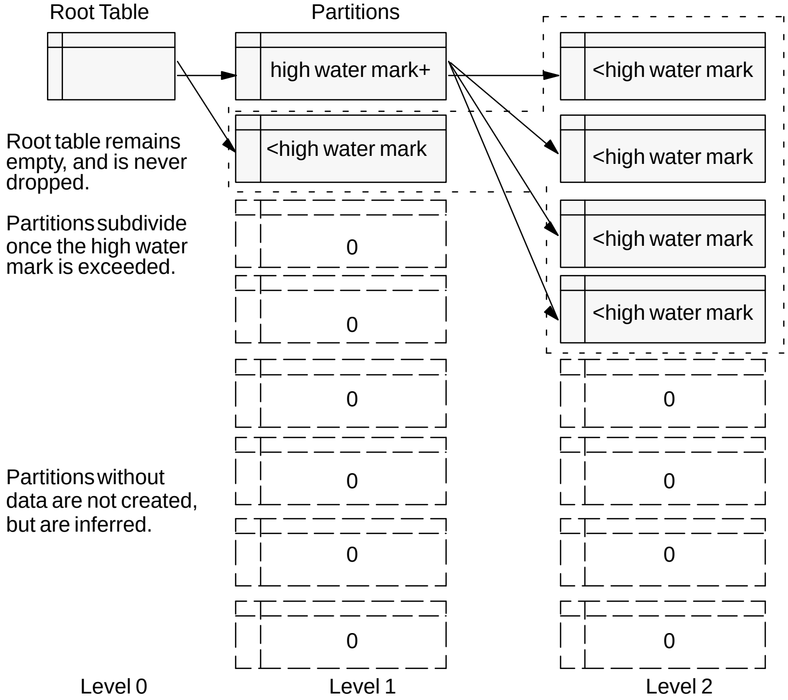

Figure 1 – 4 Table Partitioning Process

As the data is loaded using SD*Loader and the high water mark is exceeded for a partition, it subdivides into another set of up to 2^3 partitions. The parent partition is dropped, and only the partitions that actually hold data are created. The final set of partitions is made up of those tables, at any given level, that still contain data.

In Figure 1 – 4 and Figure 1 – 5, the partitions that exist after loading are surrounded by a dashed line.

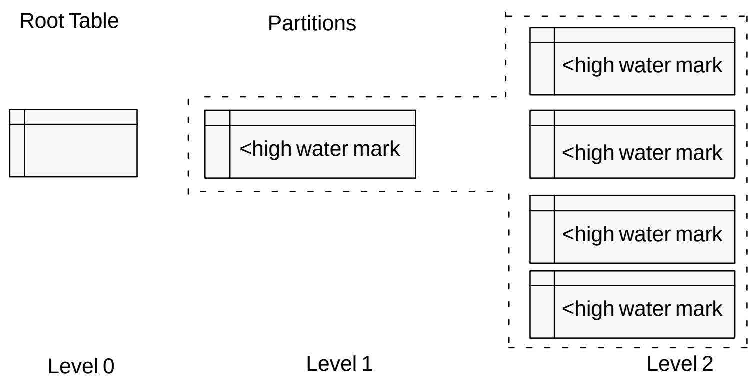

Figure 1 – 5 Partitions Remaining after Data Loading

Maximum Number of Partitions

The maximum number of partitions for a partitioned table is as follows:

```
2 * number_of_dimensions * maximum_number_of_levels
```

Spatial Data Option Data Dictionary

The Spatial Data Option data dictionary is a set of tables owned by the database user MDSYS. An extension to the Oracle7 data dictionary, it automatically maintains information about spatial tables, columns, and partitions. The Spatial Data Option data dictionary is created during the installation process of the Spatial Data Option. All non–spatial attribute information is maintained in the Oracle7 data dictionary.

As with the Oracle7 data dictionary, the Spatial Data Option data dictionary has public views that provide extensive information to users about spatial tables, columns, and partitions. These views join information in the Spatial Data Option data dictionary and the Oracle7 data dictionary. When executing a query against a partitioned table, a user queries the Spatial Data Option data dictionary to find all intersecting partitions, and then queries the partitions themselves.

For a list of Spatial Data Option data dictionary views, see Appendix A, "Spatial Data Option Data Dictionary Reference."

Note: Spatial Data Option tables must be created and registered to be listed in the Spatial Data Option data dictionary.

Partition View

Although the partition view in the Spatial Data Option data dictionary is not an index, it provides the functionality of an index by improving query performance, and serving as the access mechanism to the spatial data stored in partitioned tables. In the Spatial Data Option data dictionary, there is one entry per partition, not per data record. Because access is initially performed at partition level, not record level, the data access time is greatly reduced.

For a description of partitioning and the Spatial Data Option, see the "Partitioning" section in this chapter.

Creating a Partition Map

A partition map is a list of HHCODEs defining the extent of a data set. A partition map defines where the data is located, and determines how dense the data is. The following views in the Spatial Data Option data dictionary provide the required information to create a partition map:

- USER_MD_PARTITIONS
- ALL_MD_PARTITIONS

Both views share the column COMMON_HHCODE. This column returns the section that all HHCODES within a particular partition have in common. By using the SD*SQL HHCELLBNDRY function on the COMMON_HHCODE column, you can determine the bounding extents for each partition.

For a description of the SD*SQL HHCELLBNDRY function, see Chapter 6, "SD*SQL Kernel Functions,"

CHAPTER

2

Converting and Loading

This chapter contains information about converting source data and loading the data into spatial tables. The following topics are included:

- data conversion and loading process
- spatial load format files
- converting data
- loading data

Data Conversion and Loading Process

Converting and loading data from different formats into spatial tables involves the following processes:

- converting the data from its existing format to the format required for loading into spatial tables
- loading data into spatial tables using SD*Loader

The source data can have the following formats:

- homogeneous data files, which are fixed–length ASCII or binary files with records of one layout only
- complex or variable–length format files, such as existing proprietary data formats
- standard Oracle7 tables
- a combination of sources, such as complex data files and standard Oracle7 tables

All source data is converted into spatial load format (SLF), which is the format required by the SD*Loader utility to load data into spatial tables.

Figure 2 – 1 illustrates the general data conversion and loading process for different types of source data.

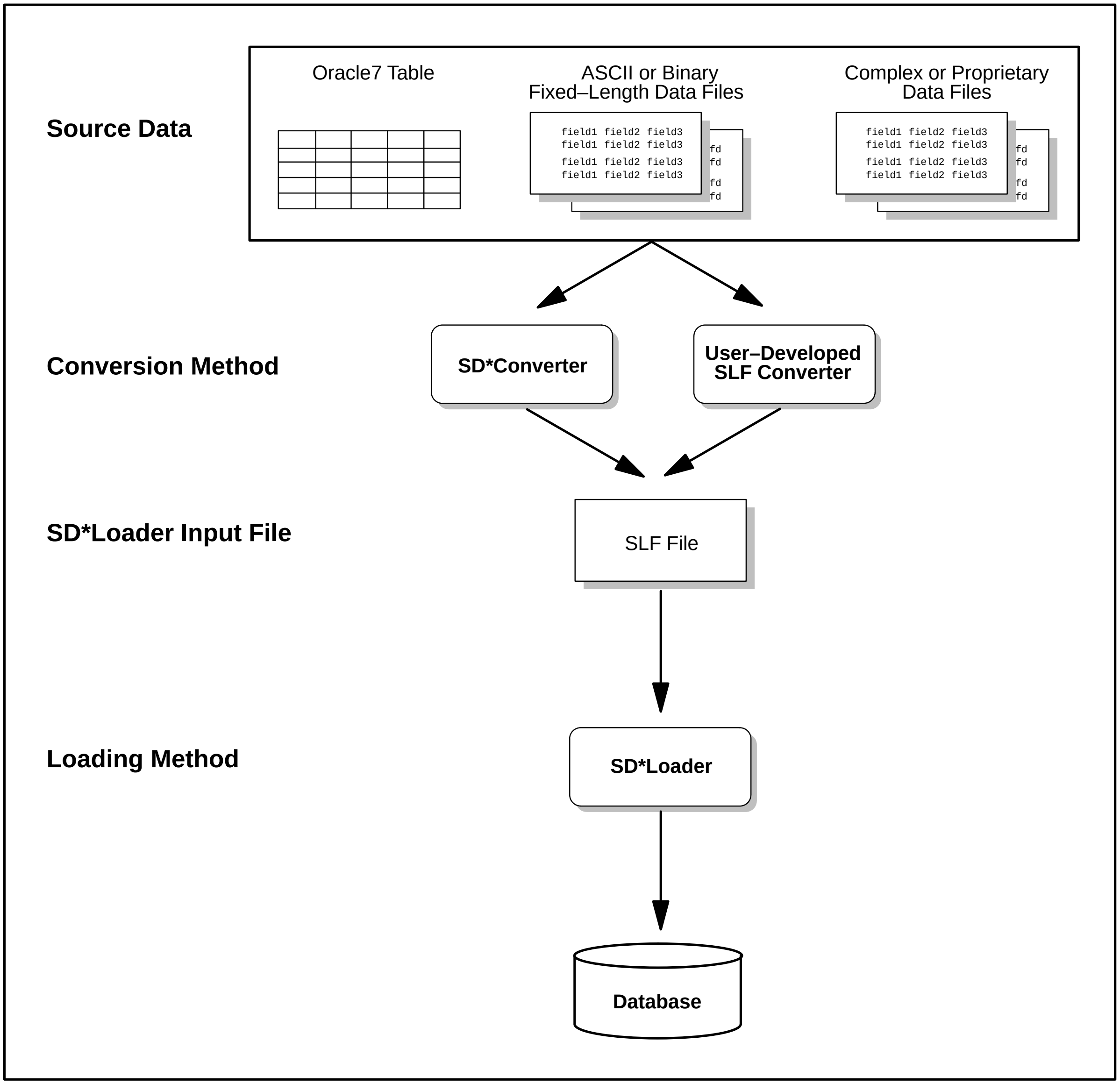

Figure 2 – 1 Data Conversion and Loading Process

Considerations for Conversion Paths

Figure 2 – 1 is a general overview of the conversion and loading process. There are many potential conversion paths that can be taken, depending on the type of source data, the amount of source data, and the resources available. The following considerations should be kept in mind when determining your individual conversion path:

- If the data is in fixed–length ASCII or binary data files or standard Oracle7 tables, SD*Converter can be used directly to convert the data.

  Although it is also possible to create a user–developed SLF converter for fixed–length ASCII or binary data files, SD*Converter is convenient and therefore recommended.

- If the data is in complex or proprietary data files, SD*Converter cannot be used to convert the data directly. One of the following methods can be used:

  – Develop a user–developed SLF converter to convert the data directly into SLF files.

  – Develop a C program or other program for translating multiple complex formats into fixed–length ASCII or binary files with records of one layout only, which can then be converted into SLF using SD*Converter, or inserted into an Oracle7 table.

  If you decide to develop a program for converting complex formats into a format for input into SD*Converter, take into account available resources, including the following:

  – additional space required for temporary files created during the intermediate processes

  – additional time and resources for the programming required

- Conversion and loading processes can be performed on separate machines if it is more efficient to do so. This is because both converting and loading are resource–intensive. Converting is CPU intensive, while loading is I/O intensive. Choose the machine that is most efficient for the process.

Considerations for Sorting

Data sorting can be performed during the conversion or the loading phase, depending on requirements. Some sorting considerations include the following:

- If the data file contains fewer records than the high water mark, then do not sort at conversion time. The table does not require partitioning at load time, and therefore does not require sorting at any stage.
- When fast load times are important, sort at the conversion stage. This is because the sorting process is both I/O and CPU intensive.
- If the data is not sorted at conversion time, SD*Loader sorts if required. If it is known that a sort will be performed, either by specifying YES for the SD*Converter SORT parameter or when sorted automatically by SD*Loader, then the sort should be performed on the machine that performs sorts most efficiently.

Spatial Load Format Files

SLF files are binary files that have been formatted for loading into spatial tables.

SLF files are created either by using the SD*Converter utility or by creating a user–developed SLF converter. These files are then loaded into spatial tables using SD*Loader.

Characteristics

SLF files have the following characteristics:

- They are binary.
- They contain fixed–length records.
- All records share the same layout.
- They are architecture–specific, which means they are only portable between machines of the same hardware platform and operating system.
- They contain a header describing the SLF data.
- They can already be sorted for loading into a partitioned table.

Considerations

Consider the following points before creating SLF files:

- It is generally better practice to create larger, fewer SLF files rather than smaller, more numerous SLF files. When SD*Loader loads only a small amount of the total data each time, it cannot determine in advance the final organization of the partitions being created. This results in more frequent reorganization of the partitions and the data within them, which is time–consuming and resource–intensive.
- SLF files are designed to be temporary, and because they are quite large, should be deleted once the data is loaded.
- Ensure that there is sufficient space in the directory where the SLF file is created to store the temporary files that are created. For instructions for calculating the space required, see ”SD*Converter” in Chapter 5, ”Utilities.”

Restrictions

The following restrictions apply to SLF files:

- All records in the SLF file must share the same layout, which means they have records of one type only.

Warning: SLF files are not portable between machine architectures. If you perform the conversion and loading on different machines, they must be the same hardware and operating system.

Converting Data

There are several methods for converting source data into SLF files. The method used depends on the format of the source data.

Whatever conversion method is used, the resulting data files must be SLF files.

SD*Converter

The SD*Converter utility creates SLF files from source data.

For detailed instructions on using SD*Converter, see "SD*Converter" in Chapter 5, "Utilities."

Input Options

SD*Converter requires a combination of several inputs, depending on the format of the source data. Input options are as follows:

- table control file
- data control file
- Oracle7 data dictionary
- Spatial Data Option data dictionary
- source data file
- standard Oracle7 table

Table Control File: The table control file provides the following information about the spatial table:

- spatial table name
- column information, including column names and datatypes, and partition key column
- dimension information, including dimension name, HHCODE column name, dimension number, dimension range, and scale

The Spatial Data Option data dictionary can be used as an alternative table control input source, if a spatial table has been registered.

Data Control File: A data control file provides the following information about source data that is not in an Oracle7 table:

- ASCII or binary file format
- record length
- column information, including column names and datatypes
- dimension information, including dimension name, HHCODE column name, position, and datatype

Oracle7 Data Dictionary: If the source data is in a standard Oracle7 table, the Oracle7 data dictionary provides the following information about the source data:

- if the Oracle7 table exists
- whether the user has access to the Oracle7 table

When converting from an Oracle7 table, the Oracle7 data dictionary must be used instead of a data control file.

Spatial Data Option Data Dictionary: The Spatial Data Option data dictionary provides the following information about the spatial table:

- if the table is registered as a spatial table
- whether the user has access to the spatial table
- column information, including column names and datatypes
- dimension information, including dimension name, sequence of dimensions, dimension range, and scale
- partition structure, including partition key column and high water mark

Note: To obtain information about the spatial table from the Spatial Data Option data dictionary, you must already have created and registered the spatial table for which you are converting the data.

A table control file can be used as an alternative table control input source, unless the source data is from a standard Oracle7 table.

Source Data File: A source data file provides the data for input into SD*Converter, when the data is not in an Oracle7 table.

Oracle7 Table: A standard Oracle7 table provides the data for input into SD*Converter when the data is not from a source data file.

Input Combinations

The following input combinations are valid for SD*Converter:

Input Combination 1: For ASCII and binary source data files, information about the source data is provided in a data control file, and the description of the spatial table is provided in a table control file, as shown in Figure 2 – 2.

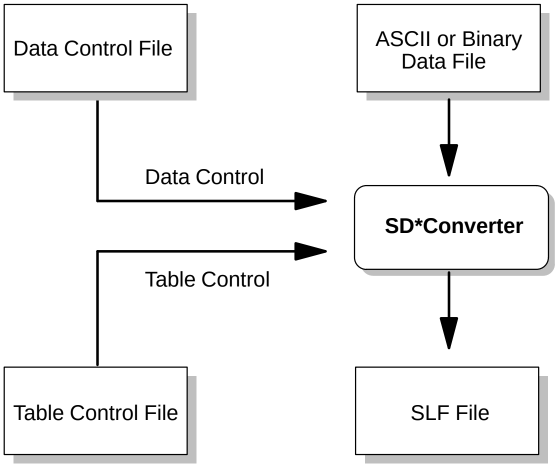

Figure 2 – 2 Input Combination 1

Input Combination 2: For ASCII or binary source data files, information about the source data is provided in a data control file, and the description of the spatial table is provided by the Spatial Data Option data dictionary, as shown in Figure 2 – 3.

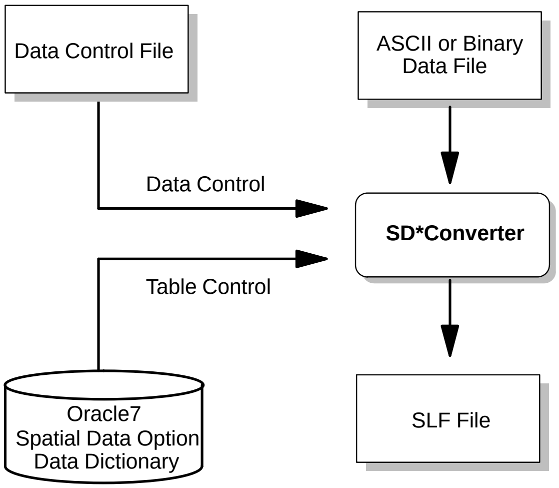

Figure 2 – 3 Input Combination 2

Input Combination 3: For Oracle7 tables, information about the source data is provided by the Oracle7 data dictionary, and the information about the spatial table is provided by the Spatial Data Option data dictionary, as shown in Figure 2 – 4.

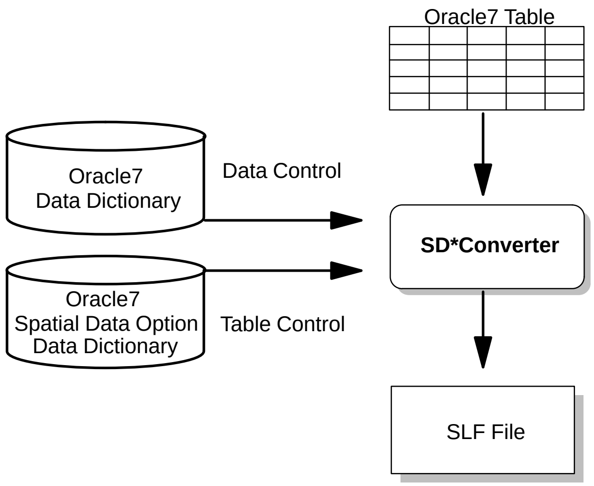

Figure 2 – 4 Input Combination 3

Input Combination 4: For complex or proprietary source data files, the data must first be converted into fixed–length ASCII or binary source data files or Oracle7 tables before they can be converted using SD*Converter, as shown in Figure 2 – 5.

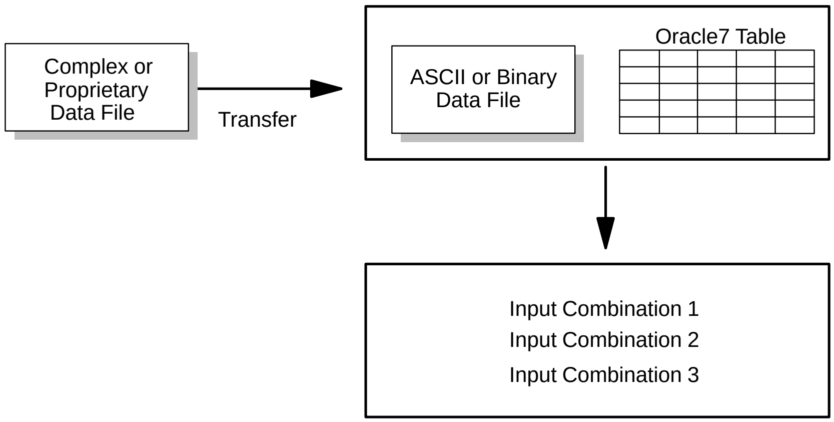

Figure 2 – 5 Input Combination 4

Considerations

Consider the following points before using SD*Converter:

- By using control files you can create SLF files without running Oracle7 while you convert data from other formats into SLF.
- Data can be sorted either during the conversion or the loading process, depending on requirements. For guidelines on selecting whether to sort, see ”Considerations for Sorting” in this chapter.
- When a sort is performed during the SD*Converter process, two temporary files are created. These files are deleted automatically once conversion is complete. Ensure you have sufficient disk space to accommodate these files. For information on calculating disk space, see ”Temporary Files” in this section.

Restrictions

The following restrictions apply when using SD*Converter:

- The source data must be in homogeneous data files, which are fixed–length ASCII or binary files whose records are of one layout only, or standard Oracle7 tables.
- You must use the UNIX TWO_TASK environment variable to enable networking on another machine. You cannot use a SQL*Net string with the username and password.
- The KEYWORD values must be entered with the parameters.

Temporary Files

When converting data using SD*Converter, two temporary files are created during the sort process.

The first temporary file is the same size as the SLF file that is created, and the second temporary file is approximately the size of the partition key HHCODE plus an additional four bytes for each record in the SLF file. These files are created in the same directory as the SLF file and are named as follows:

- **.srt_*unique_name***
- **.tmp_*unique_name***

The calculation used to allocate sufficient disk space for the data conversion process is as follows:

```
SLF file   (SLF file size)
+   temp file #1   (SLF file size)
+   temp file #2   (HHCODE size + 4) * number of records
```

For example, assume the SLF file is 100 Mb in size, contains 100,000 records, and has a partition key HHCODE size of 20 bytes. In this case, the calculations for disk space for the data conversion process are approximately as follows:

```
SLF file   = (100 Mb)
+   temp file #1   = (100 Mb)
+   temp file #2   = (20 + 4) * 100,000 = 2,400,000 bytes
=   202.4 Mb
```

User–Developed SLF Converter

A user–developed SLF file converter can be created when the source data to convert is in complex or proprietary data formats. To do this, you create a C language program that reads the source data and writes it into an SLF file using the SLF library of C functions.

For complete instructions on creating a user–developed SLF converter, see Chapter 8, ”User–Developed SLF Converter.”

Characteristics

The following characteristics apply to any user–developed SLF converter:

- The SLF library file of C functions must be included and linked to the SLF converter. For a description of this file, see ”SLF Library” in Chapter 8, ”User–Developed SLF Converter.”
- Declare and allocate the data structures in the SLF header file. For a description of this file, see ”User SLF Header File” in Chapter 8, ”User–Developed SLF Converter.”
- A source data file must be supplied to the program.
- A table control file or the Spatial Data Option data dictionary information must be provided to the program.

Considerations

Consider the following points before creating an SLF converter:

- Decide whether it is more efficient to sort the data in the converter or during loading with SD*Loader.
- Decide whether it is more efficient to convert the data into ASCII or binary files or standard Oracle7 tables using some other method and then use SD*Converter to create the SLF file.

Restrictions

The following restrictions apply when creating an SLF converter:

- You cannot use LONG and LONG RAW datatypes.
- The data converted into the SLF file must be of one layout only.

Loading Data

After data has been converted into SLF files using either SD*Converter or a user–developed converter, the data is loaded into spatial tables using the SD*Loader utility.

When a spatial table is created, information on how to store the table is provided to the Spatial Data Option data dictionary. SD*Loader checks the Spatial Data Option data dictionary to determine which data to store together, and how much to store together in a single partition before subdividing into other partitions.

Note: You can also use the SD*SQL packages with the SQL INSERT command to insert single records at a time, but SD*Loader is the standard loading method, and must be used for batch loading.

Characteristics

SD*Loader performs the following functions:

- It partitions the SLF data, and sorts if necessary, for storage in multiple partitions under the following conditions:
  - if the volume of data is higher than the high water mark
  - if the data was not sorted during the conversion process
- It loads the partitioned data into the correct partitions, creating new partitions as necessary.
- It updates the Spatial Data Option data dictionary to reflect the new partition structure.

SD*Loader distributes the partitions in a round–robin fashion using all available tablespaces. This can greatly improve extract performance because the data is spread over multiple tablespaces that optimally reside on different physical disks. For more information on tablespace striping, see ”Tablespace Management” in Chapter 4, ”Administration.”

Considerations

The following considerations apply when using SD*Loader:

- When loading multiple files, you can enhance the performance of subsequent loads by setting the high water mark for the initial load lower than the optimum level. This will force the partitioning to occur during the initial load. Reset the optimum high water mark using the MD_DDL.ALTER_MD_TABLE_HWM procedure before performing subsequent loads. For more information on resetting the high water mark, see the MD_DDL.ALTER_MD_TABLE_HWM procedure in Chapter 7, ”SD*SQL Packages.”

- When data is loaded into a partitioned table and the high water mark is reached, only the partitions that contain data are created. The remaining tables in the structure do not exist. To improve loading time, load dense data files first; the partitions that will contain data are created. When the sparse dataset is loaded, the data can immediately be placed in the correct partitions without having to create and subdivide partitions.
- To decrease loading times, you can run multiple sessions at once, provided you have sufficient SGA.
- The SLF file header information must match the spatial table description for the table into which it is being loaded. If any definition is different, the load process terminates.
- Only the values, not the keywords, are required if the parameters are passed in the order in which they appear in the keyword list.
- Ensure that you have sufficient disk space to accommodate the temporary files that are created during a load. For information on calculating sufficient space, see ”Temporary Files” in this section.
- Ensure that you have sufficient space for the temporary tables created during a load. For detailed information on how to determine how much space you need, see ”Calculating Table Size Requirements” in Chapter 4, ”Administration.”
- The SD*SQL procedures can be used with the SQL INSERT command to load individual records at a time, but it is recommended that bulk loads always be performed using SD*Loader.

For information about direct path loads and the Oracle Loader utility, see the *Oracle7 Server Utilities*.

Restrictions

Consider the following points when using SD*Loader:

- You cannot load data of the RAW or LONG RAW datatypes.
- The SLF files must contain records of one layout only.

Temporary Files

Many temporary files are created during a load, depending on the process SD*Loader is performing. The following temporary files are created:

- **ctl.*unique_name***
- **dat.*unique_name***
- **log.*unique_name***

The preceding files are created in the local directory where you execute SD*Loader. Additional temporary files can be created, depending on whether a sort is performed.

When the load finishes, these temporary files are removed. If a load fails, however, some of these files can remain, and must be deleted manually.

Temporary Tables

Many temporary tables are created during partitioning, and are automatically dropped after loading finishes. Ensure that you have sufficient tablespace to accommodate these temporary tables.

If a load fails, some temporary tables may not have been dropped, and must be dropped manually if you are not continuing the load. To determine which tables remain, use the SQL DESCRIBE command to query the USER_MD_EXCEPTIONS and ALL_MD_EXCEPTIONS views to determine the remaining tables. You can then use the MD_PART.CLEAR_EXCEPTION_TABLES procedure to drop the tables, or drop them individually using the MD_DDL.DROP_MD_TABLE procedure. For detailed instructions for using these procedures, see Chapter 7, "SD*SQL Packages." For more information on clearing exceptions in the Spatial Data Option database, see "Exception Conditions" in Chapter 4, "Administration."

Recovery Procedures

Consider the following points about recovery from a failed load:

- SD*Loader has built–in recovery procedures that allow you to proceed with interrupted loads. If you re–execute using the same SLF file name, username, and spatial table name, the program checks to see if this file has completed its load, and if not, prompts you to continue the load or cancel it.
- The MD_LOADER_ERRORS Spatial Data Option data dictionary view contains the current status of a load file. This table should be checked to see if any errors occurred during a load.

CHAPTER

3

Data Manipulation and Query

This chapter presents an overview of data manipulation and query techniques for the Spatial Data Option. The following topics are included:

- Data Definition Language (DDL) commands
- Data Manipulation Language (DML) commands
- spatial data queries
- SD*SQL kernel functions
- inserting, deleting, and updating data in spatial tables
- manipulating data using MD_DML packages

For a detailed task list of how to perform data manipulation and query for the Spatial Data Option, see Chapter 4, "Data Manipulation and Query," *Oracle7 Spatial Data Option Application Developer's Guide.*

Data Definition Language (DDL) Commands

Oracle7 DDL commands are used to define, maintain, and grant permissions on database objects by performing tasks such as creating, altering, and dropping objects and granting and revoking privileges and roles. When using DDL commands the following conditions apply:

- Oracle7 implicitly commits the current transaction after every DDL command.
- Many DDL commands cause Oracle7 to recompile or reauthorize schema objects.
- PL/SQL does not support DDL commands.

In addition to the DDL commands used with standard Oracle7 tables, there are Spatial Data Option DDL commands that are used to modify spatial tables. They are often used with DML commands.

For detailed information on DDL commands, see the *Oracle7 Server SQL Reference* and the *Oracle7 Server Concepts*.

The following tasks are discussed in this section:

- altering a spatial table
- dropping a spatial table

Altering a Spatial Table

The procedures used to alter a spatial table depend on the table type. This section describes how to alter the following spatial table types:

- partitioned table
- non–partitioned table

Partitioned Table

The following characteristics can be modified on a partitioned table:

- attribute column
- compute mode
- high water mark

Table 3 – 1 lists the procedures used to alter the characteristics that can be modified on a partitioned table.

Task	Procedure Name	Description and Considerations
alter all partitions	MD_DDL.ALTER_MD_TABLE	alters all the partitions in a partitioned table. You cannot alter HHCODE columns or add, delete, modify, enable, or disable any table or column constraints in a partitioned table.
alter compute mode	MD_DDL.ALTER_MD_TABLE_CM	changes the way the compute method is determined when data is loaded using SD*Loader.
alter high water mark	MD_DDL.ALTER_MD_TABLE_HWM	alters the high water mark for a partitioned table. The high water mark must already exist. Existing tables are not reorganized when the value of the high water mark is changed, but only when data is subsequently loaded into the partition.
alter attribute column	MD_DDL.ALTER_MD_TABLE	alters the attribute column for a partitioned table.

Table 3 – 1 Altering a Partitioned Table

Non–Partitioned Table

Table 3 – 2 lists the procedure used to modify a non–partitioned table.

Task	Procedure Name	Description and Considerations
alter non–partitioned table	SQL ALTER TABLE	alters a non–partitioned table. You can alter any column except HHCODE columns.

Table 3 – 2 Altering a Non–Partitioned Table

Dropping a Spatial Table

Table 3 – 3 lists the procedure used to drop a spatial table.

Task	Procedure Name	Description and Considerations
drop a spatial table	MD_DDL.DROP_MD_TABLE	removes all related information from the Spatial Data Option data dictionary and, for partitioned tables, drops all associated partitions followed by the spatial table.

Table 3 – 3 Dropping a Spatial Table

If the procedure fails, an application error is raised. If an error occurs, query the USER_MD_PARTITIONS view to see if the table still exists. If the table has not been dropped, rerun the procedure.

Note: You can also drop a spatial table created during an extract procedure using the MD_WEX_DROP_TARGET procedure.

Warning: Never use the SQL DROP TABLE command to drop a spatial table. If you do, the information in the Spatial Data Option data dictionary is not updated and is inconsistent with the state of the database.

Data Manipulation Language (DML) Commands

Oracle7 DML commands are used to query and manipulate data in existing schema objects by performing tasks such as deleting, inserting, querying, and updating data, and performing table locks.

When using DML commands the following conditions apply:

- DML commands do not implicitly commit the current transaction.
- PL/SQL supports DML commands.

The Spatial Data Option provides a set of functions and procedures to extend DML functions to spatial tables. Querying and manipulating data in a spatial table is a multi–step process.

All DML procedures on partitioned tables must be executed using dynamic SQL from any Oracle7 tool. This is necessary because a user must check the Spatial Data Option data dictionary using the MD_PART.GET_PARTITION_NAME function to determine the actual table or partition name on which to execute the statement, and then generate and dynamically execute the command.

For a description of the MD_PART.GET_PARTITION_NAME function, see Chapter 7, "SD*SQL Packages."

For detailed information on DML commands, see the *Oracle7 Server SQL Reference* and the *Oracle7 Server Concepts*.

Spatial Data Queries

The Spatial Data Option provides the ability to perform specialized spatial queries, called extracts, on partitioned tables. Extracts enable you to define an *n*–dimensional area of interest, called a window, and to retrieve all the data that falls within this area. An *n*–dimensional extract is performed as follows:

1. The Spatial Data Option data dictionary is scanned and the partitions that intersect the data window are identified.

2. The identified partitions are scanned if required.

   By definition, if a partition overlaps the window boundaries then some records in the partition are within the window and some are outside it. The only way to determine which records are inside the the window is to scan the partition using the HHIDROWS or HHIDLROWS function.

   If a partition is inside the window boundaries then all rows in that partition are located within the window, and the partition does not need to be scanned.

3. The set of rows meeting the query criteria is extracted. Both spatial and attribute columns are extracted for the set of rows.

   Note: To execute a DML statement on an attribute column of a partitioned table, the statement must be executed on every partition in the table.

Data can be accessed either by standard SQL SELECT statements or through PL/SQL procedural extensions. Both methods make it possible to extract windows of data from a spatial database.

Window Extract Types

The Spatial Data Option can extract spatial data for the following window types:

- range
- proximity
- polygon

Range

A range window is defined by specifying minimum and maximum values of the search range for the dimensions to be queried.

Proximity

A proximity window is defined by specifying a center point and a radius for all desired dimensions. Proximity queries assume that all dimensions in the query have the same scale. If this is not the case, results are not meaningful.

Polygon

A polygon window is defined by specifying a start and end point for each node, in two dimensions, up to a maximum of 124 nodes.

Figure 3 – 1 illustrates the window extract types.

Spatial Windows

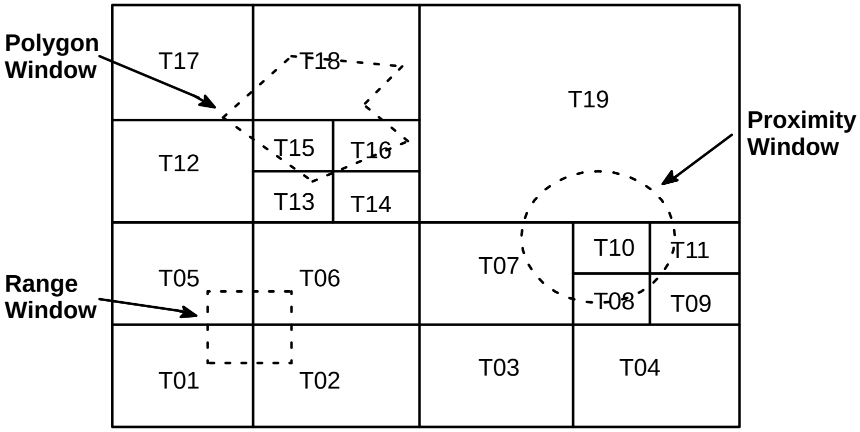

Figure 3 – 1 Window Extract Types

By providing individual access to each dimension of an HHCODE, all extract types can be performed on *n*–dimensional HHCODEs. For example, a two–dimensional search can be performed on a three–dimensional HHCODE simply by specifying the two dimensions of the HHCODE on which to operate.

Performance time for a window query is related to the total number of records in the partitions that intersect the data window, not the total number of records in all of the partitions belonging to the table.

Extracting Data

Spatial data can be extracted using either of the following methods:

- package method
- custom method

The package method requires users to invoke appropriate procedures that are already written to extract spatial data. The custom method requires users to write their own functions, using C or PL/SQL, to extract spatial data.

When deciding whether to use the package or custom method, you should choose the package method if ease of use is most important, and custom method if you want to write a custom application and speed is a priority.

Warning: Once you begin extracting data using either method, you must continue to use that method for the duration of the transaction.

Package Method

Extracting data using the package method uses a set of PL/SQL procedures contained in the MD_WEX package to retrieve spatial data from a partitioned table. The package method is easier to use than the custom method and requires less coding. Unlike the custom method, the package method stores the retrieved data in an intermediate table or view. If you perform queries repeatedly on the same area of interest, it is advantageous to perform analysis on the intermediate, non–partitioned table rather than the larger partitioned table.

Note: The package method requires enough tablespace to hold the extracted data. The extracted data is a duplication of the data from the partitioned table.

A package query involves executing the following steps:

1. The mandatory and optional parameters are set up in the MD_WEX package to perform the following tasks:
   - resetting query specifications
   - setting the SQL filter (optional)
   - setting the target type (optional)
   - setting the HHCODE type (optional)
   - setting the storage clause (optional)
   - setting the target tablespace (optional)
   - setting the dimension list (optional)
   - setting one of the following windows:
     - range window
     - proximity window
     - polygon window

   For a detailed description of the MD_WEX procedures, see Chapter 7, "SD*SQL Packages."
2. The extract procedure is run to perform the extract.

Table 3 – 4 lists the tasks to set up the mandatory and optional parameters in the MD_WEX package.

Task	Procedure Name	Description and Considerations
resetting query specifications	MD_WEX.RESET_GLOBALS	clears previous query specifications
setting the SQL filter	MD_WEX.SET_SQL_FILTER	indicates which source columns to copy to the target object. It can also specify non–spatial window extract constraints.
setting the target type	MD_WEX.SET_TARGET_TYPE	indicates what type of target object you want to create: a non–partitioned table or a view. If data is placed in a table, the records that fall within the window are copied to that table. The records can then be accessed independently of the partitions from which they were extracted.
setting the HHCODE type	MD_WEX.SET_HHCODE_ TYPE	indicates whether the partition key is an *n*–dimensional point HHCODE, or a two–dimensional line, represented by a four–dimensional HHCODE
setting the storage clause	MD_WEX.SET_ STORAGE_CLAUSE	defines the storage parameters to apply when the target object is a table
setting the target tablespace	MD_WEX.SET_ TARGET_TABLESPACE	defines the target tablespace. Default is the tablespace defined in the user profile.
setting the dimension list	MD_WEX.SET_ DIMENSION_LIST	identifies the dimensions of the partition key column to use for the window extract if not all the dimensions in the partition key are queried
setting the range window	MD_WEX.SET_RANGE_ WINDOW	defines the range window boundaries. A range window is a list of lower boundary and upper boundary values provided for each dimension.
setting the proximity window	MD_WEX.SET_ PROXIMITY_WINDOW	defines a proximity window. It is specified by defining a center point and a radius around that point.
setting the polygon window	MD_WEX.SET_ POLYGON_WINDOW	defines an *n*–point two–dimensional polygon window. Polygon windows are only valid for two dimensions.
performing an extract	MD_WEX.EXTRACT	performs the actual window extract operation after all mandatory and optional parameters have been set
deleting an extracted table or view	MD_WEX.DROP_TARGET	drops the table or view created by the extract when it is no longer required

Table 3 – 4 Setting Up MD_WEX Package Parameters

For a detailed description of MD_WEX procedures, see Chapter 7, ”SD*SQL Packages.”

Custom Method

Extracting data using the custom method requires you to explicitly specify the operations required to retrieve spatial data from a table. The custom method offers greater flexibility and control.

The following processes are used to perform a custom query:

1. All the partitions that are INSIDE, OVERLAP, EQUAL, or ENCLOSE the window are identified by accessing the Spatial Data Option data dictionary.
2. This set of partitions is queried to obtain the records that fall within the window of data.

The preceding processes require use of either point HHCODE functions or line HHCODE functions. For syntax and a description of all HHCODE functions, see Chapter 6, ”SD*SQL Kernel Functions.”

The following functions are used for point data:

HHIDPART	is a function that is used during the extraction of point data to identify the partitions that enclose, are enclosed by, overlap, are equal to, or are outside the query window.
HHIDROWS	is a function that is used during extraction of point data to identify the records that are inside, outside, or on the boundary of the query window.

The following functions are used for line data:

HHIDLPART	is a function that is used during the extraction of line data to identify the partitions that enclose, are enclosed by, overlap, are equal to, or are outside the query window.
HHIDLROWS	is a function that is used during the extraction of line data to identify the records that are inside, outside, on the boundary of, or overlap the query window.

HHIDPART and HHIDLPART are used during the partition identification process. HHIDROWS and HHIDLROWS are used during the record identification process.

Query Modification

HHCODE functions and reordering, regrouping, and modifying the WHERE clause can be used alone or together to achieve efficient data retrieval. For more information on query modifiers, see the *Oracle7 Spatial Data Option Application Developer's Guide*.

Extracting Large Amounts of Data

If a large amount of data is extracted by creating a view, an error message can be returned. This occurs because Oracle7 supports views with a maximum text length of 64 Kb.

If an error message is returned because a view length exceeds 64 Kb, the following procedures can be used to successfully extract the data:

- Data can be extracted using the MD_WEX package.
- Data can be extracted using subqueries, not by creating a view.
- Data can be extracted directly from the table, not by creating a view.
- A series of extracts can be performed that access smaller windows of data.

SD*SQL Kernel Functions

SD*SQL is a SQL and procedural extension to Oracle7 for the management and retrieval of spatial data. The SD*SQL kernel functions enhance the standard functionality of the DELETE, INSERT, SELECT, and UPDATE commands.

The SD*SQL functions can be run from both SQL and PL/SQL. They are used for the same purpose as other Oracle7 SQL and PL/SQL functions; the only difference is that SD*SQL functions operate on spatial data. The syntax to call an SD*SQL function differs depending on whether it is called from SQL or PL/SQL.

The following tasks are performed using SD*SQL kernel functions:

- creating and decoding HHCODE
- obtaining information about HHCODE
- finding HHCODE commonalities
- sorting and grouping HHCODEs
- converting dates and times in HHCODE
- performing calculations on HHCODE

For a detailed description of the SD*SQL functions, see Chapter 6, "SD*SQL Kernel Functions."

Creating and Decoding HHCODE

The following set of functions provide access to the individual dimensions of the HHCODE. Using these functions, you can create or encode an HHCODE from spatial data values, decode an HHCODE to return the original spatial values, and compose or collapse dimensions from an HHCODE to create a new HHCODE.

HHCELLBNDRY	is a function that returns the minimum and maximum of one of the dimensions of a quadrant enclosing an HHCODE to a specified level of resolution.
HHCOLLAPSE	is a function that removes one or more dimensions encoded in an HHCODE to create a new HHCODE with fewer dimensions.
HHCOMPOSE	is a function that builds a new HHCODE using the dimensional information already encoded in an HHCODE.

HHDECODE	is a function that retrieves the original dimensional value for a specific dimension, in a specified dimension range, of an HHCODE.
HHENCODE	is a function that encodes an HHCODE from original dimensional data.
HHSUBSTR	is a function that returns a portion of an HHCODE based on the start and end resolution levels.

Obtaining Information about HHCODE

The following set of functions provides information about the HHCODE. Some of the functions provide information from the Spatial Data Option data dictionary. Other functions provide information about the internal structure and external representation of an HHCODE. This information is helpful when designing effective HHCODEs.

HHBYTELEN	is a function that returns the number of bytes required to store an HHCODE.
HHLENGTH	is a function that returns the maximum number of levels of resolution in an HHCODE.
HHLEVELS	is a function that returns the number of levels of resolution encoded for a particular dimension range and scale. It is the inverse of HHPRECISION.
HHNDIM	is a function that returns the number of dimensions in an HHCODE.
HHPRECISION	is a function that returns the number of digits of scale for a given range and number of levels. It is the inverse of HHLEVELS.

Finding HHCODE Commonalities

The following set of functions provides ways to determine what information is held in common between two HHCODEs.

HHCOMMONCODE	is a function that returns the common code between two HHCODEs, representing the quadrant enclosing them both.
HHMATCH	is a function that compares two HHCODEs and returns the number of matching levels of resolution, starting from the first level.

Sorting and Grouping HHCODE

The following set of functions is used to sort and group HHCODE data. The functions are combined with the SQL ORDER BY and GROUP BY functions within a query. The trailing portion of an HHCODE column contains structural information about the HHCODE that affects the sorting of HHCODEs if the HHCODEs have differing lengths. These functions strip off the trailing portion so that the sort order is not affected.

HHGROUP	is a function that is used in a GROUP BY clause to group the HHCODE data in the table.
HHORDER	is a function that is used in an ORDER BY clause to properly sort HHCODE data.

Converting Dates and Times in HHCODE

The following set of functions converts date values when a time or date is used as an HHCODE dimension.

HHCLDATE	is a function that returns the calendar date from a Julian date. It is the inverse of HHJLDATE.
HHJLDATE	is a function that returns a Julian date from a calendar date. It is the inverse of HHCLDATE.

Performing Calculations with HHCODE

The following set of functions performs substring and distance calculations on the HHCODE.

HHCELLSIZE	is a function that returns the area or volume of a cell at a given level of resolution.
HHDISTANCE	is a function that applies a Euclidean or Manhattan distance calculation to return the distance between two HHCODEs.

Inserting, Deleting, and Updating Data

Inserting Data

The procedures used to insert data into spatial tables vary according to whether the table is partitioned or non–partitioned. These procedures are not necessary if you load data using the SD*Loader utility.

Partitioned Table

The following procedures are used to insert data into a partitioned table:

1. The partition key value is generated. The original spatial data is generated into an HHCODE using one of the following functions:

   - MD_DML.GENHHCODE
   - HHENCODE

   A partition key value is specified because it determines into which partition the row is inserted.

2. The partition name into which data will be inserted is determined using the MD_PART.GET_PARTITION_NAME function.

   The MD_PART.GET_PARTITION_NAME function also returns the status of the partition.

3. Any exception conditions are resolved, including any of the following partition exception conditions:

   3.1 If the partition does not exist, it is created using the MD_PART.CREATE_INFERRED_PARTITION procedure.

   3.2 If the partition exists, but the high water mark is exceeded, the partition can be subdivided. Even though data can still be inserted after the high water mark is exceeded, a better solution is to resolve the exception condition.

4. A SQL INSERT statement is generated and executed using the SQL INSERT command.

   If a block of records will all be located in the same partition, an array INSERT can be performed.

5. Optional. If the number of records in the partition exceeds the high water mark, the partition can be manually subdivided.

   If the SD*Loader utility is used subsequently to insert data into a partition whose high water mark is exceeded, the utility subdivides the partition during the load process.

Note: Dynamic SQL is used to insert data in spatial tables. To insert data using Pro*C, a PL/SQL block is included in the Pro*C program.

The process flow diagram in Figure 3 – 2 illustrates the execution of an INSERT statement on a partitioned table:

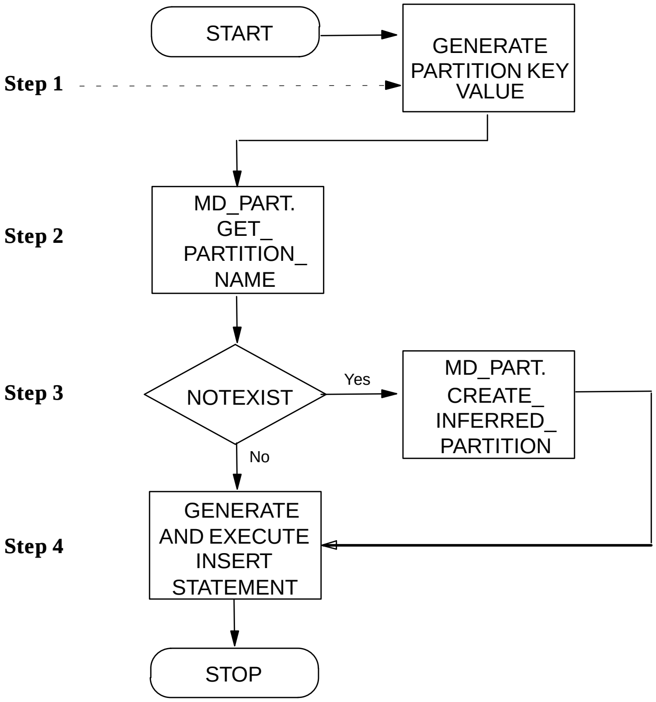

Figure 3 – 2 INSERT Process: Partitioned Table

Non–Partitioned Table

The following procedures are used to insert data into a non–partitioned table:

1. Data is converted for input to HHCODE using the MD_DML.GENHHCODE function or the HHENCODE function.

   For information on the MD_DML.GENHHCODE function, see Chapter 7, "SD*SQL Packages." For information on the HHENCODE function, see Chapter 6, "SD*SQL Kernel Functions."

2. The standard SQL INSERT command is used to insert the new data.

A non–partitioned spatial table uses the same commands to insert data as a standard Oracle7 table.

Deleting Data

The procedures used to delete data from a spatial table vary according to whether the table is partitioned or non–partitioned.

Partitioned Table

The following procedures are used to delete data from a partitioned table.

1. If the partition key is not included in the WHERE clause, all partitions are accessed, and the SQL DELETE statement is generated and executed for each.
2. If the partition key is included in the WHERE clause, the partition is located and a SQL DELETE statement is generated and executed on the partition.
3. If the partition does not exist, then there is nothing to delete, and the operation is complete.
4. A SQL DELETE statement is generated and executed using the SQL DELETE command.

Note: Dynamic SQL is used to delete data in spatial tables. To delete data in Pro*C, a PL/SQL block is included in the Pro*C program.

The process flow diagram in Figure 3 – 3 illustrates the execution of a SQL DELETE statement on a partitioned table:

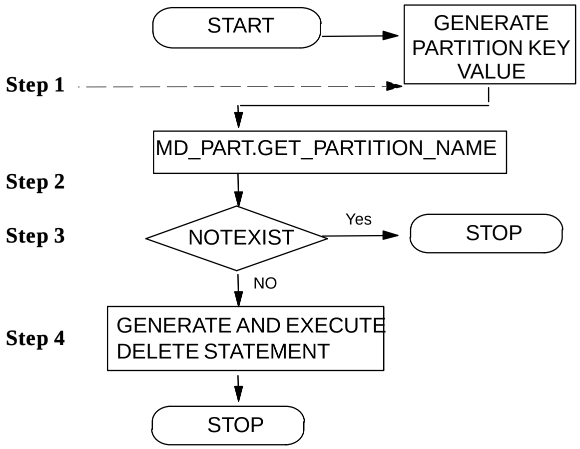

Figure 3 – 3 DELETE Process: Partitioned Table

Non–Partitioned Table

To delete data from a non–partitioned table, the standard SQL DELETE command is run. A non–partitioned spatial table uses the same commands to delete data as a standard Oracle7 table.

Updating Data

The procedures used to update data in a spatial table vary according to whether the table is partitioned or non–partitioned. The following are possible update scenarios for spatial tables:

- updating data without updating the partition key column of a partitioned table
- updating data in the partition key column of a partitioned table
- updating a non–partitioned table

Partitioned Table: Update without Updating Partition Key Column

The following procedures are performed to update data on columns except for the partition key in a partitioned table.

1. If the partition key is not included in the WHERE clause, all partitions are accessed, and the SQL UPDATE statement is generated and executed for each.
2. If the partition key is included in the WHERE clause, the partition is located and a SQL UPDATE statement is generated and executed on the partition.
3. If the partition does not exist, then there is nothing to update, and the operation is complete.
4. The SQL UPDATE statement is generated and executed using the SQL UPDATE command.

The process flow diagram in Figure 3 – 4 shows the execution of an UPDATE statement on a partitioned table that does not update the partition key column.

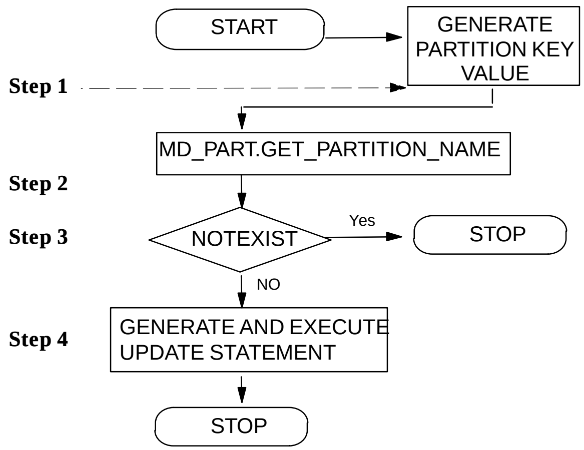

Figure 3 – 4 UPDATE Process: Partition Key Not Updated

Partitioned Table: Updating Partition Key Column

If data is updated in a spatial table and the partition key column is one of the columns to be updated, the partition is located, and a SQL UPDATE statement is generated and executed on the partition.

Because the partition key column determines partitioning, updating the partition key column requires extra reorganization. Partitions and partition key columns is checked and possibly created to ensure partitioning continues in a consistent manner.

The following procedures are performed to update the partition key column in a partitioned table.

1. If the partition key is not included in the WHERE clause, all partitions are accessed.
2. If the partition key is included in the WHERE clause, the partition is located.
3. If the partition does not exist, then there is nothing to update, and the operation is complete.
4. All partitions are accessed, and the SQL UPDATE statement is generated and executed for each partition that includes the new partition key value.
5. The partition is located.
6. If the partition does not exist, an inferred partition is created.
7. If the old partition key is the same as the new partition key, a SQL UPDATE statement is generated and executed for the partition key column and, if required, for attribute columns.
8. If the old partition key is different than the new partition key, the record is moved to its new partition.
9. If attribute columns as well as the partition key column are being updated, the SQL UPDATE statement is generated and executed using the SQL UPDATE command.

The process flow diagram in Figure 3 – 5 illustrates the execution of a SQL UPDATE statement on the partition key column of a partitioned table.

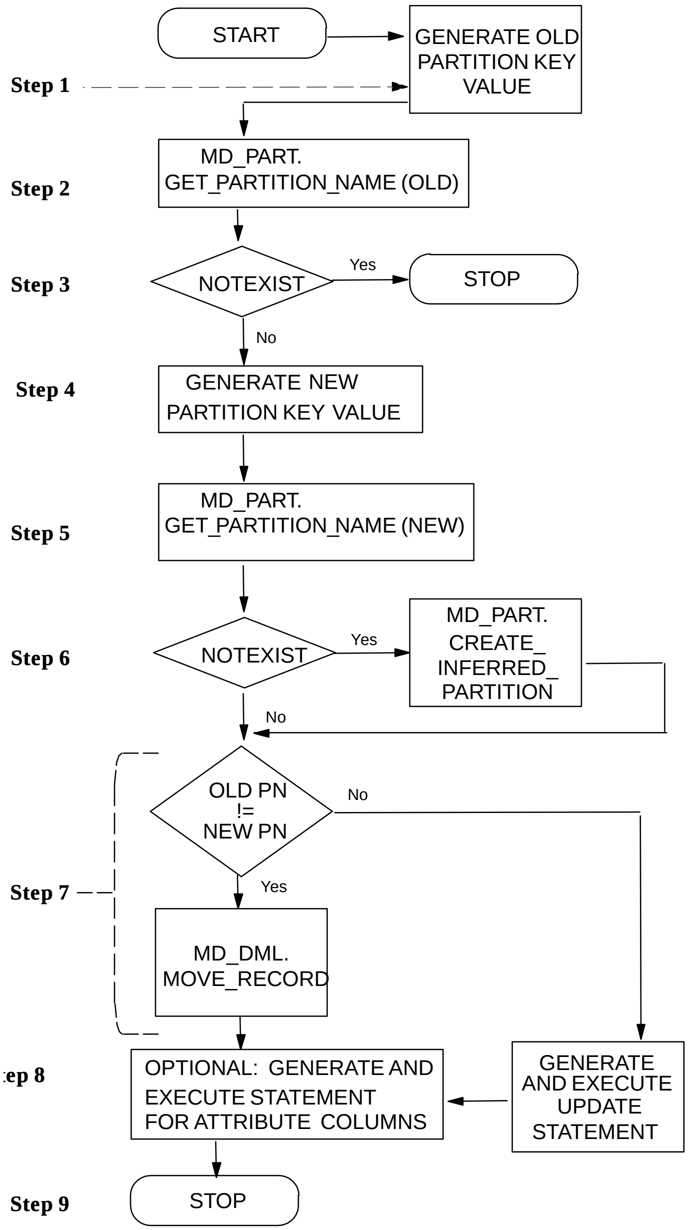

Figure 3 – 5 UPDATE Process: Partitioned Table, Partition Key Updated

Non–Partitioned Table

To perform an update on a non–partitioned table, any original data is converted to an HHCODE, and then a standard SQL UPDATE procedure id executed on the table. A non–partitioned spatial table uses the same commands to update data as a standard Oracle7 table.

Manipulating Data Using MD_DML Package

When manipulating data with the Spatial Data Option, partitioned tables have different considerations than non–partitioned tables. Because partitioned tables are composed of multiple partitions, special procedures are required to perform DML operations. DML operations to manipulate data in spatial tables are a combination of PL/SQL procedures to maintain partitioned tables, move records, and lock tables; and SQL INSERT, UPDATE, and DELETE statements using dynamic SQL. In contrast, non–partitioned tables are treated exactly like standard Oracle7 tables for DML operations, and use standard SQL commands.

The MD_DML PL/SQL package is used with the INSERT, UPDATE, and DELETE DML operations to manipulate data in partitioned tables.

For information on the MD_DML package, see Chapter 7, "SD*SQL Packages."

The structure of a partitioned table makes it impossible to determine beforehand which partitions will be involved in a DML operation. For this reason, DML operations are executed using dynamic SQL. The MD_DML functions can be invoked as a PL/SQL block from within Oracle7 tools such as SQL*Plus, Server Manager, or Pro*C. DML operations are implemented using dynamic SQL with the precompilers, OCI, or PL/SQL.

MD_DML functions are used to perform the following tasks:

- moving a record
- locking a table
- generating an HHCODE

Table 3 – 5 lists tasks performed using MD_DML functions:

Task	Function Name	Description and Considerations
moving a record	MD_DML.MOVE_RECORD	updates and, if necessary, moves the partition key column of a record in a partitioned table. This procedure locates the partition containing the record, updates the HHCODE column, and moves the record to the new partition if the new HHCODE value is located in a different partition.
locking a table	MD_DML.LOCK_MD_TABLE	locks an entire partitioned table. This procedure locks all of the partitions that belong to the table. If one of the partitions fails to lock, all other locked partitions that belong to the partitioned table are released. To release the lock, you must perform either a COMMIT or a ROLLBACK. Use the SQL LOCK command to lock a non–partitioned table or single partition.
generating HHCODE	MD_DML.GENHHCODE	generates an HHCODE from original data for a particular spatial table without first having to look up the dimension information in the Spatial Data Option data dictionary. This function performs all required logic and validation to generate an HHCODE and returns a NULL if an error is encountered.

Table 3 – 5 Tasks Using MD_DML Procedures

CHAPTER

4 Administration

This chapter describes the following administrative tasks associated with the Spatial Data Option:

- partition maintenance
- transferring data
- user privilege administration
- creating and dropping indexes
- implementing constraints and triggers
- changing the MDSYS password
- calculating table size requirements
- pinning PL/SQL packages
- tablespace management
- exception conditions
- replication

Partition Maintenance

In a partitioned table, the spatial organization of data must be maintained when inserting, updating, or deleting data. To keep the data organized correctly, partition maintenance procedures must be performed. Partition maintenance procedures can be used with SQL INSERT, UPDATE, and DELETE commands.

The partition maintenance procedures can also be called for other reasons; for example, to subdivide a partition, to create more space in the database, or to restore a partition and bring its data online.

Partition Status

The partition status describes the state of the partition. Table 4 – 1 lists and describes partition statuses.

Status	Description
NOTEXIST	The partition does not exist.
ONLINE	The partition is online.

Table 4 – 1 Partition Statuses

Note: During partition subdivision, the status of the newly–created partitions is set to OFFLINE until the subdivision is complete.

Partition Maintenance Operations

The partition maintenance operations consist of a set of PL/SQL functions and procedures in the MD_PART package. This package is used to perform the following partition maintenance operations on spatial tables:

- creating an inferred partition
- deleting rows from a partition
- dropping a partition
- identifying partition name and status
- moving a partition to another tablespace
- subdividing a partition

Table 4 – 2 lists the procedures and functions used for partition maintenance operations.

Task	Procedure or Function Name	Description and Considerations
creating an inferred partition	MD_PART.CREATE_ INFERRED_PARTITION	creates an empty partition. This function is used if the MD_PART.GET_PARTITION_NAME function returns a status of MD_PART.NOTEXIST for an inferred partition into which to insert data This function ensures that the spatial organization of the data is maintained. This function does not insert data.
deleting rows from a partition	MD_PART. TRUNCATE_PARTITION	removes rows from a partition without dropping it. This procedure either drops or reuses the space from the deleted rows allocated to the partition.
dropping a partition	MD_PART.DROP_ PARTITION	drops a partition. This procedure is used to reclaim space when the data in a partition has been removed.
identifying partition name and status	MD_PART.GET_ PARTITION_NAME	determines the name and status of a partition. The name and status of the partition must be identified before executing a DML command or performing partition maintenance operations.
moving a partition	MD_PART.MOVE_ PARTITION	moves a partition. The tablespace must be allocated and activated for the spatial table to which the partition belongs.
subdividing a partition	MD_PART.SUB DIVIDE_PARTITION SD*Loader	subdivides a partition. A partition should be subdivided if it is over or near the high water mark.

Table 4 – 2 Partition Maintenance Operations

For a detailed description of the MD_PART functions and procedures described in this section, see Chapter 7, "SD*SQL Packages."

Creating an Inferred Partition

Use the following procedure to create an inferred partition:

1. Determine the spatial table name.
2. Determine the HHCODE or a partition key value within the range of values for the nonexistent partition.
3. Invoke the MD_PART.CREATE_INFERRED_PARTITION function.

   This causes an implicit COMMIT to occur.

Example

The following example creates an inferred partition for the TABLE1 table:

```
MD_PART.CREATE_INFERRED_PARTITION ('herman', 'table1', MD.HHENCODE
(–76.2, –180, 180, 7, 47, –90, 90, 7));
```

Deleting Rows from a Partition

Use the following procedure to truncate a partition by removing all rows:

1. Determine the partition name.
2. Determine whether or not to reuse the tablespace storage for the partition.

   If the REUSE_STORAGE parameter is equal to TRUE, the space from the deleted rows remains allocated to the partition. This space can be subsequently used only by new data in the partition resulting from a SQL INSERT or UPDATE command.

   If the REUSE_STORAGE parameter is equal to FALSE, all but the space specified by the partition's MINEXTENTS parameter is deallocated. This space can subsequently be used by other objects in the tablespace.
3. Invoke the MD_PART.TRUNCATE_PARTITION procedure. This causes an implicit COMMIT to occur.

Note: Do not use the SQL TRUNCATE command directly on a partition.

Warning: Executing the following procedure removes all data from the TABLE1_P000000005 partition.

Example

The following example truncates the partition TABLE1_P000000005:

```
MD_PART.TRUNCATE_PARTITION ('table1_P000000005');
```

Dropping a Partition

Use the following procedure to drop a partition:

1. Determine the partition name.
2. Invoke the MD_PART.DROP_PARTITION procedure.

   This causes an implicit COMMIT to occur.

Example

The following example drops the partition TABLE1_P000000005:

```
MD_PART.DROP_PARTITION ('herman', 'table1_P000000005');
```

Identifying Partition Name and Status

Use the MD_PART.GET_PARTITION_NAME function to determine the name and status of a partition.

Before executing a DML command or performing partition maintenance operations you must identify the name and status of the partition that contains the data.

When a partition subdivides, new partitions are created and data is transferred from the old partition into the new partitions. Users who want to query the data during this process should execute a query on the old partition, not the new partitions.

Moving a Partition

To move a partition to another tablespace, use the MD_PART.MOVE_PARTITION procedure.

You can move a partition to another tablespace for several reasons; for example, to balance the I/O load or to drop a tablespace. The new tablespace must be allocated and currently activated for the spatial table to which the partition belongs.

Subdividing a Partition

Use the following procedure to divide a partition into child partitions:

1. Determine the high water mark for the spatial table.
2. Determine the partition name.
3. Use the SQL SELECT command to determine a row count for the partition.
4. Invoke the MD_PART.SUBDIVIDE_PARTITION procedure.

A partition should be subdivided if a row count on the table indicates that it is over or near the high water mark. Subdividing a partition causes a partition to decompose. If data is inserted using SD*Loader, subdivision occurs automatically. If a partition is subdivided using the MD_PART.SUBDIVIDE_PARTITION procedure, the next time SD*Loader inserts records into that partition it subdivides the partition if the number of records exceeds the high water mark.

The MD_PART.SUBDIVIDE_PARTITION procedure can also be used to optimize performance.

Example I

The following example subdivides a partition:

```
SQL> EXECUTE MD_PART.SUBDIVIDE_PARTITION ('TABLE1_P00000000C');
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SET PAUSE OFF
SQL> SET ECHO OFF
SQL> SET FEEDBACK OFF
SQL> SPOOL count_rows
2> SELECT 'SELECT count(*) FROM ' || partition_table_name||';'
3> FROM user_md_partitions
4> WHERE md_table_name = 'TABLE1';
SQL> SPOOL OFF
SQL> SET HEADING ON
SQL> SET PAGESIZE 24
SQL> SET PAUSE ON
SQL> SET ECHO ON
SQL> SET FEEDBACK ON
```

Example II

The following example subdivides the TABLE1_P000000008 partition of the TABLE1 table:

```
SQL> EXECUTE MD_PART.SUBDIVIDE_PARTITION ('herman',
'table1_P000000008')
```

Transferring Data

There are several ways to transfer spatial data between databases. The method that you use depends on your operating environment. One of these methods can be used to export data for backup. This section discusses the following ways to transfer spatial data:

- using the Oracle7 Server Export and Import utilities
- copying spatial data to a file for transfer to an OS file with SD*Converter and SD*Loader
- transferring data with the MD_WEX package

Oracle7 Server Export and Import Utilities

The Oracle7 Server Export and Import utilities can be used in one of the following ways to transfer Spatial Data Option data between databases:

- Performing a full database export and import preserves the Spatial Data Option data dictionary and all spatial tables.
- Performing a user export of the MDSYS and HERMAN users, as well as all database users who own spatial tables, preserves the Spatial Data Option data dictionary and all spatial tables.

For information on the Export and Import utilities, see the *Oracle7 Server Utilities*.

Copying to a FIle

Data can also be transferred by copying spatial data to an OS file for transfer to another database.

Requirements

All of these methods have the following requirements:

- The HHCODE columns of the spatial table must be decoded into their original dimensional values before they can be saved to an OS file.
- SD*Converter and SD*Loader must be used to load the data into the target database.

Methods

The following methods can be used to copy data to an OS file.

- If the original data sources exist and the data has not been modified, the original data sources can be copied to an OS file.
- SQL commands can be used to generate SQL scripts that select the data to be moved and spool results to an OS file.
- The SQL SELECT command can be used to copy the data to be moved to an Oracle7 table in the source database. From the source database the Oracle7 Export and Import utilities can be

used to load the data into an Oracle7 table in the target database. In the target database SD*Converter can be used to convert the data, using the Oracle7 table option for the data control.

- The SQL SELECT command can be used to create a view of the data to be moved by selecting all rows of the view to copy to an OS file.

Example The steps listed in the following example create a view that selects the partition key column, and decodes it and the attribute data into its original dimensional values from all partitions. The example then selects all the rows of the decoded view of the partitioned table and spools the result to an OS file.

1. Save Spatial Data Option data as OS files.

```
SQL>   CREATE OR REPLACE VIEW ptsdata AS
2>     SELECT HHDECODE(location,1,0,360) lat,
3>            HHDECODE(location,2,0,180) lon, depth
4>            FROM points_p000000001 UNION ALL
5>     SELECT HHDECODE(location,1,0,360) lat,
6>            HHDECODE(location,2,0,180) lon, depth
7>            FROM points_p000000002 UNION ALL
8>     SELECT HHDECODE(location,1,0,360) lat,
9>            HHDECODE(location,2,0,180) lon, depth
10>           FROM points_p000000003 UNION ALL
11>    SELECT HHDECODE(location,1,0,360) lat,
12>           HHDECODE(location,2,0,180) lon, depth
13>           FROM points_p000000004 UNION ALL
14>    SELECT HHDECODE(location,1,0,360) lat,
15>           HHDECODE(location,2,0,180) lon, depth
16>           FROM points_p000000005 UNION ALL
17>    SELECT HHDECODE(location,1,0,360) lat,
18>           HHDECODE(location,2,0,180) lon, depth
19>           FROM points_p000000006 UNION ALL
20>    SELECT HHDECODE(location,1,0,360) lat,
21>           HHDECODE(location,2,0,180) lon, depth
22>           FROM points_p000000007 UNION ALL
23>    SELECT HHDECODE(location,1,0,360) lat,
24>           HHDECODE(location,2,0,180) lon, depth
25>           FROM points_p000000008 UNION ALL
26>    SELECT HHDECODE(location,1,0,360) lat,
27>           HHDECODE(location,2,0,180) lon, depth
28>           FROM points_p000000009 UNION ALL
29>    SELECT HHDECODE(location,1,0,360) lat,
30>           HHDECODE(location,2,0,180) lon, depth
31>           FROM points_p00000000A;
SQL>   SET TERMOUT OFF;
SQL>   SET HEADING OFF;
SQL>   SET PAGESIZE 0;
```

```
SQL>   SPOOL ptsdata.lis
SQL>   SELECT * FROM ptsdata;
SQL>   EXIT
```

For detailed information on how to use the HHDECODE function, see Chapter 6, "SD*SQL Kernel Functions."

2. Transfer the spooled data file to the target database.

3. Edit the SD*Converter data control file to reflect the record length of the data created in the resultant spool file.

   To determine record length, enter the following command:

   ```
   % wc ptsdata.lis
   ```

   The preceding command returns a value similar to the following:

   ```
   42545  127635        3446145       ptsdata.lis
   ```

4. Perform the following calculation to determine the record length:

   ```
   # of bytes / # of lines = record length
   ```

   For the preceding example, this calculation is as follows:

   ```
   # of bytes (3446145) / # of lines (42545) = 81
   ```

5. Edit the SD*Converter data control file to reflect the spacing of the data in the resultant spool file. Position values in the SD*Converter data control file may need to be updated to reflect column spacing in the spooled output file.

   The following is an example of an updated SD*Converter data control file:

   ```
   ASCII
   FIXED  81
   DIMENSION lat  location       POSITION (1:10)       FLOAT
   DIMENSION lon  location       POSITION (12:21)      FLOAT
   COLUMN  depth                 POSITION (29:32)      INTEGER
   ```

   See Chapter 5, "Utilities," and the *Oracle7 Spatial Data Option Application Developer's Guide* for more information on SD*Converter data control files.

6. Create the spatial tables in the target database.

   For more information on how to create spatial tables, see the *Oracle7 Spatial Data Option Application Developer's Guide.*

7. Convert and load the spatial data into the target database using the SD*Converter and SD*Loader utilities.

   For more information on the SD*Converter and SD*Loader utilities, see Chapter 5, "Utilities.,"

Transferring Data with MD_WEX Package

Use the following procedure to move data from a partitioned table to another database.

1. Use the MD_WEX procedures to create a table or view on an extract window.

   For a detailed description of the MD_WEX package, see Chapter 7, ”SD*SQL Packages.”

2. Use a SQL SELECT command to select all the rows in the table or view created in Step 1

3. Use the SQL*Plus spool command to save the results in an OS file.

   For information on the SQL*Plus SPOOL command, see the *SQL*Plus User’s Guide and Reference.*

4. Transfer the spooled output file to the target database.

5. Create the SD*Converter data control file and table control file according to the structure of the spooled output file.

6. Convert the spooled output file into an SLF file using SD*Converter.

7. If the table does not already exist in the target database, use SQL commands and the MD_DDL procedures to create and register the table in the Spatial Data Option data dictionary.

8. Load the data in the SLF file into the spatial table using SD*Loader.

For a detailed description of the Spatial Data Option SD*Converter and SD*Loader utilities, see Chapter 5, ”Utilities.”

User Privilege Administration

This section discusses procedures used to grant and revoke table privileges on a spatial table.

Granting Privileges on Spatial Table

The procedure for granting privileges on spatial tables depends on whether the table is partitioned or non–partitioned.

To grant privileges on a partitioned table that has already subdivided, you must grant the privileges to the spatial table and also to all existing partitions. A SQL script file that generates SQL commands can be used to automate the process of granting privileges to existing partitions.

The Spatial Data Option propagates any constraint, trigger, index, or grant created on the spatial table to all new partitions that are created either manually or automatically.

Partitioned Table

When granting privileges to a partitioned table, the following is true for all partitioned tables:

- Any grant created on the spatial table of an empty partitioned table is propagated to subsequent partitions.
- When partitions are created or subdivided, the Spatial Data Option propagates the grants to the new partitions.

The procedure for granting privileges on partitioned tables depends on whether the table is empty or already contains data.

Empty Partitioned Table: To grant privileges to a partitioned table that does not yet contain data, use the SQL GRANT command to grant the privileges to the spatial table.

Populated Partitioned Table: To grant privileges on an existing spatial table and all associated partitions, use the following procedure.

1. Determine the spatial tablename.
2. Determine all existing partition names.
3. Raise an application error if the partition does not exist. Use the MD_PART.CREATE_INFERRED_PARTITION procedure to create necessary partitions.
4. Create a SQL command to generate a SQL script to grant rights to the spatial table and all existing partitions.

Example

The following example creates a SQL script that grants all privileges on all partitions of the TABLE1 table to a new user.

```
SQL> REM Grant a privilege on a partitioned table
SQL> GRANT all ON table1 TO new_user;
```

```
SQL> SET TERMOUT OFF
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SPOOL grant.sql
SQL>  SELECT 'GRANT SELECT ON ' || partition_table_name || 'TO
new_user;'
2 FROM user_md_partitions
3 WHERE md_table_name = 'TABLE1';
SQL> SPOOL off
SQL> START grant.SQL
```

For information on commands to grant privileges, see the *Oracle7 Server SQL Reference.*

Non–Partitioned Table

To grant privileges on non–partitioned tables, use the SQL GRANT command.

Revoking Privileges on Spatial Table

To revoke a privilege from a partitioned table, you must revoke that privilege from the spatial table and all existing partition tables.

1. Determine partition names and the spatial tablename from which you want to revoke privileges.
2. Revoke privileges from partitions and spatial table as necessary, using the SQL REVOKE command.

Example

The following example creates a SQL script that revokes the SQL SELECT command privilege from a new user from all partitions of the TABLE1 table:

```
SQL> REM Revoke a privilege from a partitioned table.
SQL> REVOKE SELECT ON TABLE1 FROM new_user;
SQL> SET TERMOUT OFF
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SPOOL revoke.sql
SQL> SELECT 'REVOKE SELECT ON ' '||partition_table_name||
2 'FROM new_user';'
3 FROM user_md_partitions
4 WHERE md_table_name = 'TABLE1';
SQL> SPOOL OFF
SQL> START revoke.sql
```

For information on commands to revoke privileges, see the *Oracle7 Server SQL Reference.*

Creating and Dropping Indexes

The Spatial Data Option supports creating and dropping indexes on spatial tables. This section describes the following procedures:

- creating an index
- dropping an index

Creating an Index on a Spatial Table

The procedure for creating an index on a spatial table differs depending on whether the table is partitioned or non–partitioned.

Partitioned Table

Use the following procedure to create an index on a partitioned table:

1. Create a SQL command to generate the SQL CREATE INDEX commands on attribute columns in all associated partitions.

   For information on the SQL CREATE INDEX command, see the *Oracle7 Server SQL Reference.*

2. Include an indication of the partition name in the index name.

Note: If an index exists on a partition that subsequently subdivides, the index is not propagated.

The generated SQL commands can be saved to a SQL*Plus spool file, and the spool file can be executed to create indexes on all partitions.

For information on the SPOOL command, see the *Oracle7 Server Utilities*.

Example I

The following example generates a SQL command to generate the SQL commands to create an index on the partition key column for all partitions of the TABLE1 table:

```
SELECT 'CREATE INDEX indx1_I' ||
    SUBSTR(partition_table_name,
    INSTR(partition_table_name,'_P')+1) ||' ON '||
    partition_table_name||' ( location );'
FROM all_md_partitions
WHERE owner = 'SCOTT'
    AND md_table_name = 'TABLE1'
```

Note: Because each index must have a unique name, the sequence number of the partition is incorporated into the index name using the substr() expression.

Example II The following example generates the SQL commands to create an index on the DEPTH column for all partitions of the TABLE1 table:

```
SQL> SET TERMOUT OFF
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SPOOL 'create_TABLE1_index.sql'
SQL> SELECT 'CREATE INDEX '|| partition_table_name ||
2>    '_depth_indx ON ' || partition_table_name || '(depth);'
3>    FROM user_md_partitions
4>    WHERE md_table_name = 'TABLE1';
SQL> SPOOL off
SQL>
SQL> START create_TABLE1_index.sql
```

Non–Partitioned Table

To create an index on a non–partitioned table, use the SQL CREATE INDEX command.

For information on the SQL CREATE INDEX command, see the *Oracle7 Server SQL Reference.*

Dropping an Index on a Spatial Table

The procedure for dropping an index on a spatial table differs depending on whether the table is partitioned or non–partitioned.

Partitioned Table

To drop an index on a partitioned table, each index created on each partition must be dropped. Use a SQL command to generate the SQL DROP INDEX commands for all associated partitions. The generated SQL commands can be saved to a SQL*Plus spool file, and the spool file can be executed to drop indexes on all partitions.

For information on the SPOOL command, see the *Oracle7 Server Utilities*.

Non–Partitioned Table

To drop an index on a non–partitioned table, use the SQL DROP INDEX command.

For information on the SQL DROP INDEX command, see the *Oracle7 Server SQL Reference.*

Example I The following example generates the SQL commands to drop an index for all partitions of the TABLE1 table:

```
SQL> SET TERMOUT OFF
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SPOOL 'drop_TABLE1_index.sql'
SELECT 'DROP INDEX indx1_I' ||
    SUBSTR(partition_table_name,
    INSTR(partition_table_name,'_P')+1)|| ';'
```

```
FROM all_md_partitions
WHERE owner = 'SCOTT'
   AND md_table_name = 'TABLE1'
SQL> SPOOL off
SQL>
SQL> START drop_TABLE1_index.sql
```

Note: The preceding example assumes that the sequence number of the partition was incorporated into the index name to make each index name unique.

Example II

The following example generates the SQL commands to drop all indexes for the TABLE1 table and its partitions:

```
SQL> SET TERMOUT OFF
SQL> SET HEADING OFF
SQL> SET PAGESIZE 0
SQL> SPOOL 'drop_TABLE1_index.sql'
SQL> SELECT 'DROP_INDEX ' || index_name || ';'
   FROM user_indexes
   WHERE table_name like 'TABLE1%';
SQL> SPOOL off
SQL>
SQL> START drop_TABLE1_index.sql
```

Implementing Constraints and Triggers

The Spatial Data Option supports the implementation of the following constraints and triggers on a spatial table:

- referential constraints and user–defined constraints
- database triggers

Referential and User–Defined Constraints

For any database, data integrity is very important. Referential and data integrity constraints can be defined on both partitioned and non–partitioned tables. By utilizing the primary key and foreign key integrity constraints you can insure that duplicate values, nulls, and relationships between columns conform to your design.

Partitioned Table

Exercise care when defining constraints and triggers on a partitioned table, because they have a major impact on space usage in the SYSTEM tablespace. They are copied to every partition; therefore, as the number of partitions increases, so does the number of constraints and triggers defined in the database.

When implementing constraints on a partitioned table, the following guidelines apply:

- The primary key is not supported; however, the HHCODE partition key column can be defined as the primary key.
- The foreign key is supported for all columns except the partition key column.
- The ON DELETE CASCADE option is supported for all foreign keys.
- The CHECK CONSTRAINT option is supported for all columns.
- The NOT NULL option is supported for all columns.
- The EXCEPTIONS INTO option is supported for all columns.
- The UNIQUE option is supported for all columns except the partition key column, but only at the partition level.
- Attribute columns can be defined as the foreign key.

Non–Partitioned Table

When implementing constraints on a non–partitioned table, the following guidelines apply:

- All constraints are supported for all columns.
- Attribute columns can be defined as the primary key and the foreign key.
- HHCODE columns can be defined as the primary key and the foreign key.

Triggers

Triggers can be created for both partitioned and non–partitioned tables. As a general rule, developers should make the trigger bodies as small as possible. For example, it is advantageous to define the trigger logic in a single procedure, and have each trigger call that procedure.

Partitioned Table

If a trigger is created on a partitioned table, the trigger propagates to each new partition when the table subdivides.

Non–Partitioned Table

Triggers on a non–partitioned table operate exactly like triggers on standard Oracle7 tables.

Changing the MDSYS Password

You must reconfigure the Spatial Data Option if you change the MDSYS user password after installation. For instructions on how to reconfigure, see the installation guide for your operating system.

Calculating Table Size Requirements

This section describes how to calculate the size requirements of a partitioned table. The actual size can vary widely, depending on the values the user chooses for PCTFREE and other storage parameters. This section describes a way to calculate the maximum and the average size requirements of a partitioned table.

Perform the following steps to calculate the maximum and the average size of a partitioned table:

1. Calculate partition size by implementing the same procedure used to estimate the size of a standard Oracle7 table. To determine the maximum size, use the high water mark. To determine the average size, enter the high water mark divided by two for the number of rows in the calculation.

   For information on how to calculate the size of a standard Oracle7 table, see the *Oracle7 Server Administrator's Guide*.

2. Calculate the number of partitions.

   The number of partitions is dependent on the distribution and density of the data. The following calculation provides a rough estimate:

   total number rows to load/(high water mark/($2^{\text{number of dimensions}}$))

3. Multiply the average size of the partition size, calculated in Step 1, by the number of partitions expected, calculated in Step 2.

Example

The following is an example of an equation that calculates the size required for a two–dimensional table. It calculates the average partition size where R equals the number of rows per database block.

```
available space        = (1768–2R) bytes (R = number of rows/block)

average row size       = SELECT AVG(NVL(VSIZE (location), 0)) +
                            AVG(NVL(VSIZE (depth), 0))
                           FROM TABLE1_P000000001;
                       = 16 bytes

total average row size = row header + F + V + average row size
                       = 3 + (1 * 2) + (3 * 0) + 16
                       = 21 bytes

R(rows/block)          = available space/total average row size
                       = (1768 bytes–2R bytes*rows/block)/21 bytes
21R bytes*rows/block   = 1768 bytes – 2R bytes * rows/block
23R bytes * rows/block = 1768 bytes
R                      = 76 rows/block
```

```
number blocks/table     = (high water mark/2)/average rows per block
                        = 5000 rows / 76 rows/block
                        = 66 blocks

average partition size  = number blocks/table * 2048 bytes/block
                        = 66 blocks * 2048 bytes/block
                        = 135168 bytes
```

The following equation calculates the number of partitions, as described in Step 2:

```
average number of partition   = total rows loaded/ (high water
                                   mark/(2^(number of dimensions)))
                              = 42545 / (10000 / (2^2))
                              = 17 partitions
```

The following equation calculates the average space required for a partitioned table, as described in Step 3:

```
average space required     = average partition size *
                                   average number of partitions
                           = 135168 bytes * 17 partitions
                           = 2297856 bytes
```

Pinning PL/SQL Packages

Data must be located in the System Global Area (SGA) to be referenced; if it is not, the system must read it into the SGA before it can be used. Infrequently accessed data is flushed from the SGA. Because the Spatial Data Option has large PL/SQL packages, you may decide to keep frequently used packages permanently in the SGA to improve performance. Designating a package to remain permanently in the SGA is known as pinning the PL/SQL package.

For more information about the SGA and Oracle memory structures, see the *Oracle7 Server Concepts*.

1. Enter the following commands as the *oracle* owner to install the DBMS_SHARED_POOL package:

```
% cd $ORACLE_HOME/rdbms/admin
% svrmgrl
SVRMGR> CONNECT INTERNAL
SVRMGR> @dbmspool.sql
SVRMGR> @prvtpool.sql
```

2. Execute the following command to pin each of the Spatial Data Option PL/SQL packages:

```
SVRMGR> EXECUTE dbms_shared_pool.keep('MDSYS.package');
```

   where *package* is one of the following packages:

   - MD
   - MDEXEC
   - MDLEXR
   - MDLIB
   - MDTRIG
   - MDVBLD
   - MD_DDL
   - MD_DML
   - MD_PART
   - MD_WEX

   If space is limited in the SGA, you may decide to pin a subset of the Spatial Data Option PL/SQL packages. If this is the case, Oracle Corporation recommends installing the following packages:

   - MDEXEC
   - MDLIB
   - MDTRIG

Tablespace Management

A database is logically divided into one or more tablespaces, which are logical storage units. One or more data files is created for each tablespace to physically store the data. The combined size of a tablespace's data files is the total storage capacity of the tablespace. The combined size of a database's tablespaces is the total storage capacity of the database. The following topics are relevant to tablespace management with the Spatial Data Option:

- tablespace striping
- partition tablespace striping
- how tablespace striping works
- tablespace activation

Tablespace Striping

Distributing a tablespace over several disk devices is known as tablespace striping. Oracle7 tablespace striping and operating system tablespace striping can be used for both non–partitioned and partitioned tables.

For more information on tablespace striping, see the *Oracle7 Server Administrator's Guide.*

Partition Tablespace Striping

Partition tablespace striping is the distribution of partitions over several tablespaces. For Spatial Data Option partition tablespace striping to occur, more than one tablespace must be defined and activated for a partitioned table. Partition tablespace striping occurs when a partition subdivides.

How Tablespace Striping Works

As with all Oracle tables, if a tablespace is specified when creating a table, the table is stored in that tablespace. If a table is created without specifying the tablespace, the table is stored in the default tablespace of the user who created the table.

With Spatial Data Option, however, the partitions are distributed among all allocated active tablespaces in circular fashion when a subdivision occurs.

For example, assume that the spatial table TABLE1 is created without specifying a tablespace. The TABLE1 table is assigned by default to TBS1, the user's default tablespace.

The TABLE1 table is registered as a partitioned table. Later, tablespaces TBS2 and TBS3 are allocated and active for the TABLE1 table using the following commands:

```
SQL> EXECUTE MD_DDL.ALLOCATE_TABLESPACE ('TABLE1', 'tbs2');
SQL> EXECUTE MD_DDL.ALLOCATE_TABLESPACE ('TABLE1', 'tbs3');
```

Data is entered into TABLE1, and its partitions are subdivided. The first TABLE1 partition is created in TBS2, the second one in TBS3, the third in TBS1, the fourth in TBS2, and so on.

Advantages

Partition tablespace striping provides the following advantages:

- It can greatly improve retrieval performance. If data for a specific region is spread over multiple tablespaces and tablespaces reside on different physical disks, parallel reads can take place.
- Administrators can perform tasks on small subsets of data.
- Designers do not need to determine exactly how much data the tables will contain in the future, or where to store the data when tables are created.

Tablespace Activation

If there is no data to be loaded into any partitioned table for which a tablespace is allocated, then that tablespace can be deactivated. If there is data to be loaded into any partitioned table for which a tablespace is allocated but not active, then that tablespace can be reactivated. This section describes how to perform the following tasks:

- deactivating a tablespace
- reactivating a tablespace

Deactivating

Use the following procedure to deactivate a tablespace:

1. Determine the spatial tablename.
2. Determine the tablespace name.
3. Invoke the MD_DDL.DEACTIVATE_TABLESPACE procedure.

Note: Spatial Data Option tablespaces cannot be deallocated, they can only be deactivated.

Example

The following example deactivates TABLESPACE5 for the TABLE1 table:

```
SQL> EXECUTE MD_DDL.DEACTIVATE_TABLESPACE ('table1', -
>'tablespace5');
```

Reactivating

Use the following procedure to reactivate a tablespace:

1. Determine the spatial tablename.
2. Determine the tablespace name.

3. Invoke the MD_DDL.ACTIVATE_TABLESPACE procedure.

Example The following example activates tablespace5 for the TABLE1 table, which had been deactivated using the DEACTIVATE_TABLESPACE procedure:

```
SQL> EXECUTE MD_DDL.ACTIVATE_TABLESPACE ('table1', 'tablespace5');
```

For information on the MD_DDL.DEACTIVATE_TABLESPACE and MD_DDL.ACTIVATE_TABLESPACE procedures, see Chapter 7, "SD*SQL Packages."

Exception Conditions

A number of exception conditions can occur when using the Spatial Data Option. This section describes possible exception conditions, and offers the following suggestions on what you can do if they occur:

- clearing partitions that exist in the Oracle7 data dictionary but are no longer registered in the Spatial Data Option data dictionary
- validating Spatial Data Option data dictionary entries for spatial tables

Clearing Partitions in Oracle7 Data Dictionary

An exception condition can occur if tables that are listed in the Oracle7 data dictionary are no longer registered in the Spatial Data Option data dictionary. This can occur under the following conditions:

- if a user drops any of the partitions of a spatial table using the SQL DROP TABLE command
- if an MD_DDL.DROP_MD_TABLE procedure is interrupted
- if the SQL DROP USER CASCADE command is run for a user who owns spatial tables

Tables that are listed in the Oracle7 data dictionary but no longer registered in the Spatial Data Option data dictionary are left in a state of inconsistency. This can be determined by cross–referencing the USER_MD_PARTITIONS view in the Spatial Data Option data dictionary with the Oracle7 data dictionary. After you have determined which tables are not part of a spatial table, you can drop those tables using the SQL DROP TABLE command.

Tables in an exception condition are listed in the DBA_MD_EXCEPTIONS view. To drop these tables, execute the MD_PART.CLEAR_EXCEPTION_TABLES procedure.

For more information about the MD_PART.CLEAR_EXCEPTION_TABLES procedure, see Chapter 7, ”SD*SQL Packages.”

Note: You cannot use the MD_PART.DROP_PARTITION procedure when this exception condition occurs, because the MD_PART.DROP_PARTITION procedure can only be used on valid partitions that are identified in the Spatial Data Option data dictionary.

Validating a Spatial Data Option Data Dictionary Entry

To validate a Spatial Data Option data dictionary entry for a spatial table, use the MDVERIFY package. It verifies that spatial tables owned by the user calling the function are consistent in the Spatial Data Option data dictionary.

You do not need to run the MDVERIFY package during normal system operation. However, the procedures can provide valuable diagnostic information for Oracle Worldwide Support if the Spatial Data Option data dictionary becomes corrupted.

For more information about the MDVERIFY package, see Chapter 7, ”SD*SQL Packages.”

Replication

Replication is a method used to view or update data at multiple sites. Spatial Data Option supports static partitioning replication, which means replication is supported as long as the following characteristics of the partitions do not change:

- partition name
- number of partitions
- partition information stored in the Spatial Data Option data dictionary

You can perform operations on partitions as long as the partitions do not subdivide; that is, the high water mark is not exceeded. To ensure that partitions do not subdivide, load the data and create all the partitions before starting replication. Once the partitions have been created and registered in the Spatial Data Option data dictionary, the user must register the partitions, update the Spatial Data Option data dictionary with the replication schema, and make sure the information about views and the stored SD*SQL packages is propagated to the different replication sites.

Replication of dynamic partitions is under consideration for subsequent releases of the Spatial Data Option. Replication support for dynamic partitioning entails efficiently informing all master sites about partition changes.

For information on using replication with Oracle7, see the *Oracle7 Server Distributed Systems, Volume II: Replicated Data.*

PART *II*

Reference

CHAPTER

5

Utilities

This chapter describes the Spatial Data Option utilities. The following topics are included:

- SD*Converter
- SD*Loader

SD*Converter

Purpose

This utility converts data from external data files or standard Oracle7 tables into SLF files.

Prerequisites

You must have the following tables or files before you can run SD*Converter:

- a data file containing the data to be converted, or a standard Oracle7 table
- a Spatial Data Option data dictionary, or a table control file describing the data dictionary information
- a data control file describing the format of the data, if the data is in a data file

For information on creating control files, see ”Table Control Files” and ”Data Control Files” in this section.

You must have sufficient space in the directory for the SLF file to be created, as well as for temporary files that are created during the sort process, if you choose to sort. For information on calculating the amount of space required and deciding whether to sort, see Chapter 2, ”Converting and Loading.”

Syntax

The following shows the possible syntax combinations of keywords and parameters, depending on the input:

Data Control File
Table Control File
and Data File

When the input combination is a data control file, a table control file, and a data file, the syntax is as follows:

```
SDCONV DATACONTROL=data_control_filename
TABLECONTROL=table_control_filename DATA=data_filename
[SLF=data_filename.SLF|slf_filename] [LOG=log_filename]
[BAD=bad_filename] [ERRORS=50|max_number_of_errors]
[BINDSIZE=65536|bindsize] [SORT=YES|NO]
```

Data Control File
Spatial Data Option Data
Dictionary and
Data File

When the input combination is a data control file, a Spatial Data Option data dictionary, and a data file, the syntax is as follows:

```
SDCONV USERID=username/password DATACONTROL=data_control_filename
TABLECONTROL=sd_tablename DATA=data_filename
[SLF=data_filename.SLF|slf_filename] [LOG=log_filename]
[BAD=bad_filename] [ERRORS=50|max_number_of_errors]
[BINDSIZE=65536|bindsize] [SORT=YES|NO]
```

Oracle7 Table and Spatial Data Option Data Dictionary

When the input combination is a Spatial Data Option data dictionary and a standard Oracle7 table, the syntax is as follows:

```
SDCONV USERID=username/password TABLECONTROL=sd_tablename
TABLE=Oracle7_tablename [SLF=Oracle7_tablename.SLF|slf_filename]
[LOG=log_filename] [BAD=bad_filename]
[ERRORS=50|max_number_of_errors] [BINDSIZE=65536|bindsize]
```

Keywords and Parameters

USERID	is the keyword to specify username and password. Required if either the data control information or the table control information is obtained from the Spatial Data Option data dictionary or the source data is an Oracle7 table. Do not use when both input sources are from control files.
username/password	specifies the username and password.
TABLECONTROL	is the keyword to specify table control input, and can be either a control file or a spatial table name, depending on the input combination used.
table_control_filename	specifies the name of the table control file when not using a Spatial Data Option data dictionary.
sd_tablename	specifies the name of the spatial table when the table control information is taken from the Spatial Data Option data dictionary.
DATACONTROL	is the keyword to specify data control filename. Do not use when the data input source is an Oracle7 table.
data_control_filename	specifies the name of the data control file.
DATA	is the keyword to specify the filename of the data to be converted when the source is an external data file.
data_input_filename	specifies the name of the file containing the data to be converted.
TABLE	is the keyword to specify source data table name when the source is an Oracle7 table. Do not use when the data input source is a data control file.
Oracle7_tablename	specifies the name of the Oracle7 table.

SLF	is the keyword to specify SLF output filename. Optional; default is the base name of the data input filename with the **.slf** extension.
data_input_filename.SLF	is the default SLF output filename when the input is from a data file.
Oracle7_tablename.SLF	is the default SLF output filename when the input is from an Oracle7 table.
SLF_filename	specifies an SLF output filename.
LOG	is the keyword to specify name of the file created by SD*Converter to store information about the conversion process. Optional; if not specified, no log file is written, and errors are displayed on standard output.
log_filename	specifies a log filename.
BAD	is the keyword to specify name of the file created by SD*Converter to store bad lines of the input file. Optional; if not specified, no file is created. In the bad file, records that could not be processed because of incorrect format or bad content follow the line number where they were found.
bad_filename	specifies a bad filename.
ERRORS	is the keyword to specify the maximum number of non–fatal errors to skip. Optional; default is 50.
50	is the default maximum number of errors.
max_number_of_errors	specifies the maximum number of errors.
BINDSIZE	is the keyword to specify size of bind array in bytes. Optional; default is 65536 bytes.
65536	is the default bind array size in bytes.
bindsize	specifies a bind array size in bytes.
SORT	is the keyword to specify whether to perform a sort. Values are YES and NO. Optional; defaults to YES.

YES — is the default SORT parameter; which specifies that a sort be performed during conversion.

NO — specifies not to perform a sort during conversion.

Usage Notes

Consider the following points when using the SD*Converter utility:

- Execute SD*Converter from the operating system command line by entering the SDCONV command followed by arguments.
- You cannot enter parameters without their associated keywords.
- Table 5 – 1 shows the required keywords for each of the three input options. Examples of each input option are provided in the following examples.

Input Option	Userid	Table Control	Data Control	Data Source
Dictionary=File Data=File		tablecontrol= *filename.ctl*	datacontrol= *filename.ctl*	data= *filename.dat*
Dictionary=Table Data= File	userid= *uid/pwd*	tablecontrol= *sdtablename*	datacontrol= *filename.ctl*	data= *filename.dat*
Dictionary=Table Data=Table	userid= *uid/pwd*	tablecontrol= *sdtablename*		table= *tablename*

Table 5 – 1 SD*Converter Input Source Options: Required Keywords

- You can sort during the conversion or during the loading process, depending on your requirements. For more information on choosing when to sort see ”Considerations for Sorting” in Chapter 2, ”Converting and Loading.”

Example I

The following example creates an SLF file called **table12b.slf** from a simple, homogeneous data file **table1.dat**, using the dictionary control information from the table control file **sdtable1.ctl** and the data control file **table1.ctl**.

```
% sdconv tablecontrol=sdtable1.ctl datacontrol=table1.ctl
data=table1.dat slf=table12b.slf
```

The output appears as follows:

```
SD*Converter: Release 7.3.2.0.0 – Production on Sun Feb 25
11:57:59 1996

Copyright (c) Oracle Corporation 1994. All rights reserved.

Converter Option:              Data File to SLF File
---------------------------------------------------
Dictionary Control File:       sdtable1.ctl
Data Control File:             table1.ctl

Input Data File:               table1.dat
Output File:                   table12b.slf
Log File:                      NONE
Bad File:                      NONE

Bind Size (bytes):             65536
Errors Allowed:                50
Sort Option:                   Yes

Converting...
Sorting...

Total Errors encountered: 0
Total Bad lines skipped:  0
SD*Converter Finished
```

The table control file TABLE12B.SLF is created in the directory where SD*Converter is executed.

An example data control file is provided in the "Data Control Files" section of this chapter. An example table control file is provided in the"Table Control Files" section of this chapter.

Example II The following example uses a Spatial Data Option data dictionary and the data control file **table1.ctl**:

```
% sdconv userid=herman/vampire tablecontrol=table1
datacontrol=table1.ctl data=table1.dat slf=table1.slf
```

The output appears as follows:

```
SD*Converter: Release 7.3.2.0.0 – Production on Sun Feb 25
11:57:59 1996

Copyright (c) Oracle Corporation 1994. All rights reserved.

Converter Option:             Data File to SLF File
---------------------------------------------------
Dictionary Control Table:     table1
Data Control File:            table1.ctl

Input Data File:              table1.dat
Output File:                  table1.slf
Log File:                     NONE
Bad File:                     NONE

Bind Size (bytes):            65536
Errors Allowed:               50
Sort Option:                  Yes

Converting...
Sorting...

Total Errors encountered: 0
Total Bad lines skipped:  0
SD*Converter Finished
```

Example III The following example uses a Spatial Data Option data dictionary, and converts the data from an Oracle7 table:

```
% sdconv userid=herman/vampire tablecontrol=table1
table=dim1dim2_table1 slf=table12c.slf
```

The output appears as follows:

```
SD*Converter: Release 7.3.2.0.0 – Production on Sun Feb 25
11:44:50 1996

Copyright (c) Oracle Corporation 1994. All rights reserved.

Converting Option:            Oracle Table to SLF File
------------------------------------------------------
Dictionary Control Table:     table1

Input Data Table:             dim1dim2_table1
Output File:                  table12c.slf
Log File:                     NONE
Bad File:                     NONE

Bind Size (bytes):            65536
Errors Allowed:               50
Sort Option:                  Yes

Converting...
Sorting...

Total Errors encountered: 0
Total Bad lines skipped:  0
SD*Converter Finished
```

Example IV The following is an example of a bad file:

```
Error occurred in line 1 around (–..76DARN6)
 –..76DARN6  46.185675    13

SDO–5221: ’–..76DARN6’ is not the expected value in the data file

Error occurred in line 5 around (DIM1)
 –760049.88  46.205923     1

SDO–1661: dimensional value for ’DIM1’ is out of bounds
```

Table Control Files

The table control file describes the spatial table where the converted data will be stored, including the following:

- column definition information, including column names and datatypes, and the partition key column
- dimension definition information, including the dimension name, HHCODE column name, dimension number, dimension range, and the scale

Table Control File Syntax

```
COLUMN column_name {VARCHAR2|NUMBER|DATE|RAW|CHAR|HHCODE
[PARTITION KEY]}
DIMENSION dimension_name hhcode_column_name (dimension_number,
lower_boundary, upper_boundary, scale)
```

For complete syntax and instructions for defining columns, see ”Table Control File Column Definition Syntax” in this section.

For complete syntax and instructions for defining dimensions, see ”Table Control File Dimension Definition Syntax” in this section.

Usage Notes

Consider the following points when creating a table control file:

- A table control file is used when a Spatial Data Option data dictionary is not used as the source of the data dictionary information for the spatial table.
- At least one HHCODE column must be specified in the table control file. If there is more than one HHCODE column, one of them must be specified as the partition key.
- The PARTITION KEY keyword is applicable only to the HHCODE datatype, and there can only be one specified per spatial table. This indicates which HHCODE column to use to partition the data when loading.
- Column and dimension names can be a maximum of 30 characters.
- The column and dimension names must exist in the spatial table into which the data will be loaded. These names are verified in either the Spatial Data Option data dictionary or the table control file during conversion. If a discrepancy exists, an error occurs.
- The size of the column datatype must be provided, except for NUMBER, which defaults to (38,10).
- NOT NULL applies to all column datatypes. The partition key column is always NOT NULL, whether or not it is specified.

- The HHCODE column name in the column definition must match the HHCODE column name in the dimension definition. Dimension names must be unique for all HHCODE columns.
- In the dimension definition, the lower boundary value must be lower than the upper boundary value.
- The order of definitions is not important; they can appear in any order. For instance, the line entry for a dimension definition can appear before the line entry for a column definition.
- Case is ignored; everything is converted to uppercase when processed. Use any combination of uppercase and lowercase you find makes the file more comprehensible when viewed.
- Empty lines, leading blanks, and tabs in the table control file are ignored by SD*Converter.
- A space is not required between the column type and the parenthesis, but everything else must be separated by spaces or tabs.
- Comments are preceded by a pound sign (#).

Example The following is an example of a table control file:

```
COLUMN     attribute     NUMBER(38,10)
COLUMN     hhcolumn      HHCODE PARTITION KEY
DIMENSION  dim1          hhcolumn (1,–180,180,7)
DIMENSION  dim2          hhcolumn (2, –90,90,7)
```

Table Control File Column Definition Syntax

```
COLUMN column_name {VARCHAR2|NUMBER|DATE|RAW|CHAR|HHCODE
[PARTITION KEY]}
```

COLUMN	is the keyword to define the column information.
column_name	specifies the name of the column. Maximum length is 30 characters.
VARCHAR2(size)	specifies column datatype as VARCHAR2.
NUMBER(prec,scale) or (prec)	specifies column datatype as NUMBER.
DATE	specifies column datatype as DATE.
RAW(size)	specifies column datatype as RAW.
CHAR(size)	specifies column datatype as CHAR.
HHCODE	specifies column datatype as HHCODE.
PARTITION KEY	specifies that it is the HHCODE column to use to partition the data. Only used with the HHCODE datatype.

Example The following is an example of a table control file column definition:

```
COLUMN     attribute      NUMBER(38,10)
COLUMN     hhcolumn       HHCODE PARTITION KEY
```

Table Control File Dimension Definition Syntax

DIMENSION *dimension_name hhcode_column_name* (*dimension_number, lower_boundary, upper_boundary, scale*)

DIMENSION	is the keyword to define the dimension information.
dimension_name	specifies the dimension name.
hhcode_column_name	specifies the HHCODE column name.
dimension_number	specifies the dimension number.
lower_boundary	specifies the lower boundary of the dimension range.
upper_boundary	specifies the upper boundary of the dimension range.
scale	specifies the scale.

Example The following is an example of a table control file dimension definition:

```
DIMENSION  dim1           hhcolumn (1,-180,180,7)
DIMENSION  dim2           hhcolumn (2, -90,90,7)
```

Data Control Files

The data control file describes the format of the data to be converted when the source of the data is not a Spatial Data Option data dictionary, including the following:

- format information: ASCII or binary
- record length of the data
- column definition information, including column names and datatypes
- dimension definition information, including the dimension name, the name of the HHCODE column containing the dimension, the position of the column, and the datatype

Data Control File Syntax

```
{ASCII|BINARY}
FIXED record_length
COLUMN column_name POSITION {((number:number))|(number)}
{DATE date_format_string|INTEGER|SMALLINT|FLOAT|DOUBLE|BYTEINT|
RAW|CHAR} [NULLIF POSITION {NE|!=|<>|EQ|==|=}'char_string']
DIMENSION dimension_name hhcode_column_name POSITION
{(number:number)|(number)}
{DATE date_format_string|INTEGER|SMALLINT|FLOAT|DOUBLE|BYTEINT}
```

ASCII	specifies the file format as ASCII.
BINARY	specifies the file format as binary.
FIXED	specifies the file format as FIXED.
record_length	specifies the record length.
COLUMN	is the keyword to define the column information.
DIMENSION	is the keyword to define the dimension information.

For complete syntax and instructions for defining columns, see "Data Control File Column Definition Syntax" in this section.

For complete syntax and instructions for defining dimensions, see "Data Control File Dimension Definition Syntax" in this section.

Usage Notes

Consider the following points when creating a data control file:

- The datatype specifications in the data control file tell SD*Converter how to interpret the information in the source data file. SD*Converter extracts data from a field in the input file and converts it into SLF, guided by the datatype specification for each column in the data control file.
- SD*Converter does not recognize datatype specifications for Oracle internal datatypes such as NUMBER or VARCHAR2, but uses the SQL*Loader datatypes which can be produced with standard programming languages (native datatypes). For more information on specifying datatypes, see "Specifying Datatypes" in Chapter 6, "SQL*Loader Control File Reference," *Oracle7 Server Utilities.*
- The data files described by the data control file must be binary or ASCII, and have fixed record lengths. If the file is in ASCII, the record length includes the End Of Line (EOL) character.
- The first two lines of the data control file must be the file type, and file format followed by the record length. The order of lines is important; the file type and file format must be the first and second non–comment lines respectively.
- The record length should be equal to the size of the data file in bytes divided by the number of records in the file.
- The order of definitions is not important; they can appear in any order. For instance, the line entry for a dimension definition can appear before the line entry for a column definition.

- Case is not important. Everything is converted to uppercase when processed.
- Column and dimension names can be a maximum of 30 characters.
- When defining dates in a data control file, you must use a valid date format string. For a list of valid formats as well as guidelines for using them, see Table 6 – 2 in the section ”Date Format Elements” in Chapter 6, ”SD*SQL Kernel Functions.”
- The DATE datatype can be used as either a column field or as a dimension. If it is used as a column field, it is a DATE field. If you use a date string as a dimension, the date is converted internally into a decimal Julian date to the specified accuracy, and then is encoded into the HHCODE.
- The MLS date format element is ignored if the data is loaded into an Oracle7 DATE field rather than as a dimension.
- Comments are preceded by a pound sign (#).

Example I The following example is the data control file **table1.ctl**:

```
ASCII

FIXED 33

DIMENSION  dim1   hhcolumn       POSITION (2:11)   FLOAT
DIMENSION  dim2   hhcolumn       POSITION (14:23)  FLOAT

COLUMN     attribute             POSITION (25:29)  INTEGER
```

Example II The following example shows how to use TIME in a data control file:

```
# Ctl file description of time file

ASCII

FIXED 81

DIMENSION dim1         hhcolumn POSITION (1:11)  FLOAT
DIMENSION dim2         hhcolumn POSITION (15:25) FLOAT
DIMENSION time         hhcolumn POSITION (50:71) DATE
'DD-MON-YY-HH-MI-SS-MLS'

COLUMN    dtime                 POSITION (29:37) DATE 'DD-MON-YY'
COLUMN    attribute             POSITION (42:47) INTEGER
COLUMN    descr                 POSITION (76:80) CHAR
```

Data Control File Column Definition Syntax

```
COLUMN column_name POSITION {(number:number)|(number)}
{DATE date_format_string|INTEGER|SMALLIN|FLOAT|DOUBLE|BYTEINT|RAW|
CHAR} [NULLIF POSITION {NE|!=|<>|EQ|==|=} 'char_string']
```

COLUMN	is the keyword to define the column information.
column_name	specifies the name of the column.
POSITION	is the keyword to specify the position of the column in the record.
number:number	specifies the position of the column using the format *start_position:end_position*.
number	specifies the position of the column as a number.
DATE	specifies the column datatype as DATE. Native datatype is char array in C.
date_format_string	specifies the format mask for a DATE datatype column.
INTEGER	specifies the column datatype as INTEGER. Datatype is natural work type for platform (short) in C.
SMALLINT	specifies the column datatype as SMALLINT. Native datatype is natural small word type (short) in C.
FLOAT	specifies the column datatype as FLOAT. Native datatype is natural floating point (float) in C.

DOUBLE	specifies the column datatype as DOUBLE. Datatype natural high precision floating point (double) in C.
BYTEINT	specifies the column datatype as BYTEINT. Native datatype is single byte integer or character (char) in C.
RAW	specifies the column datatype as RAW. Native datatype is char array in C.
CHAR	specifies the column datatype as CHAR. Native datatype is char array in C.
NULLIF	is the keyword to specify the condition if the value is NULL.
POSITION	is the keyword to specify the position of the column. Optional; only defined if NULLIF is specified.
NE	is a comparison operator specifying not equal. Optional; only defined if NULLIF is specified.
!=	is a comparison operator specifying not equal. Optional; only defined if NULLIF is specified.
<>	is a comparison operator specifying not equal. Optional; only defined if NULLIF is specified.
EQ	is a comparison operator specifying equal. Optional; only defined if NULLIF is specified.
==	is a comparison operator specifying equal. Optional; only defined if NULLIF is specified.
=	is a comparison operator specifying equal. Optional; only defined if NULLIF is specified.
string	specifies a string of characters enclosed within single or double quotation marks that is compared to the comparison field. It is a NULL indicator string. Optional; only defined if NULLIF is specified.

Example The following is an example of a data control file column definition:

```
COLUMN    attribute            POSITION (25:29) INTEGER
```

Data Control File Dimension Definition Syntax

```
DIMENSION dimension_name hhcode_column_name POSITION
{(number:number)|(number)}{DATE date_format_string
|INTEGER|SMALLINT|FLOAT|DOUBLE|BYTEINT}
```

DIMENSION	is the keyword to define the dimension definition.
dimension_name	specifies the name of the dimension.
hhcode_column_name	specifies the name of the HHCODE column containing the dimension.
POSITION	is the keyword to specify the position of the column in the record.
number:number	specifies the position of the column using the format *start_position:end_position*.
number	specifies the position of the column using a number.
DATE	specifies the column datatype as DATE. Native datatype is char array in C.
date_format_string	specifies the format mask for a DATE datatype dimension. For a list of valid formats as well as guidelines for using them, see Table 6 – 2 in the section "Date Format Elements" in Chapter 6, "SD*SQL Kernel Functions."
INTEGER	specifies the column datatype as INTEGER. Native datatype is natural word type for platform in C.
SMALLINT	specifies the column datatype as SMALLINT. Native datatype is natural small word type (short) in C.
FLOAT	specifies the column datatype as FLOAT. Native datatype is natural small word type (short) in C.
DOUBLE	specifies the column datatype as DOUBLE. Native datatype is natural high precision floating point (double) in C.
BYTEINT	specifies the column datatype as BYTEINT. Native datatype is single byte integer or character (char) in C.

Example The following is an example of a data control file dimension definition:

```
DIMENSION dim1         hhcolumn POSITION (1:11)  FLOAT
```

Related Topics

- SD*Loader utility
- user–developed SLF converter

SD*Loader

Purpose

This utility loads data from SLF files into a partitioned table in an Oracle7 database.

Prerequisites

You must have one or more SLF files in the correct format to load data.

Ensure that you have sufficient disk space to accommodate the temporary tables that are created if the data is sorted during the load. For information on calculating space requirements, see "Calculating Table Size Requirements" in Chapter 4, "Administration."

Syntax

```
SDLOAD USERID=username/password SLF=slf_filename
SDTABLE=sd_tablename [LOG=log_filename] [BINDSIZE=65536|bindsize]
[ROLLBACK=rollback_segment_name] [DIRECT=TRUE|FALSE]
[CTLKEEP=TRUE|FALSE]
```

Keywords and Parameters

USERID	is the keyword to specify username and password.
username/password	specifies the username and password.
SLF	is the keyword to specify name of the SLF file to be loaded.
slf_filename	specifies the name of the SLF file.
SDTABLE	is the keyword to specify name of the spatial table into which to load the data.
sd_tablename	specifies the name of the spatial table.
LOG	is the keyword to specify name of the log file created by SD*Loader to store information about the load process. Optional; if not specified, no log file is written, and errors are displayed on standard output.
log_filename	specifies a log filename.
BINDSIZE	is the keyword to specify size of bind array in bytes. Optional; default is 65536 bytes.
65536	is the default bind array size in bytes.
bindsize	specifies a bind array size in bytes.
ROLLBACK	is the keyword to specify a particular rollback segment to use during the partitioning process. Optional; default is selected by the Oracle7 database from available rollback segments.

rollback_segment_name	specifies a rollback segment other than the default.
DIRECT	is the keyword to specify whether to use a direct path load. Optional; values are TRUE, FALSE. Default is TRUE.
TRUE	is the default parameter for DIRECT. Specifies that a direct path load will be used.
FALSE	specifies not to use a direct path load.
CTLKEEP	is the keyword to specify whether to keep the data control files generated by SD*Loader during a load. Optional; values are TRUE, FALSE. Default is FALSE.
TRUE	specifies to keep the control files generated during a load.
FALSE	is the default parameter for CTLKEEP. Specifies not to keep the control files generated during a load.

Usage Notes

Consider the following points when running the SD*Loader utility:

- Execute SD*Loader from the operating system command line by entering the SDLOAD command followed by arguments.
- SD*Loader creates data control files during the load process, and by default discards these files afterwards. You can keep these files for debugging purposes by setting CTLKEEP to TRUE.
- You cannot load data of the RAW or LONG RAW datatype.

For information on the Oracle SQL*Loader utility, see the *Oracle7 Server Utilities*.

Example

The following example loads data from the file **table1.slf** into the TABLE1 table:

```
% sdload userid=herman/vampire slf=table1.slf sdtable=table1
```

The output appears as follows:

```
SLF file table1.slf contains 42545 records.
Starting load/partition process...

Commit point reached for TABLE1_P000000001: 8680
Logical record count in SLF file: 8680

Commit point reached for TABLE1_P000000002: 4773
Logical record count in SLF file: 13453

Commit point reached for TABLE1_P000000003: 4817
Logical record count in SLF file: 18270

Commit point reached for TABLE1_P000000004: 2686
Logical record count in SLF file: 20956

Commit point reached for TABLE1_P000000005: 1754
Logical record count in SLF file: 22710
.
.
.

Load successfully completed.
```

Related Topics

- SD*Converter utility
- user–developed SLF converter

CHAPTER

6

SD*SQL Kernel Functions

This chapter contains an introduction to and descriptions of all Spatial Data Option kernel functions, presented alphabetically.

Introduction

An essential feature of working with spatial components is the ability to select, sort, or otherwise manipulate the spatial data. The SD*SQL kernel functions provide access to spatial data. The data selected can consist of actual HHCODEs, or can be derived using other SD*SQL functions.

Some functions return information about existing spatial data; for example, the number of dimensions encoded in an HHCODE.

Some functions take original dimensional values and create HHCODEs, or decode them to return their original values. Other functions are used with the selection of windows of data for further manipulation and analysis.

These functions are used in SQL and PL/SQL in combination with SQL commands in the same manner as other Oracle SQL functions, except that the SD*SQL functions operate on the spatial data.

Considerations

The following considerations apply when using SD*SQL kernel functions with Oracle products:

Using SD*SQL Kernel Functions in PL/SQL

When calling SD*SQL kernel functions from PL/SQL, you must prefix the function name with the package name MD. The function HHDECODE() is referenced as MD.HHDECODE() as in the following example:

Example

```
DECLARE
    * NUMBER;
BEGIN
    SELECT MD.HHDECODE(location, 1, –180, 180)
        INTO *
        FROM table1_P000000001
    WHERE rownum=1;
END;
```

Using SD*SQL Kernel Functions in Pro*C

Pro*C fully parses C language syntax and SQL commands in the source file. When it tries to parse a command such as the following, it does not recognize SD*SQL kernel functions, and exits with an error message:

```
EXEC SQL SELECT HHENCODE(123.456,0,360,7,12,0,180,7)
    INTO :x
    FROM sys.dual;
```

To include SD*SQL kernel functions in Pro*C programs, use dynamic SQL as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION
   char *stmt = ”SELECT HHENCODE(123.456,0,360,7,12,0,180,7) \
                  FROM sys.dual”;
.
.
EXEC SQL END DECLARE SECTION;
.
.
EXEC SQL PREPARE S FROM :stmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C;
```

SD*SQL Kernel Functions By Type

Table 6 – 1 presents the kernel functions grouped by type, with a brief description of the purpose of each.

Function Type	Function Name	Purpose
Encoding and Decoding Functions	HHCOLLAPSE	Removes one or more dimensions encoded in an HHCODE to create a new HHCODE with fewer dimensions.
	HHCOMPOSE	Builds a new HHCODE using the dimensional information already encoded in an HHCODE.
	HHDECODE	Retrieves the original dimensional value for a specific dimension, in a specified dimension range, of an HHCODE.
	HHENCODE	Encodes an HHCODE from the original dimensional data.
HHCODE Metadata Functions	HHBYTELEN	Returns the number of bytes to allocate to store an HHCODE.
	HHCELLBNDRY	Calculates the quadrant enclosing an HHCODE to a specified level of resolution. A cell boundary of as many dimensions as have been defined can be determined.
	HHCELLSIZE	Returns the area or volume of a cell at a given level of resolution.
	HHLENGTH	Returns the maximum number of levels of resolution encoded in an HHCODE.
	HHLEVELS	Returns the number of levels of resolution encoded for a particular dimension range and scale. Inverse of HHPRECISION.
	HHNDIM	Returns the number of dimensions encoded in an HHCODE.
	HHPRECISION	Returns number of digits of scale for a given range and number of levels. Inverse of HHLEVELS.

Table 6 – 1 Kernel Functions

continued on next page

Function Type	Function Name	Purpose
HHCODE Commonality Functions	HHCOMMONCODE	Returns the common code between two HHCODEs, representing the super–quadrant enclosing both.
	HHMATCH	Compares two HHCODEs and returns the number of matching levels of resolution, starting from the first level.
Calculation Functions	HHDISTANCE	Applies a Euclidean or Manhattan distance calculation to return the distance between two HHCODEs.
	HHSUBSTR	Returns a portion of an HHCODE based on the start and end resolution levels.
Sorting and Grouping Functions	HHGROUP	Used in a GROUP BY clause to group the HHCODE data in the table.
	HHORDER	Used during ORDER BY operations to properly sort HHCODE data.
Window Identification Functions	HHIDLPART	Used during window extraction of a subset of line data; identifies the partitions that enclose, are enclosed by, overlap, are equal to, or are outside the query window.
	HHIDLROWS	Used during window extraction of a subset of line data; identifies the rows within a partition that are within the query window.
	HHIDPART	Used during window extraction of a subset of point data; identifies the partitions that enclose, are enclosed by, overlap, are equal to, or are outside the query window.
	HHIDROWS	Used during window extraction of a subset of point data; identifies the rows within a partition that are within the query window.
Date and Time Functions	HHCLDATE	Returns the calendar date, given a specific decimal Julian date created using the HHJLDATE function.
	HHJLDATE	Returns a decimal Julian date from a calendar date.

Table 6 – 1 Kernel Functions

Date Format Elements

The SD*SQL kernel functions that manipulate dates use a subset of the standard Oracle date format elements, plus an additional millisecond specification. These functions are as follows:

- HHCLDATE function
- HHJLDATE function

In addition to the SD*SQL kernel functions, the following Spatial Data Option components use the same date format when handling dates:

- data control files that use date as a dimension
- user–developed SLF converters that use date as a dimension

Table 6 – 2 lists and describes the valid format elements:

Format Element	Description	Default Value
YYYY	4 digit year	0001
AD/BC	AD/BC indicator	AD
MON	3 character month	JAN
MM	2 digit month (01–12)	01
DD	2 digit day (01–31)	01
HH	2 digit hour (00–23)	00
MI	2 digit minute (00–59)	00
SS	2 digit second (00–59)	00
MLS	3 digit millisecond (000–999)	000

Table 6 – 2 Valid Date Formats

Valid date separators are as follows:

- space
- comma ,
- dash –
- period .
- slash /
- colon :

Usage Notes

Consider the following points when formatting dates:

- The number of digits must match the mask for the correct values to be translated.

- You must specify at least a year and a month format element. The default values for the remainder of the format elements are used. For example, if you specify the year and the month as follows:

  ```
  'YYYY–MM'
  ```

  and the conversion for the year and month are 1995 and October, the function interprets as follows:

  ```
  '1995–AD–10–01–01–00–00–00–000'
  ```

- The optional modifiers AD and BC are entered as shown in Table 6 – 2.
- The default hour format is 24 hour, which differs from the standard format used in the Oracle7 database.
- Use a combination of formats and separators that can be easily understood, or use the conventional notation for your organization.

Example The following is an example of how to specify the date format:

```
'DD/MM/YYYY HH.MI.SS.MLS'
```

HHBYTELEN

Purpose

This function determines the size, in bytes, of an HHCODE based on the number of dimensions and levels encoded in the dimensions.

Syntax

```
HHBYTELEN (number_of_dimensions, maximum_levels)
```

Keywords and Parameters

number_of_ dimensions	specifies the number of dimensions. Datatype is NUMBER.
maximum_levels	specifies the maximum number of levels encoded for all dimensions. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- Use HHBYTELEN in Pro*C programs to determine how much space to allocate for an HHCODE.
- Use the HHLEVELS function to determine the number of levels of resolution from the scale and boundaries of a dimension.

Example

The following is an example of of the HHBYTELEN function:

```
SQL> SELECT HHBYTELEN(3,32) hsize FROM dual;
```

The output appears as follows:

```
HSIZE
-----
   18
```

Related Topics

- HHLEVELS function

HHCELLBNDRY

Purpose

This function is used to calculate the quadrant enclosing an HHCODE to a specified level of resolution. It returns a minimum and maximum boundary, one boundary at a time, for one dimension at a time, in the original coordinate system values. You can determine a cell of as many dimensions as were encoded.

Syntax

```
HHCELLBNDRY (hhcode_expression, dimension_number, lower_boundary,
upper_boundary, number_of_levels, {'MIN'|'MAX'})
```

Keywords and Parameters

hhcode_expression	is an expression that evaluates to an HHCODE. Datatype is RAW.
dimension_number	specifies the dimension number. Datatype is NUMBER.
lower_boundary	specifies the lower boundary of the dimension range. Datatype is NUMBER.
upper_boundary	specifies the upper boundary of the dimension range. Datatype is NUMBER.
number_of_levels	specifies the number of levels of resolution. Datatype is NUMBER.
MIN	is the minimum bounding coordinate indicator. Datatype is VARCHAR2.
MAX	is the maximum bounding coordinate indicator. Datatype is VARCHAR2.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- This function returns the boundaries of the cell you are decoding along each dimension. The boundaries are defined by two values; a lower boundary value and an upper boundary value. Since the kernel functions can only return one value at a time, MIN and MAX are used to specify which boundary the user wants to select.
- Because the HHCODE itself is a cell, however small, rather than a specific point, the level of resolution you define with HHCELLBNDRY determines the granularity of the result.

- The HHCELLBNDRY function is valid for point data only.
- Figure 6 – 1 is a gridded space representing a two–dimensional region. The gridlines show how the space is decomposed into quadrants, which decrease in size by each level of resolution. A quadrant number for a specifically defined cell is determined by appending numbers as the space subdivides. A point has been marked in the upper right corner to represent an HHCODE whose cell boundaries are to be determined at different levels of resolution.

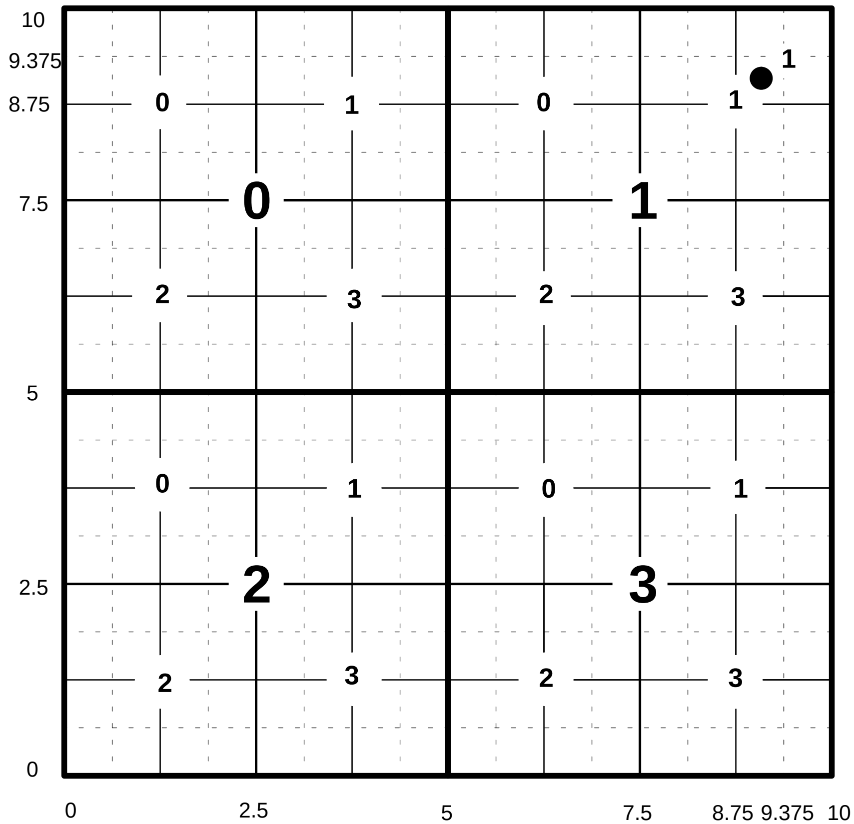

Figure 6 – 2 HHCELLBNDRY Function

In Figure 6 – 2, the value returned indicates both the level and the quadrant:

at level 1, quadrant 1:	*x* is between 5 and 10 *y* is between 5 and 10
at level 2, quadrant 11:	*x* is between 7.5 and 10 *y* is between 7.5 and 10

at level 3, quadrant 111:	*x* is between 8.75 and 10 *y* is between 8.75 and 10
at level 4, quadrant 1112:	*x* is between 8.75 and 9.375 *y* is between 8.75 and 9.375

As the number of levels is increased, the cell boundaries converge upon the point. The higher the level specified, the greater the granularity.

Example I The following example returns the minimum and maximum boundaries for DIMENSION_1 and DIMENSION_2 at level 19, in the original coordinate system values, in the TABLE1 table:

```
SQL> SELECT HHCELLBNDRY(hhcolumn,1,–180,180,19,’MIN’) min_dim1,
2 HHCELLBNDRY(hhcolumn,1,–180,180,19,’MAX’) max_dim1,
3 HHCELLBNDRY(hhcolumn,2,–90,90,19,’MIN’) min_dim2,
4 HHCELLBNDRY(hhcolumn,2,–90,90,19,’MAX’) max_dim2
5 FROM table1_p000000001
6 WHERE rownum <= 10;
```

The output appears as follows:

```
MIN_DIM1   MAX_DIM1   MIN_DIM2   MAX_DIM_2
---------- ---------- ---------- ----------
–76.590500 –76.589813 46.0011292 46.0014725
–76.588440 –76.587753 46.0011292 46.0014725
–76.586380 –76.585693 46.0011292 46.0014725
–76.590500 –76.589813 46.0025024 46.0028458
–76.589813 –76.589127 46.0038757 46.0042191
–76.588440 –76.587753 46.0025024 46.0028458
–76.586380 –76.585693 46.0025024 46.0028458
–76.587753 –76.587067 46.0038757 46.0042191
–76.590500 –76.589813 46.0052490 46.0055923
–76.589813 –76.589127 46.0062790 46.0066223

10 rows selected.
```

Example II HHCELLBNDRY can be used to determine the extent of a partition. For example, consider a two–dimensional example where the dimensions are defined as follows:

dimension 1:	– 180 to + 180
dimension 2:	– 90 to + 90

The query to decode any partition extent in the Spatial Data Option data dictionary is as follows:

```
SQL> SELECT partition_table_name,
    HHCELLBNDRY(common_hhcode,1,–180,180,common_level,’MIN’)
                   min_dim1,
    HHCELLBNDRY(common_hhcode,1,–180,180,common_level,’MAX’)
```

```
                 max_dim1,
   HHCELLBNDRY(common_hhcode,2,–90,90,common_level,’MIN’)
                 min_dim2,
   HHCELLBNDRY(common_hhcode,2,–90,90,common_level, ’MAX’)
                 max_dim2
      FROM user_md_partitions
      WHERE md_table_name=’TABLE1’
      ORDER BY partition_table_name;
```

The output appears as follows:

```
PARTITION_TABLE_NAME  MIN_DIM1   MAX_DIM1   MIN_DIM2   MAX_DIM2
--------------------- ---------- ---------- ---------- ----------
TABLE1_P000000001   –76.640625  –75.9375   45.703125  46.0546875
TABLE1_P000000002   –76.640625  –76.464844 46.0546875 46.1425781
TABLE1_P000000003   –76.640625  –76.464844 46.1425781 46.2304688
TABLE1_P000000004   –76.464844  –76.289063 46.0546875 46.1425781
TABLE1_P000000005   –76.464844  –76.289063 46.1425781 46.2304688
TABLE1_P000000006   –76.640625  –76.289063 46.2304688 46.40625
TABLE1_P000000007   –76.289063  –76.113281 46.0546875 46.1425781
TABLE1_P000000008   –76.289063  –76.113281 46.1425781 46.2304688
TABLE1_P000000009   –76.113281  –75.9375   46.0546875 46.1425781
TABLE1_P00000000A   –76.113281  –76.069336 46.1425781 46.1645508
TABLE1_P00000000B   –76.069336  –76.025391 46.1425781 46.1645508
TABLE1_P00000000C   –76.069336  –76.025391 46.1645508 46.1865234
TABLE1_P00000000D   –76.113281  –76.025391 46.1865234 46.2304688
TABLE1_P00000000E   –76.025391  –76.014404 46.1810303 46.1865234
TABLE1_P00000000F   –76.181803  –76.181803 47.0004119 47.000412
TABLE1_P00000000G   –76.640625  –75.9375   47.8125    48.1640625

16 rows selected.
```

Related Topics

None

HHCELLSIZE

Purpose

This function returns the *n*–dimensional volume of a cell at a given level of resolution.

Syntax

```
HHCELLSIZE (lower_boundary, upper_boundary, number_of_levels
[lower_boundary, upper_boundary, number_of_levels...])
```

Keywords and Parameters

lower_boundary specifies the lower boundary of the dimension range. Datatype is NUMBER.

upper_boundary specifies the upper boundary of the dimension range. Datatype is NUMBER.

number_of_levels specifies the number of levels of resolution. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- To determine the size of a cell in *n*–dimensions, specify the level of resolution, and the lower and upper boundaries for each dimension.
- HHCELLSIZE does not require an HHCODE as an argument, because at any given level the area or volume of a cell can be predicted from the dimension information.
- This function is valid for point data only.

Example I The following example returns the cell size at level 1:

```
SQL> SELECT HHCELLSIZE(–180,180,1,–90,90,1)
2 FROM dual;
```

The output appears as follows:

```
HHCELLSIZE(–180,180,1,–90,90,1)
-------------------------------
16200
```

Example II The following example returns the cell size at level 2:

```
SQL> SELECT HHCELLSIZE(–180,180,2,–90,90,2)
2 FROM dual;
```

The output appears as follows:

```
HHCELLSIZE(–180,180,2,–90,90,2)
-------------------------------
4050
```

Example III The following example returns the cell size at level 3:

```
SQL> SELECT HHCELLSIZE(–180,180,3,–90,90,3)
2 FROM dual;
```

The output appears as follows:

```
HHCELLSIZE(–180,180,3,–90,90,3)
-------------------------------
1012.5
```

Related Topics

None

HHCLDATE

Purpose

This function returns the calendar date when given a decimal Julian date that was obtained using the HHJLDATE function.

Syntax

HHCLDATE (*julian_date*, '*date_format_string*')

Keywords and Parameters

julian_date	specifies the decimal Julian date. Datatype is NUMBER.
date_format_string	specifies the format mask to use. Datatype is VARCHAR2

Returns

This function returns a character string.

Usage Notes

You must use a valid date format string. For a list of valid formats as well as guidelines for using them, see Table 6 – 2 in the section "Date Format Elements" in this chapter.

Example

The following is an example of the HHCLDATE function:

```
SQL> SELECT HHCLDATE(2438316.190473264, 'YYYY/MM/DD-HH:MI:SS:MLS')
2 FROM dual;
```

The output appears as follows:

```
HHCLDATE(2438316.190473264,'YYYY/MM/DD-HH:MI:SS:MLS')
-----------------------------------------------------
1963/10/13-04:34:16:890
```

Related Topics

- HHJLDATE function

HHCOLLAPSE

Purpose

This function removes one or more dimensions encoded in an HHCODE to create a new HHCODE with fewer dimensions.

Syntax

```
HHCOLLAPSE (hhcode_expression, dimension_number
[, hhcode_expression, dimension_number, ...])
```

Keywords and Parameters

hhcode_expression	is an expression that evaluates to an HHCODE. Datatype is RAW.
dimension_number	specifies the dimension number. Datatype is NUMBER.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- At least one dimension must be retained.
- Use the HHCOLLAPSE function to eliminate a small number of dimensions from an HHCODE. To eliminate many dimensions and leave only a small number, or to reorder dimensions, use the HHCOMPOSE function.

Example

The following is an example of the HHCOLLAPSE function:

```
SQL> SELECT HHDECODE(HHCOLLAPSE(hhcolumn,1),1,–90,90) dim2,
      HHDECODE(hhcolumn,2,–90,90) dim2
    FROM table1_p000000001
    WHERE rownum=1;
```

The previous example uses the HHDECODE function to decode the original values. The output appears as follows:

```
DIM2       DIM2
---------- ---------
46.001333  46.001333
```

Related Topics

- HHCOMPOSE function
- HHDECODE function

HHCOMMONCODE

Purpose

This function returns the common code between two HHCODEs. The common code represents the super–quadrant containing both of the specified HHCODEs.

Syntax

```
HHCOMMONCODE (hhcode_expression, hhcode_expression)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

Returns

This function returns an HHCODE.

Usage Notes

Both HHCODEs must have the same number and types of dimensions.

Example

The following is an example of the HHCOMMONCODE function:

```
SQL> SELECT HHCOMMONCODE(MIN(hhcolumn),MAX(hhcolumn)) superquad
2 FROM table1_p000000001;
```

The output appears as follows:

```
SUPERQUAD
----------------
7083800909020000
```

Related Topics

None

HHCOMPOSE

Purpose

This function builds a new HHCODE using the dimension information already encoded in an HHCODE. It can be used to create an HHCODE of fewer dimensions, or to reorder the dimensions.

Syntax

```
HHCOMPOSE (hhcode_expression, dimension_number
[, dimension_number...])
```

Keywords and Parameters

hhcode_expression	is an expression that evaluates to an HHCODE. Datatype is RAW.
dimension_number	specifies the dimension number. Datatype is NUMBER.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- You can specify up to as many dimensions as are encoded in the original HHCODE. You can also enter them in any order, which allows you to change the order of the dimensions if you wish.
- This function saves you from having to decode an HHCODE column, and then encode only the dimensions you want.
- Use HHCOMPOSE to build an HHCODE with a small number of dimensions from an original HHCODE with many dimensions. To remove only a few dimensions from an HHCODE, use the HHCOLLAPSE function. To reorder dimensions, use HHCOMPOSE.

Example

The following is an example of the HHCOMPOSE function:

```
SQL> SELECT HHCOMPOSE(hhcolumn,1)
2 FROM table1_p000000001 WHERE rownum=1;
```

The output appears as follows:

```
HHCOMPOSE(HHCOLUMN,1)
---------------------
4989277F20010000
```

Related Topics

- HHCOLLAPSE function

HHDECODE

Purpose

This function returns the value in its original format for a specific dimension, in a specified dimension range, of an HHCODE.

Syntax

HHDECODE (*hhcode_expression*, *dimension_number*, *lower_boundary*, *upper_boundary*)

Keywords and Parameters

hhcode_expression	is an expression that evaluates to an HHCODE. Datatype is RAW.
dimension_number	specifies the dimension number. Datatype is NUMBER.
lower_boundary	specifies the lower boundary of the dimension range. Datatype is NUMBER.
upper_boundary	specifies the upper boundary of the dimension range. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- HHDECODE is called once for each dimension to be decoded.
- This function returns a number that is the value for a specified dimension decoded to the specified range. This does not have to be the range that was originally encoded.

Example

The following is an example of the HHDECODE function:

```
SQL> SELECT HHDECODE(hhcolumn,1,–180,180) dim1, HHDECODE
2 (hhcolumn,2,–90,90) dim2
3 FROM table1_p000000001
4 WHERE rownum=1;
```

The output appears as follows:

```
DIM1       DIM2
---------- ----------
–76.590339 46.001333
```

Related Topics

- HHENCODE function

HHDISTANCE

Purpose

This function calculates either a Euclidean or Manhattan squared distance between two HHCODE values, in *n* dimensions.

Syntax

```
HHDISTANCE ({'EUCLID'|'MANHATTAN'}, hhcode_expression_1,
hhcode_expression_2, lower_boundary_1, upper_boundary_1
[, lower_boundary_n, upper_boundary_n...])
```

Keywords and Parameters

EUCLID	specifies the distance type EUCLID. This specifies using the Euclidean distance calculation to compute the distance. Dataype is VARCHAR2.
MANHATTAN	specifies the distance type MANHATTAN. This specifies using the Manhattan distance calculation to compute the distance. Dataype is VARCHAR2.
hhcode_ expression_1	is an expression that evaluates to an HHCODE. Datatype is RAW.
hhcode expression_2	is an expression that evaluates to an HHCODE. Datatype is RAW.
lower_boundary_1	specifies the lower boundary of the first dimension range. Datatype is NUMBER.
upper_boundary_1	specifies the upper boundary of the first dimension range. Datatype is NUMBER.
lower_boundary_n	specifies the lower boundary of the *n*th dimension range. Datatype is NUMBER.
upper_boundary_n	specifies the upper boundary of the *n*th dimension range. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- This function is valid for point data only.
- The HHCODEs must have the same number and types of dimensions.
- The distance value returned is the distance squared. To get the actual distance, take the square root of the returned value.

- For each HHCODE, provide *n* pairs of lower and upper boundary values. Figure 6 – 3 illustrates this in two dimensions:

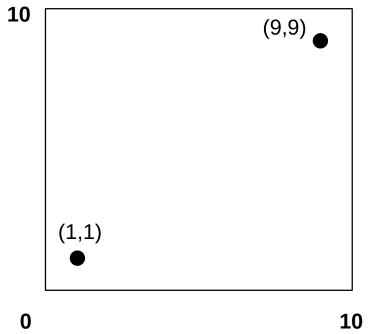

Figure 6 – 3 HHDISTANCE Calculation

Example I

The following example calculates a Euclidean distance using the two–dimensional coordinate values in Figure 6 – 3:

```
SQL> SELECT HHDISTANCE ('EUCLID', HHENCODE (1,0,10,0,1,0,10,0),
2 HHENCODE (9,0,10,0,9,0,10,0),0,10,0,10) distance
3 FROM dual;
```

The output appears as follows:

```
DISTANCE
--------
11.4904852
```

Example II

The following example calculates a Manhattan distance using the two–dimensional coordinate values in Figure 6 – 3:

```
SQL> SELECT HHDISTANCE ('MANHATTAN',HHENCODE (1,0,10,0,1,0,10,0),
 HHENCODE (9,0,10,0,9,0,10,0),0,10,0,10) distance
3 FROM dual;
```

The output appears as follows:

```
DISTANCE
--------
16.25
```

Related Topics

None

HHENCODE

Purpose

This function encodes an HHCODE.

Syntax

HHENCODE (*value*, *lower_boundary*, *upper_boundary*, *scale* [, *value*, *lower_boundary*, *upper_boundary*, *scale* ...])

Keywords and Parameters

value specifies the value to encode for the particular dimension. Datatype is NUMBER.

lower_boundary specifies the lower boundary of the dimension range. Datatype is NUMBER.

upper_boundary specifies the upper boundary of the dimension range. Datatype is NUMBER.

scale specifies the scale. Datatype is NUMBER.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- HHENCODE is the manual equivalent of the conversion process using SD*Converter.
- Parameters are grouped to describe a single dimension. You must repeat them for each value to encode, as follows:

  value, *lower_boundary*, *upper_boundary*, *scale*

- When encoding points, the order of dimensions is not important, but must be consistent for all rows within the table. However, to encode lines, you must use the following format:

  [*x*1, *y*1, *x*2 ,*y*2]

Example

The following is an example of the HHENCODE function:

```
SQL> SELECT hhcolumn, HHENCODE (HHDECODE
2 (hhcolumn,1,-180,180),-180,180,7,
3 HHDECODE(hhcolumn,2,-90,90),-90,90,7) hhcolumn1
4 FROM table1_p000000001
5 WHERE rownum=1;
```

The output appears as follows:

```
HHCOLUMN                   HHCOLUMN1
-------------------------- --------------------------
708394D2492F2AAA1F20020000 708394D2492F2AAA1F20020000
```

Related Topics

- HHDECODE function

HHGROUP

Purpose

This function is used in a GROUP BY clause to group the HHCODE data in a spatial table.

Syntax

```
HHGROUP (hhcode_expression)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- If you are including an HHCODE expression in a GROUP BY clause, you must apply the HHGROUP function.
- If all the HHCODEs in the table you are grouping are the same number of bytes, and the dimension information is the same, then you do not need to apply this function.

Example The following is an example of the HHGROUP function:

```
SQL> SELECT MIN (partition_table_name)
2 FROM user_md_partitions
3 GROUP BY HHGROUP (HHSUBSTR (common_hhcode,1,
4 common_level – 1));
```

The output appears as follows:

```
MIN(PARTITION_TABLE_NAME)
-------------------------
TABLE1_P000000002
TABLE1_P000000003
TABLE1_P000000008
TABLE1_P00000000B
TABLE1_P00000000F
```

Related Topics

- HHORDER function

HHIDLPART

Purpose

This function identifies the location of a partition of line data in a partitioned table in relation to a two–dimensional query window. It identifies whether the partition is inside, equal to, overlaps, outside, or encloses the query window.

Syntax

```
HHIDLPART ({'RANGE'|'PROXIMITY'|'POLYGON'}, COMMON_HHCODE,
lower_boundary_1, upper_boundary_1, lower_boundary_2,
upper_boundary_2, window_definition)
```

Keywords and Parameters

RANGE	specifies a two–dimensional range window to query. Datatype is VARCHAR2.
PROXIMITY	specifies a two–dimensional proximity window to query. Datatype is VARCHAR2.
POLYGON	specifies a two–dimensional polygon window to query. Datatype is VARCHAR2.
COMMON_ HHCODE	is the COMMON_HHCODE column in either the ALL_MD_PARTITIONS or the USER_MD_PARTITIONS Spatial Data Option data dictionary data dictionary views.
lower_boundary_1	specifies the lower boundary of the first dimension range. Datatype is NUMBER.
upper_boundary_1	specifies the upper boundary of the first dimension range. Datatype is NUMBER.
lower_boundary_2	specifies the lower window boundary of the second dimension range. Datatype is NUMBER.
upper_boundary_2	specifies the upper boundary of the second dimension range. Datatype is NUMBER.
window_definition	specifies the window definition. Varies depending on whether it is a range, proximity, or polygon query window. For the syntax and description of each query type, see "Usage Notes" in this section.

Returns

This function returns one of the following descriptions of the relationship between the row and the window, as follows:

INSIDE	The partition is inside the specified window.
EQUAL	The partition is equal to the specified window.
OVERLAP	The partition overlaps the specified window.
OUTSIDE	The partition is outside the specified window.
ENCLOSES	The partition encloses the specified window.

Note: The relationships between the preceding definitions and the specified window are illustrated in the description of the HHIDPART function, in this section.

Usage Notes

Consider the following points when using this function:

- This function is valid for line data only.
- The COMMON_HHCODE column must have four dimensions, because it represents a two–dimensional line. If it does not, an error message is returned.

Range Window Syntax

This is defined by the lower and upper boundaries of the range in two dimensions as follows:

```
lower_window_boundary_1, upper_window_boundary_1,
lower_window_boundary_2, upper_window_boundary_2
```

lower_window_ boundary_1	specifies the lower window boundary of the first dimension. Datatype is NUMBER.
upper_window_ boundary_1	specifies the upper window boundary of the first dimension. Datatype is NUMBER.
lower_window_ boundary_2	specifies the lower window boundary of the second dimension. Datatype is NUMBER.
upper_window_ boundary_2	specifies the upper window boundary of the second dimension. Datatype is NUMBER.

Example

The following example determines which partitions intersect a range window, defined as follows:

dimension 1:	–76.15 to –76.05
dimension 2:	45.8 to 46.185

```
SQL> SELECT partition_table_name,
2 HHIDLPART('RANGE',common_hhcode,–180,180,–90,90,
3 –76.15,–76.05,45.8,46.185) RELATIONSHIP
4 FROM user_md_partitions
5WHERE md_table_name = 'LINES';
```

The output appears as follows:

```
PARTITION_TABLE_NAME              RELATIONSHIP
-----------------------------------------------
LINES_p000000001                  OVERLAP
LINES_P000000002                  OUTSIDE
.
.
.
```

Proximity Window Syntax

This is defined by a center point and radius in two dimensions as follows:

center_1, center_2, radius

center_1	specifies the first dimensional value of the center point. Datatype is NUMBER.
center_2	specifies the second dimensional value of the center point. Datatype is NUMBER.
radius	specifies the radius in the coordinate system used. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of two center values must be specified.
- The radius and coordinate system are specified in the same units.
- The coordinate system must be Cartesian.

Example

The following example determines which partitions intersect a proximity window, defined by the following:

center point:	76.1, 46.17
radius:	0.025

```
SQL> SELECT partition_table_name,
2 HHIDLPART('PROXIMITY',common_hhcode,-180,180,-90,90,
3 -76.1,46.17,0.025) RELATIONSHIP
4 FROM user_md_partitions
5WHERE md_table_name = 'LINES';
```

The output appears as follows:

```
PARTITION_TABLE_NAME              RELATIONSHIP
-----------------------------------------------
LINES_P000000001                  OVERLAP
LINES_P000000002                  OUTSIDE
.
.
.
```

Polygon Window Syntax

This is defined by a set of 3 to 124 vertex points in two dimensions, as follows:
x1, y1, x2, y2, x3, y3, [*xn, yn...*]

x1	specifies the *x* coordinate of the first vertex point. Datatype is NUMBER.
y1	specifies the *y* coordinate of the first vertex point. Datatype is NUMBER.
x2	specifies the *x* coordinate of the second vertex point. Datatype is NUMBER.

y2	specifies the *y* coordinate of the second vertex point. Datatype is NUMBER.
x3	specifies the *x* coordinate of the third vertex point. Datatype is NUMBER.
y3	specifies the *y* coordinate of the third vertex point. Datatype is NUMBER.
xn	specifies the *x* coordinate of the *n*th vertex point. Datatype is NUMBER.
yn	specifies the *y* coordinate of the *n*th vertex point. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of three vertex points must be specified.
- The vertex points must be specified in order.
- Do not close the polygon window. This means that you do not specify the first and last vertex points as the same.

Example The following example determines which partitions intersect the polygon window, defined by the following 4 points:

point 1:	–76.15, 45.8
point 2:	–76.15, 46.185
point 3:	76.05, 46.185
point 4:	–76.05, 45.8

```
SQL> SELECT partition_table_name,
2 HHIDLPART('POLYGON',common_hhcode,–180,180,–90,90,
3 –76.15,45.8,–76.15,46.185,–76.05,46.185,
4 –76.05,45.8) RELATIONSHIP
5 FROM user_md_partitions
6 WHERE md_table_name = 'LINES';
```

The output appears as follows:

```
PARTITION_TABLE_NAME              RELATIONSHIP
-----------------------------------------------
LINES_P000000001                  OVERLAP
LINES_P000000002                  OUTSIDE
.
.
.
```

Related Topics

- HHIDLROWS function

HHIDLROWS

Purpose

This function identifies the location of a two–dimensional line in relation to a two–dimensional query window. It identifies whether the line is inside, outside, on the boundary of, or overlaps the query window.

Syntax

```
HHIDLROWS ({'RANGE'|'PROXIMITY'|'POLYGON'}, partition_key,
lower_boundary_1, upper_boundary_1, lower_boundary_2,
upper_boundary_2, window_definition)
```

Keywords and Parameters

RANGE	specifies a two–dimensional range window to query. Datatype is VARCHAR2.
PROXIMITY	specifies a two–dimensional proximity window to query. Datatype is VARCHAR2.
POLYGON	specifies a two–dimensional polygon window to query. Datatype is VARCHAR2.
partition_key	specifies the partition key column in a partitioned table; specifies the HHCODE column in a non–partitioned table.
lower_boundary_1	specifies the lower boundary of the first dimension range. Datatype is NUMBER.
upper_boundary_1	specifies the upper boundary of the first dimension range. Datatype is NUMBER.
lower_boundary_2	specifies the lower boundary of the second dimension range. Datatype is NUMBER.
upper_boundary_2	specifies the upper boundary of the second dimension range. Datatype is NUMBER.
window_definition	specifies the window definition. Varies depending on whether it is a range, proximity, or polygon query window. For the syntax and description of each query type, see "Usage Notes" in this section.

Returns

This function returns one of the following descriptions of the relationship between the line and the window:

INSIDE	The line is inside the specified window.
OUTSIDE	The line is outside the specified window.

BOUNDARY	The line is on the boundary of the specified window.
OVERLAP	The line overlaps the specified window.

Usage Notes

Consider the following points when using this function:

- This function is valid for line data only.
- The HHCODE column must have four dimensions, because it represents a two–dimensional line. If it does not, an error message is returned.
- Treat all other arguments as two–dimensional.

Range Window Syntax

This is defined by the lower and upper boundaries of the range window in two dimensions, as follows:

```
lower_window_boundary_1, upper_window_boundary_1,
lower_window_boundary_2, upper_window_boundary_2
```

lower_window_ boundary_1	specifies the lower window boundary of the first dimension. Datatype is NUMBER.
upper_window_ boundary_1	specifies the upper window boundary of the first dimension. Datatype is NUMBER.
lower_window_ boundary_2	specifies the lower window boundary of the second dimension. Datatype is NUMBER.
upper_window_ boundary_2	specifies the upper window boundary of the second dimension. Datatype is NUMBER.

Example

The following example determines how many lines from the partition TABLE1_P000000001 lie within a range window, defined as follows:

dimension 1:	–76.15 to –76.05
dimension 2:	45.8 to 46.185

```
SQL> SELECT count(*)
2 FROM lines_p000000001
3 WHERE HHIDLROWS('RANGE',hhcolumn,-180,180,-90,90,
4 -76.15,-76.05,45.8,46.185) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
--------
    1193
```

Proximity Window Syntax

This is defined by a center point and a radius in two dimensions, as follows:

center_1, center_2, radius

center_1	specifies the first dimensional value of the center point.
center_2	specifies the second dimensional value of the center point.
radius	specifies the radius in the coordinate system used.

Guidelines are as follows:

- Only two dimensions can be specified. If more than two dimensions are defined, an error occurs.
- The radius must be specified in the same units as the coordinate system.
- The coordinate system must be Cartesian.

Example

The following example determines how many lines from the partition TABLE1_P00000000D lie within the proximity window defined as follows:

center point:	–76.1, 46.17
radius:	0.025

```
SQL> SELECT count(*)
2 FROM lines_p00000000d
3 WHERE HHIDLROWS('PROXIMITY',hhcolumn,–180,180,–90,90,
4 –76.1,46.17,0.025) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
--------
    7707
```

Polygon Window Syntax

This is defined by a set of 3 to 124 vertex points in two dimensions, as follows:

x1, y1, x2, y2, x3, y3, [*xn, yn*...]

x1	specifies the *x* coordinate of the first vertex point. Datatype is NUMBER.
y1	specifies the *y* coordinate of the first vertex point. Datatype is NUMBER.
x2	specifies the *x* coordinate of the second vertex point. Datatype is NUMBER.
y2	specifies the *y* coordinate of the second vertex point. Datatype is NUMBER.

x3 specifies the *x* coordinate of the third vertex point. Datatype is NUMBER.

y3 specifies the *y* coordinate of the third vertex point. Datatype is NUMBER.

xn specifies the *x* coordinate of the *n*th vertex point. Datatype is NUMBER.

yn specifies the *y* coordinate of the *n*th vertex point. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of three vertex points must be specified.
- When defining a polygon window, the vertex points must be specified in order.
- Do not close the polygon window. This means that you do not specify both the first vertex point and the last vertex point as the same.

Example The following example determines how many lines from the partition TABLE1_P000000001 lie within the polygon window defined as follows:

dimension 1: (–76.15, 45.8) and (–76.15, 46.185)

dimension 2: (–76.05, 46.185) and (–76.05, 45.8)

```
SQL> SELECT count(*)
2 FROM lines_p000000001
3 WHERE HHIDLROWS('POLYGON',hhcolumn,-180,180,-90,90,
4 -76.15,45.8,-76.15,46.185,-76.05,46.185,
5 -76.05,45.8) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
--------
    1193
```

Related Topics

- HHIDLPART function

HHIDPART

Purpose

This function identifies the relationship between a partition of a partitioned table and an *n*–dimensional window of point data. It identifies whether the partition is inside, equal to, overlaps, outside, or encloses the query window.

Syntax

```
HHIDPART ({'RANGE'|'PROXIMITY'|'POLYGON'}, COMMON_HHCODE,
lower_boundary_1, upper_boundary_1, [ lower_boundary_n,
upper_boundary_n,...] window_definition)
```

Keywords and Parameters

RANGE	specifies an *n*–dimensional range window to query. Datatype is VARCHAR2.
PROXIMITY	specifies an *n*–dimensional proximity window to query. Datatype is VARCHAR2.
POLYGON	specifies a two–dimensional polygon window to query. Datatype is VARCHAR2.
COMMON_ HHCODE	is the COMMON_HHCODE column in either the ALL_MD_PARTITIONS or the USER_MD_PARTITIONS Spatial Data Option data dictionary data dictionary views.
lower_boundary_1	specifies the lower boundary of the first dimension range. Datatype is NUMBER.
upper_boundary_1	specifies the upper boundary of the first dimension range. Datatype is NUMBER.
lower_boundary_n	specifies the lower boundary of the *n*th dimension range. Datatype is NUMBER.
upper_boundary_n	specifies the upper boundary of the *n*th dimension range. Datatype is NUMBER.
window_definition	specifies the window definition. Varies depending on whether it is a range, proximity, or polygon query window. For the syntax and description of each query type, see "Usage Notes" in this section.

Returns

One of the following descriptions of the relationship between the partition and the window is returned:

INSIDE	The partition is inside the specified window.
EQUAL	The partition is equal to the specified window.
OVERLAP	The partition overlaps the specified window.

OUTSIDE	The partition is outside the specified window.
ENCLOSES	The partition encloses the specified window.

Figure 6 – 4 illustrates the relationships between the preceding definitions and the specified window:

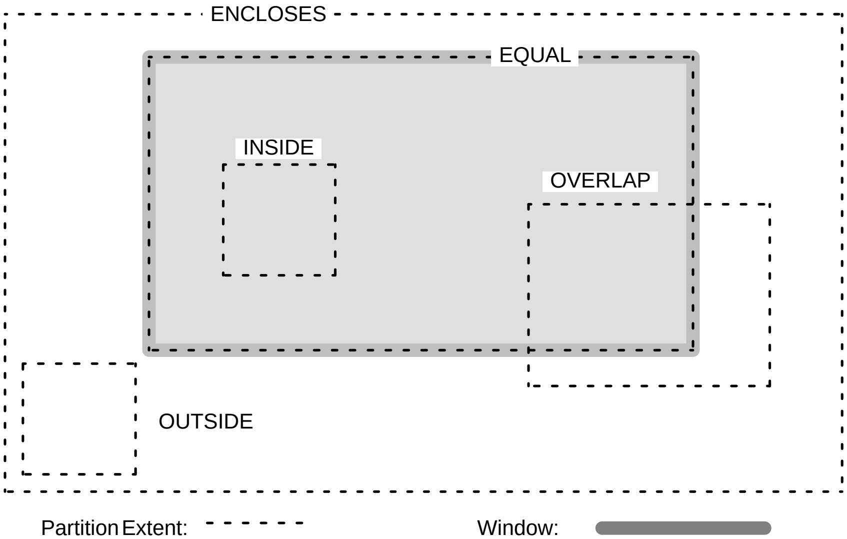

Figure 6 – 4 Partition/Window Relationships for a Two–Dimensional Case

Usage Notes

Consider the following points when using this function:

- This function is valid for point data only.
- To use a subset of the encoded dimensions, apply an HHCOMPOSE or HHCOLLAPSE function on the COMMON_HHCODE column. The following is an example of using HHCOMPOSE:

  ```
  HHCOMPOSE (common_hhcode, 1,3,5)
  ```

- Once the partitions are identified, use the HHIDROWS function to determine which points in the partitions fall within the specified window.

Range Window Syntax

This is defined by the lower and upper boundaries of the range window in 1 to 32 dimensions, as follows:

```
lower_window_boundary_1, upper_window_boundary_1
[, lower_window_boundary_n, upper_window_boundary_n...]
```

lower_window_ boundary_1	specifies the lower window boundary of the first dimension.

upper_window_ boundary_1 specifies the upper window boundary of the first dimension.

lower_window_ boundary_n specifies the lower window boundary of the *n*th dimension.

upper_window_ boundary_n specifies the upper window boundary of the *n*th dimension.

Example The following example determines which partitions intersect a range window defined as follows:

dimension 1: –76.15 to –76.05

dimension 2: 45.8 to 46.185

```
SQL> SELECT partition_table_name,
2 HHIDPART('RANGE',common_hhcode,–180,180,–90,90,
3 –76.15,–76.05,45.8,46.185) RELATIONSHIP
4 FROM user_md_partitions
5WHERE md_table_name = 'TABLE1';
```

The output appears as follows:

```
PARTITION_TABLE_NAME           RELATIONSHIP
------------------------------ ------------
TABLE1_P000000001              OVERLAP
TABLE1_P000000002              OUTSIDE
.
.
.
```

Proximity Window Syntax This is defined by a center point and a radius in from 2 to 32 dimensions, as follows:

center_1, center_2, (center_n, ...) radius

center_1 specifies the first dimensional value of the center point. Datatype is NUMBER.

center_2 specifies the second dimensional value of the center point. Datatype is NUMBER.

center_n specifies the *n*th dimensional value of the center point. Datatype is NUMBER.

radius specifies the radius in the coordinate system used. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of two center values must be specified.

- The radius must be specified in the same units as the coordinate system.
- The coordinate system must be Cartesian.

Example The following example determines which partitions intersect a proximity window, defined as follows:

center point: –76.1, 46.17

radius: 0.025

```
SQL> SELECT partition_table_name,
2 HHIDPART('PROXIMITY',common_hhcode,–180,180,–90,90,
3 –76.1,46.17,0.025) RELATIONSHIP
4 FROM user_md_partitions
5WHERE md_table_name = 'TABLE1';
```

The output appears as follows:

```
PARTITION_TABLE_NAME           RELATIONSHIP
------------------------------ ------------
TABLE1_P000000001              OUTSIDE
TABLE1_P000000002              OUTSIDE
.
.
.
```

Polygon Window Syntax This is defined by a set of 3 to 124 vertex points in two dimensions as follows:

x1, *y1*, *x2*, *y2*, *x3*, *y3*, [*xn*, *yn*...]

x1 specifies the *x* coordinate of the first vertex point. Datatype is NUMBER. Datatype is NUMBER.

y1 specifies the *y* coordinate of the first vertex point. Datatype is NUMBER. Datatype is NUMBER.

x2 specifies the *x* coordinate of the second vertex point. Datatype is NUMBER. Datatype is NUMBER.

y2 specifies the *y* coordinate of the second vertex point. Datatype is NUMBER. Datatype is NUMBER.

x3 specifies the *x* coordinate of the third vertex point. Datatype is NUMBER. Datatype is NUMBER.

y3 specifies the *y* coordinate of the third vertex point. Datatype is NUMBER. Datatype is NUMBER.

xn — specifies the *x* coordinate of the *n*th vertex point. Datatype is NUMBER. Datatype is NUMBER.

yn — specifies the *y* coordinate of the *n*th vertex point. Datatype is NUMBER. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of three vertex points must be specified.
- When defining a polygon window, the vertex points must be specified in order.
- Do not close the polygon window. This means that you do not specify both the first vertex point and the last vertex point as the same.
- Polygon windows can only be two–dimensional; if more than two dimensions are specified, an error message is returned.

Example The following example determines which partitions intersect a polygon window defined by 4 points as follows:

point 1: –76.15, 45.8

point 2: –76.15, 46.185

point 3: 76.05, 46.185

point 4: –76.05, 45.8

```
SQL> SELECT partition_table_name,
2 HHIDPART('POLYGON',common_hhcode,-180,180,-90,90,
3 -76.15,45.8,-76.15,46.185,-76.05,46.185,
4 -76.05,45.8) RELATIONSHIP
5 FROM user_md_partitions
6 WHERE md_table_name = 'TABLE1';
```

The output appears as follows:

```
PARTITION_TABLE_NAME           RELATIONSHIP
------------------------------ ------------
TABLE1_P000000001              OVERLAP
TABLE1_P000000002              OUTSIDE
.
.
.
```

Related Topics

- HHIDROWS function

HHIDROWS

Purpose

This function identifies the location of an *n*–dimensional point in relation to an *n*–dimensional window. It identifies whether the point is inside, outside, or on the boundary of the query window.

Syntax

```
HHIDROWS ({'RANGE'|'PROXIMITY'|'POLYGON'}, partition_key,
(lower_boundary_1, upper_boundary_1, [lower_boundary_n,
upper_boundary_n,...] window_definition)
```

Keywords and Parameters

RANGE	specifies an *n*–dimensional range window to query. Datatype is VARCHAR2.
PROXIMITY	specifies an *n*–dimensional proximity window to query. Datatype is VARCHAR2.
POLYGON	specifies a two–dimensional polygon window to query. Datatype is VARCHAR2.
partition_key	specifies the partition key column of a partitioned table; is an HHCODE column in a non–partitioned table.
lower_boundary_1	specifies the lower boundary of the first dimension range. Datatype is NUMBER.
upper_boundary_1	specifies the upper boundary of the first dimension range. Datatype is NUMBER.
lower_boundary_n	specifies the lower boundary for the *n*th dimension range. Datatype is NUMBER.
upper_boundary_n	specifies the upper boundary for the *n*th dimension range. Datatype is NUMBER.
window_definition	specifies the window definition. Varies depending on whether it is a range, proximity, or polygon query window. For the syntax and description of each query type, see ”Usage Notes” in this section.

Returns

One of the following descriptions of the relationship between the point and the window is returned:

INSIDE	The point is inside the specified window.
OUTSIDE	The point is outside the specified window.

BOUNDARY	The point is on the boundary of the specified window.

Usage Notes

Consider the following points when using this function:

- This function can be used with both partitioned and non–partitioned tables.
- This function is valid for point data only.

Range Window Syntax

This is defined by the lower and upper boundaries of the range window in 1 to 32 dimensions, as follows:

```
lower_window_boundary_1, upper_window_boundary_1
[, lower_window_boundary_n, upper_window_boundary_n...]
```

lower_window_ boundary_1	specifies the lower window boundary of the first dimension. Datatype is NUMBER.
upper_window_ boundary_1	specifies the upper window boundary of the first dimension. Datatype is NUMBER.
lower_window_ boundary_n	specifies the lower window boundary of the *n*th dimension. Datatype is NUMBER.
upper_window_ boundary_n	specifies the upper window boundary of the *n*th dimension. Datatype is NUMBER.

Example

The following example determines how many points from the partition TABLE1_P000000001 lie within a range window, defined as follows:

dimension 1:	–76.15 to –76.05
dimension 2:	45.8 to 46.185

```
SQL> SELECT count(*)
2 FROM table1_p000000001
3 WHERE HHIDROWS('RANGE',hhcolumn,–180,180,–90,90,
4 –76.15,–76.05,45.8,46.185) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
-------
645
```

Proximity Window Syntax

This is defined by a center point and a radius in from 2 to 32 dimensions, as follows:

```
center_1, center_2, (center_n, ...) radius
```

center_1	specifies the first dimensional value of the center point. Datatype is NUMBER.

center_2 specifies the second dimensional value of the center point. Datatype is NUMBER.

center_n specifies the *n*th dimensional value of the center point. Datatype is NUMBER.

radius specifies the radius in the coordinate system used. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of two center values must be specified.
- The radius must be specified in the same units as the coordinate system.
- The coordinate system must be Cartesian.

Example The following example determines how many points from the partition TABLE1_P00000000D lie within the proximity window defined as follows:

center point: –76.1, 46.17

radius: 0.025

```
SQL> SELECT count(*)
2 FROM table1_p00000000d
3 WHERE HHIDROWS('PROXIMITY',hhcolumn,–180,180,–90,90,
4 –76.1,46.17,0.025) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
--------
10
```

Polygon Window Syntax

This is defined by a set of 3 to 124 vertex points in two dimensions, as follows:

x1, *y1*, *x2*, *y2*, *x3*, *y3* [, *xn*, *yn*...]

x1 specifies the *x* coordinate of the first vertex point. Datatype is NUMBER.

y1 specifies the *y* coordinate of the first vertex point. Datatype is NUMBER.

x2 specifies the *x* coordinate of the second vertex point. Datatype is NUMBER.

y2 specifies the *y* coordinate of the second vertex point. Datatype is NUMBER.

x3 specifies the *x* coordinate of the third vertex point. Datatype is NUMBER.

y3 specifies the *y* coordinate of the third vertex point. Datatype is NUMBER.

xn specifies the *x* coordinate of the *n*th vertex point. Datatype is NUMBER.

yn specifies the *y* coordinate of the *n*th vertex point. Datatype is NUMBER.

Guidelines are as follows:

- A minimum of 3 vertex points must be specified.
- When defining a polygon window, the vertex points must be specified in order.
- Do not close the polygon window. This means that you do not specify both the first vertex point and the last vertex point as the same.
- Polygon windows can only be two–dimensional; if more than two dimensions are specified, an error message is returned.

Example The following example determines how many points from the partition TABLE1_P000000001 lie within the polygon window defined by the following 4 points:

point 1:	–76.15, 45.8
point 2:	–76.15, 46.185
point 3:	–76.05, 46.185
point 4:	–76.05, 45.8

```
SQL> SELECT count(*)
2 FROM table1_p000000001
3 WHERE HHIDROWS('POLYGON',hhcolumn,–180,180,–90,90,
4 –76.15,45.8,–76.15,46.185,–76.05,46.185,
5–76.05,45.8) != 'OUTSIDE';
```

The output appears as follows:

```
COUNT(*)
--------
645
```

Related Topics

- HHIDPART function

HHJLDATE

Purpose

This function takes a calendar date and converts it to a decimal Julian date from January 1, 4712 BC.

Syntax

```
HHJLDATE ('calendar_date','date_format_string')
```

Keywords and Parameters

calendar_date	specifies the calendar date.
date_format_string	specifies the format mask to use. Datatype is VARCHAR2

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- You must use a valid date format string. For a list of valid formats as well as guidelines for using them, see Table 6 – 2 in the section "Date Format Elements" in this chapter.
- To retrieve the original calendar date use the HHCLDATE function.

Example

The following example converts a calendar date into a Julian date:

```
SQL> column djd format 9999999.999999999
SQL> SELECT HHJLDATE ('1963/10/13-04:34:16:890',
2 'YYYY/MM/DD-HH:MI:SS:MLS') djd
3 FROM dual;
```

The output appears as follows:

```
DJD
-----------------
2438316.190473263
```

Related Topics

- HHCLDATE function

HHLENGTH

Purpose

This function returns the maximum number of levels of resolution encoded in an HHCODE, or for specific dimensions in an HHCODE.

Syntax

HHLENGTH (*hhcode_expression* [, *dimension_number*])

Keywords and Parameters

hhcode_expression	is an expression that evaluates to an HHCODE. Datatype is RAW.
dimension_number	specifies the dimension number. Optional; default is all dimensions. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

Consider the following points when using this function:

- Because an HHCODE can contain dimensions encoded to different levels of resolution, this function provides a mechanism for determining the maximum level of resolution encoded within the HHCODE, and the number of levels of resolution encoded for a particular dimension.
- This information can also be obtained from the Spatial Data Option data dictionary. The following is an example of how to determine the length of a particular dimension from the Spatial Data Option data dictionary:

```
SQL> SELECT recursion_level
2 FROM user_md_dimensions
3 WHERE dimension_name = 'DIM1'
4 AND column_name = 'HHCOLUMN'
5 AND md_table_name = 'TABLE1';
```

The following is an example of how to determine the maximum length of the HHCODE from the Spatial Data Option data dictionary for all dimensions:

```
SQL> SELECT dimension_name(recursion_level)
2 FROM user_md_dimensions
3 WHERE column_name = 'HHCOLUMN'
4 AND md_table_name = 'TABLE1';
```

The output appears as follows:

```
DIMENSION_NAME     RECURSION_LEVEL
----------------------------------
LAT                             32
LONG                            31
```

Example

The following is an example of the HHLENGTH function:

```
SQL> SELECT HHLENGTH(hhcolumn, 1)
2 FROM table1_p000000001
3 WHERE rownum=1;
```

The output appears as follows:

```
HHLENGTH(HHCOLUMN,1)
--------------------
32
```

Related Topics

- HHLEVELS function

HHLEVELS

Purpose

This function takes a scale and dimension range, and returns the number of levels of resolution needed to represent this scale.

Syntax

```
HHLEVELS (lower_boundary, upper_boundary, scale)
```

Keywords and Parameters

lower_boundary specifies the lower boundary of the dimension range. Datatype is NUMBER.

upper_boundary specifies the upper boundary of the dimension range. Datatype is NUMBER.

scale specifies the scale. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

This function performs the inverse of HHPRECISION.

Example I The example returns the level of resolution of the first dimension in a two–dimensional HHCODE:

```
SQL> SELECT HHLEVELS(-180,180,7)
2 FROM dual;
```

The output appears as follows:

```
HHLEVELS(-180,180,7)
--------------------
                  32
```

Example II The example returns the level of resolution of the second dimension in a two–dimensional HHCODE:

```
SQL> SELECT HHLEVELS(-90,90,7)
2 FROM dual;
```

The output appears as follows:

```
HHLEVELS(-90,90,7)
------------------
                31
```

Related Topics

- HHPRECISION function

HHMATCH

Purpose

This function compares two HHCODEs and returns the number of matching levels of resolution, starting from the first level. The HHCODEs must have the same number and descriptions of dimensions.

Syntax

```
HHMATCH (hhcode_expression, hhcode_expression)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

Returns

This function returns an integer.

Usage Notes

Figure 6 – 5 is a gridded space representing a two–dimensional area. The gridlines show how the space is decomposed into quadrants, which decrease in size by each level of resolution. A quadrant number for a specifically defined cell is determined by prefixing numbers as the space decomposes; for example, the quadrant representing the lower left quarter of the diagram is given the number 2; the lower left quadrant on the next level of decomposition is also numbered 2, but its position and area are referenced as 22. This numbering convention for decomposition is used here to illustrate levels of resolution of the HHCODEs.

The HHMATCH function returns the matching level of resolution, as represented in Figure 6 – 5 by quadrants of two HHCODEs.

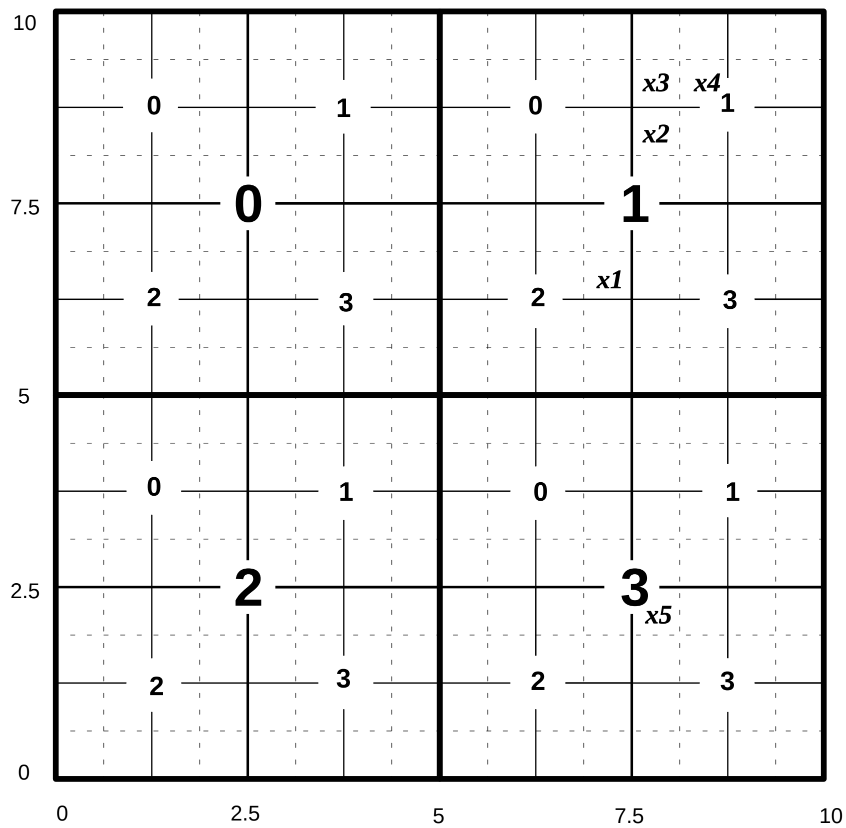

Figure 6 – 5 The HHMATCH Function

Five points are marked on Figure 6 – 5 to represent five HHCODEs. Performing a series of HHMATCH queries on pairs of these HHCODEs determines the quadrant containing both.

The following relationships are demonstrated by Figure 6 – 5:

- *x1* and *x5* do not share a quadrant, return value is 0
- *x1* and *x2* share quadrant 1, return value is 1
- *x2* and *x*3 share quadrant 11, return value is 2
- *x3* and *x4* share quadrant 110, return value is 3

To determine the quadrant containing the two HHCODEs, execute an HHSUBSTR function on either one of the HHMATCH arguments, to the number of levels of resolution returned by HHMATCH.

For example, the following command returns a value of 3:

```
HHMATCH(hhcode_expression_1,hhcode_expression_2)
```

In this case, either of the following commands returns the quadrant containing both HHCODEs, which is the common code:

```
HHSUBSTR(hhcode_expression_1,3)
```

or

```
HHSUBSTR(hhcode_expression_2,3)
```

Example The following is an example of the HHMATCH function:

```
SQL> SELECT HHMATCH(min(hhcolumn), max(hhcolumn))
2 FROM table1_p000000001;
```

The output appears as follows:

```
HHMATCH(MIN(HHCOLUMN),MAX(HHCOLUMN))
------------------------------------
                                   9
```

Related Topics

- HHCOMMONCODE function
- HHSUBSTR function

HHNDIM

Purpose

This function returns the number of dimensions encoded within an HHCODE.

Syntax

```
HHNDIM (hhcode_expression)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

Returns

This function returns an integer.

Usage Notes

This information can also be obtained from the Spatial Data Option data dictionary by querying either the USER_MD_DIMENSIONS or ALL_MD_DIMENSIONS views.

Example

The following is an example of the HHNDIM function:

```
SQL> SELECT HHNDIM(hhcolumn) number_of_dimensions
2 FROM table1_p000000001
3 WHERE rownum=1;
```

The output appears as follows:

```
NUMBER_OF_DIMENSIONS
--------------------
                   2
```

Related Topics

None

HHORDER

Purpose

This function must be used during ORDER BY operations, so that the HHCODEs are ordered correctly.

Syntax

```
HHORDER (hhcode_expression)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- Use HHORDER when you want to perform an ORDER BY on an HHCODE column.
- If all the HHCODEs in the table you are ordering are the same number of bytes, and the dimension information is the same, then you do not need to apply this function.

Example

The following is an example of the HHORDER function:

```
SQL> SELECT COMMON_HHCODE
2 FROM user_md_partitions
3 ORDER BY HHORDER (common_hhcode);
```

The output appears as follows:

```
COMMON_HHCODE
----------------
7083800909020000
7083C00B0B020000
7083C40B0B020000
7083C80B0B020000
7083CC0B0B020000
7083D00A0A020000
7083E00B0B020000
7083E40B0B020000
```

Related Topics

- HHGROUP function

HHPRECISION

Purpose

This function returns the scale accuracy of an HHCODE maintained by the given level of resolution for a given dimension range.

Syntax

```
HHPRECISION (lower_boundary, upper_boundary, number_of_levels)
```

Keywords and Parameters

lower_boundary specifies the lower boundary of the dimension range. Datatype is NUMBER.

upper_boundary specifies the upper boundary of the dimension range. Datatype is NUMBER.

number_of_levels specifies the number of levels of resolution of the dimension. Datatype is NUMBER.

Returns

This function returns a number.

Usage Notes

The scale accuracy is the number of decimal digits after the decimal in the original data that were encoded into the HHCODE.

Example The following is an example of the HHPRECISION function:

```
SELECT HHPRECISION (–180,180,32)
2 FROM DUAL;
```

The output appears as follows:

```
HHPRECISION(–180,180,32)
------------------------
                       7
```

The previous example determines the scale maintained by 32 levels of resolution, with a lower boundary of –180 and an upper boundary of 180.

Related Topics

- HHLEVELS function

HHSUBSTR

Purpose

This function returns a portion, or substring, of an HHCODE based on the start and end resolution levels.

Syntax

```
HHSUBSTR (hhcode_expression, start_level, end_level)
```

Keywords and Parameters

hhcode_expression is an expression that evaluates to an HHCODE. Datatype is RAW.

start_level specifies the start resolution level. Datatype is NUMBER.

end_level specifies the end resolution level. Datatype is NUMBER.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- The start level is typically level one for most applications; however, you can start at any level.
- This function provides fuzzy search capability by returning all HHCODEs within a subscribed region, or window.

Example

The following is an example of the HHSUBSTR function:

```
SQL> SELECT HHSUBSTR(hhcolumn,1,12) FROM table1_p000000001
2 WHERE row_num < 10;
```

The output appears as follows:

```
MIN_SUBSTR              MAX_SUBSTR
----------------------------------------
7083900A0A020000        7083B00A0A020000
.
.
.
```

The examples for the HHGROUP function illustrate additional cases where HHSUBSTR is useful.

Related Topics

- HHGROUP function

CHAPTER

7

SD*SQL Packages

This chapter contains descriptions of the following Spatial Data Option functions and procedures that comprise the SD*SQL PL/SQL packages:

- MD_DDL package
- MD_DML package
- MD_PART package
- MD_WEX package
- MDVERIFY package

Summary of SD*SQL Packages

The SD*SQL PL/SQL packages fall into the following categories:

- Data Definition Language (DDL) procedures
- Data Manipulation Language (DML) functions and procedures
- Partition maintenance functions and procedures
- Window extract procedures for extracting Spatial Data Option data from partitioned tables

These packages can be invoked from any Oracle tool, such as SQL*Plus and Server Manager, as well as from within Pro*C as a PL/SQL block.

For detailed information about dynamic SQL, see the following Oracle7 server documentation:

- *Pro*C Supplement to the Oracle Precompilers Guide*
- *Programmer's Guide to the Oracle Precompilers*
- *Programmer's Guide to the Oracle Call Interfaces*

MD_DDL Package

The MD_DDL package is used to perform the following tasks:

- create, alter, and drop spatial tables
- allocate, activate, and deactivate tablespaces for spatial tables

The MD_DDL package contains all DDL procedures for spatial tables.

Note: The procedures in the MD_DDL package must always be preceded by the MD_DDL package name.

MD_DML Package

The MD_DML package is used with SQL INSERT, UPDATE, and DELETE statements and the MD_PART partition maintenance package to perform DML operations on spatial tables.

In addition, the MD_DML package allows you to perform the following operations:

- generate an HHCODE from the original dimensional data
- lock a partitioned table
- move a record from one partition to another

Note: The procedures in the MD_DML package must always be preceded by the MD_DML package name.

MD_PART Package

The MD_PART package is used to perform the following partition maintenance operations on spatial tables:

- identify partitions and determine whether they exist
- create, drop, and truncate partitions
- move and subdivide partitions

Note: The functions and procedures in the MD_PART package must always be preceded by the MD_PART package name.

MD_WEX Package

The MD_WEX package is used to extract data from partitioned tables, to allow subsets of data to be copied to another table or to create a view. These procedures perform the following operations:

- set a SQL filter
- set a storage clause when creating a table
- set a target tablespace when creating a table
- define a subset of dimensions to extract
- define the type of query window to use: range, proximity, or polygon
- extract the data in the defined window, as either a table or a view

Note: The functions and procedures in the MD_WEX package must always be preceded by the MD_WEX package name.

MDVERIFY Package

The MDVERIFY package is used to verify the consistency of the Spatial Data Option data dictionary. You do not need to run the MDVERIFY procedures during normal operations, but they can provide valuable diagnostic information to Oracle World Wide Support if the Spatial Data Option data dictionary becomes corrupted.

MD_DDL.ACTIVATE_TABLESPACE

Purpose

This procedure activates the use of a tablespace that has been deactivated for a partitioned table.

Prerequisites

You must have the privileges from the resource role. These privileges are granted to you by the DBA using the SQL ALTER USER command.

Syntax

```
MD_DDL.ACTIVATE_TABLESPACE ([schema.|username,] sd_tablename,
tablespace_name)
```

Keywords and Parameters

schema — specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.

username — specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.

sd_tablename — specifies the name of the partitioned table. Datatype is VARCHAR2.

tablespace_name — specifies the name of the tablespace to activate. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Tablespaces for partitioned tables cannot be deallocated. They can only be deactivated so that no new rows are added. The deactivated tablespace name remains on the tablespace list in the Spatial Data Option data dictionary, but is not used. For more information on tablespace management, see ”Tablespace Management” in Chapter 4, ”Administration.”
- This procedure does not create the tablespace. Create the tablespace using the SQL CREATE TABLESPACE command. For more information on creating tablespaces, see the *Oracle7 Server SQL Language Reference Manual.*

Example

The following example activates TABLESPACE5 for the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ACTIVATE_TABLESPACE (’table1’, ’tablespace5’);
```

Related Topics

- MD_DDL.ALLOCATE_TABLESPACE procedure
- MD_DDL.DEACTIVATE_TABLESPACE procedure

MD_DDL.ADD_HHCODE_COLUMN

Purpose

This procedure is used to define the HHCODE columns in the spatial table after it is registered in the Spatial Data Option data dictionary.

Prerequisites

You must own the table or have ALTER TABLE privilege on the table.

Syntax

MD_DDL.ADD_HHCODE_COLUMN ([*schema.*|*username,*] *sd_tablename*, *hhcode_column_name,* [*partition_key,*] [*not_null,*] *dimension_name*, *lower_boundary*, *upper_boundary*, *scale* [, *dimension_name*, *lower_boundary*, *upper_boundary*, *scale*...])

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the spatial table. Datatype is VARCHAR2.
hhcode_column_name	specifies the name of the HHCODE column. This column must already exist and be defined as a RAW (255).
partition_key	specifies whether the column is the partition key. Values are TRUE, FALSE. Datatype is Boolean. If partition key is TRUE, then the column is created with the constraint NOT NULL. Optional; default is FALSE.
not_null	specifies whether the column is NULL or NOT NULL. Values are TRUE, FALSE. TRUE equals NOT NULL. FALSE equals NULL. Optional; default is NOT NULL. Datatype is BOOLEAN. This parameter is ignored if the partition key is set to TRUE.
dimension_name	specifies the dimension name. Datatype is VARCHAR2.
lower_boundary	specifies the lower boundary of the dimension range. Datatype is NUMBER.

upper_boundary	specifies the upper boundary of the dimension range. Datatype is NUMBER.
scale	specifies the scale of the dimension. Datatype is NUMBER.

Usage Notes

Consider the following points when using this procedure:

- This procedure is used to define the HHCODE columns for partitioned and non–partitioned tables.
- The spatial table must already exist, and have been registered using the MD_DDL.REGISTER_MD_TABLE procedure before you can use this procedure.
- The spatial table must be empty. If it is not, an error occurs.
- For each dimension, specify the name, lower and upper boundaries of the dimension range, and scale accuracy of the dimension. You can define 1 to 32 dimensions.
- The HHCODE column name you specify must match the HHCODE column name created with the SQL CREATE TABLE command.
- The scale of the dimension is the number of digits to the right of the decimal point at which the encoded data is stored. You can define any scale up to the accuracy of the original data. If you define a scale that is less than the original scale, however, you cannot decode the data from the HHCODE to its original accuracy.
- If you are creating a partitioned table and you have more than one HHCODE column, you must indicate the HHCODE column that is used to organize the data by defining the partition key as TRUE.
- Only one partition key column is allowed per partitioned table. If you try to add another, an error occurs. An attempt to add a partition key column to a non–partitioned table is ignored.
- If the partition key is set to TRUE, then the column is created with the constraint of NOT NULL.
- If the MD_DDL.ADD_HHCODE_COLUMN procedure fails, a rollback occurs. If it is the first HHCODE column added since the table was registered, you must re–execute the MD_DDL.REGISTER_MD_TABLE procedure.
- The MD_DDL.ADD_HHCODE_COLUMN procedure causes an implicit COMMIT to occur.

Example I The following example adds the HHCODE column HHCOLUMN, with two dimensions, to the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ADD_HHCODE_COLUMN ('table1','hhcolumn', –
> TRUE, 'dim1', –180, 180, 7, 'dim2', –90, 90, 7);
```

TABLE1	is the name of the spatial table.
HHCOLUMN	is the HHCODE column name.
TRUE	indicates that it is the partition key column.
DIM1	is the dimension name for the first dimension.
–180	is the lower boundary for the first dimension.
180	is the upper boundary for the first dimension.
7	is the scale for the first dimension.
DIM2	is the dimension name for the second dimension.
–90	is the lower boundary for the second dimension.
90	is the upper boundary for the second dimension.
7	is the scale for the second dimension.

Related Topics

- MD_DDL.REGISTER_TABLE procedure

MD_DDL.ALLOCATE_TABLESPACE

Purpose

This procedure allocates a tablespace to a partitioned table and makes it active. It adds the tablespace name to the list of allocated tablespaces for the partitioned table.

Prerequisites

The tablespace must already be created with the SQL CREATE TABLESPACE command before it can be allocated.

You must have the privileges from the resource role. These privileges are granted to you by the DBA using the QUOTA parameter of the SQL ALTER USER command.

You must have an object privilege on the partitioned table.

Syntax

```
MD_DDL.ALLOCATE_TABLESPACE ([schema.|username,] sd_tablename,
tablespace_name)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the partitioned table. Datatype is VARCHAR2.
tablespace_name	specifies the name of the tablespace to allocate. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- This procedure adds the tablespace name to the list of allocated tablespaces for the partitioned table in the Spatial Data Option data dictionary.
- The tablespace must already exist before it can be allocated.
- Once allocated, the tablespace is active and is available for storage when new rows are inserted into the table. For information on how rows are distributed among tablespaces, see "Tablespace Management" in Chapter 4, "Administration."
- Once a tablespace is allocated, it is active unless deactivated using the MD_DDL.DEACTIVATE_TABLESPACE procedure.

- Tablespaces for partitioned tables cannot be deallocated. They can only be deactivated so that no new rows are added. The deactivated tablespace name remains on the tablespace list in the Spatial Data Option data dictionary, but is not used.
- Do not drop a tablespace that is still allocated and active. Before you drop any tablespace you must perform the following operations:
  1. Deactivate the tablespace for all spatial tables using the MD_DDL.DEACTIVATE_TABLESPACE procedure.
  2. Move all partitions in the tablespace to another tablespace using the MD_PART.MOVE_PARTITION procedure, or drop the table using the MD_DDL.DROP_MD_TABLE procedure.
  3. Drop the tablespace using the SQL DROP TABLESPACE command.

Example The following example allocates a new tablespace to the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ALLOCATE_TABLESPACE ('table1', 'tablespace4');
```

Related Topics

- MD_DDL.ACTIVATE_TABLESPACE procedure
- MD_DDL.DEACTIVATE_TABLESPACE procedure

MD_DDL.ALTER_MD_TABLE

Purpose

This procedure is used to alter a partitioned table. It uses the SQL ALTER command as a template to alter all the partitions in a partitioned table.

Prerequisites

You must own the table or you must have the ALTER TABLE privilege on the partitions in the spatial table.

Syntax

```
MD_DDL.ALTER_MD_TABLE ([schema.|username,] sd_tablename,
sql_template [, continue_on_error])
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table to alter. Datatype is VARCHAR2.
sql_template	is a SQL ALTER TABLE command template that is performed on all partitions of the table.
continue_on_error	specifies whether the command should continue and try to complete or stop when it encounters an error. Values are TRUE, FALSE. TRUE indicates continue; FALSE indicates to stop when an error is encountered. Optional; default is TRUE. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Use this procedure to alter a partitioned table in the same ways that the SQL ALTER TABLE command can alter standard Oracle7 tables. This includes the following:
  - to add one or more new attribute columns
  - to modify an existing attribute column definition
  - to modify data block space usage parameters PCTFREE and PCTUSED

– to modify transaction entry settings INITRANS and MAXTRANS

- Use the SQL ALTER TABLE command to alter non–partitioned tables.
- Errors are logged in the USER_MD_EXCEPTIONS table.
- The SQL template should not include the name of the spatial table. The substitution variable `%s` should be used instead. The variable is replaced by the name of the partition to be altered.
- If you have set this procedure to continue on error, check the USER_MD_EXCEPTIONS view to make sure no errors occurred during its execution.

Warning: Never alter partitioned tables using the SQL ALTER TABLE command.

Example I

The following example adds a column to the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE ('table1', 'ALTER TABLE %s –
> ADD (column2 NUMBER(10,2)');
```

Example II

The following example modifies an existing column in the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE ('table1', 'ALTER TABLE %s –
> MODIFY (column2 NUMBER(10,2)');
```

Example III

The following example modifies space usage parameters of the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE ('table1', 'ALTER TABLE %s –
> PCTFREE 10 PCTUSER 90');
```

Example IV

The following example modifies the transaction setting of the TABLE1 table:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE ('table1', 'ALTER TABLE %s –
> INITTRANS 5');
```

Related Topics

- MD_DDL.REGISTER_MD_TABLE procedure

MD_DDL.ALTER_MD_TABLE_CM

Purpose

This procedure alters the compute mode, which is used by SD*Loader when data is loaded into the partitioned table.

Prerequisites

You must own the table or you must have the ALTER TABLE privilege on the table.

Syntax

```
MD_DDL.ALTER_MD_TABLE_CM ([schema.|username,] sd_tablename,
{'EXACT'|'ESTIMATE'})
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table to alter. Datatype is VARCHAR2.
EXACT	specifies that the compute mode is EXACT. SD*Loader performs an exact count of the number of rows. Datatype is VARCHAR2.
ESTIMATE	specifies that the compute mode is ESTIMATE. SD*Loader estimates the number of rows. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Use this procedure to change the way the compute mode is determined when data is loaded.
- EXACT specifies that a SELECT count (*) is performed on the partition. For more information on the SQL COUNT function, see the *Oracle7 Server SQL Reference.*
- ESTIMATE specifies that the ANALYZE command estimates the number of rows in a partition during loading. For more information on the SQL ANALYZE command, see the *Oracle7 Server SQL Reference.*
- ESTIMATE is generally sufficiently accurate, and is usually faster than EXACT.

Example

The following example changes compute mode for the TABLE1 table to ESTIMATE:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE_CM ('table1', 'ESTIMATE');
```

Related Topics

- MD_DDL.REGISTER_MD_TABLE procedure

MD_DDL.ALTER_MD_TABLE_HWM

Purpose

This procedure alters the high water mark of a partitioned table.

Prerequisites

You must own the table or you must have ALTER privilege on the table.

Syntax

```
MD_DDL.ALTER_MD_TABLE_HWM ([schema.|username,] sd_tablename,
high_water_mark)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table to alter. Datatype is VARCHAR2.
high_water_mark	specifies the new high water mark. Datatype is NUMBER.

Usage Notes

Data in existing partitions is not re–distributed. The new high water mark is effective only on subsequently created partitions.

Example

The following example alters the high water mark for the TABLE1 table to 1500:

```
SQL> EXECUTE MD_DDL.ALTER_MD_TABLE_HWM('table1',1500);
```

Related Topics

- MD_DDL.REGISTER_MD_TABLE procedure

MD_DDL.DEACTIVATE_TABLESPACE

Purpose

This procedure deactivates a tablespace for a partitioned table. The deactivated tablespace is no longer used when new rows are inserted into the partitioned table.

Prerequisites

You must have an object privilege on the partitioned table.

Syntax

```
MD_DDL.DEACTIVATE_TABLESPACE ([schema.|username,] sd_tablename,
tablespace_name)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table. Datatype is VARCHAR2.
tablespace_name	specifies the name of the tablespace to deactivate. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- The deactivated tablespace name remains on the tablespace list in the Spatial Data Option data dictionary, but is not used.
- The tablespace can be re–activated using the MD_DDL.ACTIVATE_TABLESPACE procedure.
- Tablespaces for partitioned tables cannot be deallocated. They can only be deactivated so that no new rows are added.

Example

The following example deactivates TABLESPACE5 for the TABLE1 table:

```
SQL> EXECUTE MD_DDL.DEACTIVATE_TABLESPACE ('table1', –
> 'tablespace5');
```

Related Topics

- MD_DDL.ACTIVATE_TABLESPACE procedure
- MD_DDL.ALLOCATE_TABLESPACE procedure

MD_DDL.DROP_MD_TABLE

Purpose

This procedure drops spatial tables, including partitioned tables and all associated partitions.

Prerequisites

You must own the table or you must have the DROP ANY TABLE system privilege.

Syntax

```
MD_DDL.DROP_MD_TABLE ([schema.|username,] sd_tablename)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table to drop. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Do not use the SQL DROP TABLE command to drop a spatial table. If you do so, the information in the Spatial Data Option data dictionary is not updated and is inconsistent with the state of the database.
- If the MD_DDL.DROP_MD_TABLE procedure fails, an application error is raised. Query the USER_MD_TABLES view. If the table still exists, re–execute the procedure.
- When a partitioned table is dropped, all partitions are dropped, followed by the root table.

Warning: Never drop the root table using the SQL DROP TABLE command.

Example

The following example drops the TABLE1 table, all associated partitions, and the root table:

```
SQL> EXECUTE MD_DDL.DROP_MD_TABLE ('table1');
```

Related Topics

- MD_PART.DROP_PARTITION procedure
- MD_PART.TRUNCATE_PARTITION procedure

MD_DDL.REGISTER_MD_TABLE

Purpose

This procedure registers the table in the Spatial Data Option data dictionary, defines whether or not the table is partitioned and optionally sets a compute mode.

Prerequisites

You must own the table or have ALTER privilege on the table.

Syntax

```
MD_DDL.REGISTER_MD_TABLE ([schema.|username,] sd_tablename
[,high_water_mark [, NULL, ['EXACT'|'ESTIMATE']]])
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table to register in the Spatial Data Option data dictionary. This table must already exist as an empty Oracle7 table. Datatype is VARCHAR2.
high_water_mark	specifies the high water mark. If no high water mark is defined, then the table is considered to be non–partitioned. Datatype is NUMBER.
NULL	specifies a null value in the parameter list. This is used as a place holder, and is only required if a compute mode is also specified. Datatype is VARCHAR2.
EXACT	specifies the compute mode so that SD*Loader counts the number of rows during a load exactly. Datatype is VARCHAR2.
ESTIMATE	specifies the compute mode so that SD*Loader estimates the number of rows during a load. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- To register the table as partitioned, specify the high water mark. If it is not set, then the table is registered as non–partitioned.
- If the optional compute mode parameter is set, you must pass a null value between the high water mark parameter and the compute mode parameter, as follows:

```
high_water_mark_value, NULL, compute_mode_value
```

- EXACT specifies that a SELECT count(*) is performed on the partition when data is loaded. For information on the SQL COUNT function, see the *Oracle7 Server SQL Reference*.
- ESTIMATE specifies that the ANALYZE command estimates the number of rows in a partition when data is loaded. For more information on the SQL ANALYZE command, see the Oracle7 Server *SQL Reference*.
- ESTIMATE is usually fairly accurate, and faster than EXACT.
- The MD_DDL.REGISTER_MD_TABLE procedure does not cause an implicit COMMIT; therefore the table is not fully registered until the first HHCODE column definition is added with the MD_DDL.ADD_HHCODE_COLUMN procedure, which does cause a COMMIT to occur.
- **Note:** When you register a spatial table, the tablespace that is registered with it is the tablespace in which the root table was created, not the user's default tablespace.

Example The following example registers the spatial table TABLE1:

```
SQL> EXECUTE MD_DDL.REGISTER_MD_TABLE ('table1',10000, NULL, –
> 'EXACT');
```

table1	is the name of the table to register.
10000	is the high water mark. This indicates that the table is partitionable.
NULL	specifies a null placeholder.
EXACT	is the compute mode.

Related Topics

- MD_DDL.ALTER_MD_TABLE_CM procedure
- MD_DDL.ALTER_MD_TABLE_HWM procedure

MD_DML.GENHHCODE

Purpose

This function generates an HHCODE from original dimensional data.

Prerequisites

You must own the table or have an object privilege on the table.

Syntax

```
MD_DML.GENHHCODE ([schema.|username,] sd_tablename,
[hhcode_column_name,] dimension_1 [, dimension_n...]
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the function. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the function. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table. Datatype is VARCHAR2.
hhcode_column_name	specifies the name of the HHCODE column. Optional; if not specified, the function assumes that the HHCODE is being generated for the partition key column of the specified partitioned table, and uses its dimension definition. Datatype is VARCHAR2.
dimension_1	specifies the value for the first dimension. Datatype is NUMBER.
dimension_n	specifies the value for the *n*th dimension. Datatype is NUMBER.

Returns

This function returns an HHCODE.

Usage Notes

Consider the following points when using this function:

- This function generates an HHCODE value without having to first look up the dimension information in the Spatial Data Option data dictionary and then call the HHENCODE function. MD_DML.GENHHCODE performs all required logic and validation to generate an HHCODE for a spatial table.
- This function returns a NULL if an error is encountered generating the HHCODE.

- The generated HHCODE can now be used in a SQL INSERT command or a SQL UPDATE command to modify a partitioned table.

Example I The following example of a PL/SQL block generates an HHCODE using the dimension definition of the partition key of the TABLE1 table:

```
declare
   hhc raw (255);
begin
   hhc := MD_DML.GENHHCODE
     ('herman','table1','hhcolumn',–76.1,46.1)
end;
```

Example II The preceding example of a PL/SQL block is equivalent to the following:

```
declare
   hhc raw(255);
begin
   hhc := MD.HHENCODE(12.3120849,–180,180,7,1.2333591,–90,90,7);
end;
```

The difference between the two methods is as follows:

- In Example I, you provide the table name and column names to use to generate the HHCODE, and then provide the values. All required dimension definitions are looked up in the Spatial Data Option data dictionary.
- In Example II, you must provide the dimension definitions.

Related Topics

- HHENCODE function

MD_DML.LOCK_MD_TABLE

Purpose

This procedure locks the spatial table and all the partitions of a partitioned table.

Prerequisites

You must own the table or have an object privilege on the table.

Syntax

```
MD_DML.LOCK_MD_TABLE ([schema.|username,] sd_tablename [,
'EXCLUSIVE'|'ROW EXCLUSIVE'|'ROW SHARE'|'SHARE'|'SHARE
UPDATE'|'SHARE ROW EXCLUSIVE', [lock_wait_mode]])
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the partitioned table to lock. Datatype is VARCHAR2.
EXCLUSIVE	is an optional lock mode. Specifies an EXCLUSIVE lock on the partitioned table.
ROW EXCLUSIVE	is an optional lock mode. Specifies a ROW EXCLUSIVE lock on the partitioned table.
ROW SHARE	is the default lock mode. Specifies a ROW SHARE lock on the partitioned table. Datatype is VARCHAR2.
SHARE	is an optional lock mode. Specifies a SHARE lock on the partitioned table. Datatype is VARCHAR2.
SHARE UPDATE	is an optional lock mode. Specifies a SHARE UPDATE lock on the partitioned table. Datatype is VARCHAR2.
SHARE ROW EXCLUSIVE	is an optional lock mode. Specifies a SHARE ROW EXCLUSIVE lock on the partitioned table. Datatype is VARCHAR2.
lock_wait_mode	specifies the lock wait mode. Optional; values are TRUE, FALSE. Default is TRUE. Datatype is BOOLEAN.

Usage Notes

Consider the following points when using this procedure:

- To lock a single partition or a non–partitioned table, use the SQL LOCK TABLE command.
- To release the lock, you must perform a COMMIT or ROLLBACK.
- If one of the partitions or the spatial table fails to lock, all other locks are released.
- If you do not specify the optional lock wait mode, you must provide a NULL value in its place.
- The MD_DML.LOCK_MD_TABLE procedure attempts to lock with a row–exclusive lock on all the Spatial Data Option data dictionary records for the partitions belonging to the table. This procedure then locks the spatial table and each partition according to the specified lock wait mode.
- If the lock wait mode is equal to TRUE then the procedure waits until the locked objects become available. If the lock wait mode is equal to FALSE then the procedure does not wait and the procedure fails when objects are not available.
- There are two predefined Boolean global constants in the MD_DML package that can also be used. MD_DML.LOCK_WAIT can be used in place of TRUE. MD_DML.LOCK_NOWAIT can be used in place of FALSE.
- For detailed information on lock modes and lock wait modes, see the SQL LOCK TABLE command in the *Oracle7 Server SQL Reference.*
- For general information on locking tables, see the *Oracle7 Server Concepts.*

Example

The following example locks the spatial table TABLE1:

```
SQL> EXECUTE MD_DML.LOCK_MD_TABLE('table1', 'EXCLUSIVE');
```

Related Topics

None

MD_DML.MOVE_RECORD

Purpose

This procedure allows you to update the partition key column of a record in a partitioned table, and move the record to another partition if required.

Prerequisites

You must have INSERT privilege on the spatial table.

Syntax

```
MD_DML.MOVE_RECORD ([schema.|username,] sd_tablename, old_hhcode,
new_hhcode [, attribute_where_clause])
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the table. Datatype is VARCHAR2.
old_hhcode	specifies the old HHCODE value. Datatype is RAW.
new_hhcode	specifies the new HHCODE value. Datatype is RAW.
attribute_ where_clause	specifies a WHERE clause using attribute data in the partitioned table. Optional; default is none. Datatype is VARCHAR2.

Usage Notes

Updating a record whose partition key column equals the old HHCODE and setting the partition key column to equal the new HHCODE may cause the record to move to a different partition. This procedure updates the partition key column. The record is moved to the new partition after the update, if required.

Example

The following example moves a record from one partition to another in the TABLE1 table:

```
SQL> EXECUTE MD_DML.MOVE_RECORD('table1', –
> MD.HHENCODE (–76.589978,–180,180,7,46.14301,–90,90,7),) –
> MD.HHENCODE (–75.57,–180,180,7,46.14,–90,90,7));
```

Related Topics

None

MD_PART.CLEAR_EXCEPTION_TABLES

Purpose

This procedure drops all views and tables listed in the Spatial Data Option data dictionary views ALL_MD_EXCEPTIONS or USER_MD_EXCEPTIONS for the user executing the procedure.

Prerequisites

You must own the objects being dropped.

Syntax

```
MD_PART.CLEAR_EXCEPTION_TABLES
```

Keywords and Parameters

None

Usage Notes

Consider the following points when using this procedure:

- After any operation using the Spatial Data Option, the exception table should be check to make sure that no exceptions are listed. If there are, use this procedure to clear these exceptions, which means that all tables and views will be dropped and their records cleared from the Spatial Data Option data dictionary views.
- This procedure only clears exceptions raised by the user executing the procedure.
- To confirm that exceptions have been cleared, you can query the USER_MD_EXCEPTIONS view. Once this procedure is executed, the query should result in no rows being selected.
- For more information on clearing exceptions, see "Exception Conditions" in Chapter 4, "Administration."

Example

The following example displays and clears the exceptions raised by the current user:

```
SQL> SELECT * from USER_MD_EXCEPTIONS;
```

The following output appears:

```
NAME              OPERATION   CCHH
----------------- ---------- -----
TABLE1_P00000000A DROP TABLE
```

The previous output shows that a partition of the table was not successfully dropped. Now clear the exceptions table, as follows:

```
SQL> EXECUTE MD_PART.CLEAR_EXCEPTION_TABLES;
```

Related Topics

None

MD_PART.CREATE_INFERRED_PARTITION

Purpose

This function creates an empty partition with HHCOMMONCODE derived from the value specified in the HHCODE parameter, and returns the name of the partition created. This function is used when the MDPART.GET_PARTITION_NAME function returns a status of MD_PART.NOTEXIST.

Prerequisites

You must own the table or you must have CREATE TABLE privilege in the schema.

Syntax

```
MD_PART.CREATE_INFERRED_PARTITION ([schema.|username,]
sd_tablename, hhcode_expression)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the function. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the function. Datatype is VARCHAR2.
sd_tablename	specifies the partitioned table name. Datatype is VARCHAR2.
hhcode_expression	specifies the HHCODE of the record to be inserted into the new partition. Datatype is RAW.

Returns

This function returns the name of the newly created partition.

Usage Notes

Consider the following points when using this function:

- The HHCODE value is the same value used in the MD_PART.GET_PARTITION_NAME function.
- This function causes an implicit COMMIT to occur.

Example

The following example of a PL/SQL block creates a new partition for the TABLE1 table:

```
declare
    cpart   varchar2(30);
begin
    cpart := MD.PART.CREATE_INFERRED_PARTITION ('herman',
      'table1', MD.HHENCODE (–76.2,–180,180,7,47,–90,90,7));
end;
/
```

Related Topics

None

MD_PART.DROP_PARTITION

Purpose

This procedure drops a partition of a partitioned table.

Prerequisites

You must own the table or you must have DROP TABLE privilege in the schema.

Syntax

```
MD_PART.DROP_PARTITION ([schema.|username,] partition_name)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
partition_name	specifies the name of the partition to be dropped. Datatype is VARCHAR2.

Usage Notes

The MD_PART.DROP_PARTITION procedure causes an implicit COMMIT to occur.

Warning: Never drop a partition with the SQL DROP command.

Example

The following example drops the partition TABLE1_P000000005:

```
SQL> EXECUTE MD_PART.DROP_PARTITION ('herman', –
> 'table1_p000000005');
```

Related Topics

- MD_DML.DROP_MD_TABLE procedure

MD_PART.GET_PARTITION_NAME

Purpose

This function identifies the name of a partition in a partitioned table, and reports whether the partition exists. This information is then used to determine where to insert a record in the partitioned table.

Prerequisites

You must have an object privilege on the partitioned table.

Syntax

MD_PART.GET_PARTITION_NAME ([*schema.*|*username,*] *sd_tablename*, *hhcode_expression*, *lock_wait_mode, partition_name*)

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the function. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the function. Datatype is VARCHAR2.
sd_tablename	specifies the name of the partitioned table. Datatype is VARCHAR2.
hhcode_expression	specifies the HHCODE of the record to be inserted. Datatype is RAW.
lock_wait_mode	specifies the lock wait mode. Optional; values are TRUE, FALSE. Default is TRUE. Datatype is BOOLEAN.
partition_name	specifies the partition name returned from the function. Datatype is VARCHAR2.

Returns

This function returns the partition status as follows:

NOTEXIST	A partition does not exist, because no data for the region it represents has been loaded or inserted yet. The partition must therefore be created before you use the SQL INSERT command to insert the record into the partition. The MD_PART.CREATE_INFERRED_PARTITION procedure is used to create the partition.
ONLINE	A partition is ready for data to be inserted using the SQL INSERT command.

Usage Notes

Consider the following points when using this function:

- This function is used to determine the correct partition in which to insert the row, to maintain the spatial organization of the data.

- Once the status of the partition is determined, you can decide whether to perform the data manipulation statement, or perform a partition maintenance procedure, such as creating the partition.
- This function attempts to lock the record in the Spatial Data Option data dictionary for the specified partition. If successful, it also locks the partition itself in ROW SHARE mode.

  If the lock_wait_mode is set to TRUE, the function waits to exclusively lock the Spatial Data Option data dictionary record for the specified partition, if this record is already locked by another user. If FALSE, this specifies that the function does not wait, and fails if it cannot acquire the necessary locks. If the wait mode is not specified, no locking is attempted.

  There are two pre–defined Boolean global constants in the MD_DML package that can also be used. MD_DML.LOCK_WAIT can be used in place of TRUE. MD_DML.LOCK_NOWAIT can be used in place of FALSE.

  If you do not want locking to occur, or you want to perform locking manually, then do not specify a wait mode in the procedure call.

Example The following example of a PL/SQL block gets the partition name of an HHCODE value:

```
declare
    ps      integer;
    pn      varchar2(30);
begin
    ps := MD_PART.GET_PARTITION_NAME ('table1',
      MD.HHENCODE (–76.2,–180,180,7,47,–90,90,7),pn);
end;
/
```

Related Topics

- MD_PART.CREATE_INFERRED_PARTITION procedure
- MD_PART.SUBDIVIDE_PARTITION procedure

MD_PART.MOVE_PARTITION

Purpose

This procedure moves a partition from the tablespace in which it currently resides into another tablespace.

Prerequisites

You must own the table or have CREATE TABLE privilege in the schema. Both you and the owner of the table must have the privileges from the resource role granted for the target tablespace.

Syntax

MD_PART.MOVE_PARTITION ([*schema.*|*username,*] *partition_name*, *tablespace_name*)

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
partition_name	specifies the name of the partition to be moved. Datatype is VARCHAR2.
tablespace_name	specifies the name of the tablespace where the partition is to be moved. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- The MD_PART.MOVE_PARTITION procedure allows partitions to be moved, to balance I/O or drop tablespaces.
- This procedure moves a partition from one tablespace to another, renames the partition to the next available partition name in the sequence, and drops the old partition name.
- The tablespace where the partition is moved must be allocated to the partitioned table to which the partition belongs.
- This procedure causes an implicit COMMIT to occur.

Example

The following example moves the partition TABLE1_P000000005 to the TBS2 tablespace:

```
SQL> EXECUTE MD_PART.MOVE_PARTITION (’herman’, –
> ’table1_p000000005’, ’tbs2’);
```

Related Topics

None

MD_PART.SUBDIVIDE_PARTITION

Purpose

This procedure subdivides a partition, and is used primarily when a manual data load using the SQL INSERT command has caused the partition to exceed its high water mark.

Prerequisites

You must have CREATE TABLE privilege in the schema of the partitioned table. You must have quota assigned on all the tablespaces allocated to the partitioned table.

Syntax

```
MD_PART.SUBDIVIDE_PARTITION ([schema.|username,] partition_name)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
partition_name	specifies the name of the partition to subdivide. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Subdivide a partition if a row count on the table indicates that it is over or near the high water mark.
- When a partition subdivides, the data is moved from the current partition into the newly created partitions. Once the process is complete, the original partition is dropped and the Spatial Data Option data dictionary is updated accordingly.
- This procedure causes an implicit COMMIT to occur.
- **Note:** When data is loaded using SD*Loader, table partitioning occurs automatically.

Example

The following example subdivides the TABLE1_P000000008 partition of the TABLE1 table:

```
SQL> EXECUTE MD_PART.SUBDIVIDE_PARTITION ('herman', –
> 'table1_p000000008');
```

Related Topics

- SD*Loader utility

MD_PART.TRUNCATE_PARTITION

Purpose

This procedure truncates a partition of a partitioned table by removing all rows.

Prerequisites

You must have the DELETE privilege on the partition to be truncated.

Syntax

```
MD_PART.TRUNCATE_PARTITION ([schema.|username,] partition_name [,
reuse_storage])
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
partition_name	specifies the name of the partition to be truncated. Datatype is VARCHAR2.
reuse_storage	specifies how the freed space is reused. Optional; values are TRUE, FALSE. TRUE specifies to reuse storage. FALSE specifies to drop storage. Default is FALSE. Datatype is BOOLEAN.

Usage Notes

Consider the following points when using this procedure:

- This procedure removes all rows of a partition.
- If TRUE is specified for storage reuse, the space from the deleted rows remains allocated to the partition. This space can be subsequently used only by new data in the partition resulting from a SQL INSERT command.
- If FALSE is specified for storage reuse, all but the space specified by the partition's MINEXTENTS parameter for the table is deallocated. This space can subsequently be used by other objects in the tablespace.
- This procedure is used to remove large amounts of data quickly.
- You cannot roll back this procedure.
- This procedure causes an implicit COMMIT to occur.

Warning: Do not use the SQL TRUNCATE statement directly on a partition. You must use the MD_PART.TRUNCATE_PARTITION procedure.

- For more information on truncating, see the SQL TRUNCATE command in the *Oracle7 Server SQL Language Reference Manual*.

Example

The following example truncates the partition TABLE1_P000000005:

```
SQL> EXECUTE MD_PART.TRUNCATE_PARTITION ('table1_p000000005');
```

Related Topics

- MD_DDL.DROP_MD_TABLE procedure

MD_WEX.DROP_TARGET

Purpose

This procedure is used to drop a table or view that was created using the window extract procedures.

Prerequisites

You must have the DROP TABLE privilege on the table or view to be dropped.

Syntax

```
MD_WEX.DROP_TARGET ([schema.|username,] target_object_name)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
target_object_name	specifies the name of the target object; table or view. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- If the target object is a view, you must use this procedure to drop the view, so that all levels of a complex view are dropped.
- If the target object is a table, you can use this procedure, or you can use the MD_DDL.DROP_MD_TABLE procedure.

Warning: Never use the SQL DROP TABLE or DROP VIEW commands to drop spatial tables or views. Always drop spatial objects using SD*SQL procedures.

Example

```
SQL> EXECUTE MD_WEX.DROP_MD_TABLE –
> (’herman’,’extract_table);
```

Related Topics

- MD_DDL.DROP_MD_TABLE procedure

MD_WEX.EXTRACT

Purpose

This procedure performs the window extract operation after the extract type is set using the other window extract procedures.

Prerequisites

You must have the SELECT privilege on the partitions to be extracted from as well as the spatial table.

Syntax

```
MD_WEX.EXTRACT ([schema.|username,] source_sd_tablename,
[schema.|username,] target_object_name)
```

Keywords and Parameters

schema	specifies the schema that contains the source table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the source table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
source_sd_tablename	specifies the name of the source table. Datatype is VARCHAR2.
schema	specifies the schema of the target object, a table or view. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the target object, a table or view. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
target_object_name	specifies the name of the target object, a table or view. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- Before running this procedure, you must first set your extract type using one of the following procedures:
  - MD_WEX.SET_POLYGON_WINDOW
  - MD_WEX.SET_PROXIMITY_WINDOW
  - MD_WEX.SET_RANGE_WINDOW
- If the target object is a table it is registered as non–partitioned.
- This procedure performs an implicit COMMIT.

Example

The following example extracts a window of TABLE1 data into a table:

```
SQL> EXECUTE MD_WEX.EXTRACT –
> ('herman','table1','herman','table1_extract');
```

Related Topics

- MD_WEX package

MD_WEX.RESET_GLOBALS

Purpose

This procedure resets the the window extract package to its default settings between uses of the package.

Prerequisites

None

Syntax

```
MD_WEX.RESET_GLOBALS
```

Keywords and Parameters

None

Usage Notes

Consider the following points when using this procedure:

- Use this procedure to reset global settings to their defaults when you execute the MD_WEX package more than once.
- The default MD_WEX settings are as follows:

Dimension list	is a blank dimension list. If not defined when MD_WEX is next executed, all dimensions in the partition key column of the specified table are used.
SQL filter	is the default SQL filter, which is the equivalent of: `'SELECT * from %s'`
Storage	is none. If not defined when MD_WEX is next executed, the default tablespace storage for the specified table is used.
Target type	is a table.
Window	is the window defined when MD_WEX is next executed.
Window type	is range.

Example

The following example resets the global MD_WEX settings to their defaults:

```
SQL> EXECUTE MD_WEX.RESET_GLOBALS
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_DIMENSION_LIST

Purpose

This procedure can be used to limit a window extract to portions of the partition key column. For example, a particular table may have a partition key with five dimensions defined in it, but you may only wish to extract based on two of the five dimensions.

Prerequisites

None

Syntax

```
MD_WEX.SET_DIMENSION_LIST (dimension_number [,
dimension_number...])
```

Keywords and Parameters

dimension_number	specifies the dimension number of the first dimension to extract. Datatype is INTEGER.
dimension_number	specifies the dimension number of the *n*th dimension to extract. Datatype is INTEGER.

Usage Notes

Setting the dimension list is optional. If it is not set, the dimension list is obtained from the Spatial Data Option data dictionary. These are the dimensions of the partition key column, in the order in which they were defined.

Example The following example indicates that the extract is based on the values of dimension 1 and dimension 4 of the 5 dimensions encoded.

```
SQL> EXECUTE MD_WEX.SET_DIMENSION_LIST (1,4);
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_HHCODE_TYPE

Purpose

This procedure is used to set the type of data to be queried to a be a point or line HHCODE.

Prerequisites

None

Syntax

```
MD_WEX.SET_HHCODE_TYPE ('POINT'|'LINE')
```

Keywords and Parameters

POINT	is the default HHCODE type, which is point data. Datatype is VARCHAR2.
LINE	specifies that the HHCODE type is line data. Datatype is VARCHAR2.

Usage Notes

If this procedure is not used before the extract process, the HHCODE type is assumed to be point.

Example

The following example sets the HHCODE type to line:

```
SQL> EXECUTE MD_WEX.SET_HHCODE_TYPE ('LINE');
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_POLYGON_WINDOW

Purpose

This procedure defines an *n*–point two–dimensional polygon window using a series of 3 to 124 vertex points for the window.

Prerequisites

None

Syntax

```
MD_WEX.SET_POLYGON_WINDOW (x1,y1,x2,y2,x3,y3 [, xn, yn...])
```

Keywords and Parameters

x1	specifies the *x* value for the first vertex point. Datatype is NUMBER.
y1	specifies the *y* value for the first vertex point. Datatype is NUMBER.
x2	specifies the *x* value for the second vertex point. Datatype is NUMBER.
y2	specifies the *y* value for the second vertex point. Datatype is NUMBER.
x3	specifies the *x* value for the third vertex point. Datatype is NUMBER.
y3	specifies the *y* value for the third vertex point. Datatype is NUMBER.
xn	specifies the *x* value for the *n*th vertex point. Datatype is NUMBER.
yn	specifies the *y* value for the *n*th vertex point. Datatype is NUMBER.

Usage Notes

Consider the following points when using this procedure:

- A minimum of three vertex points must be specified.
- Polygon windows are only valid in two dimensions.
- The polygon must be non–crossing between vertex points.
- Do not close the polygon. MD_WEX assumes that the last vertex specified is connected to the first by a straight line.

Example

The following example sets a polygon window:

```
SQL> EXECUTE MD_WEX.SET_POLYGON_WINDOW
          (–76.1,46.1,–75.2,47.1,–74.3,46.1);
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_PROXIMITY_WINDOW

Purpose

This procedure defines an *n*–dimensional proximity window by specifying a center point and a radius around that point in from 2 to 32 dimensions.

Prerequisites

None

Syntax

```
MD_WEX.SET_PROXIMITY_WINDOW (dimension_value_1, dimension_value_2,
[dimension_value_n,...] radius)
```

Keywords and Parameters

dimension_value_1	specifies the first dimension value of the center point. Datatype is NUMBER.
dimension_value_2	specifies the second dimension value of the center point. Datatype is NUMBER.
dimension_value_n	specifies the *n*th dimension value of the center point. Datatype is NUMBER.
radius	specifies the radius in the coordinate system used. Datatype is NUMBER.

Usage Notes

Consider the following points when using this procedure:

- The order of the dimension values must correspond to the order in which the dimension list was defined, if specified.
- Proximity windows are only valid for Cartesian coordinate systems.
- Only two–dimensional proximity windows can be specified for the LINES type.

Example

The following example sets a two–dimensional proximity window:

```
SQL> EXECUTE MD_WEX.SET_PROXIMITY_WINDOW (–76.1,46.1,0.102);
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_RANGE_WINDOW

Purpose

This procedure defines an *n*–dimensional range window by specifying the lower and upper range boundaries in from 1 to 32 dimensions.

Prerequisites

None

Syntax

```
MD_WEX.SET_RANGE_WINDOW (lower_window_boundary_1,
upper_window_boundary_1 [, lower_window_boundary_n,
upper_window_boundary_n...])
```

Keywords and Parameters

lower_window_boundary_1	specifies the lower window boundary value for the first dimension. Datatype is NUMBER.
upper_window_boundary_1	specifies the upper window boundary value for the first dimension. Datatype is NUMBER.
lower_window_boundary_n	specifies the lower window boundary value for the *n*th dimension. Datatype is NUMBER.
upper_window_boundary_n	specifies the upper window boundary value for the *n*th dimension. Datatype is NUMBER.

Usage Notes

Consider the following points when using this procedure:

- A range window extract is created from a list of lower boundary and upper boundary values for each dimension.
- A minimum of two coordinate pairs must be specified.
- Range windows are valid from 1 to 32 dimensions.
- Each lower boundary and upper boundary pair must correspond to and be in the order of the dimensions defined in the dimension list. For example, if the dimension list was set using the following:

  ```
  MD_WEX.SET_DIMENSION_LIST (2,1)
  ```

  the corresponding SET_RANGE_WINDOW call is as follows:

  ```
  MD_WEX.SET_RANGE_WINDOW
  (lower_window_boundary_2,upper_window_boundary_2,
  lower_window_boundary_1,upper_window_boundary_1)
  ```

Example

The following example sets a two–dimensional range window:

```
SQL> EXECUTE MD_WEX.SET_RANGE_WINDOW (–76.1,–76.0,46.1,46.2);
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_SQL_FILTER

Purpose

This procedure determines which source columns are to be copied to the target object (table or view). It can also be used to specify any non–spatial constraints to be applied to the window extract.

Prerequisites

None

Syntax

```
MD_WEX.SET_SQL_FILTER (sql_filter)
```

Keywords and Parameters

sql_filter	specifies the SELECT statement to use as the SQL filter. Optional; default is as follows: `'SELECT * FROM attribute_column(s)'`

Usage Notes

Consider the following points when using this procedure:

- The table name in the FROM clause is specified by using the %s substitution variable. The %s is replaced with the correct partition names by the procedure during its execution.
- If the target is a table, you must select at least the HHCODE, or portion or manipulation of it, since the target table is registered in the Spatial Data Option data dictionary as a non–partitioned table. This means that you cannot use the following type of SQL filter if the target is a table:

  ```
  SELECT attribute FROM %s
  ```

Example

The following example applies an ATTRIBUTE constraint as a SQL filter:

```
SQL> EXECUTE MD_WEX.SET_SQL_FILTER ('SELECT hhcolumn,attribute
          FROM %s WHERE attribute = 50');
```

This filter specifies that not only must all the records be within the specified window, but only those records with ATTRIBUTE values that equal 50 are copied to the target table or view.

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_STORAGE_CLAUSE

Purpose

This procedure defines a storage clause to use when creating a TABLE from the extracted window data.

Prerequisites

You must have the privileges necessary to use the SQL STORAGE clause.

Syntax

```
MD_WEX.SET_STORAGE_CLAUSE (storage_clause_string)
```

Keywords and Parameters

storage_clause_ string	specifies the storage clause to use. Optional; defaults are the storage parameters defined for the tablespace in which the table is created. Datatype is VARCHAR2.

Usage Notes

Setting the storage clause is optional. The default is to use the default tablespace storage parameters.

For supplementary information about setting a storage clause, see the SQL STORAGE clause in the *Oracle7 Server SQL Reference.*

Example The following example sets a storage clause for the target table:

```
SQL> EXECUTE MD_WEX.SET_STORAGE_CLAUSE ('STORAGE (initial 50K
           next 100K)');
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_TARGET_TABLESPACE

Purpose

This procedure is used to direct the target table to a particular tablespace.

Prerequisites

You must have sufficient privileges and space quota on the tablespace to create the target.

Syntax

```
MD_WEX.SET_TARGET_TABLESPACE (tablespace_name)
```

Keywords and Parameters

tablespace_name	specifies the name of the target tablespace. Optional; default is the user's default tablespace. Datatype is VARCHAR2.

Usage Notes

This procedure can be used to direct the table to a tablespace where there is sufficient space to accommodate it.

Example

The following example sets the tablespace to TABLESPACE4 for the target table:

```
SQL> EXECUTE MD_WEX.SET_TARGET_TABLESPACE ('tablespace4');
```

Related Topics

- MD_WEX.EXTRACT procedure

MD_WEX.SET_TARGET_TYPE

Purpose

This procedure specifies which type of target object you wish to create, a table or a view. When satisfying a window extract request, MD_WEX can either copy all the records that fall within a window into a new, non–partitioned table, or it can create a view.

Prerequisites

None

Syntax

```
MD_WEX.SET_TARGET_TYPE ({'TABLE'|'VIEW'})
```

Keywords and Parameters

TABLE	is the default target type. Optional; datatype is VARCHAR2.
VIEW	specifies the optional target type VIEW. Datatype is VARCHAR2.

Usage Notes

Use the MD_WEX.DROP_TARGET procedure when you wish to drop a table or view created for a temporary purpose, such as querying.

Example

The following example sets the type to VIEW for the target table:

```
SQL> EXECUTE MD_WEX.SET_TARGET_TYPE ('VIEW');
```

Related Topics

- MD_WEX.EXTRACT procedure

MDVERIFY.CHECK_TABLE

Purpose

This procedure checks the consistency of the Spatial Data Option data dictionary for a single spatial table.

Prerequisites

This procedure should be run from SQL*Plus or Server Manager.

Syntax

```
MDVERIFY.CHECK_TABLE ([schema.|username,] sd_tablename)
```

Keywords and Parameters

schema	specifies the schema that contains the table. Optional; default is the schema name of the user executing the procedure. Datatype is VARCHAR2.
username	specifies the username of the owner of the table. Optional; default is the username of the user executing the procedure. Datatype is VARCHAR2.
sd_tablename	specifies the name of the spatial table. Datatype is VARCHAR2.

Usage Notes

Consider the following points when using this procedure:

- When this procedure fails, an error is normally reported indicating the number of inconsistencies encountered. A more detailed report can be generated if the SERVEROUTPUT flag is set in SQL*Plus or Server Manager. The following is an example of setting the SERVEROUTPUT parameter to on:

```
SQL> SET SERVEROUTPUT on
```

- This procedure verifies the following for any spatial table:
  - The specified spatial table exists.
  - There are no partitions registered if the spatial table is non–partitioned.
  - All tablespaces allocated to the spatial table exist.
  - All partitions registered in the Spatial Data Option data dictionary exist.
  - Each HHCODE column defined exists.
  - The number of dimensions defined for an HHCODE column are defined in the Spatial Data Option data dictionary correctly.
  - There is only one partition key column defined for each partitioned table.

Example I

The following example verifies the TABLE1 table and succeeds:

```
SQL> EXECUTE MDSYS.MDVERIFY.CHECK_TABLE ('TABLE1');
```

The output appears as follows:

```
Spatial Table HERMAN.TABLE1 verified!
```

Example II The following example verifies the TABLE2 table and reports inconsistencies:

```
SQL> EXECUTE MDSYS.MDVERIFY.CHECK_TABLE ('TABLE2');
```

The output appears as follows:

```
partition HERMAN.TABLE2_P00000000A does not exist
Spatial Table HERMAN.TABLE2 has 1 error(s).
begin mdsys.mdverify.check_table('TABLE2');end;

*
ERROR at line 1:
ORA–20000: Verification failed with 1 error(s).
ORA–06512: at "MDSYS.MDVERIFY", line 152
ORA–06512: at "MDSYS.MDVERIFY", line 158
ORA–06512: at line 1
```

Related Topics

- MDVERIFY.CHECK_TABLES procedure

MDVERIFY.CHECK_TABLES

Purpose

This procedure checks the consistency of all the spatial tables owned by the user executing the procedure. If the user is MDSYS, then all tables in the Spatial Data Option data dictionary are checked.

Prerequisites

This procedure should be run from SQL*Plus or Server Manager.

Syntax

```
MDVERIFY.CHECK_TABLES
```

Keywords and Parameters

None

Usage Notes

Consider the following points when using this procedure:

- When this procedure fails, an error is normally reported indicating the number of inconsistencies encountered. A more detailed report can be generated if the SERVEROUTPUT parameter is set in SQL*Plus or Server Manager as in the following example:

  ```
  SQL> SET SERVEROUTPUT on
  ```

- This procedure verifies the following for any spatial table:
  - The specified spatial table exists.
  - No partitions are registered for a non–partitioned table.
  - All tablespaces allocated to the spatial table exist.
  - All partitions registered in the Spatial Data Option data dictionary exist.
  - Each HHCODE column defined exists.
  - The number of dimensions for an HHCODE column are defined correctly in the Spatial Data Option data dictionary.
  - One partition key column is defined per partitioned table.

Example I

The following example verifies all Spatial Data Option tables owned by the current user and succeeds:

```
SQL> EXECUTE MDSYS.MDVERIFY.CHECK_TABLES;
```

The output appears as follows:

```
Spatial Table HERMAN.TABLE1 verified!
```

Example II

The following example verifies all Spatial Data Option tables owned by the current user and reports inconsistencies:

```
SQL> EXECUTE MDSYS.MDVERIFY.CHECK_TABLES;
```

The output appears as follows:

```
partition HERMAN.TABLE2_P00000000A does not exist
Spatial Table HERMAN.TABLE2 has 1 error(s).
begin mdsys.mdverify.check_table('TABLE2');end;
```

```
*
ERROR at line 1:
ORA–20000: Verification failed with 1 error(s).
ORA–06512: at ”MDSYS.MDVERIFY”, line 166
ORA–06512: at line 1
```

Related Topics

- MDVERIFY.CHECK_TABLE procedure

CHAPTER

8

User–Developed SLF Converter

This chapter contains information on how to create a user–developed SLF converter. The following topics are discussed:

- introduction to user–developed SLF converters
- SLF library
- mdsisl
- mdswhd
- mdswhf
- msdwsf
- msdwst
- mdsssf
- user SLF header file
- creating the SLF converter

Introduction to User–Developed SLF Converters

Purpose

Because so many diverse and complex proprietary data formats currently exist, Spatial Data Option functions have been developed to translate proprietary data formats into a single format for encoding spatial data. As an application developer, you can use the SLF library and the Spatial Data Option data dictionary definitions to create a user–developed SLF converter to convert any format–specific data into SLF files.

The principle is straightforward: you know the data, its format, and how to read it. To translate it into the SLF format, you create a program that reads the existing data and uses the SLF library to write it into an SLF file. The SLF file contains all the spatial information in the correct format that can then be loaded into the spatial table with SD*Loader.

The data converted into an individual SLF file must be of one type only; for example, two–dimensional points. You can modify the SLF converter to convert different types of data, such as three–dimensional points or two–dimensional lines, into separate SLF files, and load them into separate spatial tables.

SLF Source Files

The following files are provided for inclusion in a user–developed SLF converter:

- SLF library file of C functions
- user SLF header file

Data Dictionary Sources

In addition to the data file that you provide in your existing format, an SLF converter requires Spatial Data Option data dictionary information as input. This information can come from one of the following:

- table control file
- Spatial Data Option data dictionary

During program execution, the data dictionary information is written to the header of the SLF file.

Table Control File

This file contains information about the Spatial Data Option data dictionary. For details on its structure and how to create one, see ”Table Control Files” in section ”SD*Converter” in Chapter 5, ”Utilities.”

Using a table control file has the following advantages:

- It allows offsite developers to create SLF converters without requiring an onsite database.
- It allows resource–intensive conversion to take place on a separate machine without an active database, and separated from the loading procedure.

Spatial Data Option Data Dictionary

You can create an SLF converter that accesses the Spatial Data Option data dictionary for the data dictionary information instead of using a table control file.

Using a Spatial Data Option data dictionary has the following advantages:

- You can add new data to an existing spatial table that already contains data from another source.
- You do not need to create a table control file, if the spatial table is already defined in the Spatial Data Option data dictionary.

Guidelines for Using Date and Time

Consider the following points when using date and time in a user–developed SLF converter:

- When defining dates in a data control file, you must use a valid date format string. For a list of valid formats as well as guidelines for using them, see Table 6 – 2 in the section ”Date Format Elements” in Chapter 6, ”SD*SQL Kernel Functions.”
- A date can be used as either a column date or as a dimensional date. If it is used as part of the COL (column data) structure, it is a DATE field. If you use a date string as a dimension, the date is converted internally into a decimal Julian date to the accuracy of milliseconds, and it is encoded into the HHCODE. The COL structure and the other data structures are described in the section ”User SLF Header File” in this chapter.
- You can use any combination of date format elements. If one is not explicitly stated, the default value is assumed.

  Note: The date format element MLS does not apply if the data is loaded into an Oracle DATE field, and is ignored.

For a definition of the COL, REC, and DIM data structures, see ”User SLF Header File” in this chapter.

Example I The following is an example of how to input a date string into a COL (column data) structure:

```
COL[i].colname="DATEFIELD";
COL[i].coldtype=DATE;
COL[i].colfmtstr="YYYY-MM-DD-HH-MI-SS";
```

The corresponding REC (record data) structure is as follows:

```
REC[i].value[j].data.s.str="1992-04-03-14-12-34";
REC[i].value[j].data.s.len=19;
```

Example II The following is an example of how to input a date string into a DIM (dimension data) structure:

```
DIM[i].dimname="time";
DIM[i].dimhhname="location";
DIM[i].dimdtye=DATE;
DIM[i].dimfmtstr="YYYY-MM-DD-HH-MI-SS-MLS";
```

The corresponding REC structure is as follows:

```
REC[i].value[j].data.s.str="1992-04-03-14-12-34-127";
REC[i].value[j].data.s.len=23;
```

SLF Library

As an application developer, you write a program in the C programming language that calls the SLF functions from the SLF library file. This library is used to link your functions with the following functions:

mdsisl ()	is the function that initializes the SLF library. It is called once.
mdswhd ()	is the function that writes the header of the SLF file from the Spatial Data Option data dictionary. It is called once, if mdswhf is not called**.**
mdswhf ()	is the function that writes the header of the SLF file from a table control file. It is called once, if mdswhd is not called.
msdwsf ()	is the function that writes the source data into SLF records in the SLF file. It is called *n* times, depending on how many data records can fit into memory at once.
msdwst ()	is the function that writes trailer information to the SLF file. It is called once.
mdsssf ()	is the function that sorts the SLF file. It is called once.

Note: The order in which you call the functions from within the program is important; they must appear in the file in the preceding order.

The next sections describe each of the SLF functions and present them in the order in which they are used. Each example presented is taken from the sample SLF converter program at the end of this chapter.

mdsisl

Purpose

This function initializes the SLF library and must be called before any other library functions.

Syntax

```
int mdsisl ()
```

Returns

This function returns either a 0 (zero) for success or a number for an error code.

Keywords and Parameters

None

Usage Notes

This function is only called once, at the beginning of the program.

Example

The following is an example of how to use the mdsisl function:

```
int slfError;
.
.
.
slfError = mdsisl();
```

mdswhd

Purpose

This function writes the header of the SLF file from the Spatial Data Option data dictionary.

Syntax

```
int mdswhd (char *slf_filename, char *username, char *password,
            char *sd_tablename, int *file_descriptor)
```

Returns

This function returns either a 0 (zero) for success or a number for an error code.

Keywords and Parameters

slf_filename	is the name of the slf file to be created.
username	is the Oracle username.
password	is the password to the user's account.
sd_tablename	is the name of the table into which the data is to be loaded. The column information of the spatial table is also accessed.
file_descriptor	is a pointer to the file descriptor. The value of this variable is set in mdswhd and is required later for other library functions.

Usage Notes

This function is used if the mdswhf function is not called. The mdswhd function uses a table control file.

Example

The following is an example of how to use the mdswhd function:

```
int   slfError;
int   *fdOut;
char  *slf_filename;
char  *username;
char  *password;
char  *sd_tablename;
.
.
.
slfError = mdswhd (slf_filename, username, password,
                   sd_tablename, &fdOut);
```

mdswhf

Purpose

This function writes the header of the SLF file from a table control file.

Syntax

```
int mdswhf (char *table_control_filename, char *slf_filename,
            int *file_descriptor, char *bad_line,
            int *bad_line_number)
```

Returns

This function returns either a 0 (zero) for success or a number for an error code.

Keywords and Parameters

table_control_filename — is the name of the table control file that contains the table control information.

slf_filename — is the name of the SLF file to be created.

&file_descriptor — is the file descriptor. The value of this variable is set in mdswhf and is required later for other library functions.

bad_line — is the text of the bad line.

bad_line_number — is the line number where any errors occur.

Usage Notes

This function is used if the mdswhd function is not called. The mdswhf function uses the Spatial Data Option data dictionary.

Example

The following is an example of how to use the mdswhf function:

```
int   slfError;
int   bad_line_number;
int   file_descriptor;
char  *table_control_filename;
char  *slf_filename;
char  *bad_line;
.
.
.
slfError = mdswhf (table_control_filename, slf_filename,
                  &file_descriptor, bad_line, &bad_line_number);
```

msdwsf

Purpose

This function writes the user data into SLF records in the SLF file, and can be called as many times as necessary, depending on how many data records fit into memory.

Syntax

```
int msdwsf (int file_descriptor, int number_of_dimensions,
            int number_of_columns, int number_of_records,
            int max_number_of_errors,
            DIM *dimensional_value_description,
            COL *column_value_description, REC *record_values,
            ERR *error_description)
```

Returns

This function returns a 0 (zero) for success, a number less than 0 if a fatal error occurred, or a number greater than 0 indicating number of non–fatal errors.

Keywords and Parameters

file_descriptor	is the file descriptor, which is passed back from either mdswhd or mdswhf.
number_of_ dimensions	is the number of dimensions to fill in the DIM structure.
number_of_ columns	is the number of columns to fill in the COL structure.
number_of_records	is the number of records passed to the function, which is the number of records filled in the REC structure.
max_number_of_ errors	is the maximum number of non–fatal errors that can occur before the function terminates. This must be specified (no default).
dimensional_ value_description	describes the dimensional values of the HHCODE columns being passed down in the REC structure. These include dimension name, dimension datatype, and, optionally, dimensional format string.
column_value_ description	describes the non–HHCODE column values being passed down to the REC structure. This includes datatype, column name, and optionally date format string.

record_values — is the record containing the actual dimension and column values.

error_description — returns error code, record numbers where errors occurred, and a brief text field that explains the error.

Usage Notes

This function must be called for each set of data records.

See "User SLF Header File" in this chapter for a description of the DIM, COL, REC, and ERR data structures.

Example

The following is an example of how to use the msdwsf function:

```
int   slfError;
int   file_descriptor;
int   dimension_number;
int   record_number;
int   maximum_number_of_errors;
DIM   *dimPtr;
COL   *colPtr;
REC   *recPtr;
VAL   *valPtr;
ERR   *errorPtr;
.
.
.
slfError = mdswsf (file_descriptor, dimension_number,
                  record_number, maximum_number_of_errors, dimPtr,
                  colPtr, recPtr, errorPtr);
```

msdwst

Purpose

This function writes trailer information to the SLF file.

Syntax

```
int msdwst (int file_descriptor)
```

Returns

This function returns either a 0 (zero) for success or a number for an error code.

Keywords and Parameters

file_descriptor is the file descriptor of the SLF file.

Usage Notes

The following is an example of how to use the msdwst function.

Example

The following example writes the trailer information to the SLF file:

```
int   slfError
int   file_descriptor
.
.
.
slfError = mdswst (file_descriptor);
```

mdsssf

Purpose

This function sorts the SLF file for loading in the spatial table with SD*Loader.

Syntax

```
int mdsssf (char *slf_filename, int bindsize)
```

Returns

This function returns either a 0 (zero) for success or a number for an error code.

Keywords and Parameters

slf_filename is the name of the SLF file to be sorted.

bindsize is the buffer size in bytes, which is the sort area size in memory to perform the sort.

Usage Notes

This function is optional; if necessary, the data can be sorted by the load function.

During the sorting process, two temporary files are created on disk. These files are created with the extensions **.tmp** and **.srt**, and are created in the same directory as the SLF file. If a sort is successful, these files are deleted, but if a sort fails, the temporary files remain on disk and must be removed manually.

Note: As a general rule, a large buffer size is recommended; however, once the maximum physical free memory on your machine is reached, it begins to swapping occurs and performance degrades severely. You must tune the performance for your working environment.

Example

The following is an example of how to use the mdsssf function:

```
int   slfError;
int   bindsize
char  *slf_filename;
.
.
.
slfError = mdsssf (slf_filename, bindsize);
```

User SLF Header File

The user SLF header file contains the description of the data structures that must be used with the msdwsf function. This include file must be referenced in your SLF converter. For an example, see Line 7 of the SLF file in the section ”Creating the SLF Converter.”

Datatypes

The SLF library supports the following datatypes:

STRING	0	is a CHAR * value (CHAR, VARCHAR2, or RAW).
LONG	1	is a LONG value (NUMBER).
DOUBLE	2	is a DOUBLE value (NUMBER).
DATE	3	is a CHAR * value (DATE).

Data Structures

The data structures described are as follows:

VAL — is the value data structure. It stores the actual values being passed to the function msdwsf. It is a union of three different data types. You must pass the length of a string if you are using the STRING datatype. This structure is defined as follows:

```
typedef struct
{
   short isnull;  /* 0 – False, !0 – True  */
   union
   {
      struct
      {
         char *str;
         int   len;
      } s;
      long   l;
      double d;
   } data;
} VAL;
```

REC — is the record data structure. This structure is made up of one or more value structures. You must allocate one for each dimension value and one for each column value. This structure is to be filled with all the dimensional values first, followed by all the column values. Do not include the dimensional column in this list because it is made up of the dimensional values passed to it. This structure is defined as follows:

```
typedef struct
{
   VAL *value;
} REC;
```

DIM

is the dimension data structure. This structure describes the dimensional value that is passed down to the function msdwsf. You must allocate one for each dimension value passed down, corresponding directly with the placement of the dimensional values in the record structure. The first value in this list describes the first data value passed in the record structure, and so on. The DIMDTYPE must be one of the following: LONG, DOUBLE, or DATE. The DIMHHNAME must be the dimensional element column name to which it corresponds. The DIMNAME is the dimensional column name. This structure is defined as follows:

```
typedef struct
{
   int   dimdtype;
   char *dimhhname;
   char *dimname;
   char *dimfmtstr;
} DIM;
```

COL

is the column data structure. This structure describes the non–dimensional, or attribute, columns. You must allocate one for each column value passed down, corresponding directly with the column values being passed down to the msdwsf function. If you have a total of three dimensions, the first column structure in this list corresponds to the fourth data value of the record structure, and so on. The COLDTYPE must be one of the following: LONG, STRING, DOUBLE, or DATE. The COLNAME is the column name it corresponds to in the spatial table. If the COLDTYPE is DATE, then you must specify the format string of the date strings that is passed. This structure is defined as follows:

```
typedef struct
{
   int   coldtype;
   char *colname;
   char *colfmtstr;
} COL;
```

ERR

is the error data structure. In the structure, ERR_NUM_SLFERR contains a value from the list defined in this header, REC_NUM_SLFERR is the record number where the error occurred, and EXTRA_TEXT_SLFERR contains supplemental textual information, such as a column or dimension name. This structure is defined as follows:

```
typedef struct
{
   int   err_num_slferr;
   int   rec_num_slferr;
   char  extra_text_slferr [31];
} ERR;
```

Usage Notes

The data structures are used in passing column and dimensional information about the associated data being passed to the function msdwsf. These structures allow the user to pass different types of data and determine whether the data being passed is null.

You must allocate memory for these data structures and provide the appropriate values.

The SLF converter does not create either a log file or a bad file, unlike SD*Converter. The status of each function is returned as an error code, and bad records are referenced by the ERR structure.

A copy of the user SLF header file follows. Before creating an SLF converter, study the structure of the file. Also, you may want to compare the structure with the sample SLF converter program in the following section.

User SLF Header File

```
#ifndef MDUSLF_ORACLE
#define MDUSLF_ORACLE
/*-----------------------------------------------------------------------------*
 * Four data types supported by the SLF library.                             *
 *-----------------------------------------------------------------------------*/
#define STRING    0    /* ”char *” value (CHAR, VARCHAR2, RAW)     */
#define LONG      1    /* long value (NUMBER)                      */
#define DOUBLE    2    /* double value (NUMBER)                    */
#define DATE      3    /* ”char *” value (DATE)                    */
/*-----------------------------------------------------------------------------*
 * Value structure: This stores the actual values being passed to the        *
 * function mdswsf.  It is a union of three different data types. You must   *
 * passed the length of a string if you are using the STRING datatype.       *
 * Along with the data being passed, you must also mention if the data value *
 * is a null value or not.                                                    *
 *-----------------------------------------------------------------------------*/
typedef struct
{
   short isnull;  /* 0 – False, !0 – True  */
   union
   {
      struct
      {
         char *str;
         int   len;
      } s;
      long   l;
      double d;
   } data;
} VAL;
/*-----------------------------------------------------------------------------*
 * Record structure: The record structure is made up of one or more Value    *
 * structures. You must allocate one for each dimension value and one for    *
 * each column value. This record structure is to be filled with all the     *
 * dimensional values first and then all the column values. Note: Do not     *
 * include the spatial column in this list because it is made up of the      *
 * dimensional values passed to it.                                           *
 *-----------------------------------------------------------------------------*/
typedef struct
{
   VAL *value;
} REC;
/*-----------------------------------------------------------------------------*
 * Dimensional structure: The dimensional structure describes the            *
 * dimensional value that is passed down to the function mdswsf. You must    *
 * allocate one for each dimension passed down and corresponds directly with *
```

```
 * the placement of the dimensional values in the Record structure. The      *
 * first one in this list describes the first data value passed in the       *
 * Record structure, the second one in this list describes the second data   *
 * value passed in the Record structure and so on. The dimdtype must be one  *
 * of the following: LONG, DOUBLE or DATE. The dimhhname must be the     *
 * spatial element column name that it corresponds to, and dimname is the    *
 * dimensional column name.                                                  *
 *---------------------------------------------------------------------------*/
typedef struct
{
   int   dimdtype;
   char *dimhhname;
   char *dimname;
   char *dimfmtstr;
} DIM;
/*---------------------------------------------------------------------------*
 * Column structure: The column structure describes the non-spatial columns  *
 * (attributes). You must allocate one for each column value passed down and *
 * corresponds directly with the column values being passed down (after the  *
 * dimensional values) to the function mdswsf. If you have a total of 3      *
 * dimensions, the first column structure in this list would correspond to   *
 * the 4th data value if the Record structure, and so on. The coldtype must  *
 * be one of the following: LONG, STRING, DOUBLE or DATE. The colname is the *
 * column name it corresponds to in the spatial table. If the coldtype is    *
 * DATE then you must specify the format string of the date strings that     *
 * is passed.                                                                *
 *---------------------------------------------------------------------------*/
typedef struct
{
   int   coldtype;
   char *colname;
   char *colfmtstr;
} COL;
/*---------------------------------------------------------------------------*
 * Error structure: err_num_slferr will contain a value from the list defined*
 * in this header, extra_text_slferr will contain supplemental information,  *
 * like a column or dimension name.                                          *
 *---------------------------------------------------------------------------*/
typedef struct
{
   int   err_num_slferr;               /* Error code from the following list */
   int   rec_num_slferr;               /* Record number where error occurred */
   char  extra_text_slferr [31];       /* Extra text, like a field name      */
} ERR;
#endif /* MDUSLF_ORACLE */
```

Creating the SLF Converter

To create an SLF converter, write a C program that uses the SLF library of C functions and includes the SLF header file to convert the data and write it to an SLF file.

Conventions

Use the sample SLF converter program as a model for creating the SLF converter. The following conventions are used in the sample program:

- The numbers at the left are line numbers, which are referenced in the explanation following the sample program. Do not use them in your program file.
- When a line continues onto the next line without a new line number, it is not a carriage return, merely a wrapped line.
- The /* indicates a comment. Anything following to the corresponding */ is ignored. The comments in the sample program provide information about the program itself, and can offer direction in creating your own.
- The bold lines indicate that they are referenced in the explanation.

The sample SLF converter shown in the next section is a C program that reads a defined data file, then uses library calls to write the data to an SLF file. A description of the data is passed to the write function, using the included header file, which provides the necessary data structure. For a description of this file, see ”User SLF Header File” in this chapter.

Requirements

Keep the following requirements in mind when writing the SLF converter:

- The user SLF header file must be included.
- The SLF library file must be included.

The source data is described following the sample program, and is referenced by the line number appearing on the left. Do not include line numbers when you write the SLF converter.

Sample SLF Converter Program

The following is a sample user–developed SLF converter that converts data from the **points.dat** file to SLF, creating the file **pts2slf1.slf**.

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Converter header file
 */
#include <mduslf.h>


/*
 * Conversion Utility :
 *
 * Format of the points data file :
 *  Fixed record length of 33 bytes
 *  0 based positions for all columns in the file are :
 *    lon    bytes 1 to 10
 *    lat    bytes 13 to 22
 *    depth  bytes 24 to 28
 *
 * lon & lat are making up the 2 dimensions of the hhcode
 *
 * This conversion utility will access the Spatial Data Option data dictionary
 * via initialization routine mdswhd.
 *
 * Here is the spatial table definition :
 *  location hhcode with dimensions (lon dim (–180,180,7), lat dim (–90,90,7)),
 *  depth    number (38,10))
 *
 * This conversion utility will only work on the points data file
 * since everything is hard coded.
 *
 *
 */

#define  USER_NAME     ”herman”
#define  PASSWORD      ”vampire”

/*
 * Spatial table name.  Must have been created in the database before the
 * conversion takes place since the Spatial Data Option data dictionary is looked at.
 * Look at the script points.sql in the create_table directory.
 */
#define  MD_TABLE      ”points”
#define  RECORD_LENGTH  (33+1)

```

```
#define  DATA_FILE     ”points.dat”
#define  SLF_FILE      ”pts2slf1.slf”

/*
 * points table contains one spatial column composed of two
 * dimensions : lon and lat.
 */
#define DIM_NO          2

/* depth column */
#define  COL_NO         1

/* We will process 10000 records at a time */
#define REC_NO          10000
#define MAX_ERRORS      10


void reportErrors (slfError, errorPtr)
int  slfError;
ERR *errorPtr;

/*
 * Simple error handler.
 * If an error is fatal, just exit the converter.
 * If the error is non fatal, report it and continue.
 */

{
   int  j;

   /*
    * A negative value means there was a fatal error (may contain non–fatal).
    * A positive value means there were non–fatal errors.
    * eg. A –5 means (4 non–fatal errors and 1 fatal)
    * eg. A  5 means there were 5 non–fatal errors.
    *
    */

   for (j = 0; j < abs (slfError); j++)
   {
      printf (”Record No (%d) ”, errorPtr [j].rec_num_slferr);
      printf (”Error No (%d) ”, errorPtr [j].err_num_slferr);
      printf (”with (%s)\n”, errorPtr [j].extra_text_slferr);
   }

   if (slfError < 0)
      exit (1);
}
```

```
main ()

{
   int    i;
   int    fdOut;
   int    slfError;

   FILE  *fdIn;

   ERR   *errorPtr;
   DIM   *dimPtr = NULL;
   COL   *colPtr = NULL;
   REC   *recPtr = NULL;
   VAL   *valPool = NULL;

   VAL   *longitudePtr;
   VAL   *latitudePtr;
   VAL   *depthPtr;

   char  *dataFile;
   char  *slfFile;
   char  *dataRecordPtr;

   char   dataRecord [RECORD_LENGTH];


   fdIn = fopen (DATA_FILE, ”r”);
   if (fdIn == NULL)
   {
      printf (”Cannot open file (%s) for read\n”, DATA_FILE);
      exit (1);
   }

   /*
    * Initialize the converter and write header information into the slf file.
    * All following calls to mdswsf will use the returned file handle fdOut.
    */

   slfError = mdsisl ();
   if (slfError != 0)
   {
      printf (”Fatal error occurred in Initializing library: %d\n”, slfError);
      exit (1);
   }

   slfError = mdswhd (SLF_FILE, USER_NAME, PASSWORD, MD_TABLE, &fdOut);
   if (slfError != 0)
   {
      printf (”Cannot create (%s) header for userName (%s), password (%s), \n”,
```

```
                SLF_FILE, USER_NAME, PASSWORD);
      printf (”and MDtable (%s)\n”, MD_TABLE);
      exit (1);
   }

   /* Allocate all data structures */

   dimPtr = (DIM *) malloc (sizeof (DIM) * DIM_NO);
   colPtr = (COL *) malloc (sizeof (COL) * COL_NO);
   recPtr = (REC *) malloc (sizeof (REC) * REC_NO);
   valPool = (VAL *) malloc (sizeof (VAL) * REC_NO * (DIM_NO + COL_NO));
   errorPtr = (ERR *) malloc (sizeof (ERR) * MAX_ERRORS);

   if ((dimPtr != NULL) && (colPtr != NULL) &&
       (recPtr != NULL) && (valPool != NULL) && (errorPtr != NULL))
   {
      /*
       * Initialize the value array in the REC structure.
       */

      for (i = 0; i < REC_NO; i++)
      {
         recPtr[i].value = valPool + (i * (DIM_NO + COL_NO));
      }

      /*
       * Initialize the DIM and COL structures with information about
       * the MD table.
       * What is important in this initialization is that the order the
       * dimensions and the columns appear in the DIM array and the col
       * array has to match the order in the REC structure.
       * All dimensions comes first and then all columns.
       */

      dimPtr [0].dimdtype = DOUBLE;
      dimPtr [0].dimhhname = ”location”;
      dimPtr [0].dimname = ”lon”;
      dimPtr [0].dimfmtstr = NULL;
      dimPtr [1].dimdtype = DOUBLE;
      dimPtr [1].dimhhname = ”location”;
      dimPtr [1].dimname = ”lat”;
      dimPtr [1].dimfmtstr = NULL;
      colPtr [0].coldtype = LONG;
      colPtr [0].colname = ”depth”;
      colPtr [0].colfmtstr = NULL;

      if (fdOut != NULL)
      {
         /*
```

```
    * Read in the data file.
    * For each chunck of REC_NO, call the conversion routine
    * mdswsf
    */

   i = 0;

   printf ("\n");

   while (fgets (dataRecord, RECORD_LENGTH, fdIn) != NULL)
   {
      /*
       * First value in the record will be the longitude, followed
       * by the latitude and then by depth.
       * Note that this order matches the order the DIM and COL arrays
       * were initialized (dimension values first then column values;
       * in the same order as the DIM and COL arrays).
       */

      longitudePtr = recPtr[i].value;
      latitudePtr = longitudePtr+1;
      depthPtr = latitudePtr+1;
      longitudePtr->isnull = 0;
      latitudePtr->isnull =0;
      depthPtr->isnull = 0;

      longitudePtr->data.d = atof (dataRecord);
      latitudePtr->data.d = atof (&dataRecord [13]);
      depthPtr->data.l = atol (&dataRecord [25]);

      if (i == REC_NO - 1)
      {
         i = 0;

         printf ("Writing %-5d records\n", REC_NO);
         slfError = mdswsf (fdOut, DIM_NO, COL_NO, REC_NO, MAX_ERRORS,
                            dimPtr, colPtr, recPtr, errorPtr);
         if (slfError != 0)
         {
            reportErrors (slfError, errorPtr);
         }
      }
      else
         i++;
   }

   /*
    * Only call the conversion routine if more records have to be
    * processed
```

```
       */

      if (i > 0)
      {
         printf ("Writing %-5d records\n", i);
         slfError = mdswsf (fdOut, DIM_NO, COL_NO, i, MAX_ERRORS,
                            dimPtr, colPtr, recPtr, errorPtr);
         if (slfError != 0)
         {
            reportErrors (slfError, errorPtr);
         }
      }
   }

   /* Free memory */

   free (dimPtr);
   free (colPtr);
   free (recPtr);
   free (valPool);
}
else
{
   printf ("Could not allocate essential memory.\n");
}

/*
 * Terminate the process : write the slf footer and
 * close the input file
 */

slfError = mdswst (fdOut);
if (slfError != 0)
{
   printf ("Fatal error occurred in Writing Trailer: %d\n", slfError);
   exit (1);
}

fclose (fdIn);

/*
 * Optional call to sort the slf file
 */

printf ("\nSorting...\n");
slfError = mdsssf (SLF_FILE, 1000000);
if (slfError != 0)
{
```

```
290        printf ("Fatal error occurred in Sorting: %d\n", slfError);
291        exit (1);
```

The preceding program is an example of a user–defined SLF converter. You create a C program that uses library calls to produce the SLF file. You interpret your data, then the library writes your data to the SLF file. To do this you must describe the data being passed to the write functions. The user SLF header file provides the necessary data structures for you to describe the data passed. In the example, the spatial table already exists and the SLF file is generated from the existing data dictionary and your data.

Example Explanation

The following is an explanation of the preceding example, defining the terms used. Line numbers in the explanation correspond with the line numbers in the example.

Line	**Description**
line 7	The name of the SLF header file must be included in the C program. You must allocate the appropriate number of elements for each data structure in the file.
line 133	Before calling any of the library functions, call the function mdsisl to initialize the SLF library.
line 140	To create the header of the SLF file, use either the mdswhd or mdswhf function to specify whether the information is obtained from the active Spatial Data Option data dictionary or from a table control file. The example gets the information from the Spatial Data Option data dictionary, so it specifies the mdswhd function.
line 151	The DIM structure allocates space for all dimensional values passed; that is, the total dimensions of all the dimensional elements.
line 152	The COL structure allocates space for all non–dimensional values passed; that is, the attribute elements.
line 153	The REC structure allocates space for the number of records being passed to the write function.
line 154	The VAL structure allocates space for the number of values for each record.
line 155	The ERR structure allocates space for the number of errors allowed; that is, non–fatal errors.

You must fill in the preceding structures with descriptions to let the library functions know what types of value are being passed.

line 164 – 167
: Assign space for each value being passed in the record structure. You must assign space for each dimension value and column values.

line 178 – 185
: Describe each dimension value that is passed to the write function. You must specify the name of the spatial element that the dimension belongs to, the dimension name, and the type of value being passed. The datatypes allowed are stated in the user SLF header file. Dimensional values must precede attribute elements.

line 186 – 188
: Describe the column values that are passed to the write function.

line 212 – 213
: Place the dimension information in the REC structure which correspond to the descriptions in the DIM structure. In the example, the value being placed in the structure corresponds to the first element in the `DIM` structure, and so on.

line 214
: Place the column values in the REC structure. For string datatypes, you must also pass the length of the string.

line 215 – 217
: State whether or not the value being passed down is a NULL value, as demonstrated on line 215. One represents a NULL value. Zero is not a NULL value.

lines 219 – 221
: Fill the record structure with data. The values in the record structure must be dimension (DIM) values and then the column (COL) values.

lines 228 – 229
: Pass the information to the msdwsf write function. You must pass the following:

- the file descriptor, which is passed back by either the mdswhd or mdswhf function
- number of dimensions
- number of columns
- number of records
- number of errors allowed

- the DIM, COL, REC, and ERR structures. The function returns a zero if successful, while a number greater than zero indicates the number of non–fatal errors that occurred. A number less than zero indicates that a fatal error occurred.

line 258 – 261	Free up the allocated space.
line 273	Once the writing is complete, write a trailer to the SLF file using the msdwst function.
line 287	Call the sort function. You must pass the SLF filename and a bindsize for the sorting function. The bindsize is the buffer size in bytes, which is the sort area size in memory to do the sorting.

PART

III Appendices

APPENDIX

A

Spatial Data Option Data Dictionary Reference

This appendix contains descriptions of the views that are available and includes the following topics:

- consistency
- views

Consistency

To ensure that Spatial Data Option data dictionary table entries are consistent, the PL/SQL package MDVERIFY has been provided. You do not need to run procedures that call MDVERIFY during normal system operation; however, the procedures can provide valuable diagnostic information to Oracle Worldwide Support if the Spatial Data Option data dictionary becomes corrupted.

Use procedures that call the MDVERIFY package just as you would any stored procedure.

For information on how to use the MDVERIFY package, see Chapter 7, "SD*SQL Packages."

Views

This section is an alphabetical reference to the Spatial Data Option data dictionary views accessible to all users.

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_DIMENSION
- 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 Option data dictionary.

For information on Oracle7 data dictionary views, see Appendix B in the *Oracle7 Server Administrator's Guide.*

ALL_MD_DIMENSIONS

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

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

ALL_MD_EXCEPTIONS

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

OWNER	is the owner of the object.
NAME	is the object name.
OPERATION	is the operation during which the fail occurred.
CCHH	is the common code HHCODE.

ALL_MD_LOADER_ERRORS

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

OWNER	is the owner of the table where the error occurred.
MD_TABLE_NAME	is the spatial table name.
FILENAME	is the SLF file name.
ROWS_LOADED	is the 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**

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

ALL_MD_TABLES **Returns a list of all the user–accessible spatial tables**

OWNER	is the owner of the table.
MD_TABLE_NAME	is the name of the spatial table.
CLASS	is the class of table: PARTITIONED or NON–PARTITIONED.
PTAB_SEQ	is the number of last partitioned table created.
HIGH_WATER_MARK	is the maximum number of rows that can be inserted into a partitioned table.
OFFLINE_PATH	is the complete pathname to directory where the table is archived.
COUNT_MODE	is the 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**

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

DBA_MD_COLUMNS **Returns a list of all columns that are part of Spatial Data Option tables**

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

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

DBA_MD_EXCEPTIONS — **Contains information about spatial tables that should be dropped as a result of some failed operation, such as a failed load**

OWNER	is the owner of the object.
NAME	is the object name.
OPERATION	is the operation during which the fail occurred.
CCHH	is the common code HHCODE.

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

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

DBA_MD_PARTITIONS — **Returns a list of all the partitioned tables**

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

DBA_MD_TABLES — **Returns a list of all the spatial tables**

OWNER	is the owner of the table.
MD_TABLE_NAME	is the name of the spatial table.
CLASS	is the class of table: PARTITIONED or NON–PARTITIONED.
PTAB_SEQ	is the number of last partitioned table created.
HIGH_WATER_MARK	is the maximum number of rows that can be inserted into a partitioned table.
OFFLINE_PATH	is the complete pathname to directory where the table is archived.
COUNT_MODE	is the 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**

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

ALL_MD_COLUMNS

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

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

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

MD_TABLE_NAME	is the name of the spatial table.
COLUMN_NAME	is the name of the column.
DATA_TYPE	is the datatype of the column.
DATA_LENGTH	is the length of the column in bytes.
DATA_PRECISION	is the scale for NUMBER datatype, binary precision for FLOAT datatype, and NULL for all other datatypes.
DATA_SCALE	is the digits to right of the decimal point in an HHCODE or a number.
NDIM	is the number of dimensions in the HHCODE column. It is NULL for all other datatypes.
MAX_LEVEL	is the maximum number of levels in the column.
NULLABLE	indicates if column allows NULLs.
PARTITION_KEY	indicates if column is the partition key column; only one allowed per partitioned table.
COLUMN_ID	is the sequence number of the column as created.
DEFAULT_LENGTH	is the length of the default value for the column.
NUM_DISTINCT	is the number of distinct values in each column of the table.
LOW_VALUE	is the 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	is the 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**

MD_TABLE_NAME	is the name of the spatial table.
COLUMN_NAME	is the name of the column.
DIMENSION_NAME	is the name of the dimension.
DIMENSION_NUMBER	is the dimension number.
LOWER_BOUND	is the lower boundary of dimension range.
UPPER_BOUND	is the upper boundary of dimension range.
SCALE	is the scale of the dimension.
RECURSION_LEVEL	is the number of levels encoded in the HHCODE.

USER_MD_EXCEPTIONS — **Contains information about spatial tables that should be dropped as a result of some failed operation, such as a failed load**

NAME	is the object name.
OPERATION	is the operation during which the fail occurred.
CCHH	is the common code HHCODE.

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

MD_TABLE_NAME	is the spatial table name.
FILENAME	is the SLF file name.
ROWS_LOADED	is the 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**

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

USER_MD_TABLES — **Returns a list of all the spatial tables owned by the user**

MD_TABLE_NAME	is the name of the spatial table.
CLASS	is the class of table: PARTITIONED or NON–PARTITIONED.
PTAB_SEQ	is the number of last sequence created.
HIGH_WATER_MARK	is the maximum number of rows that can be inserted into a partitioned table.
OFFLINE_PATH	is the complete pathname to directory where the table is archived.
COUNT_MODE	is the 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**

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

APPENDIX

B Quick Reference

This appendix is a quick reference to all utilities, functions, and procedures you can use, for the Spatial Data Option. The following topics are included:

- utilities
- SD*SQL kernel functions
- SD*SQL packages
- user–developed SLF converter

Utilities

SD*Converter

The following syntax shows the combinations of keywords and parameters for SD*Converter, depending on the input combinations.

Data Control File Table Control File and Data File

When the input combination is a data control file, a table control file, and a data file, the syntax is as follows:

```
SDCONV DATACONTROL=data_control_filename
TABLECONTROL=table_control_filename DATA=data_filename
[SLF=data_filename.SLF|slf_filename] [LOG=log_filename]
[BAD=bad_filename] [ERRORS=50|max_number_of_errors]
[BINDSIZE=65536|bindsize] [SORT=YES|NO]
```

Data Control File Spatial Data Option Data Dictionary and Data File

When the input combination is a data control file, a Spatial Data Option data dictionary, and a data file, the syntax is as follows:

```
SDCONV USERID=username/password DATACONTROL=data_control_filename
TABLECONTROL=sd_tablename DATA=data_filename
[SLF=data_filename.SLF|slf_filename] [LOG=log_filename]
[BAD=bad_filename] [ERRORS=50|max_number_of_errors]
[BINDSIZE=65536|bindsize] [SORT=YES|NO]
```

Oracle7 Table and Spatial Data Option Data Dictionary

When the input combination is a Spatial Data Option data dictionary and a standard Oracle7 table, the syntax is as follows:

```
SDCONV USERID=username/password TABLECONTROL=sd_tablename
TABLE=Oracle7_tablename [SLF=Oracle7_tablename.SLF|slf_filename]
[LOG=log_filename] [BAD=bad_filename]
[ERRORS=50|max_number_of_errors] [BINDSIZE=65536|bindsize]
```

Table Control File

```
COLUMN column_name {VARCHAR2|NUMBER|DATE|RAW|CHAR|HHCODE
[PARTITION KEY]}
DIMENSION dimension_name hhcode_column_name (dimension_number,
lower_boundary, upper_boundary, scale)
```

Data Control File

```
{ASCII|BINARY}
FIXED record_length
COLUMN column_name POSITION {(number:number)|(number)}
{DATE date_format_string|INTEGER|SMALLINT|FLOAT|DOUBLE|
BYTEINT|RAW|CHAR} [NULLIF POSITION
{NE|!=|<>|EQ|==|=}'char_string']
DIMENSION dimension_name hhcode_column_name POSITION
{(number:number)|(number)} {DATE
date_format_string|INTEGER|SMALLINT|FLOAT|DOUBLE|BYTEINT}
```

SD*Loader

```
SDLOAD USERID=username/password SLF=slf_filename
SDTABLE=sd_tablename [LOG=log_filename] [BINDSIZE=65536|bindsize]
[ROLLBACK=rollback_segment_name] [DIRECT=TRUE|FALSE]
[CTLKEEP=TRUE|FALSE]
```

SD*SQL Kernel Functions

HHBYTELEN HHBYTELEN (*number_of_dimensions*, *maximum_levels*)

HHCELLBNDRY HHCELLBNDRY (*hhcode_expression*, *dimension_number*, *lower_boundary*, *upper_boundary*, *number_of_levels*, {'MIN'|'MAX'})

HHCELLSIZE HHCELLSIZE (*lower_boundary*, *upper_boundary*, *number_of_levels* [, *lower_boundary*, *upper_boundary*, *number_of_levels*...])

HHCLDATE HHCLDATE (*julian_date*, *'date_format_string'*)

HHCOLLAPSE HHCOLLAPSE (*hhcode_expression*, *dimension_number* [, *dimension_number*...])

HHCOMMONCODE HHCOMMONCODE (*hhcode_expression*, *hhcode_expression*)

HHCOMPOSE HHCOMPOSE (*hhcode_expression*, *dimension_number* [, *dimension_number*...])

HHDECODE HHDECODE (*hhcode_expression*, *dimension_number*, *lower_boundary*, *upper_boundary*)

HHDISTANCE HHDISTANCE ({'EUCLID'|'MANHATTAN'}, *hhcode_expression_1*, *hhcode_expression_2*, *lower_boundary_1*, *upper_boundary_1* [, *lower_boundary_n*, *upper_boundary_n*...])

HHENCODE HHENCODE (*value*, *lower_boundary*, *upper_boundary*, *scale* [, *value*, *lower_boundary*, *upper_boundary*, *scale*...])

HHGROUP HHGROUP (*hhcode_expression*)

HHIDLPART HHIDLPART ({'RANGE'|'PROXIMITY'|'POLYGON'}, COMMON_HHCODE, *lower_boundary_1*, *upper_boundary_1*, *lower_boundary_2*, *upper_boundary_2*, *window_definition*)

Window Definition The following syntax is used to define the window extract area:

Range: *lower_window_boundary_1*, *upper_window_boundary_1*, *lower_window_boundary_2*, *upper_window_boundary_2*

Proximity: *center_1, center_2, radius*

Polygon: *x1*, *y1*, *x2*, *y2*, *x3*, *y3* [, *xn*, *yn*...]

HHIDLROWS HHIDLROWS ({'RANGE'|'PROXIMITY'|'POLYGON'}, *partition_key*, *lower_boundary_1*, *upper_boundary_1*, *lower_boundary_2*, *upper_boundary_2*, *window_definition*)

Window Definition

The following syntax is used to define the window extract area:

Range: *lower_window_boundary_1*, *upper_window_boundary_1*, *lower_window_boundary_2*, *upper_window_boundary_2*

Proximity: *center_1, center_2, radius*

Polygon: *x1*, *y1*, *x2*, *y2*, *x3, y3* [*, xn, yn*...]

HHIDPART HHIDPART ({'RANGE'|'PROXIMITY'|'POLYGON'}, COMMON_HHCODE, *lower_boundary_1*, *upper_boundary_1*, [*lower_boundary_n*, *upper_boundary_n*,...] *window_definition*)

Window Definition

The following syntax is used to define the window extract area:

Range: *lower_window_boundary_1*, *upper_window_boundary_1* [, *lower_window_boundary_n*, *upper_window_boundary_n*...]

Proximity: *center_1, center_2, (center_n, ...) radius*

Polygon: *x1*, *y1*, *x2*, *y2*, *x3, y3* [*, xn, yn*...]

HHIDROWS HHIDROWS ({'RANGE'|'PROXIMITY'|'POLYGON'}, *partition_key*, *lower_boundary_1*, *upper_boundary_1*, [*lower_boundary_n*, *upper_boundary_n*,...] *window_definition*)

Window Definition

The following syntax is used to define the window extract area:

Range: *lower_window_boundary_1*, *upper_window_boundary_1* [, *lower_window_boundary_n*, *upper_window_boundary_n*...]

Proximity: *center_1, center_2, (center_n, ...) radius*

Polygon: *x1*, *y1*, *x2*, *y2*, *x3, y3* [*, xn, yn*...]

HHJLDATE HHJLDATE ('*calendar_date*','*date_format_string*')

HHLENGTH HHLENGTH (*hhcode_expression* [, *dimension_number*])

HHLEVELS HHLEVELS (*lower_boundary*, *upper_boundary*, *scale*)

HHMATCH HHMATCH (*hhcode_expression*, *hhcode_expression*)

HHNDIM HHNDIM (*hhcode_expression*)

HHORDER HHORDER (*hhcode_expression*)

HHPRECISION HHPRECISION (*lower_boundary*, *upper_boundary*, *number_of_levels*)

HHSUBSTR HHSUBSTR (*hhcode_expression*, *start_level*, *end_level*)

SD*SQL Packages

MD_DDL. ACTIVATE_ TABLESPACE	`MD_DDL.ACTIVATE_TABLESPACE ([`*schema.*`|`*username,*`]` *sd_tablename, tablespace_name*`)`
MD_DDL. ADD_HHCODE_ COLUMN	`MD_DDL.ADD_HHCODE_COLUMN ([`*schema.*`|`*username,*`]` *sd_tablename, hhcode_column_name,*`[`*partition_key,*`] [`*not_null,*`]` *dimension_name, lower_boundary, upper_boundary, scale* `[,` *dimension_name, lower_boundary, upper_boundary, scale*`...])`
MD_DDL. ALLOCATE_ TABLESPACE	`MD_DDL.ALLOCATE_TABLESPACE ([`*schema.*`|`*username,*`]` *sd_tablename, tablespace_name*`)`
MD_DDL. ALTER_MD_TABLE	`MD_DDL.ALTER_MD_TABLE ([`*schema.*`|`*username,*`]` *sd_tablename, sql_template* `[,` *continue_on_error*`])`
MD_DDL. ALTER_ MD_TABLE_CM	`MD_DDL.ALTER_MD_TABLE_CM ([`*schema.*`|`*username,*`]` *sd_tablename,* `{'EXACT'|'ESTIMATE'})`
MD_DDL. ALTER_ MD_TABLE_HWM	`MD_DDL.ALTER_MD_TABLE_HWM ([`*schema.*`|`*username,*`]` *sd_tablename, high_water_mark*`)`
MD_DDL. DEACTIVATE_ TABLESPACE	`MD_DDL.DEACTIVATE_TABLESPACE ([`*schema.*`|`*username,*`]` *sd_tablename, tablespace_name*`)`
MD_DDL. DROP_MD_TABLE	`MD_DDL.DROP_MD_TABLE ([`*schema.*`|`*username,*`]` *sd_tablename*`)`
MD_DDL. REGISTER_MD_ TABLE	`MD_DDL.REGISTER_MD_TABLE ([`*schema.*`|`*username,*`]` *sd_tablename* `[,` *high_water_mark* `[,` *NULL*`, ['EXACT'|'ESTIMATE']]])`
MD_DML. GENHHCODE	`MD_DML.GENHHCODE ([`*schema.*`|`*username,*`]` *sd_tablename,* `[`*hhcode_column_name*`],` *dimension_1* `[,` *dimension_n...*`]`
MD_DML. LOCK_MD_TABLE	`MD_DML.LOCK_MD_TABLE ([`*schema.*`|`*username,*`]` *sd_tablename* `[, 'EXCLUSIVE'|'ROW EXCLUSIVE'|'ROW SHARE'|'SHARE'|'SHARE UPDATE'|'SHARE ROW EXCLUSIVE', [`*lock_mode*`]])`

MD_DML. MOVE_RECORD	`MD_DML.MOVE_RECORD ([`*schema.*`|`*username,*`]` *sd_tablename*, *old_hhcode*, *new_hhcode* `[,` *attribute_where_clause*`])`
MD_PART. CLEAR_EXCEPTION_ TABLES	`MD_PART.CLEAR_EXCEPTION_TABLES`
MD_PART. CREATE_INFERRED_ PARTITION	`MD_PART.CREATE_INFERRED_PARTITION ([`*schema.*`|`*username,*`]` *sd_tablename*, *hhcode_expression*`)`
MD_PART. DROP_PARTITION	`MD_PART.DROP_PARTITION ([`*schema.*`|`*username,*`]` *partition_name*`)`
MD_PART. GET_PARTITION_ NAME	`MD_PART.GET_PARTITION_NAME ([`*schema.*`|`*username,*`]` *sd_tablename*, *hhcode_expression*, *wait_mode*, *partition_name*`)`
MD_PART. MOVE_PARTITION	`MD_PART.MOVE_PARTITION ([`*schema.*`|`*username,*`]` *partition_name*, *tablespace_name*`)`
MD_PART. SUBDIVIDE_ PARTITION	`MD_PART.SUBDIVIDE_PARTITION ([`*schema.*`|`*username,*`]` *partition_name*`)`
MD_PART. TRUNCATE_ PARTITION	`MD_PART.TRUNCATE_PARTITION ([`*schema.*`|`*username,*`]` *partition_name* `[,` *reuse_storage*`])`
MD_WEX. DROP_TARGET	`MD_WEX.DROP_TARGET ([`*schema.*`|`*username,*`]` *target_object_name*`)`
MD_WEX. EXTRACT	`MD_WEX.EXTRACT ([`*schema.*`|`*username,*`]` *source_sd_tablename*, `[`*schema.*`|`*username,*`]` *target_object_name*`)`
MD_WEX. RESET_GLOBALS	`MD_WEX.RESET_GLOBALS`
MD_WEX. SET_DIMENSION_ LIST	`MD_WEX.SET_DIMENSION_LIST (`*dimension_number* `[,` *dimension_number...*`])`

MD_WEX. SET_HHCODE_TYPE

`MD_WEX.SET_HHCODE_TYPE ('POINT'|'LINE')`

MD_WEX. SET_POLYGON_ WINDOW

`MD_WEX.SET_POLYGON_WINDOW (x1,y1,x2,y2,x3,y3 [, xn, yn...])`

MD_WEX. SET_PROXIMITY_ WINDOW

`MD_WEX.SET_PROXIMITY_WINDOW (dimension_value_1, dimension_value_2, [ dimension_value_n,...] radius)`

MD_WEX. SET_RANGE_ WINDOW

`MD_WEX.SET_RANGE_WINDOW (lower_window_boundary_1, upper_window_boundary_1 [, lower_window_boundary_n, upper_window_boundary_n...])`

MD_WEX. SET_SQL_FILTER

`MD_WEX.SET_SQL_FILTER (sql_filter)`

MD_WEX. SET_STORAGE_ CLAUSE

`MD_WEX.SET_STORAGE_CLAUSE (storage_clause_string)`

MD_WEX. SET_TARGET_TABLE SPACE

`MD_WEX.SET_TARGET_TABLESPACE (tablespace_name)`

MD_WEX. SET_TARGET_TYPE

`MD_WEX.SET_TARGET_TYPE ({'TABLE'|'VIEW'})`

MDVERIFY. CHECK_TABLE

`MDVERIFY.CHECK_TABLE ([schema.|username,] sd_tablename)`

MDVERIFY. CHECK_TABLES

`MDVERIFY.CHECK_TABLES`

User–Developed SLF Converter

mdsisl

```
int mdsisl ()
```

mdswhd

```
int mdswhd (char *slf_filename, char *username, char *password,
               char *sd_tablename, int *file_descriptor)
```

mdswhf

```
int mdswhf (char *table_control_filename, char *slf_filename,
                int *file_descriptor, char *bad_line,
                int *bad_line_number)
```

msdwsf

```
int msdwsf (int file_descriptor, int number_of_dimensions,
                int number_of_columns, int number_of_records,
                int max_number_of_errors,
                DIM *dimensional_value_description,
                COL *column_value_description, REC *record_values,
                ERR *error_description)
```

msdwst

```
int msdwst (int file_descriptor)
```

mdsssf

```
int mdsssf (char *slf_filename, int bindsize)
```

APPENDIX

C Messages and Codes

This appendix contains Oracle7 Spatial Data Option messages and codes, SDO–00000 – SDO–07512, along with a cause and action for each.

Oracle7 Spatial Data Option Messages and Codes

SDO–00000 successful completion

Cause An operation has completed normally, having met no exceptions.

Action No user action required.

SDO–00002 too many errors in '%s' – aborting

Cause The max errors allowed has been reached.

Action Fix errors and retry or increase the max errors allowed.

SDO–00200 could not allocate memory for initialization of internal heap manager

Cause Could not allocate the essential memory.

Action Increase amount of memory available or wait until more memory becomes available.

SDO–00201 failed to initialize message handler

Cause Possibly could not allocate the essential memory.

Action Increase amount of memory available or wait until more memory becomes available.

SDO–00202 out of memory while performing essential allocations

Cause Could not allocate the essential memory.

Action Increase amount of memory available or wait until more memory becomes available.

SDO–00203 invalid maximum bind array size

Cause The space allocated to do the necessary processing was not enough.

Action Increase the value of the bindsize.

SDO–00250 unable to open '%s' for processing

Cause Could not open the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00251 cannot close ’%s’

Cause Could not close the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00252 unable to read ’%s’

Cause Could not read the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00253 unable to write to ’%s’

Cause Could not write to the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00254 unable to seek in ’%s’

Cause Could not seek the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00255 unable to write to log file

Cause Could not write to log file.

Action Check the operating system message(s) accompanying this message.

SDO–00256 unable to delete ’%s’

Cause Could not delete the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00257 unable to rename ’%s’

Cause Could not rename the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00258 unable to open a temporary file for processing

Cause Could not open the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00259 cannot close a temporary file

Cause Could not close the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00260 unable to read from a temporary file

Cause Could not read the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00261 unable to write to a temporary file

Cause Could not write to the named file.

Action Check the operating system message(s) accompanying this message.

SDO–00262 unable to seek in a temporary file

Cause Could not seek the named file for processing.

Action Check the operating system message(s) accompanying this message.

SDO–00263 unable to create a unique temporary file name

Cause Could not create a unique file name.

Action Check the operating system message(s) accompanying this message.

SDO–00264 unable to delete a temporary file

Cause Could not delete a temporary file.

Action Check the operating system message(s) accompanying this message.

SDO–00265 unable to rename a temporary file

Cause Could not rename a temporary file.

Action Check the operating system message(s) accompanying this message.

SDO–00267 failed to translate environment variable in path

Cause The environment variable is invalid or not defined.

Action Verify that the environment variable is correct.

SDO–01000 invalid keyword or value for argument number %s

Cause Incorrect spelling of keyword, keyword is not valid, or incorrect type for argument's value.

Action Check the command syntax and your spelling, then retry.

SDO–01001 mandatory keyword is missing

Cause Keyword was left out on the command line.

Action Place mandatory keyword on command line.

SDO–01002 argument for the keyword '%s' is invalid

Cause Argument is either missing or out of bounds.

Action Verify argument is correct for the specified keyword.

SDO–01003 '%s' is not a valid keyword or missing value for the keyword

Cause Incorrect spelling of keyword, keyword is not valid, or value of the keyword is missing.

Action Check the command syntax and your spelling, then retry.

SDO–01004 invalid combination on command line

Cause A keyword was used that could not be combined with a previous keyword on the command line.

Action Check the command line syntax.

SDO–01005 error detected on the command line

Cause A command line parameter is improperly defined.

Action Verify that all command line parameters are properly specified.

SDO–01006 a userid was not specified on the command line

Cause A userid was specified on the command line.

Action A userid must be specified on the command line. (e.g.: USERID=sims/sims)

SDO–01007 failed to parse connect string

Cause The connect string specified could not be properly parsed.

Action Verify that the connect string is valid. If it is, document messages and contact Oracle Worldwide Support.

SDO–01008 a null userid was specified

Cause The user specified a null userid.

Action A userid must be entered.

SDO–01009 a null password was specified

Cause The user specified a null password.

Action A password must be entered.

SDO–01200 internal error: [%s]

Cause An error occurred during an internal match manipulation.

Action Document messages and contact Oracle Worldwide Support.

SDO–01201 internal error: [%s]

Cause An error occurred during an internal substring manipulation.

Action Document messages and contact Oracle Worldwide Support.

SDO–01202 internal error: [%s]

Cause An error occurred during an internal increment manipulation.

Action Document messages and contact Oracle Worldwide Support.

SDO–01203 internal error: [%s]

Cause An error occurred during an internal set id manipulation.

Action Document messages and contact Oracle Worldwide Support.

SDO–01204 internal error: [%s]

Cause An error occurred during an internal get id manipulation.

Action Document messages and contact Oracle Worldwide Support.

SDO–01500 invalid username/password; logon denied

Cause An invalid username or password was entered in an attempt to log on to Oracle.

Action Enter a valid username/password combination in the correct format.

SDO–01501 failed to log onto Oracle

Cause Log onto Oracle failed.

Action Verify that the Oracle connect string is valid.

SDO–01502 failed to connect as MDSYS

Cause The connection to the Oracle MDSYS account failed.

Action Verify that this account exists and was set up correctly.

SDO–01503 partition is locked

Cause Partition is currently being used.

Action Wait until the partition is free.

SDO–01507 partition is locked with an unknown status

Cause Status of partition is not a known status.

Action Verify that the Spatial Data Option data dictionary is correct.

SDO–01508 inserted a partition record after someone else did

Cause Multiple concurrent loads created same partition (only one wins).

Action Re–run the load process.

SDO–01600 failed to create table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to create a table.

SDO–01601 failed to create temporary table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to create a table.

SDO–01602 failed to drop table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to drop a table.

SDO–01603 failed to drop a temporary table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to drop a table.

SDO–01604 failed to truncate table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to truncate a table.

SDO–01605 failed to truncate temporary table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to truncate a table.

SDO–01606 real table does not exist

Cause An attempt was made to archive in an archive file and the truncated table does not exist.

Action Document messages and contact Oracle Worldwide Support.

SDO–01607 HHCODE column has no dimensional information

Cause The dimensional information for the HHCODE column was not found.

Action Verify that the dimensional information is correct.

SDO–01608 HHCODE column is missing dimensional information in ’%s’

Cause No dimensional information was entered.

Action Verify that the dimensional information is correct.

SDO–01609 spatial table name does not exist for specified owner

Cause Either the spatial table name is misspelled or it does not exist.

Action Verify that table exists and check spelling.

SDO–01610 null was passed in not null field ’%s’

Cause Null flag was set for a column that was set as a not null field.

Action Set flag for not null setting.

SDO–01611 dimensional value for ’%s’ is out of bounds

Cause The dimensional value is not in the range of the lower and upper bound of the dimension.

Action Verify that the data is correct or if the lower and upper bound of the dimension is correct.

SDO–01612 the column ’%s’ has an unsupported data type

Cause The column is defined with an unsupported data type.

Action Use another data type for this column.

SDO–01613 ’%s’ was not found as a non–partitioned spatial table

Cause Specified table does not exist or is not a non–partitioned spatial table.

Action Verify that specified table exists and is a non–partitioned spatial table.

SDO–01614 failed to find dimension information from temporary table

Cause Table may not exist.

Action Verify that table exists and is working properly.

SDO–01615 failed to find HHCODE column info. from temporary column table

Cause Temporary table is incorrect.

Action Verify that table exists and is working properly.

SDO–01616 the spatial table contains no HHCODE columns

Cause No HHCODE columns exist in specified spatial table.

Action Verify that the HHCODE column exists in specified spatial table.

SDO–01617 there is no partition key column for the specified table

Cause There was no HHCODE column specified as the partition key.

Action Alter the spatial table to have a partition key.

SDO–01618 dimensional information was not found for a HHCODE column

Cause Dimensional information does not exist for a HHCODE column

Action Verify the Spatial Data Option data dictionary for the corresponding spatial table.

SDO–01619 past the maximum level of subdivision

Cause The high water mark is being exceeded on a partition at the max subdivision point.

Action Alter the spatial table with a higher high water mark.

SDO–01620 failed to assign a transaction to specified rollback segment

Cause The specified rollback segment is invalid.

Action Verify that the rollback segment is correct.

SDO–01621 failed to create temporary view

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to drop a VIEW.

SDO–01622 failed to drop temporary view

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to drop a VIEW.

SDO–01700 failed to create base trigger on partition

Cause Failed to create trigger.

Action Check the Oracle message accompanying this message.

SDO–01701 failed to enable base trigger on partition

Cause Failed to enable trigger.

Action Check the Oracle messages accompanying this message.

SDO–01702 failed to select from table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to select from a table.

SDO–01703 failed to select from temporary table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to select from a table.

SDO–01704 failed to select information from view user_users

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to select from this view.

SDO–01705 failed to select from temporary column table

Cause Table may not exist.

Action Verify that table exists and is working properly.

SDO–01706 failed to select from temporary table

Cause Table may not exist.

Action Verify that table exists and is working properly.

SDO–01707 failed to insert into temporary table

Cause Check the Oracle messages accompanying this message.

Action Verify that you are able to insert into a table.

SDO–01708 failed to select information from view all_tables

Cause Either VIEW does not exist or you do not have access.

Action Verify that you have access to the VIEW and it is working properly.

SDO–01800 failed to select from sys.col$

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.col$.

SDO–01801 failed to select from sys.obj$

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.obj$.

SDO–01802 failed to select from sys.ts$

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.ts$.

SDO–01803 failed to select from sys.tab$

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.tab$.

SDO–01804 failed to select from sys.dba_tables

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.dba_tables.

SDO–01805 failed to select information from sys.dba_users

Cause Check the Oracle messages accompanying this message.

Action Verify that MDSYS can select from sys.dba_users.

SDO–01850 invalid HHCODE column name in '%s'

Cause HHCODE column name was not found for specified spatial table.

Action Verify HHCODE column exist in specified spatial table.

SDO–01851 no column information was found in view all_md_columns

Cause No non–HHCODE columns were found in specific spatial table.

Action Verify that the view ALL_MD_COLUMNS is correct.

SDO–01852 no class information was found in view all_md_tables

Cause The class information for the spatial table was not found.

Action Verify that the view ALL_MD_TABLES is correct.

SDO–01853 failed to select information from view all_md_tables

Cause Either view does not exist or is inconsistent.

Action Verify Spatial Data Option data dictionary

SDO–01854 HHCODE column name '%s' was not found for spatial table

Cause HHCODE column name was not found for specified spatial table.

Action Verify HHCODE column exist in specified spatial table.

SDO–01855 dimension name '%s' was not found for specified HHCODE column

Cause Either dimension name does not exist in specified spatial table or name is incorrectly spelled.

Action Verify that given dimension name is correct or verify that it exists in specified spatial table for the specified HHCODE column.

SDO–01856 column name '%s' was not found for specified spatial table

Cause Either column name does not exist in specified spatial table or name is incorrectly spelled.

Action Verify that given column name is correct and exists in specified spatial table.

SDO–01857 dimension name '%s' was not found for partition key column

Cause Either dimension name does not exist in specified spatial table or name is incorrectly spelled.

Action Verify that given dimension name is correct or verify that it exists in specified spatial table for the specified partition key column.

SDO–01858 not null field in '%s' not passed

Cause A column value was not passed that was set as a not null field.

Action Pass this column with a value other than null.

SDO–01859 data type for '%s' is invalid

Cause The COL structure data type description does not correspond to the data type describing the spatial table.

Action Verify that the data type describing the data is correct or the column in the spatial table has the correct data type.

SDO–01860 partition name sequence overflow

Cause Went through all possible values of the sequence.

Action Document messages and contact Oracle Worldwide Support.

SDO–01900 failed to select information from table md$col

Cause Either table does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01901 failed to select information from table md$dim

Cause Either table does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01902 failed to select information from table md$ptab

Cause Either table does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01903 failed to select information from view mdv$tab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01904 failed to select information from view mdv$col

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01905 failed to select information from view mdv$dim

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01906 failed to select information from view all_md_dimensions

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01907 failed to select information from view all_md_columns

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01908 failed to select information from view all_tab_columns

Cause Either view does not exist or you do not have access.

Action Verify that you have access to the view and it is working properly.

SDO–01909 failed to find dimension information from view all_md_dimensions

Cause View may not exist.

Action Verify that view exists and is working properly.

SDO–01910 failed to find column information from view all_md_columns

Cause View may not exist.

Action Verify that view exists and is working properly.

SDO–01911 failed to select information from table md$ler

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01912 failed to select information from table md$pts

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01913 failed to select information from table md$tab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01914 failed to insert information from table md$exc

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01915 failed to insert information from table md$ler

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01916 failed to insert information from table md$ptab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01917 failed to delete information from table md$exc

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01918 failed to delete information from table md$ler

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01919 failed to delete information from table md$ptab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01920 failed to update information in table md$ptab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01921 failed to update information from table md$ler

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01922 failed to update information from table md$pts

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01923 failed to update information from table md$tab

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01924 failed to select information from view dba_md_columns

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–01925 failed to select information from view dba_md_dimensions

Cause Either view does not exist or is inconsistent.

Action Verify the Spatial Data Option data dictionary.

SDO–05000 SLF header: %s does not match spatial data dictionary: %s

Cause The SLF file is built for either a partitioned spatial table or non–partitioned spatial table. The SLF file is being loaded into the wrong class.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05001 SLF header: %s does not match spatial data dictionary: %s

Cause Total number of HHCODE columns in SLF file does not match total in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05002 SLF header: %s does not match spatial data dictionary: %s

Cause Total number of dimensions for each HHCODE column in SLF file does not match total in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05003 SLF header: %s does not match spatial data dictionary: %s

Cause Total number of non–HHCODE columns in SLF file does not match total in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05004 SLF header: %s does not match spatial data dictionary: %s

Cause HHCODE column name in SLF file does not match name in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05005 SLF header: %s does not match spatial data dictionary: %s

Cause Number of dimensions of HHCODE column in SLF file does not match number in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05006 SLF header: %s does not match spatial data dictionary: %s

Cause Max level of dimensions of HHCODE column in SLF file does not match max level in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05007 SLF header: %s does not match spatial data dictionary: %s

Cause Size of HHCODE column in SLF file does not match size in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05008 SLF header: %s does not match spatial data dictionary: %s

Cause Null field of HHCODE column in SLF file does not match null field in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05009 SLF header: %s does not match spatial data dictionary: %s

Cause Dimension name of HHCODE column in SLF file does not match name in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05010 SLF header: %s does not match spatial data dictionary: %s

Cause Dimension number of HHCODE column in SLF file does not match number in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05011 SLF header: %s does not match spatial data dictionary: %s

Cause Dimension level of HHCODE column in SLF file does not match level in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05012 SLF header: %s does not match spatial data dictionary: %s

Cause Dimension lower bound of HHCODE column in SLF file does not match lower bound in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05013 SLF header: %s does not match spatial data dictionary: %s

Cause Dimension upper bound of a HHCODE column in SLF file does not match upper bound in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05014 SLF header: %s does not match spatial data dictionary: %s

Cause Column name in SLF file does not match name in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05015 SLF header: %s does not match spatial data dictionary: %s

Cause Null field of column in SLF file does not match null field in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05016 SLF header: %s does not match spatial data dictionary: %s

Cause Column type in SLF file does not match type in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05017 SLF header: %s does not match spatial data dictionary: %s

Cause Precision field of column in SLF file does not match precision field in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05018 SLF header: %s does not match spatial data dictionary: %s

Cause Scale field of column in SLF file does not match scale field in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05019 SLF header: %s does not match spatial data dictionary: %s

Cause Size of column in SLF file does not match size in spatial table. Either the spatial table definition was changed after the SLF file was created or the SLF file is being loaded into the wrong spatial table.

Action Verify the SLF file is being loaded into the correct spatial table or regenerate the SLF file.

SDO–05020 no data found in SLF file (just header information)

Cause During the conversion process the writing SLF record process died.

Action Recreate the SLF file verifying that the SLF data is written.

SDO–05200 line in '%s' was too complex to parse

Cause Data in control file exceeded max field limits.

Action Document messages and contact Oracle Worldwide Support.

SDO–05201 insufficient number of fields in '%s'

Cause There were fewer than 3 fields in the control file.

Action Check the control file for accuracy.

SDO–05202 did not find the expected line or keyword in '%s'

Cause Expected keyword or line was not found.

Action Verify that the specified keywords are in the correct order.

SDO–05203 an invalid number of partition key columns was specified in '%s'

Cause More than one partition key was specified in the named control file.

Action Verify only one HHCODE is specified as the partition key in the named control file.

SDO–05204 HHCODE column is missing a dimension sequence in '%s'

Cause When entering the dimensional information into the named control file, a dimensional sequence was left out.

Action Verify that the sequence values for the dimensions are correct.

SDO–05205 file type keyword not found in ’%s’

Cause The required file type keyword in named file is missing.

Action Verify that the control file has the correct information.

SDO–05206 unexpected keyword or value found in ’%s’

Cause There was more than one field found in the control file.

Action Check the control file for accuracy.

SDO–05207 ’%s’ is an invalid file type specified in control file

Cause The file type was spelled incorrectly. The only valid file types are BINARY and ASCII.

Action Verify that the file type specified in the control file is correct.

SDO–05208 format keyword not found in ’%s’

Cause The required format keyword in named file is missing.

Action Verify that the control file has the correct information.

SDO–05209 unexpected keyword or value found in ’%s’

Cause There were more than two fields found in the control file.

Action Check the control file for accuracy.

SDO–05210 ’%s’ is an invalid keyword specified in control file

Cause The keyword FIXED was not found in control file.

Action Verify that the keyword specified in the control file is correct.

SDO–05211 ’%s’ is an invalid record length specified for the data file

Cause Record length specified is invalid for given data file.

Action Verify that the record length specified is correct.

SDO–05212 ’%s’ is an invalid keyword specified in control file

Cause The only valid keywords are DIMENSION and COLUMN.

Action Verify that the file type specified in the control file is correct.

SDO–05213 ’%s’ is an invalid keyword specified in control file

Cause The missing keyword POSITION was not found in the control file.

Action Verify that the keyword specified in the control file is correct.

SDO–05214 invalid position value specified in ’%s’

Cause Field position was incorrectly specified.

Action Verify that the field positions specified are correct.

SDO–05215 ’%s’ is an invalid data type specified in control file

Cause Data type specified was spelled incorrectly. The only valid data types are INTEGER, SMALLINT, FLOAT, DOUBLE, BYTEINT, DATE, RAW, and CHAR.

Action Verify that the data type specified is correct.

SDO–05216 specified position length is not compatible with ’%s’

Cause Field position value was incorrectly specified.

Action Verify that the field positions specified are correct.

SDO–05217 ’%s’ is an invalid keyword specified in control file

Cause The missing keyword NULLIF was not found in control file.

Action Verify that the keyword specified in the control file is correct.

SDO–05218 ’%s’ is an invalid null indicator in control file

Cause Null indicator was incorrectly specified.

Action Verify that the null indicator is correct.

SDO–05219 no spatial information was found in ’%s’

Cause Spatial information was not found in the named file.

Action Enter the dimension information into the named file.

SDO–05220 no dimensional or column information was found in ’%s’

Cause No information was not found in the named file.

Action Enter the column and dimension information into the named file.

SDO–05221 ’%s’ is not the expected value in the data file

Cause Expected value was not found.

Action Verify that the specified file is correct.

SDO–05222 data conversion with column ’%s’ failed

Cause Either bad data was passed or incorrect format string was used.

Action Verify that the data and format string are correct.

SDO–05500 user aborted SD*Loader

Cause User aborted program.

Action No action required.

SDO–05501 SQL*Loader failed to load all records to the partition

Cause Some records were bad or rejected.

Action Check the log file and correct any errors indicated.

SDO–05502 SQL*Loader failed

Cause The SQL*Loader process died.

Action Check log file and correct any errors indicated.

SDO–07510 bad boolean value

Cause A bad boolean value was specified.

Action Verify that the value is TRUE or FALSE.

SDO–07511 failed to read user input from terminal

Cause An error was encountered while attempting to prompt the user for input from the terminal.

Action This is an internal error. Document messages and contact Oracle Worldwide Support.

Glossary

aggregation A process of grouping distinct data so that the aggregated dataset has a smaller number of data elements than the original dataset.

API See *Application Programming Interface.*

Application Programming Interface (API) A set of language calls allowing the user to interface with the product the API supports.

area An extent or region of dimensional space.

attribute Descriptive information characterizing a geographical feature such as a point, line, or area.

attribute data Non-dimensional data which provides additional descriptive information about multidimensional data, for example a class or feature such as a bridge or a road.

boundary The lower or upper extent of the range of a dimension in an HHCODE, expressed by a numeric value.

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.

coordinate system A reference system for the unique definition for the location of a point in *n*-dimensional space.

coordinates An *n*-tuple of values uniquely defining a point in an *n*-dimensional coordinate system.

data control file Provides information about the format of a spatial data file. It can be either ASCII or binary and must be a fixed length.

data dictionary A repository of information about data. A data dictionary stores relational information on all of the objects in a database.

data definition language (DDL) Statements that define, alter the structure of, and drop schema objects.

data transfer The process of saving spatial data in a form that can be moved from one computer system to another.

data manipulation language (DML) Statements that query or manipulate data in existing schema objects.

DDL See *data definition language.*

DML See *data manipulation language.*

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.

elevation A vertical distance above or below a reference surface. Terrain elevation is expressed with reference to mean sea level (MSL).

extent A rectangle bounding a map, the size of which is determined by the minimum and maximum map coordinates.

feature An object in a spatial database with a distinct set of characteristics.

format conversion The conversion of data from one format to another.

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.

generalization The process removing detail to reveal structural components and general shapes.

geographically referenced data See *spatiotemporal data.*

georeferenced data See *spatiotemporal data.*

GIS See *geographical information system*.

geographical information system A computerized database management system used for the capture, conversion, storage, retrieval, analysis, and display of spatial data.

hexahedron A six-faced solid figure.

HHCODE A datatype representing the intersection point of multiple dimensions. It encodes these multiple dimensions into a unique, linear value. The HHCODEs are stored in a single column of a Spatial Data Option table.

high water mark Expressed in number of records and associated with the Spatial Data Option 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 Data Option table.

homogeneous Spatial data of one feature type such as points, lines, or regions.

hyperspatial data In mathematics, any space comprised of 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.

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.

non-partitioned table Structure used if the table will contain an amount of data manageable by a single table. A non-partitioned table is a single table which has at least one HHCODE column, never subdivides, and grows to a maximum size as defined by the MAXEXTENTS parameter. If the table does not have an HHCODE column defined, it is considered an Oracle7 table and not a spatial table.

overlap That part of a map sheet that is duplicated on another map sheet.

partition The spatial table that is part of a partitioned table.

partitioned table The spatial logical table structure that contains one or more partitions. Use partitioned tables if the table will contain a large amount of 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 which is used to dimensionally partition the data. One HHCODE datatype column must be identified as the partition key for the table to be registered as partitionable in the Spatial Data Option data dictionary. There can only be one partition key per spatial table.

polygon A class of spatial objects having nonzero area and perimeter, and representing a closed boundary region of uniform characteristics.

profile See *cross section*.

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. Oracle is a relational database management system. Other types of database systems are called hierarchical or network database systems.

reserved word Part of a computer language definition set that cannot be used as a user-defined variable or constant name. Synonym for *keyword*.

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 Reads external datafiles and converts them into SLF. SLF is the format needed to bulk load data into spatial tables.

SD*Loader Sorts and loads SLF file data into spatial tables and creates and reorganizes table partitions as required.

SD*SQL A set of procedures and functions for handling dimensional data stored in an Oracle7 database.

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 Data Option data dictionary An extension of the Oracle7 data dictionary. It keeps track of the number of partitions created in a spatial table. The Spatial Data Option data dictionary is owned by MDSYS.

set function An operation that acts on an ensemble of values or sets instead of on a single value. Operations on spatial databases are frequently represented by set operations such as UNION, INTERSECTION, and SUBTRACTION.

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 database A database containing information indexed by location.

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

spatial data model A model of how objects are located on a spatial context.

Spatial Load Format (SLF) The format used to load data into spatial tables. SLF files are designed to be temporary and should be deleted once the data is loaded. SLF files are not portable between different architectures.

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*.

table control file A text file which describes the column names, datatypes, and dimensional information of a particular Spatial Data Option table.

temporal database A database containing information indexed by time.

temporal key Represents the translocation, through time, of an *n*-dimensional object.

temporal grid A data structure providing a framework for the storage of temporal data. The addresses of cells in a temporal grid are defined using the time domain.

tiling The process of creating a polygon using HHCODEs.

very large database (VLDB) A database of very large size.

union A set operation. The outcome of the union of sets is a set with all the elements of all input sets.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Reader's Comment Form

Oracle7 Spatial Data Option™ Reference and Administrator's Guide
Part No. A43694–1

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 topic, chapter, and page number below:

Please send your comments to:

Documentation Manager Government Products Division
Oracle Corporation
500 Oracle Parkway Box 659204
Redwood Shores, California 94065
Phone: 1.415.506.2503 FAX: 1.415.506.7408

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.