Oracle9*i*

Data Cartridge Developer's Guide

Release 2 (9.2)

March 2002

Part No. A96595-01

Oracle9*i* Data Cartridge Developer's Guide, Release 2 (9.2)

Part No. A96595-01

Primary Author: William Gietz

Graphic Designer: Valarie Moore

Contributing Author: C. Dupree

Contributors: A. Yoaz

Contents

Send Us Your Comments

Oracle9*i* Data Cartridge Developer's Guide, Release 2 (9.2)

Part No. A96595-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. 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?

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

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

Preface

The *Oracle9i Data Cartridge Developer's Guide* describes how to build and use *data cartridges* to create custom extensions to the Oracle server's indexing and optimizing capabilities.

This preface contains these topics:

Audience

Oracle9i Data Cartridge Developer's Guide is intended for developers who want to learn how to build and use data cartridges to customize the indexing and optimizing functionality of the Oracle server to suit exotic kinds of data.

To use this document, you need to be familiar with using Oracle and should have a background in an Oracle-supported programming language such as C, C++, or Java used to write external procedures.

Organization

This document contains:

Introduction

Chapter 1, "What Is a Data Cartridge?" and Chapter 2, "Roadmap to Building a Data Cartridge" provide basic information and lay the groundwork for a comprehensive example used throughout the later chapters.

Building Data Cartridges

Chapters 3 through 9 lay out the components that go into building a data cartridge.

Advanced Topics

Chapters 10, 11, and 12 discuss design considerations, user-defined aggregate functions, and pipelined and parallel table functions.

Scenarios and Examples

Chapters 13 and 14 elaborate the Power Utility example scenario developed in Chapter 3, to illustrate extensible indexing. Appendix A supplements Chapter 12 with two extended examples of how to implement a table function, in C and in Java.

Reference

Chapters 15 through 20 provide reference information on data cartridge-specific APIs.

Related Documentation

For information on cartridge services using C, see the chapter on cartridge services in the *Oracle Call Interface Programmer's Guide*.

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
`[ ]`	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
`{ }`	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

`|`	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

`...`	Horizontal ellipsis points indicate either:
	■ That we have omitted parts of the code that are not directly related to the example
	■ That you can repeat a portion of the code
		`CREATE TABLE ... AS `*`subquery`*`;`
		
		`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`------------------------------------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected.`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

Convention	Meaning	Example

`Italics`	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME = `*`database_name`*

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.

File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32 is the same as`
		`C:\WINNT\SYSTEM32`

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"`
		`C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, dept)`

HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_ BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ C:\orant for Windows NT ▪ C:\orawin98 for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is C:\oracle. If you install the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is C:\oracle\ora*nn*, where *nn* is the latest release number. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Started for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*\ORACLE_ HOME\rdbms\admin directory.

Documentation Accessibility

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

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

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What’s New in Data Cartridges?

This chapter describes features relating to data cartridges that are new in Oracle9*i* Release 2 (9.2).

Oracle9*i* New Features for Data Cartridges

Note About the New Features

Oracle9*i* adds partitioned, local domain indexes and a number of other new features of interest to developers of data cartridges. To support local domain indexes in particular, both the Extensible Indexing interface (the `ODCIIndex*` routines) and the Extensible Optimizer interface (the `ODCIStats*` routines) have undergone some changes: new routines have been added, and most of the existing routines have acquired an additional parameter of the new system-defined type `ODCIEnv`.

Existing code that uses the Oracle8*i* version of the `ODCIIndex*` and `ODCIStats*` interfaces does not need to be changed to run under Oracle9*i* unless you want to use new features that require the new interfaces. Some new features—for example, user-defined aggregate functions and table functions—do not require the new interfaces, but local domain indexes do.

To implement local domain indexes, you must adopt the new `ODCIIndex*` interface. This means that, if you have existing code based on the Oracle8*i* version of the interface, you must migrate that code in its entirety to the Oracle9*i* version of the interfaces: you cannot simply supplement it with calls to a few new functions and leave the rest of the code unchanged. And, if you do adopt the Oracle9*i* `ODCIIndex*` interface, you can use only the Oracle9*i* `ODCIStats*` interface with it: you cannot use the Oracle8*i* version.

Oracle supports both the Oracle8*i* and Oracle9*i* versions of the `ODCIIndex*` and `ODCIStats*` interfaces. The string you return in the `ODCIObjectList` parameter of `ODCIGetInterfaces` tells the system which version your code implements. (Details on using this routine are explained in the reference chapters on the Extensible Indexing and Extensible Optimizer interfaces later in this book.)

To sum up: If you have Oracle8*i* code, that code will still work. To continue to use the Oracle8*i* interface, do not implement Oracle9*i* versions of any of the `ODCIIndex*` or `ODCIStats*` routines.

List of New Features in Oracle9*i* Release 2 (9.2)

- **Restriction removed**

  It is now possible to create and rebuild domain indexes and local domain index partitions in parallel.

- **Table function enhancement**

  A table function can now return the generic collection type SYS.AnyDataSet.

See Also: "Pipelined Table Functions" on page 12-7

List of New Features in Oracle9*i* Release 1 (9.0.1)

- **Local domain indexes**

  Discrete domain indexes, called local domain indexes, can be built on the partitions of a range-partitioned table. Local domain indexes are equipartitioned with the underlying table: all keys refer only to rows stored in the local domain index's corresponding table partition.

  Note: The partitioning scheme provided with Oracle9*i* makes it possible to create local domain indexes. This scheme may be changed in future releases of Oracle to be more transparent. Domain-index creators who adopt the present scheme should be aware that later on they may need to change to a new syntax and semantics for partitioning of indexes.

  See Also: Chapter 7, "g Building Domain Indexes"

- **Collection of user-defined statistics for partitioned tables**

  The extensible optimizer supports collection of user-defined statistics—partition level and aggregate—for partitioned tables.

  See Also: Chapter 8, "Query Optimization"

- **New package DBMS_ODCI**

  New package `DBMS_ODCI` contains a utility to help better estimate the cost of user-defined functions.

  See Also: Chapter 8, "Query Optimization"

- **Support for a NULL association of a statistics type**

  Instances of an indextype or object inherit an association of a statistics type. Now you can replace this with a `NULL` association for occasions when the benefit of using a better plan may not outweigh the added cost of compiling the cost or selectivity functions implemented by the statistics type.

  See Also: Chapter 8, "Query Optimization"

- **User-defined aggregate functions**

  Custom aggregate functions can be defined for working with complex data.

  See Also:

  - Chapter 11, "User-Defined Aggregate Functions"
  - Chapter 19, "Reference: User-Defined Aggregates Interface"

- **Table functions**

  Table functions can be used in the `FROM` clause of a query to return a collection (either a nested table or a varray) of rows as output. A table function can use parallel execution, and result rows from can be pipelined—that is, iteratively returned.

  See Also:

  - Chapter 12, "Pipelined and Parallel Table Functions"
  - Chapter 20, "Reference: Pipelined and Parallel Table Functions"

- **Generic and transient datatypes**

  External procedures can be given fields or parameters of a generic type that can contain values of any scalar or user-defined type, making it unnecessary to implement multiple versions of the same external procedure just to handle multiple datatypes.

  See Also: Chapter 12, the section "Transient and Generic Types"

Part I

Introduction

1 What Is a Data Cartridge?

In addition to the efficient and secure management of data ordered under the relational model, Oracle now also provides support for data organized under the object model. Object types and other features such as large objects (`LOBs`), external procedures, extensible indexing and query optimization can be used to build powerful, reusable server-based components called *data cartridges*.

This chapter introduces the following introductory information about data cartridges:

- What Are Data Cartridges?
- Why Build Data Cartridges?
- Extending the Server—Services and Interfaces
- Extensibility Services
- Extensibility Interfaces
- Cartridges as Software Components

What Are Data Cartridges?

Within the framework of the Oracle Extensibility Architecture, *data cartridges* are the mechanism for extending the capabilities of the Oracle server. What does this mean?

First, Oracle lets you capture the business logic and processes associated with domain-specific data in user-defined datatypes. In some cases, where the data cartridge provides new behavior without needing new attributes, the vehicle of implementation may be packages rather than formal types. Once you have defined these types using either of these approaches, Oracle enables you to determine the manner in which the server interprets, stores, retrieves, and indexes the data. Ultimately, *data cartridges* are the means to package this functionality as software components that can then be plugged into a server to extend its capabilities into a new domain.

This is all possible because the database has itself been made *extensible*. That is, you can now customize the indexing and query optimization mechanisms of the database management system for user-defined business objects and rich types. Where the native implementation of indexing and query optimization service could be improved for some specialized processing you require, you can provide your own implementations of these services. You use the **extensibility interfaces** to register your implementations with the server. Registering them causes the server to use your implementations instead its own when doing your specialized processing.

This is all possible because the database has itself been made *extensible*. That is, you can now customize the database management system so that it treats user-defined business objects and rich types on a par with native types with regard to server mechanisms such as indexing and query optimization.

The extensibility interfaces consist of functions that the server calls as needed to execute pieces of the custom indexing or optimizing behavior implemented for a data cartridge. The interfaces are defined by Oracle; you, the cartridge developer, must actually implement the functions (also frequently called *interfaces*) to embody the specialized behavior you require. In general, you implement the functions as static methods of an object type. An object type that implements the extensible indexing interface is called an **indextype**; an object type that implements the extensible optimizing interface is called a **statistics type**.

The key characteristics of data cartridges are as follows:

- *Data cartridges are server-based*. Their constituents reside on the server or are accessed from the server. The bulk of processing for data cartridges occurs at the server, or is dispatched from the server in the form of an external procedure.

- *Data cartridges extend the server.* They define new types and behavior to provide componentized, solution-oriented capabilities previously unavailable in the server. Users of data cartridges can freely use the new types in their application to get the new behavior. Having loaded an Image data cartridge, the user can define a table `Person` with a column `Photo` of type `Image`.
- *Data cartridges are integrated with the server.* The extensions made to the server by defining new types are integrated with the server engine so that the optimizer, query parser, indexer and other server mechanisms recognize and respond to the extensions. The Oracle Extensibility Framework defines a set of interfaces that enable data cartridges to integrate with the components of the server engine. For example, the interface to the indexing engine allows for domain-specific indexing. Optimizer interfaces similarly allow data cartridges to define domain-specific ways of assessing the CPU and I/O cost of accessing cartridge data.
- *Data cartridges are packaged.* A data cartridge is installed as a unit. Once installed, the data cartridge handles all access issues arising out of the possibility that its target users might be in different schemas, have different privileges and so on.

Why Build Data Cartridges?

The Need to Handle Complex Data Objects

Over the years, virtually every industry has evolved sophisticated models to handle complex data objects that make up the essence of their business. By *data objects*, we mean both the structures that relate different units of information and the operations that are performed on them.

The simple names given these data objects often conceal considerable complexity of the expertise they embody. For example, the banking industry has many different types of *bank accounts*. Each bank account has customer demographic information, balance information, transaction information, and rules that embody its behavior (deposit, withdrawal, interest accrual, and so forth).

As the following sections describe, data cartridges allow you to leverage this expertise by encapsulating this business logic in software components that integrate with the Oracle server. The notion of adding logic to data in a database has been available for some time by way of stored procedures. With the addition of object-relational extensions, the Oracle server can now be enhanced by application programmers and independent software vendors to support a new generation of data types, processes, and logic in order to model business objects.

The Need to Operate on Complex and Multimedia Datatypes

At the same time as business models have led to the development of increasingly complex data objects, the revolution in information technology has made it necessary to work with new kinds of data: satellites images, X-rays, animal sounds, seismic vibrations, chemical models — all these complex and multimedia datatypes are now forms of information that have to be stored and retrieved, queried and analyzed.

Today's web-based applications routinely include many different kinds of complex data. The ability to extend the database to include application-specific data types as well as the business logic associated with these types requires a new class of networked, content-rich, multitiered, distributed applications. As the following sections describe, data cartridges allow you to meet this need by combining scalar and unstructured datatypes in domain-specific components. You can further combine these components to provide both horizontal (across industries) and vertical (niche specific) functionality.

Data Cartridge Domains

The complexity of data objects, which may entail the need to handle specialized data, gives rise to application *domains*. Put another way: a data cartridge is typically domain-specific. Domains are characterized by content and scope.

In terms of *content*, a data cartridge can accommodate either scalar data or complex and multimedia forms of data. Scalar data is data that can be modeled using native SQL types such as `INTEGER`, `NUMBER`, or `CHAR`. Complex forms of data include matrixes, temperature and magnetic grids, and compound documents. Multimedia types include video, voice, and image data.

In terms of *scope*, a data cartridge can have broad horizontal (cross-industry) coverage or it can be specialized for a specific type of business. For example, a data cartridge for general storage and retrieval of textual data is cross-industry in scope, whereas a data cartridge for the storage and retrieval of legal documents for litigation support is industry-specific.

Table 1–1 shows a way of classifying data cartridge domains according to their content (type of data) and scope (cross-industry or industry-specific), with some examples.

Oracle enables you to utilize built-in scalar datatypes to construct more complex user-defined types. The Object-Relational Database Management System has evolved to the point that Oracle now provides foundational cartridges that package multimedia and complex data which can be used as bases for applications across many different industries:

Table 1–1 Data Cartridge Domains by Content and Scope

Content	Scope: Cross-Industry Uses	Scope: Industry-Specific Extensions
Scalar Data	Statistical conversion	Financial and Petroleum
Multimedia and Complex Data	Text	Image
Audio/Video	Spatial	Legal
Medical	Broadcasting	Utilities

Table 1–2 Oracle Cartridges as Bases for Development

Cartridge	Database Model	Behavior
Time	Ordered list of tuples	Compute Rolling Averages, Compare Time Periods, Construct Calendars...
Text	Tokenized serial byte stream	Display, Compress, Reformat, Index...
Image	Structured large object	Compress, Crop, Scale, Rotate, Reformat...
Spatial	Geometric objects such as points, lines, polygons	Project, Rotate, Transform, Map...
Video	Structured large object of serial (dynamic) image data	Compress, Play, Rewind, Pause...

Following from this, you can see that another way of viewing the relationship of cartridges to domains is to view basic multimedia datatypes as forming a foundation that can be extended in specific ways by specific industries. For example, Table 1–3 shows cartridges that could be built for medical applications:

Table 1–3 Medicine-Specific Extensions to Basic Cartridges

Text	Image	Audio	Video	Spatial
Records	MRI	Heartbeat	Teaching	Demographic Analysis

A cartridge providing basic services may be deployed across many industries, as a text cartridge may be utilized within both law and medicine. A cartridge can also leverage domain expertise across an industry, as an image cartridge may provide basic functionality for both X-rays and Sonar within medicine. These cartridges can in turn be further extended for more specialized vertical applications. For instance, any of the cartridges mentioned previously could be specialized by being extended by other cartridges:

Table 1–4 Examples of Extensions to a Basic Cartridge

Image
Image -> MRI -> Brain MRI -> Neonatal Brain MRI

In other words, you can develop a cartridge for both horizontal and vertical market penetration.

In summary: data cartridges allow you to define new datatypes and behavior which can then provide, in component form, solution-oriented capabilities previously unavailable in the server. In some cases, where the data cartridge provides new behavior without needing new attributes, the data cartridge may provide PL/SQL packages but not new datatype definitions. Users of data cartridges can freely use the new datatypes in their application to take advantage of the new behavior. For example, after an image data cartridge is installed, you can define a table called *Person* with a *Photo* column of type *Image*

Extending the Server—Services and Interfaces

The Oracle server provides services for basic data storage, query processing, optimization, and indexing. Various applications use these services to access database capabilities. However, data cartridges have specialized needs because they incorporate domain-specific data. To accommodate these specialized applications, the basic services have been made *extensible*.

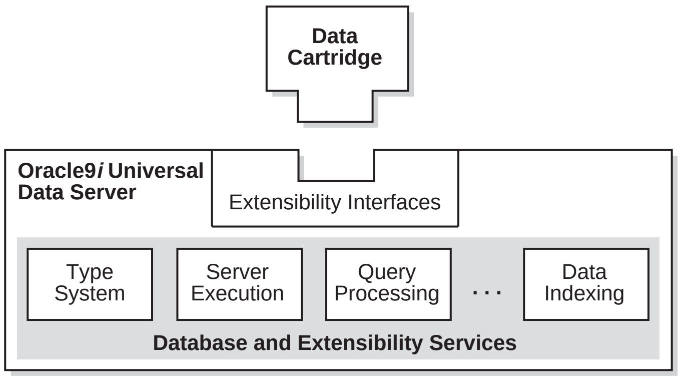
That is, where some aspects of a standard Oracle service are not adequate for the processing a data cartridge requires, you as the data cartridge developer can provide services that are specially tuned to your cartridge. Every data cartridge can provide its own implementations of these services. These specialized implementations are registered with the server using the Oracle extensibility interfaces.

For example, suppose you want to build a spatial data cartridge for geographical information systems (GIS) applications. In this case, you may need to implement routines that create a spatial index, insert an entry into the index, update the index, delete from the index, and perform any other required operations. To do this you would register your implementations with the Oracle server using extensible indexing interface, and then the server will invoke your implementation every time indexing operations were needed for spatial data. In effect, you extend the indexing service of the server.

Extensibility Services

Figure 1–1 shows the standard services implemented by the Oracle server. This section describes some of these services, not to provide exhaustive descriptions but to highlight major Oracle capabilities as they relate to data cartridge development.

Figure 1–1 Oracle Services



Extensible Type System

The Oracle universal data server provides both native and extensible type system services. Historically, mainstream applications have focused on accessing and modifying corporate data that is stored in tables composed of native SQL datatypes, such as `INTEGER`, `NUMBER`, `DATE`, and `CHAR`. Oracle adds support for new types, including:

- User-defined objects
- Collections:
  - VARRAY (varying length array)
  - Multi-set (nested table)
- REF (relationship)

- Internal large object types:
  - `BLOB` (binary large object)
  - `CLOB` (character large object)
- `BFILE` (external file)

This section discusses these new types.

Object Types

An *object type* differs from native SQL datatypes in that it is user-defined and it specifies both the underlying persistent data (attributes) and the related behaviors (methods). Object types are used to extend the modeling capabilities provided by the native datatypes. You can use object types to make better models of complex entities in the real world by binding data attributes to semantic behaviors.

An object type can have one or more *attributes*. Each attribute has a name and a type. The type of an attribute can be a native SQL type, a LOB, a collection, another object type, or a REF type. The syntax for defining object types is discussed in Chapter 3.

A *method* is a procedure or a function that is part of an object type definition. Methods can access and manipulate attributes of the related object type. Methods can run within the execution environment of the Oracle server. Methods can also be dispatched outside the server as part of the extensible server execution environment.

Collection Types

Collections are SQL datatypes that contain multiple elements. Each element, or value, for a collection is the same datatype. In Oracle, collections of complex types can be VARRAYs or nested tables.

A *VARRAY* contains a variable number of ordered elements. The VARRAY datatype can be used for a column of a table or an attribute of an object type. The element type of a VARRAY can be either a native datatype, such as NUMBER, or an object type.

A *nested table* can be created using Oracle SQL to provide the semantics of an unordered collection. As with a VARRAY, a nested table can be used to define a column of a table or an attribute of an object type.

Relationship Types (REF)

If you create an object table in Oracle, you can obtain a *reference* that acts as a database pointer to an associated row object. References are important for navigating among object instances, particularly in client-side applications.

The `REF` operator obtains a reference to a row object. Because `REFs` rely on the underlying object identity, you can use `REF` only with an object stored as a row in an object table or objects composed from an object view.

For further information about the `REF` operator and examples of its use, see the chapter on object types in the *PL/SQL User's Guide and Reference*.

Large Objects

Oracle provides *large object* (`LOB`) types to handle the storage demands of images, video clips, documents, and other forms of non-structured data. For an extensive coverage of Large Objects, please see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*. Large objects are stored in a way that optimizes space utilization and provides efficient access. Large objects are composed of locators and the related binary or character data. The `LOB` locators are stored in-line with other table columns and, for internal `LOBs` (`BLOB`, `CLOB`, and `NCLOB`), the data can be in a separate database storage area. For external `LOBs` (`BFILE`), the data is stored outside the database tablespaces in operating system files. A table can contain multiple `LOB` columns (in contrast to the limit of one `LONG RAW` column for each table). Each `LOB` column can be stored in a separate tablespace, and even on different secondary storage devices.

Oracle SQL data definition language (DDL) extensions let you create, modify, and delete tables and object types that contain large objects (`LOBs`). The Oracle SQL data manipulation language (DML) includes statements to insert and delete complete `LOBs`. There is also an extensive set of statements for piece-wise reading, writing, and manipulating of `LOBs` with PL/SQL and the Oracle Call Interface (OCI) software.

For internal `LOB` types, both the locators and related data participate fully in the transactional model of the Oracle server. The data for `BFILEs` does not participate in transactions; however, `BFILE` locators are fully supported by Oracle server transactions. For more information about `LOBs` and transactions, see the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

Unlike scalar quantities, a `LOB` value cannot be indexed using built-in indexing schemes. However, you can use the various `LOB` APIs to build modules, including methods of object types, to access and manipulate `LOB` content. The extensible indexing framework lets you define the semantics of data residing in `LOBs` and manipulate the data using these semantics.

Oracle provides you a variety of interfaces and environments to access and manipulate `LOBs`, which are described in great detail in *Oracle9i Application Developer's Guide - Large Objects (LOBs).*The use of `LOBs` to store and manipulate binary and character data to represent your domain is discussed Chapter 6, "Working with Multimedia Datatypes".

Extensible Server Execution Environment

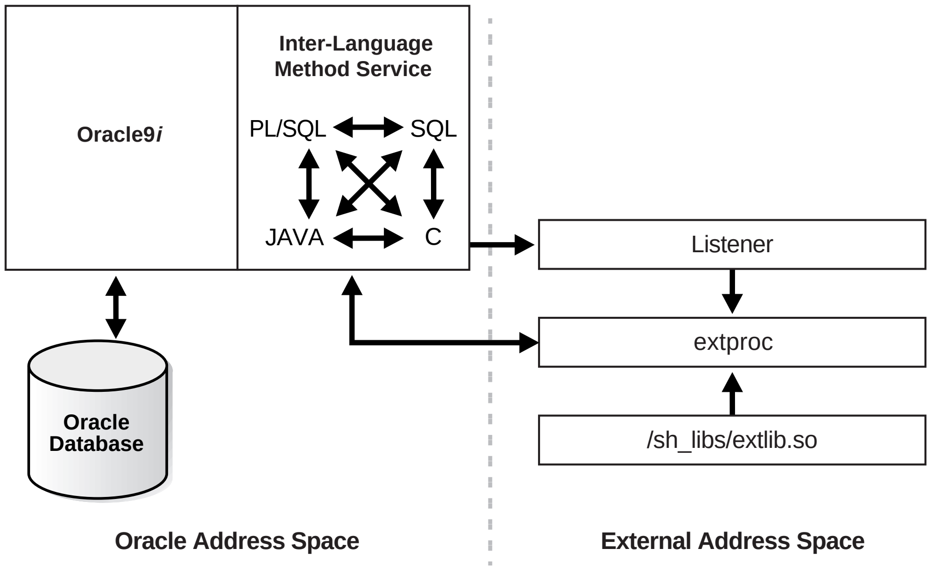
The Oracle type system decouples the implementation of a member method for an object type from the specification of the method. Components of an Oracle data cartridge can be implemented using any of the popular programming languages. In Oracle, methods, functions, and procedures can be developed using PL/SQL, external C language routines, or Java. Thus, the database server runtime environment can be extended by user-defined methods, functions, and procedures.

In Oracle, Java offers data cartridge developers a powerful implementation choice for data cartridge behavior. In addition, PL/SQL offers a data cartridge developer a powerful procedural language that supports all the object extensions for SQL. With PL/SQL, program logic can execute on the server and perform traditional procedural language operations such as loops, if-then-else clauses, and array access.

While PL/SQL and Java are powerful, certain computation-intensive operations such as a Fast Fourier Transform or an image format conversion are handled more efficiently by C programs. With the Oracle Server, you can call C language programs from the server. Such programs are executed as in a separate address space than the server. This ensures that the database server is insulated from any program failures that might occur in external procedures and, under no circumstances, can an Oracle database be corrupted by such failures.

With certain reasonable restrictions, external procedures can *call back* to the Oracle Server using OCI. Callbacks are particularly useful for processing `LOBs`. For example, by using callbacks an external procedure can perform piece-wise reads or writes of `LOBs` stored in the database. External procedures can also use callbacks to manipulate domain indexes stored as Index-Organized Tables in the database.

Figure 1–2 External Program Executing in Separate Address Space



Extensible Indexing

Typical database management systems support a few types of access methods (B+Trees, Hash Indexes) on a limited set of data types (numbers, strings, and so on). For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. In recent years, however, databases are being used to store different types of data such as text, spatial, image, video and audio that require content-based retrieval. This raises the need for indexing complex data types and also specialized indexing techniques.

Complex data types have application-specific formats, indexing requirements, and selection predicates. For example, there are many different means of document encoding (ODA, XML, plain text) and information retrieval techniques (keyword, full-text boolean, similarity, probabilistic, and so on). Similarly, R-trees are an efficient method of indexing spatial data. No database server can be built with support for all possible kinds of complex data and indexing. Oracle's solution is to build an extensible server which lets you define new index types as required.

Such user-defined indexes are called **domain indexes** because they index data in an application-specific domain. The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The physical index can be stored in the Oracle database as tables or externally as a file.

A domain index is a schema object. It is created, managed, and accessed by routines implemented as methods of a user-defined object type, called an **indextype**. The routines that an indextype must implement, and the kinds of things that the routines must do, are described in this document. Actual implementation of the routines is specific to an application and so must be done by the cartridge developer. Once a new indextype is implemented by a data cartridge, Oracle uses the indextype's specialized implementation of these routines for the data cartridge instead of the indexing implementation native to the server.

With extensible indexing, the application

- Defines the structure of the domain index,
- Stores the index data either inside or outside the Oracle database, and
- Manages, retrieves and uses the index data to evaluate user queries.

When the database system handles the physical storage of domain indexes, data cartridges

- Define the format and content of an index. This enables cartridges to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures. Note that this is a significant departure from the medicine indexing features provided for simple SQL data types. Also, since an index is modeled as a collection of tuples, in-place updating is directly supported.
- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

Typical relational and object-relational database management systems do not support extensible indexing. Consequently, many applications maintain file-based indexes for complex data residing in relational database tables. A considerable amount of code and effort is required to maintain consistency between external indexes and the related relational data, support compound queries (involving tabular values and external indexes), and to manage a system (backup, recovery, allocate storage, and so on) with multiple forms of persistent storage (files and databases). By supporting extensible indexes, the Oracle Server significantly reduces the level of effort needed to develop solutions involving high-performance access to complex datatypes.

Extensible Optimizer

The extensible optimizer functionality allows authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions. This information is used by the optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information; the rule-based optimizer is unchanged.

The optimizer generates an execution plan for a SQL statement. An execution plan includes an access method for each table in the `FROM` clause, and an ordering (called the join order) of the tables in the `FROM` clause. System-defined access methods include indexes, hash clusters, and table scans. The optimizer chooses a plan by generating a set of join orders or permutations, computing the cost of each, and selecting the one with the lowest cost. For each table in the join order, the optimizer computes the cost of each possible access method and join method and chooses the one with the lowest cost. The cost of the join order is the sum of the access method and join method costs. The costs are calculated using algorithms which together compose the cost model. A cost model can include varying level of detail about the physical environment in which the query is executed. Our present cost model includes only the number of disk accesses with minor adjustments to compensate for the lack of detail. The optimizer uses statistics about the objects referenced in the query to compute the costs. The statistics are gathered using the `ANALYZE` command. The optimizer uses these statistics to calculate cost and selectivity. The selectivity of a predicate is the fraction of rows in a table that will be chosen by the predicate. It is a number between 0 and 100 (expressed as percentage).

Extensible indexing functionality allows users to define new operators, index types, and domain indexes. For such user-defined operators and domain indexes, the extensible optimizer functionality will allow users to control the three main components used by the optimizer to select an execution plan: *statistics, selectivity,* and *cost*.

Note: Oracle Corporation recommends that you use the `DBMS_STATS` package instead of the SQL `ANALYZE` statement to collect optimizer statistics. In a future release, functionality to collect optimizer statistics will be removed from `ANALYZE`.

See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about `DBMS_STATS`.

Extensibility Interfaces

Extensibility interfaces fall into the following classes:

- DBMS interfaces
- Cartridge basic service interfaces
- Data cartridge interfaces

DBMS Interfaces

The DBMS interfaces are the simplest kind of extensibility services. DBMS interfaces are made available through extensions to SQL or to the Oracle Call Interface (OCI). For example, the extensible type manager utilizes the CREATE TYPE syntax in SQL. Similarly, extensible indexing uses DDL and DML support for specifying and manipulating indexes.

Cartridge Basic Service Interfaces

Generic interfaces provide basic services like memory management, context management, internationalization, and cartridge-specific management. These cartridge basic interface services are used by data cartridges to implement behavior for new datatypes in the context of the server's execution environment. These services provide helper routines that make it easy for data cartridge developers to write robust, portable server-side methods.

Data Cartridge Interfaces

Sometimes the DBMS needs to call the data cartridge functions for implementations provided by the data cartridge developer. So, for user-defined indexing, the DBMS must use the implementation of the index interface whenever an index search or fetch operation is performed. For user-defined query optimization, the query optimizer must call functions implemented by the data cartridge to compute cost of user-defined operators or functions.

These standard data cartridge functions are similar to callback functions that the DBMS can invoke. In the future, data cartridge interfaces will be made available to enable the data cartridge to include the specifications for such functions.

Cartridges as Software Components

The accumulated expertise that underlies a set of data objects comprises a knowledge base that can be marketed as a standalone cartridge, or as a cartridge that could be extended in different ways by different users. But how does one achieve this? The data and rules that apply to the software components are often spread across many different applications. With data cartridges, you gather the definition and rules together for use throughout the data processing environment. Packaging domain-specific component expertise in a data cartridge allows the cartridge to access the corporate information repository and add both organizational and operational value to the data. Such *software components* are applications that can be "plugged" into other software components, and which are themselves "pluggable".

Their constituents reside at the server or are accessed from the server. Most processing for data cartridges occurs at the server or is dispatched from the server in the form of an external procedure.

The Structure of a Data Cartridge

A data cartridge generally defines one or more object types. Object types from this and potentially other data cartridges can provide users with new or extended capabilities conveniently packaged. A data cartridge includes both the definition of object types and the code that implements their capabilities. A data cartridge can be used as the foundation for the definition of other data cartridges.

Each object type includes two components. The order in which these components are made available to the server (that is, the order in which they are defined) is important. The major components include:

- Object type specification
- Object type body code

In addition, a data cartridge may use the extended server execution environment. The use of external procedures involves two additional components:

- External library linkage specification
- External library code

Simple data cartridges consist of these components, which are described in this section. More complex data cartridges will use the extensibility services and interfaces (see Chapter 9, "Using Cartridge Services"). Complex Data Cartridges contain domain operators and domain indextypes (see Chapter 7, "g Building

Domain Indexes"), and optimization functions (see Chapter 8, "Query Optimization").

Object Type Specification

A data cartridge consists of one or more of these domain-specific objects packaged and integrated with the server. Each domain-specific type is an *object type* (or ODT, for object data type) and includes both of the following:

- *Attribute* data that holds object state information

  Attributes can be defined using built-in datatypes or other object types.

- *Methods* that incorporate the object's behavior

  Methods can be simple (such as adding two numbers) or complex (such as computing prices of financial derivatives), and can be coded either in PL/SQL or in a third-generation language (3GL) such as C.

The *object type specification* gives the object a name, and it defines the types of persistent data, called attributes, that an instance of this object will include. It also specifies names, return values, and argument types of the related behaviors, or methods. Much like a C++ class definition in a header (.h) prefix file, the type specification lays out the object framework (attributes and method signatures), but does not include the actual method code that performs the functions. The object type specifications for the various object types defined by your data cartridge will be written in SQL and stored in a SQL script that will be input to the server at cartridge installation time.

Object Type Body Code

The *type body* provides the code that implements the object type's methods. Method code can be implemented in PL/SQL, Java, C, C++, or any other 3GL. Most simple methods can be written in PL/SQL and Java. (See the *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL syntax.)

Code written in C, C++, and other 3GLs must be packaged in a runtime or dynamic link library. This is described in "External Library Linkage Specification" on page 1-16 and "External Library Code" on page 1-17.

External Library Linkage Specification

If the implementation of your methods is in C, C++, or some other 3GL, the methods must be packaged within a runtime or dynamic link library. The external library linkage specification is necessary to tell the server about this library, including its location, the binding of the type's methods to the library's entry points, and the methods' parameters.

Any 3GL code dispatched through the external library linkage specification will run in a separate process from the Oracle server. As such, the dispatch involves communication overhead. In deciding which methods should be implemented in external libraries, you should be aware of this overhead. In general, the cost of dispatch is less significant for methods that are complex or computation intensive.

External Library Code

The external library is the runtime or dynamic link library that contains any 3GL method code. You implement the 3GL methods in a language such as C, and then use operating-system-specific commands to build a shared-object library on UNIX platforms or a DLL on Windows NT systems.

Installing a Data Cartridge

Data cartridges are packaged so that their constituents (type definitions, PL/SQL packages, external procedures, users, roles, synonyms, and so forth) can be installed into or de-installed from the Oracle universal data server as a unit.

See Also: *Oracle Universal Installer Concepts Guide*

2 Roadmap to Building a Data Cartridge

This chapter describes a recommended development process, including relationships and dependencies among parts of the process. Topics include:

Development Process

The simplest questions are the most profound: *Who? What? When? Where? How?* You could say that this chapter is concerned with the *when* and *where* of *how.* But before we examine the road-map to building data cartridges, it would be wise to give a moment to viewing the project as a whole.

What

The very first step in developing a data cartridge is to establish the domain-specific value you intend to provide. Clearly define the new capabilities the cartridge will make available. More specifically: What are the objects that cartridge will expose to users as a means to accessing to these capabilities?

Who

Who are the intended users of this cartridge? Are they other developers — in which case the extensibility of the cartridge is of crucial importance. Are they end- users — in which case the cartridge must be highly attuned to the domain in question. Building a cartridge is a non-trivial project that should be founded in a business model that clearly distinguishes who these users are.

Being realistic about the complexity of building a data cartridge, raises the question of who it is that will perform the task. Are all the necessary skills present in the development team? Most essentially, the developers (be they one or many) must be able to bridge the object-relational database management system with the domain.

When and Where

What are the deliverables? How much time is there for development? Is there a software development process? The project is much more likely to succeed if there are clearly defined expectations and milestones. This chapter should aid you in mapping out a realistic development path.

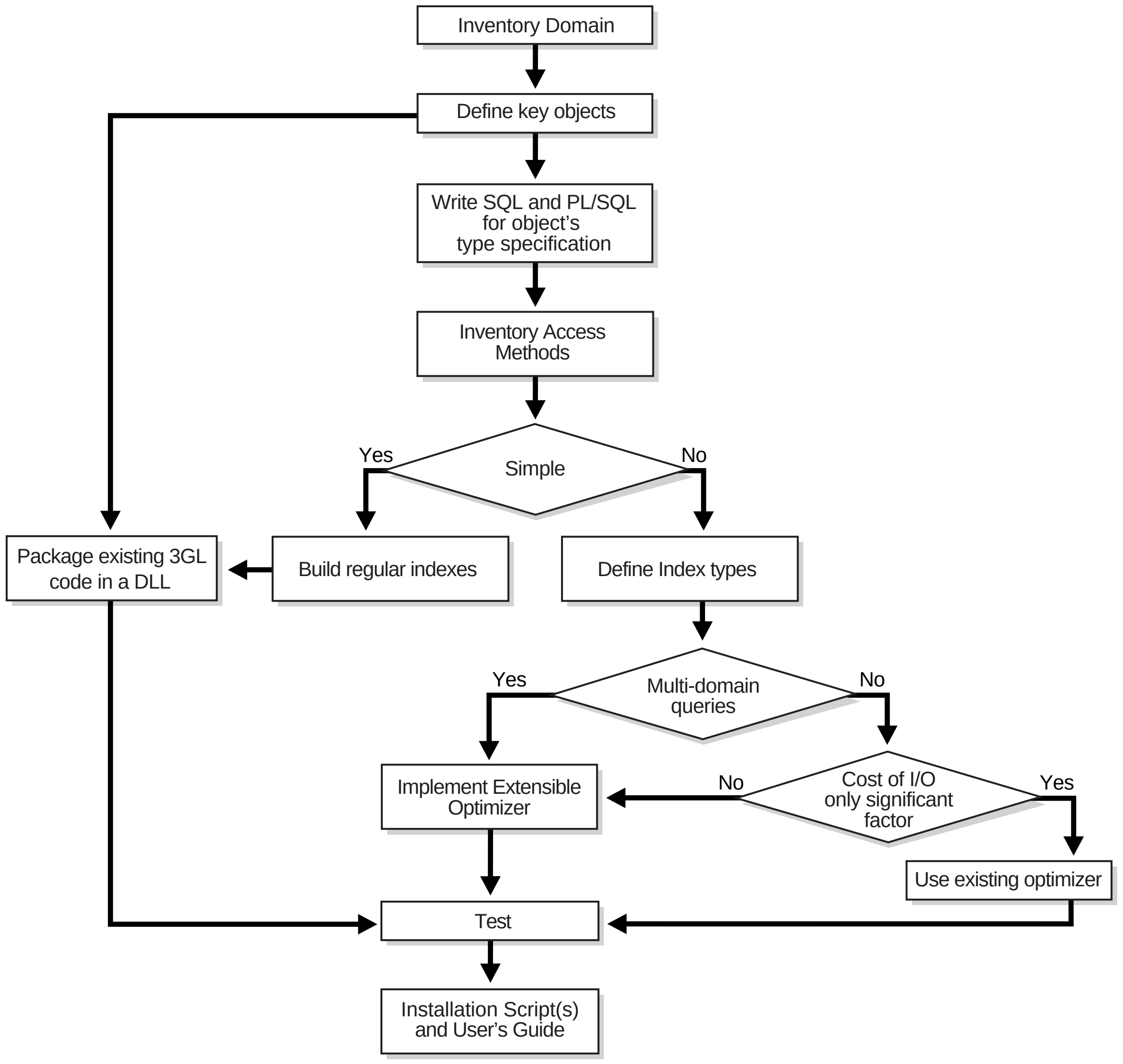
How

Choose and design objects so that their names and semantics are familiar in the developer's and users' domain. Given the complexity of the project, you should consider using one of the standard object-oriented design methodologies.

In defining a collection of objects, give care to the interface between the SQL side of object methods and the 3GL code that incorporates your value-added technology. Keep this interface as simple as possible by limiting the number of methods that call out to library routines and by allowing the 3GL code to do a block of work independently. Avoid defining hundreds of calls into low-level library entry points.

With this interface defined, you can proceed along parallel paths, as illustrated in Figure 2–1. You can complete the paths sequentially or alternately work among the paths until you complete all three.

Figure 2–1 Cartridge Development Process


Inventory Domain
Define key objects
Write SQL and PL/SQL for object's type specification
Inventory Access Methods
Simple
Yes
No
Package existing 3GL code in a DLL
Build regular indexes
Define Index types
Multi-domain queries
Yes
No
Implement Extensible Optimizer
Cost of I/O only significant factor
No
Yes
Use existing optimizer
Test
Installation Script(s) and User's Guide

The ’leftmost’ of these parallel paths packages any existing 3GL code that performs operations relevant to your domain in a DLL, possibly with new entry points on top of old code. The DLL will be called by the SQL component of the object's method code. Where possible, this code should all be tested in a standalone fashion using a 3GL test program.

The ’middle’ path defines and writes the object's type specification and the PL/SQL components of the object's method code. Some methods may be written entirely in PL/SQL, while others may call into the external library. If your application requires an external library, provide the library definition and the detailed bindings to library entry routines.

The direction you take at the choice point results from the simplicity or complexity of the access methods you need to deploy, which in turn derives from the nature of the data as represented by columns in the table. If you the methods you need to query your data are relatively simple, you can build regular indexes. By contrast, dealing with complex data means you will need to define complex index types as the basis for making use of Oracle’s extensible indexing technology. If you are in addition faced with implementing multi-domain queries, you should choose to make use of Oracle’s extensible optimizer technology.

It may be that you do not have execute queries on multiple domains. If I/O is the only significant factor affecting performance, you can make use of standard optimizing techniques. If, however, there are other factors in play, you may still need to utilize the extensible optimizer.

Finally, you will want to test the application and create the necessary installation scripts.

Installation and Use

Before you can use a data cartridge, you must install it. Installation is the process of assembling the sub-components so that the server can locate them and understand the object type definitions.

Putting the sub-components in place involves defining object types and tables in the server (usually accomplished by running SQL scripts), putting dynamic link libraries in the location expected by the linkage specification, and copying on-line documentation, help files, and error message files to a managed location.

Telling the server about the object types involves running SQL scripts that load the individual object types defined by the cartridge. This step must be done from a privileged account.

Finally, users of the cartridge must be granted the necessary privileges to use it.

Requirements and Guidelines for Data Cartridge Constituents

The following requirements and guidelines apply to certain database objects associated with the data cartridge.

Schema

The database components that make up each cartridge must be installed in a schema of the same name as the cartridge name. If a cartridge needs multiple schemas, the first 10 characters of the schema must be the same as the cartridge name. Note that the maximum permissible length of schema names in Oracle is 30 bytes (30 characters in single-byte languages.)

The following database components of a data cartridge must be placed in the cartridge schema:

- Type names
- Table names
- View names
- Directory names
- Library names
- Package names

The choice of a schema name determines the Oracle username, because the schema name and username are always the same in Oracle.

Globals

Some database-level constituents of cartridges may be global in scope, and so not within the scope of a particular user (schema) but visible to all users. Examples of such globals are:

- Roles
- Synonyms
- Sequences

All globals should start with the cartridge name. For example, a global role for the Acme video cartridge should have a unique global name like *C$ACMEVID1ROL1*, and not merely *ROL1*.

Error Message Names or Error Codes

Currently, error codes 20000-20999 are reserved for user errors or application errors. When a cartridge encounters an error, it should generate an error of the form *ORA 20000: %s*, where *%s* is a place holder for a cartridge-specific error message. Cartridge developers must ensure that their error messages are unique. You can ensure uniqueness by having all cartridge-specific error messages consist of a cartridge message name in the format C$pppptttm-nnnn plus message text. For example, an error raised by the Acme video cartridge might reported as:

```
ORA 20000: C$ACMEVID1-0001: No such file
```

In this example:

- `ORA 20000` is the server error code.
- `C$ACMEVID1` is the cartridge name.
- `0001` is the number assigned by Acme for this specific error.
- `No such file` is the description of the error, as written by Acme.

Cartridge Installation Directory

In many cases, a cartridge installation directory is desirable. All the operating system-level components of the cartridge, such as shared libraries, configuration files, and so on, can be put under a directory that is specific to a vendor or organization.

This directory name should be the same as the prefix chosen by the organization, and the directory should be created under the root directory for the platform. For example, if the Acme Cartridge Company needs to store any files, libraries, or directories, it must create a directory /ACME, and then store any files in cartridge-specific subdirectories.

Files

Message files that associate cartridge error or message numbers with message text can be put in one or more cartridge-specific subdirectories.

Configuration files can be placed in a cartridge-specific subdirectory. For example:

```
/ACME/VID1/Config
```

Shared Library Names for External Procedures

Use one of the following guidelines for each shared library (.so or .dll file):

- Place it in the cartridge installation directory. In this case, ensure that all library names are unique.
- Place it in a directory other than the cartridge installation directory. In this case, the file name should start with the cartridge name without the *C$* part. If there are multiple such libraries, the name should start with the first seven letters of the cartridge name without the *C$* part.

Deployment Checklist

At the *deployment* level, you will face a number of common issues. The most optimal approach to these problems will depend on the particular needs of your application. We list the tasks that we think should form the basis of your checklist, and in some cases propose solutions.

- You will need a way to install and de-install your cartridge components. This includes libraries, database objects, flat files, programs, configuration tools, administration tools, and other objects. Explore whether your cartridge might be able to utilize the *Oracle Universal Installer.*

  See Also: *Oracle Universal Installer Concepts Guide*

- You should allow for installation of multiple versions of a cartridge to provide backward compatibility and availability. Make use of Oracle's migration facilities as part of your larger strategy.
- You will need to track which cartridges are installed in order to install data cartridges that depend on other data cartridge, or to handle different versions of installed components.
- You will need to provide an upgrade path for migrating to newer versions of cartridges. Make use of Oracle's migration facilities as part of your larger strategy.
- To be able to limit access to cartridge components to specific users and roles, deploy Oracle's security mechanisms together with a blend of procedures that operate under invoker's and definer's rights depending on the need.

- You will need to be able to keep track of which users have access to a cartridge (for ease of administration). Consider making use of a table with appropriate triggers.
- How do you know where cartridges are installed? This is more of a security / administration concern than a requirement. There is currently no easy way of knowing which cartridges are installed in a particular database or what users have access to the cartridge or any of its components.

Naming Conventions

This section discusses how the components of a data cartridge should be named. It is intended for independent software vendors (ISVs) and others who are creating cartridges to be used by others.

Note: Most examples in this manual do not follow the naming conventions, because they are intended to be as simple and generic as possible. However, as your familiarity with the technology increases and you consider building data cartridges to be used by others, you should understand and follow these naming conventions.

The naming conventions in this chapter assume a single-byte character set. See "Internationalization" on page 2-12 for a discussion of using other character sets.

See Also: Chapter 9, the section "Globalization Support"

Need for Naming Conventions

In a production environment, an Oracle database may have multiple data cartridges installed. These data cartridges may be from different development groups or vendors, and may have been developed in isolation. Each data cartridge consists of various schema objects inside the database, as well as other components visible at the operating system level, such as external procedures in shared libraries. If multiple data cartridges tried to use the same names for schema objects or operating system-level entities, the result would be incorrect and inconsistent behavior.

Furthermore, because exception conditions during the runtime operation of data cartridges can cause the Oracle server to return errors, it is important to prevent conflicts between error or message codes of different data cartridges. These conflicts can arise if, for example, two cartridges use the same error code for different error conditions. Having unique error and message codes ensures that the origin of the exception condition can be readily identified.

Unique Name Format

To prevent multiple data cartridge components from having the same name, Oracle recommends the following convention to ensure unique naming of data cartridges. Naming is to be done for each vendor or supplier. That is, each organization developing data cartridges must choose an unique name, and Oracle will provide a name reservation service.

Each organization should choose an reserve a prefix. Oracle will add C$ to the start of the string chosen by the organization, to ensure a unique prefix. This prefix can then be used to name the database schema in which the database components of the data cartridge reside, or to name the directory in which the operating-system components of the data cartridge are placed.

Data cartridges and their components should have names of the following format:

C$pppptttm.ccccccccc

The following table describes the parts of this naming convention format.

Table 2–1 Data Cartridge Naming Conventions

Part	Explanation	Example
C$	Recommended by Oracle for all data cartridges.	
pppp	Prefix selected by the data cartridge creator. (Must be exactly four characters.)	ACME
ttt	Type of cartridge, using an abbreviation meaningful to the creator. Three characters.	AUD (for *audio*)
m	Miscellaneous information indicator, to allow a designation meaningful to the creator. One character.	1 (perhaps a version number)
. (period)	Period required if specifying an object in full *schema.object* form.	
ccccccccc	Component name. Variable length.	mf_set_volume (method function adjusting volume)

Oracle recommends that except for the dollar sign ($) as the second character, all characters in the name should be alphanumeric (letters and numbers, with underscores and hyphens permitted).

For example, Acme Cartridge Company chooses and registers a prefix of *ACME*. It provides an audio data cartridge and a video data cartridge, and chooses *AUD* and *VID* as the type codes, respectively. It has no other information to include in the cartridge name, and so it chooses an arbitrary number *1* for the miscellaneous information indicator. As a result, the two cartridge names are:

- `C$ACMEAUD1`
- `C$ACMEVID1`

For each cartridge, a separate schema must be created, and Acme uses the cartridge name is the schema name. Thus, all database components of the audio cartridge must be created under the schema C$ACMEAUD1, and all database components of the video cartridge must be created under the schema C$ACMEVID1. Examples of some components might include:

- `C$ACMEVID1.mf_rewind`
- `C$ACMEVID1.vid_ops_package`
- `C$ACMEVID1.vid_stream_lib`

Each organization is responsible for specific naming requirements after the *C$pppp* portion of the object name. For example, Acme Cartridge Company must ensure that all of its cartridges have unique names and that all components within a cartridge have unique names.

Cartridge Registration

In order to make a naming scheme work, you need to have a registration process that will handle the administration of names of components that make up a data cartridge.

Directory Structure and Standards

You need some directory standard to know where to put your binaries, support files, messages files, administration files, and libraries.

You also need to define a database user who will install your cartridges. One possible solution is to use `EXDSYS`, for External Data Cartridge System user.

Note:

The EXDSYS user is a user with special privileges required for running cartridges. This user could be installed as part of cartridge installation, but would better be part of the database installation. To do this, you will need to move this process into a standard database creation script.

Your long range planning should consider ways to integrate directory structure with the Network Computer Architecture (NCA).

Cartridge Upgrades

Administrators need a safe way to upgrade a cartridge and its related metadata to a newer version of the cartridge. You also require a process for upgrading data and removing obsolete data. This may entail installation support (Enterprise Manager) and database support for moving to newer database cartridge types

Administrators also require a means to update tables using cartridge types when a cartridge changes.

Import and Export

To import and export objects, you need to understand how Oracle's import and export facilities handle Oracle objects. In particular, you need to know how types are handled and whether the type methods are imported and exported, and also whether user-defined methods are supported.

Cartridge Versioning

There are two types of cartridge versioning problems that need to be addressed. They are:

- Internal Versioning
- External Versioning

Internal Versioning

Internal versioning is the harder problem. Ideally, you would like a mechanism to support multiple versions of a cartridge in the database. This would provide backward compatibility and also make for high availability.

Types are amenable to changing methods, but not to changing the type attributes themselves. This implies that upgrades are complicated for types that change over time. You may need a way to use multiple versions of type, and some method to insure that administrators can gradually update your technology.

External Versioning

External versioning is the easier of the two versioning problems. You need to be able to track a cartridge version number and be able to take action accordingly upon installation or configuration based on versioning information.

Internationalization

You may want to internationalize your cartridges. This means they will need to be able to support multiple languages and have access to Globalization Support facilities for messages and parsing. For details on Globalization Support, see the *Oracle9i Globalization Support Guide*. It includes a chapter on the Globalization Support data cartridge service.

It is recommended that the names for data cartridge components be chosen using the ASCII character set.

If you must name the data cartridge components in a character set other than ASCII, Oracle will still assign you a four-character unique prefix. This will, however, increase the number of bytes required to hold the prefix. The names of all Oracle schema objects must fit into 30 bytes. In ASCII, this equals 30 characters. If you have, for example, a six-byte character set and request a four-character prefix string, Oracle may truncate your request to a smaller number of characters.

External Access

- How do administrators know who has access to a cartridge?

  Administrators need to administer access rights to internal and external components such as programs and data files to specific users and roles.

Internal Access

- How do administrators restrict access to certain tables, types, views, and other cartridge components to individual users and roles?

  Administrators for security reasons must be allowed to restrict access to types on an individual basis.

For instance, some data cartridges, such as Oracle's Image Cartridge, have few security issues. These cartridges may grant privileges to every user in the database. Other cartridges that are more complex may need differing security models. In building complex data cartridges, you will need a way to identify the various components that make up a cartridge and also an instance of a cartridge and be able to grant and revoke security roles on identifiable components.

It may be that Oracle will provide a visual tool will identify components of a cartridge and allow roles be assigned to each component.

Invoker's Rights

Invoker's rights is a special privilege that allows the system to access database objects that it wouldn't normally have access to. This has been the case for the special SYS user. It also will need to be done for cartridges under whatever user you use (such as EXDSYS).

If you don't have invoker's rights, then any types you construct in a central user space (such as EXDSYS) will have to grant privileges to public, which is not necessarily desirable.

Test and Debug Services

You will need a way to test and debug your cartridges.Please refer to the guides which pertains to your operating environment (PL/SQL, Java, C/C++)

Administration

Configuration

Data Cartridges need a front end to handle deployment issues, such as installation, as well as configuration tools. While each data cartridge may have differing security needs, a basic front end that allows a user to install, configure, and administer data cartridge components is necessary.

This front end may just be some form of knowledge base or on-line documentation. In any case, it should be on-line, easily navigable, and contain templates exhibiting standards and starting points.

Suggested Development Approach

In developing a data cartridge, it is best to take a systematic approach, starting with small, easy tasks and building incrementally toward a comprehensive solution.This section presents a suggested approach.

To create a prototype data cartridge:

1. Read this book and try the examples on disk and in example chapter.
2. Create the prototype of your own data cartridge, creating a single object type and a few data elements and methods. You can add object types, data elements, and methods, specific indextypes, and user-defined operators as you expand the cartridge's capabilities.)
3. Begin by implementing your methods entirely in SQL, and add callouts to 3GL code (if any) later.
4. Test and debug your cartridge.

After you have the prototype working, you may want to follow a development process that includes these steps:

1. Identify your areas of domain expertise.
2. Identify those areas of expertise that are relevant to persistent data.
3. Consider the feasibility of packaging one or more of these areas as a new data cartridge or as an extension to an existing cartridge.
4. Use an object-oriented methodology to help decide what object types to include in data cartridges.
5. Build and test the cartridges, one at a time.

Part II

Building Data Cartridges

3 Defining Object Types

This chapter provides an example of starting with a schema for a data cartridge. Object types are crucial to building data cartridges in that they enable domain-level abstractions to be captured in the database.

Topics include:

- Objects and Object Types
- Assigning an OID to an Object Type
- Constructor Methods
- Object Comparison

See Also: The following manuals contain additional information about creating and using object types:

- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle9i Database Concepts*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *PL/SQL User's Guide and Reference*.

Objects and Object Types

In the Oracle ORDBMS ("Object-Relational Database Management System"), you use object types to model real-world entities. An object type has attributes, which reflect the entity's structure, and methods, which implement the operations on the entity. Attributes are defined using built-in types or other object types. Methods are functions or procedures written in PL/SQL or an external language like C and stored in the database.

A typical use for an object type is to impose structure on some part of the data kept in the database. For example, an object type named *DataStream* could be used by a cartridge to store large amounts of data in a character LOB (a data type for large objects). This object type has attributes such as an identifier, a name, a date, and so on. The following statement defines the *DataStream* datatype:

```
CREATE OR REPLACE TYPE DataStream AS OBJECT (
   id INTEGER,
   name VARCHAR2(20),
   createdOn DATE,
   data CLOB,
   MEMBER FUNCTION DataStreamMin  RETURN pls_integer,
   MEMBER FUNCTION DataStreamMax  RETURN pls_integer,
   MAP MEMBER FUNCTION DataStreamToInt  return integer,
   PRAGMA restrict_references(DataStreamMin, WNDS, WNPS),
   PRAGMA restrict_references(DataStreamMax, WNDS, WNPS));
```

A method is a procedure or function that is part of the object type definition and that can operate on the object type data attributes. Such methods are called **member methods**, and they take the keyword `MEMBER` when you specify them as a component of the object type. The `DataStream` type definition declares three methods. The first two, `DataStreamMin` and `DataStreamMax`, calculate the minimum and maximum values, respectively, in the data stream stored inside the character `LOB`.

The third method (`DataStreamToInt`), a **map method**, governs comparisons between instances of data stream type.

> **See Also:** "Object Comparison" on page 3-5 for information about map methods

The pragma (compiler directive) `RESTRICT_REFERENCES` is necessary for security, and is discussed in the following sections.

After declaring the type, define the type body. The body contains the code for type methods. The following example shows the type body definition for the `DataStream` type. It defines the member function methods (`DataStreamMin` and *DataStreamMax*) and the map method (`DataStreamToInt`).

```
CREATE OR REPLACE TYPE BODY DataStream IS
    MEMBER FUNCTION DataStreamMin RETURN pls_integer IS
      a pls_integer := DS_Package.ds_findmin(data);
      BEGIN RETURN a; END;
    MEMBER FUNCTION DataStreamMax RETURN pls_integer IS
      b pls_integer := DS_Package.ds_findmax(data);
      BEGIN RETURN b; END;
    MAP MEMBER FUNCTION DataStreamToInt RETURN integer IS
      c integer := id;
      BEGIN RETURN c; END;
END;
```

`DataStreamMin` and `DataStreamMax` involve calling routines in a PL/SQL package called `DS_Package`. Since these methods are likely to be compute-intensive (they process numbers stored in the CLOB to determine minimum and maximum values), they are defined as external procedures and implemented in C. The external dispatch is routed through a PL/SQL package named `DS_Package`. Such packages are discussed in *Oracle9i Supplied PL/SQL Packages and Types Reference*.

The third method (`DataStreamToInt`), the map method, is implemented in PL/SQL. Because we have a identifier (*id*) attribute in `DataStream`, this method can return the value of the identifier attribute. (Most map methods, however, are more complex than `DataStreamToInt`.).

See Also:

- Chapter 6, "Working with Multimedia Datatypes", for information about using LOBs with data cartridges
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for general information about LOBs

Assigning an OID to an Object Type

The `CREATE TYPE` statement has an optional keyword `OID`, which associates a user-specified object identifier (`OID`) with the type definition. This feature was available effective with release 8.0.3; however, it was not documented because it is intended for use primarily by Oracle product developers and by developers of data cartridges. However, it should be used by anyone who creates an object type that will be used in more than one database.

Each type has an OID. If you create an object type and do not specify an OID, Oracle generates an OID and assigns it to the type. Oracle uses the OID internally for operations pertaining to that type. Using the same OID for a type is important if you plan to share instances of the type across databases for such operations as export/import and distributed queries.

Note: Most other Oracle documentation refers to the use of OIDs with rows in object tables. In CREATE TYPE with OID, an OID is assigned to the type itself. Of course, each row created in a table with a column of the specified type will also still have a row-specific OID.

For example, assume that you want to create a type named `SpecialPerson` and then instantiate that type in two different databases with tables named `SpecialPersonTable1` and `SpecialPersonTable2`. The RDBMS needs to know that the `SpecialPerson` type is the same type in both instances, and therefore the type must be defined using the same OID in both databases. If you do not specify an OID with CREATE TYPE, a unique identifier is created automatically by the RDBMS.

The syntax for specifying an OID for an object type is as follows:

```
CREATE OR REPLACE TYPE type_name OID 'oid' AS OBJECT (attribute datatype
[,...]);
```

In the following example, the SELECT statement generates an OID, and the CREATE TYPE statement uses the OID in creating an object type named *mytype*. Be sure to use the SELECT statement to generate a different OID for each object type to be created, because this is the only way to guarantee that each OID is valid and globally unique.

```
SQLPLUS> SELECT SYS_OP_GUID() FROM DUAL;
SYS_OP_GUID()
--------------------------------
19A57209ECB73F91E03400400B40BBE3
1 row selected.
```

```
SQLPLUS> CREATE TYPE mytype OID '19A57209ECB73F91E03400400B40BBE3'
     2> AS OBJECT (attrib1 NUMBER);
Statement processed.
```

Constructor Methods

The system implicitly defines a **constructor method** for each object type that you define. The name of the constructor method is the same as the name of the object type. The parameters of the constructor method are exactly the data attributes of the object type, and they occur in the same order as the attribute definition for the object type. At present, only one constructor method can be defined, and thus you cannot define other constructor methods.

For example, when the system executes the following statement to create a type named *rational_type*, it also implicitly creates a constructor method for this object type.

```
CREATE TYPE rational_type (
     numerator integer,
     denominator integer);
```

When you instantiate an object of *rational_type*, you invoke the constructor method. For example:

```
CREATE TABLE some_table (
     c1 integer, c2 rational_type);
INSERT INTO some_table
     VALUES (42, rational_type(223, 71));
```

Object Comparison

SQL performs comparison operations on objects. Comparisons can be explicit, using the comparison operators (=, <, >, <>, <=, >=, !=) and the `BETWEEN` and `IN` predicates. Comparisons can be implicit, as in the `GROUP BY`, `ORDER BY`, `DISTINCT`, and `UNIQUE` clauses.

Comparison of objects makes use of special member functions of the object type: map methods and order methods. To perform object comparison, you must implement either a map method or order method in the `CREATE TYPE` and `CREATE TYPE BODY` statements.

For example, the type body for the *DataStream* type, implements the map member function for *DataStream* comparison as:

```
MAP MEMBER FUNCTION DataStreamToInt RETURN integer IS
      c integer := id;
      BEGIN RETURN c; END;
```

This definition of the map member function relies on the presence of the *id* attribute of the *DataStream* type to map instances to integers. Whenever a comparison operation is required between objects of type *DataStream*, the map function *DataStreamToInt ()* is called implicitly by the system.

The object type *rational_type* does not have a simple *id* attribute like that for *DataStream*. For *rational_type*, the map member function is slightly more complicated. Because a map function can return any of the built-in types; *rational_type* can return a value or type `REAL`:

```
MAP MEMBER FUNCTION RationalToReal RETURN REAL IS
     BEGIN
         RETURN numerator/denominator;
     END;
...
```

If you have not defined a map or order function for an object type, only equality comparisons are allowed on objects of that type. Oracle SQL performs the comparison by doing a field-by-field comparison of the attributes of that type.

4 Methods: Using C/C++ and Java

This chapter describes how to use C, C++, and Java to implement the methods of a data cartridge. Methods are procedures and functions that define the operations permitted on data defined using the data cartridge.

This chapter focuses on issues related to developing and debugging external procedures, including:

External Procedures

PL/SQL is powerful language for database programming. However, because some methods can be complex, it may not be possible to code such a method optimally using PL/SQL. For example, a routine to perform numerical integration will probably run faster if it is implemented in C than if it is implemented in PL/SQL.

To support such special-purpose processing, PL/SQL provides an interface for calling routines written in other languages. This makes the strengths and capabilities of 3GLs like C available through calls from a database server. Such a 3GL routine, called an **external procedure**, is stored in a shared library, registered with PL/SQL, and called from PL/SQL at runtime to perform special-purpose processing. Details on external procedures and their use can be found in the *PL/SQL User's Guide and Reference*.

External procedures are an important tool for data cartridge developers. They can be used not only to write fast, efficient, computation-intensive routines for cartridge types, but also to integrate existing code with the database as data cartridges. Shared libraries already written and available in other languages, such as a Windows NT DLL with C routines to perform format conversions for audio files, can be called directly from a method in a type implemented by an audio cartridge. Similarly, you can use external procedures to process signals, drive devices, analyze data streams, render graphics, or process numerical data.

Using Shared Libraries

A **shared library** is an operating system file, such as a Windows DLL or a Solaris shared object, that stores the coded implementation of external procedures. Access to the shared library from Oracle occurs by using an **alias library**, which is a schema object that represents the library within PL/SQL. For security, creation of an alias library requires DBA privileges. To create the alias library (such as *DS_Lib* in the following example), you must decide on the operating system location for the library, log in as a DBA or as a user with the `CREATE LIBRARY PRIVILEGE`, and then enter a statement such as the following:

```
CREATE OR REPLACE LIBRARY DS_Lib AS
     '/data_cartridge_dir/libdatastream.so';
```

This example creates the alias library schema object in the database. After the alias library is created, you can refer to the shared library by the name *DS_Lib* from PL/SQL.

The example just given specifies an absolute path for the library. If you have copies of the library on multiple systems, to support distributed execution of external procedures by designated (or "dedicated") agents, you can use an environment variable to specify the location of the libraries more generally. For example:

```
CREATE OR REPLACE LIBRARY DS_Lib AS
  '${DS_LIB_HOME}/libdatastream.so' AGENT 'agent_link';
```

This statement uses the environment variable `${DS_LIB_HOME}` to specify a common point of reference or root directory from which the library can be found on all systems. The string following the `AGENT` keyword specifies the agent (actually, a database link) that will be used to run any external procedure declared to be in library `DS_Lib`.

> **See Also:** For more information on using dedicated external procedure agents to run an external procedure, see *PL/SQL User's Guide and Reference*

Registering an External Procedure

To call an external procedure, you must not only tell PL/SQL the alias library in which to find the external procedure, but also how to call the procedure and what arguments to pass to it.

Earlier, the type *DataStream* was defined, and certain methods of type *DataStream* were defined by calling functions from a package *DS_Package*. Also, this package was specified. The following statement defines the body of this package (*DS_ Package*).

```
CREATE OR REPLACE PACKAGE BODY DS_Package AS
     FUNCTION DS_Findmin(data  CLOB) RETURN PLS_INTEGER IS EXTERNAL
     NAME "c_findmin" LIBRARY DS_Lib LANGUAGE C WITH CONTEXT;
     FUNCTION DS_Findmax(data CLOB) RETURN PLS_INTEGER IS EXTERNAL
     NAME "c_findmax" LIBRARY DS_Lib LANGUAGE C WITH CONTEXT;
   END;
```

In the `PACKAGE BODY` declaration clause of this example, the package functions are tied to external procedures in a shared library. The `EXTERNAL` clause in the function declaration registers information about the external procedure, such as its name (found after the `NAME` keyword), its location (which must be an alias library, following the `LIBRARY` keyword), the language in which the external procedure is written (following the `LANGUAGE` keyword), and so on. For a description of the parameters that can accompany an `EXTERNAL` clause, see the *PL/SQL User's Guide and Reference.*

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals*, the chapter on external procedures, for information on how to format the call specification when passing an object type to a C routine

The final part of the `EXTERNAL` clause in the example is the `WITH CONTEXT` specification. This means that a context pointer is passed to the external procedure. The context pointer is opaque to the external procedure, but is available so that the external procedure can call back to the Oracle server, to potentially access more data in the same transaction context. The `WITH CONTEXT` clause is discussed in "Using the WITH CONTEXT Clause" on page 4-10.

Although the example describes external procedure calls from object type methods, a data cartridge can use external procedures from a variety of other places in PL/SQL. External procedure calls can appear in:

- Anonymous blocks
- Standalone and packaged subprograms
- Methods of an object type
- Database triggers
- SQL statements (calls to packaged functions only)

How PL/SQL Calls an External Procedure

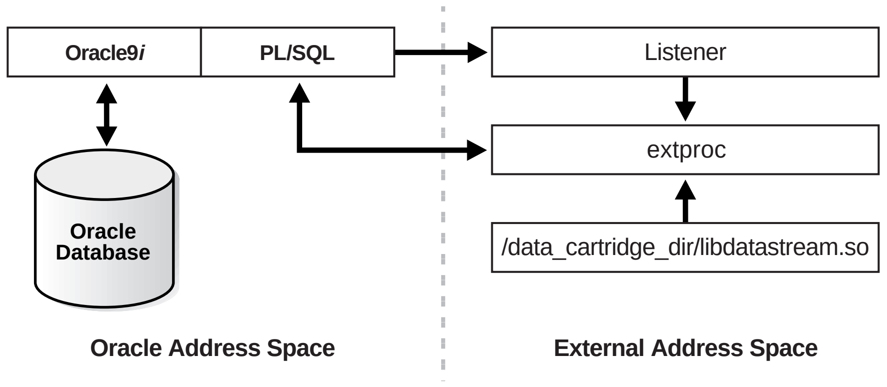
To call an external procedure, PL/SQL must know the DLL or shared library in which the procedure resides. PL/SQL looks up the alias library in the `EXTERNAL` clause of the subprogram that registered the external procedure. The data dictionary is used to determine the actual path to the operating system shared library or DLL.

PL/SQL alerts a Listener process, which in turn spawns (launches) a session-specific agent. Unless some other particular agent has been designated, either in the `CREATE LIBRARY` statement for the procedure's specified library or in the agent argument of the `CREATE PROCEDURE` statement, the default agent `extproc` is launched. The Listener hands over the connection tothe agent. PL/SQL passes the agent the name of the DLL, the name of the external procedure, and any parameters passed in by the caller.

The rest of this account assumes that the agent launched is the default agent `extproc`. For more information on using dedicated external procedure agents to run an external procedure, see *PL/SQL User's Guide and Reference*.

After receiving the name of the DLL and the external procedure, `extproc` loads the DLL and runs the external procedure. Also, `extproc` handles service calls (such as raising an exception) and callbacks to the Oracle server. Finally, `extproc` passes to PL/SQL any values returned by the external procedure. Figure 4–1 shows the flow of control.

Figure 4–1 How an External Procedure is Called



Note: The Listener must start `extproc` on the system that runs the Oracle server. Starting `extproc` on a different system is not supported.

After the external procedure completes, `extproc` remains active throughout your Oracle session. (When you log off, `extproc` is killed.) Thus, you incur the cost of spawning `extproc` only once, no matter how many calls you make. Still, you should call an external procedure only when the computational benefits outweigh the cost.

See Also: For information about administering `extproc` and external procedure calls, see the *Oracle9i Database Administrator's Guide*.

Configuration Files for External Procedures

The configuration files listener.ora and tnsnames.ora must have appropriate entries so that the Listener can dispatch the external procedures.

The Listener configuration file listener.ora must have a `SID_DESC` entry for the external procedure. For example:

```
# Listener configuration file
# This file is generated by stkconf.tsc

CONNECT_TIMEOUT_LISTENER = 0

LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=ipc)(KEY=o8))
 (ADDRESS=(PROTOCOL=tcp)(HOST=unix123)(PORT=1521))
)

SID_LIST_LISTENER = (SID_LIST=
(SID_DESC=(SID_NAME=o8)(ORACLE_HOME=/rdbms/u01/app/oracle/product/8.0
.3))

(SID_DESC=(SID_NAME=extproc)(ORACLE_HOME=/rdbms/u01/app/oracle/product/
8.0.3)(PROGRAM=extproc))
)
```

This listener.ora example assumes the following:

- The Oracle instance is called *o8*.
- The system or node on which the Oracle server runs is named *unix123*.
- The installation directory for the Oracle server is /rdbms/u01.
- The port number for Oracle TCP/IP communication is the default Listener port 1521.

The tnsnames.ora file (network substrate configuration file) must also be updated to refer to the external procedure. For example:

```
o8      =
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=unix123)(PORT=1521))(CONNECT_
DATA=(SID=o8)))
```

```
extproc_connection_data =
(DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=o8))(CONNECT_DATA=(SID=extproc)))
```

This tnsnames.ora example assumes that IPC mechanisms are used to communicate with the external procedure. You can also use, for example, TCP/IP for communication, in which case the `PROTOCOL` parameter must be set to *tcp*.

For more information about configuring the listener.ora and tnsnames.ora files, see the *Oracle9i Database Administrator's Guide*.

Passing Parameters to an External Procedure

Passing parameters to an external procedure is complicated by several circumstances:

- The set of PL/SQL datatypes does not correspond one-to-one with the set of C datatypes.
- PL/SQL parameters can be null, whereas C parameters cannot. (Unlike C, PL/SQL includes the RDBMS concept of nullity.)
- The external procedure might need the current length or maximum length of `CHAR`, `LONG RAW`, `RAW`, and `VARCHAR2` parameters.
- The external procedure might need character set information about `CHAR`, `VARCHAR2`, and `CLOB` parameters.
- PL/SQL might need the current length, maximum length, or null status of values returned by the external procedure.

In the following sections, you learn how to specify a parameter list that deals with these circumstances.

An example of parameter passing is shown in "Doing Callbacks" on page 4-11, where the package function *DS_Findmin(data CLOB)* calls the C routine *c_findmin* and the CLOB argument is passed to the C routine as an *OCILobLocator* *.

Specifying Datatypes

You do not pass parameters to an external procedure directly. Instead, you pass them to the PL/SQL subprogram that registered the external procedure. So, you must specify PL/SQL datatypes for the parameters. For guidance, see Table 4–1. Each PL/SQL datatype maps to a default external datatype. (In turn, each external datatype maps to a C datatype.)

Table 4–1 Parameter Datatype Mappings

PL/SQL Type	Supported External Types	Default External Type
BINARY_INTEGER, BOOLEAN, PLS_INTEGER	CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, INT, UNSIGNED INT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T	INT
NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE	CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, INT, UNSIGNED INT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T	UNSIGNED INT
FLOAT, REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR, CHARACTER, LONG, ROWID, VARCHAR, VARCHAR2	STRING	STRING
LONG RAW, RAW	RAW	RAW
BFILE, BLOB, CLOB	OCILOBLOCATOR	OCILOBLOCATOR

In some cases, you can use the `PARAMETERS` clause to override the default datatype mappings. For example, you can re-map the PL/SQL datatype `BOOLEAN` from external datatype `INT` to external datatype `CHAR`.

To avoid errors when declaring C prototype parameters, refer to Table 4–2, which shows the C datatype to specify for a given external datatype and PL/SQL parameter mode. For example, if the external datatype of an `OUT` parameter is `CHAR`, specify the datatype **char *** in your C prototype.

Table 4–2 External Datatype Mappings

External Datatype	IN, RETURN	IN by Reference, RETURN by Reference	IN OUT, OUT
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILobLocator *	OCILobLocator *	OCILobLocator **

Using the Parameters Clause

You can optionally use the PARAMETERS clause to pass additional information about PL/SQL formal parameters and function return values to an external procedure. You can also use this clause to reposition parameters.

See Also: For more information about the PARAMETERS clause, see the *PL/SQL User's Guide and Reference.*

Using the WITH CONTEXT Clause

Once launched, an external procedure may need to access the database. For example, *DS_Findmin* does not copy the entire CLOB data over to *c_findmin*, because doing so would vastly increase the amount of stack that the C routine needs. Instead, the PL/SQL function just passes a LOB locator to the C routine, with the intent that the database will be re-accessed from C to read the actual LOB data.

When the C routine reads the data, it can use the OCI buffering and streaming interfaces associated with LOBs (see the *Oracle Call Interface Programmer's Guide* for details), so that only incremental amounts of stack are needed. Such re-access of the database from an external procedure is known as a **callback**.

To be able to call back to a database, you need to use the WITH CONTEXT clause to give the external procedure access to the database environment, service, and error handles. When an external procedure is called using WITH CONTEXT, the corresponding C routine automatically gets as its first parameter an argument of type OCIExtProcContext *. (The order of the parameters can be changed using the PARAMETERS clause.) You can use this context pointer to fetch the handles using the OCIExtProcGetEnv call, and then call back to the database. This procedure is shown in "Doing Callbacks" on page 4-11.

OCIExtProcGetEnv

This service routine enables OCI callbacks to the database during an external procedure call. Use the OCI handles obtained by this function only for callbacks. If you use them for standard OCI calls, the handles establish a new connection to the database and cannot be used for callbacks in the same transaction. In other words, during an external procedure call, you can use OCI handles for callbacks or a new connection but not for both.

The C prototype for this function follows:

```
sword OCIExtProcGetEnv(
   OCIExtProcContext *with_context,
   OCIEnv    **envh,
   OCISvcCtx **svch,
   OCIError  **errh);
```

The parameter *with_context* is the context pointer, and the parameters *envh*, *svch*, and *errh* are the OCI environment, service, and error handles, respectively. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

"Doing Callbacks" on page 4-11 shows how `OCIExtProcGetEnv` might be used in callbacks. For a working example, see the script extproc.sql in the PL/SQL demo directory. (For the location of this directory, see your Oracle installation or user's guide.) This script demonstrates the calling of an external procedure. The companion file extproc.c contains the C source code for the external procedure. To run the demo, follow the instructions in `extproc.sql`. You must use the *SCOTT/TIGER* account, which must have `CREATE LIBRARY` privileges.

Doing Callbacks

An external procedure executing on the Oracle server can call the access function `OCIExtProcGetEnv` to obtain OCI environment and service handles. With the OCI, you can use callbacks to execute SQL statements and PL/SQL subprograms, fetch data, and manipulate LOBs. Moreover, callbacks and external procedures operate in the same user session and transaction context, so they have the same user privileges.

The following example is a version of `c_findmin` that is simplified to illustrate callbacks. The complete listing is available on the disk that is included with this kit.

```
Static  OCIEnv   *envhp;
Static  OCISvcCtx  *svchp;
Static OCIError   *errhp;
Int   c_findmin (OCIExtProcContext *ctx, OCILobLocator  *lobl) {
sword  retval;
retval = OCIExtProcGetEnv (ctx, &envhp, &svchp, &errhp);
if ((retval != OCI_SUCCESS) && (retval !=  OCI_SUCCESS_WITH_INFO))
   exit(-1);
   /* Use lobl to read the CLOB, compute the minimum, and store the value
       in retval. */
return retval;
}
```

Restrictions on Callbacks

With callbacks, the following SQL statements and OCI routines are not supported:

- Transaction control statements such as `COMMIT`

- Data definition statements such as `CREATE`
- Object-oriented OCI routines such as `OCIRefClear`
- Polling-mode OCI routines such as `OCIGetPieceInfo`
- All these OCI routines:

  ```
  OCIEnvInit
  OCIInitialize
  OCIPasswordChange
  OCIServerAttach
  OCIServerDetach
  OCISessionBegin
  OCISessionEnd
  OCISvcCtxToLda
  OCITransCommit
  OCITransDetach
  OCITransRollback
  OCITransStart
  ```

- Also, with OCI routine `OCIHandleAlloc`, the following handle types are not supported:

  ```
  OCI_HTYPE_SERVER
  OCI_HTYPE_SESSION
  OCI_HTYPE_SVCCTX
  OCI_HTYPE_TRANS
  ```

OCI Access Functions for External Procedures

When called from an external procedure, a service routine can raise exceptions, allocate memory, and get OCI handles for callbacks to the server. To use the functions, you must specify the `WITH CONTEXT` clause, which lets you pass a context structure to the external procedure. The context structure is declared in header file ociextp.h as follows:

```
typedef struct OCIExtProcContext OCIExtProcContext;
```

This section describes how service routines use the context information. For more information and examples of usage, see the chapter on external procedures in the*Oracle9i Application Developer's Guide - Fundamentals.*

OCIExtProcAllocCallMemory

This service routine allocates *n* bytes of memory for the duration of the external procedure call. Any memory allocated by the function is freed as soon as control returns to PL/SQL.

Note: Do not use any other function to allocate or free memory.

The C prototype for this function follows:

```
dvoid *OCIExtProcAllocCallMemory(
   OCIExtProcContext *with_context,
   size_t amount);
```

The parameters *with_context* and *amount* are the context pointer and number of bytes to allocate, respectively. The function returns an untyped pointer to the allocated memory. A return value of zero indicates failure.

OCIExtProcRaiseExcp

This service routine raises a predefined exception, which must have a valid Oracle error number in the range 1..32767. After doing any necessary cleanup, the external procedure must return immediately. (No values are assigned to `OUT` or `IN OUT` parameters.) The C prototype for this function follows:

```
int OCIExtProcRaiseExcp(
   OCIExtProcContext *with_context,
   size_t error_number);
```

The parameters *with_context* and *error_number* are the context pointer and Oracle error number. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

OCIExtProcRaiseExcpWithMsg

This service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```
int OCIExtProcRaiseExcpWithMsg(
   OCIExtProcContext *with_context,
   size_t error_number,
   text   *error_message,
   size_t  len);
```

The parameters `with_context`, `error_number`, and `error_message` are the context pointer, Oracle error number, and error message text. The parameter *len* stores the length of the error message. If the message is a null-terminated string, *len* is zero. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

Common Potential Errors

This section presents several kinds of errors you might make in running external procedures.

Calls to External Functions

```
Can't Find DLL
ORA-06520: PL/SQL: Error loading external library
ORA-06522: Unable to load DLL
ORA-06512: at "<name>", line <number>
ORA-06512: at "<name>", line <number>
ORA-06512: at line <number>
```

You may have specified the wrong path or wrong name for the DLL file, or you may have tried to use a DLL on a network mounted drive (a remote drive).

RPC Time Out

```
ORA-28576: lost RPC connection to external procedure agent
ORA-06512: at "<name>", line <number>
ORA-06512: at "<name>", line <number>
ORA-06512: at line <number>
```

This error might occur after you exit a debugger while debugging a shared library or DLL. Simply disconnect your client and reconnect to the database.

Debugging External Procedures

Usually, when an external procedure fails, its C prototype is faulty. That is, the prototype does not match the one generated internally by PL/SQL. This can happen if you specify an incompatible C datatype. For example, to pass an `OUT` parameter of type `REAL`, you must specify float *. Specifying float, double *, or any other C datatype will result in a mismatch.

In such cases, you might get a lost RPC connection to external procedure agent error, which means that agent `extproc` terminated abnormally because the external procedure caused a core dump. To avoid errors when declaring C prototype parameters, refer to Table 4–2

Using Package DEBUG_EXTPROC

To help you debug external procedures, PL/SQL provides the utility package `DEBUG_EXTPROC`. To install the package, run the script `dbgextp.sql`, which you can find in the PL/SQL demo directory.

To use the package, follow the instructions in dbgextp.sql. Your Oracle account must have `EXECUTE` privileges on the package and `CREATE LIBRARY` privileges.

Note: `DEBUG_EXTPROC` works only on platforms with debuggers that can attach to a running process.

Debugging C Code in DLLs on Windows NT Systems

If you are developing on a Windows NT system, you may perform the following additional actions to debug external procedures:

1. Invoke the Windows NT Task Manager (press Ctrl+Alt+Del.and select Task Manager).
2. In the Processes display, select ExtProc.exe.
3. Right click, and select Debug.
4. Select OK in the message box.

   At this point, if you have built your DLL in a debug fashion with Microsoft Visual C++, Visual C++ is activated.
5. In the Visual C++ window, select Edit > Breakpoints.
6. Use the breakpoint identified in *dbgextp.sql* in the PL/SQL demo directory.

Guidelines for Using External Procedures with Data Cartridges

In future releases, `extproc` might be a multithreaded process. Therefore, be sure to write thread-safe external procedures. That way, they will continue to run properly if `extproc` becomes multithreaded. In particular, avoid using static variables, which can be shared by routines running in separate threads. Otherwise, you might get unexpected results.

For help in creating a dynamic link library, look in the RDBMS subdirectory /public, where a template makefile can be found.

When calling external procedures, never write to IN parameters or overflow the capacity of `OUT` parameters. (PL/SQL does no runtime checks for these error conditions.) Likewise, never read an `OUT` parameter or a function result. Also, always assign a value to `IN OUT` and `OUT` parameters and to function results. Otherwise, your external procedure will not return successfully.

If you include the `WITH CONTEXT` and `PARAMETERS` clauses, you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list. If you omit the `PARAMETERS` clause, the context pointer is the first parameter passed to the external procedure.

If you include the `PARAMETERS` clause and the external procedure is a function, you must specify the parameter `RETURN` (not `RETURN` *property*) in the last position.

For every formal parameter, there must be a corresponding parameter in the `PARAMETERS` clause. Also, make sure that the datatypes of parameters in the `PARAMETERS` clause are compatible with those in the C prototype because no implicit conversions are done.

A parameter for which you specify `INDICATOR` or `LENGTH` has the same parameter mode as the corresponding formal parameter. However, a parameter for which you specify `MAXLEN`, `CHARSETID`, or `CHARSETFORM` is always treated like an `IN` parameter, even if you also specify `BY REFERENCE`.

With a parameter of type `CHAR`, `LONG RAW`, `RAW`, or `VARCHAR2`, you must use the property `LENGTH`. Also, if that parameter is `IN OUT` or `OUT` and null, you must set the length of the corresponding C parameter to zero.

Java Methods

In order to utilize Java Data Cartridges, it is important that you know how to load Java class definitions, about how to call stored procedures, and about context management. For details on these issues, see Chapters 1 and 2 of the *Oracle9i Java Stored Procedures Developer's Guide.*. Information on ODCI classes can also be found in Chapter 15 of this manual.

5 Methods: Using PL/SQL

This chapter describes how to use PL/SQL to implement the methods of a data cartridge. Methods are procedures and functions that define the operations permitted on data defined using the data cartridge. Topics include:

Methods

A **method** is procedure or function that is part of the object type definition, and that can operate on the attributes of the type. Such methods are also called **member methods**, and they take the keyword MEMBER when you specify them as a component of the object type.

See the *Oracle9i Database Concepts* manual for information about:

- Method specification
- Method names
- Method name overloading

Map methods, which govern comparisons between object types, are discussed in the previous sections.

The following sections show simple examples of implementing a method, invoking a method, and referencing an attribute in a method. For further explanation and more detailed examples, see the chapter on object types in the *PL/SQL User's Guide and Reference*.

Implementing Methods

To implement a method, create the PL/SQL code and specify it within a CREATE TYPE BODY statement.

For example, consider the following definition of an object type named *rational_type*:

```
CREATE TYPE rational_type AS OBJECT
( numerator INTEGER,
  denominator INTEGER,
  MAP MEMBER FUNCTION rat_to_real RETURN REAL,
  MEMBER PROCEDURE normalize,
  MEMBER FUNCTION plus (x rational_type)
       RETURN rational_type);
```

The following definition is shown merely because it defines the function gcd, which is used in the definition of the normalize method in the CREATE TYPE BODY statement later in this section.

```
CREATE FUNCTION gcd (x INTEGER, y INTEGER) RETURN INTEGER AS
-- Find greatest common divisor of x and y. For example, if
-- (8,12) is input, the greatest common divisor is 4.
-- This will be used in normalizing (simplifying) fractions.
-- (You need not try to understand how this code works, unless
--  you are a math wizard. It does.)
--
   ans INTEGER;
BEGIN
   IF (y <= x) AND (x MOD y = 0) THEN
      ans := y;
   ELSIF x < y THEN
      ans := gcd(y, x);  -- Recursive call
   ELSE
      ans := gcd(y, x MOD y);  -- Recursive call
   END IF;
   RETURN ans;
END;
```

The following statement implements the methods (`rat_to_real`, `normalize`, and `plus`) for the object type `rational_type`:

```
CREATE TYPE BODY rational_type
( MAP MEMBER FUNCTION rat_to_real RETURN REAL IS
   -- The rat-to-real function converts a rational number to
   -- a real number. For example, 6/8 = 0.75
   BEGIN
      RETURN numerator/denominator;
   END;

   -- The normalize procedure simplifies a fraction.
   -- For example, 6/8 = 3/4
   MEMBER PROCEDURE normalize IS
      divisor INTEGER := gcd(numerator, denominator);
   BEGIN
      numerator := numerator/divisor;
      denominator := denominator/divisor;
   END;

   -- The plus function adds a specified value to the
   -- current value and returns a normalized result.
   -- For example, 1/2 + 3/4 = 5/4
   --
   MEMBER FUNCTION plus(x rational_type)
         RETURN rational_type IS
         -- Return sum of SELF + x
   BEGIN
      r = rational_type(numerator*x.demonimator +
             x.numerator*denominator,
             denominator*x.denominator);
```

```
                -- Example adding 1/2 to 3/4:
                -- (3*2 + 1*4) / (4*2)
      -- Now normalize (simplify). Here, 10/8 = 5/4
      r.normalize;
      RETURN r;
   END;
END;
```

Note: If an object type has no methods, no CREATE TYPE BODY statement for that object type is required.

Invoking Methods

To invoke a method, use the following syntax:

<object_name>.<method_name>([parameter_list])

In SQL statements only, you can use the following syntax:

<correlation_variable>.<method_name>([parameter_list])

The following PL/SQL example invokes a method named *get_emp_sal*:

```
DECLARE
   employee employee_type;
   salary number;
   ...
BEGIN
   salary := employee.get_emp_sal();
   ...
END;
```

An alternative way to invoke a method is by using the SELF built-in parameter. Because the implicit first parameter of each method is the name of the object on whose behalf the method is invoked, the following example performs the same action as the line after BEGIN in the preceding example:

```
salary := get_emp_sal(SELF => employee);
```

In this example, *employee* is the name of the object on whose behalf the get_emp_sal method is invoked.

Referencing Attributes in a Method

As shown in the example in “Implementing Methods” on page 3-1, member methods can reference the attributes and member methods of the same object type without using a qualifier. A built-in reference is always provided to the object on whose behalf the method is invoked. This reference is called SELF.

Consider the following trivial example, in which two statements set the value of variable *var1* to 42:

```
CREATE TYPE a_type AS OBJECT (
   var1 INTEGER,
   MEMBER PROCEDURE set_var1);
CREATE TYPE BODY a_type (
   MEMBER PROCEDURE set_var1 IS
   BEGIN
      var1 := 42;
      SELF.var1 := 42;
   END set_var1;
);
```

In this example, *var1 := 42* and *SELF.var1 := 42* are in effect the same statement. Because *var1* is the name of an attribute of the object type *a_type* and because *set_var1* is a member method of this object type, no qualification is required to access *var1* in the method code. However, for code readability and maintainability, you can use the keyword SELF in this context to make the reference to *var1* more clear.

PL/SQL Packages

A **package** is a group of PL/SQL types, objects, and stored procedures and functions. The **specification** part of a package declares the public types, variables, constants, and subprograms that are visible outside the immediate scope of the package. The **body** of a package defines the objects declared in the specification, as well as private objects that are not visible to applications outside the package.

The following example shows the package specification for the package named *DS_package*. This package contains the two stored functions *ds_findmin* and *ds_findmax*, which implement the *DataStreamMin* and *DataStreamMax* functions defined for the *DataStream* object type.

```
CREATE OR REPLACE PACKAGE DS_package AS
    FUNCTION  ds_findmin(data clob) RETURN pls_integer;
    FUNCTION  ds_findmax(data clob) RETURN pls_integer;
     PRAGMA restrict_references(ds_findmin, WNDS, WNPS);
    PRAGMA restrict_references(ds_findmax, WNDS, WNPS);
END;
```

For the *DataStream* type and type body definitions, see Chapter 2, "Roadmap to Building a Data Cartridge".

For more information about PL/SQL packages, see the chapter about using procedures and packages in the *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Pragma RESTRICT_REFERENCES

To execute a SQL statement that calls a member function, Oracle must know the **purity level** of the function, that is, the extent to which the function is free of side effects. The term **side effect**, in this context, refers to accessing database tables, package variables, and so forth for reading or writing. It is important to control side effects because they can prevent the proper parallelization of a query, produce order-dependent (and therefore indeterminate) results, or require impermissible actions such as the maintenance of package state across user sessions.

A member function called from a SQL statement can be restricted so that it cannot:

- Insert into, update, or delete database tables
- Be executed remotely or in parallel if it reads or writes the values of packaged variables
- Write the values of packaged variables unless it is called from a `SELECT`, `VALUES`, or `SET` clause
- Call another method or subprogram that violates any of these rules
- Reference a view that violates any of these rules

For more information about the rules governing purity levels and side effects, see the *PL/SQL User's Guide and Reference*.

You use the pragma (compiler directive) `RESTRICT_REFERENCES` to enforce these rules. For example, the purity level of the *DataStreamMax* method of type *DataStream* is asserted to be *write no database state* (`WNDS`) and *write no package state* (`WNPS`) in the following way:

```
CREATE TYPE DataStream AS OBJECT (
         ....
PRAGMA RESTRICT_REFERENCES (DataStreamMax, WNDS, WNPS)
         ... );
```

Member methods that call external procedures cannot do so directly but must route the calls through a package. The reason is that currently the arguments to external procedures cannot be object types. A member function automatically gets a SELF reference (a reference to that specific instance of the object type) as its first argument. Therefore, member methods in objects types cannot call out directly to external procedures.

Collecting all external calls into a package makes for a better design. The purity level of the package must also be asserted. Therefore, when the package named DS_Package is declared and all external procedure calls from type DataStream are routed through this package, the purity level of the package is also declared, as follows:

```
CREATE OR REPLACE PACKAGE DS_Package AS
   ...
PRAGMA RESTRICT_REFERENCES (ds_findmin, WNDS, WNPS)
   ...
END;
```

In addition to WNDS and WNPS, it is possible to specify two other constraints: *read no database state* (RNDS) and *read no package state* (RNPS). These two constraints are normally useful if you have parallel queries.

Each constraint is independent of the others and does not imply another. Choose the set of constraints based on application-specific requirements. For more information about controlling side effects using the RESTRICT_REFERENCES pragma, see the *Oracle9i Application Developer's Guide - Fundamentals*.

You can also specify the keyword DEFAULT instead of a method or procedure name, in which case the pragma applies to all member functions of the type (or procedures of the package). For example:

```
PRAGMA RESTRICT_REFERENCES (DEFAULT, WNDS, WNPS)
```

Privileges Required to Create Procedures and Functions

To create a standalone procedure or function, or package specification or body, you must have the CREATE PROCEDURE system privilege to create a procedure or package in your schema, or the CREATE ANY PROCEDURE system privilege to create a procedure or package in another user's schema.

For the compilation of the procedure or package, the *owner* of the procedure or package must have been explicitly granted the necessary object privileges for all objects referenced within the body of the code. *The owner cannot have obtained required privileges through roles.*

For more information about privilege requirements for creating procedures and functions, see the chapter about using procedures and packages in the *Oracle9i Application Developer's Guide - Fundamentals*.

Debugging PL/SQL Code

One of the simplest ways to debug PL/SQL code is to try each method, block, or statement interactively using SQL*Plus, and fix any problems before proceeding to the next statement. If you need more information on an error message, enter the statement `SHOW ERRORS`. Also consider displaying statements for runtime debugging, such as those of the general form:

```
Location in module: <location>
Parameter name: <name>
Parameter value: <value>
```

You can debug stored procedures and packages using the `DBMS_OUTPUT` package. You insert `PUT` and `PUTLINE` statements in your code to output the value of variables and expressions to your terminal. The `DBMS_OUTPUT` package is described in the *Oracle9i Supplied PL/SQL Packages and Types Reference* and the *PL/SQL User's Guide and Reference.*

To debug stored procedures and packages, *though not object type methods at present*, you can use Procedure Builder, which is a part of the Oracle Developer/2000 tool set. Procedure Builder lets you execute PL/SQL stored procedures and triggers in a controlled debugging environment, and you can set breakpoints, list the values of variables, and perform other debugging tasks. See the *Oracle9i Java Stored Procedures Developer's Guide*

A PL/SQL tracing tool provides more information about exception conditions in application code. You can use this tool to trace the execution of server-side PL/SQL statements. Object type methods cannot be traced directly, but you can trace any PL/SQL functions or procedures that a method calls. The tracing tool also provides information about exception conditions in the application code. The trace output is written to the Oracle server trace file.

Note: Only the database administrator has access to this trace file. The tracing tool is described in the *Oracle9i Application Developer's Guide - Fundamentals*.

Notes for C and C++ Programmers

If you are a C or C++ programmer, several PL/SQL conventions and requirements may differ from your expectations. Note the following about PL/SQL:

- = means equal (not assign).
- := means assign (as in Algol).
- `VARRAY`s begin at index 1 (not 0).
- Comments begin with two hyphens (--), not with // or /*.
- The `IF` statement requires the `THEN` keyword.
- The IF statement must be concluded with the `END IF` keyword (which comes after the `ELSE` clause, if there is one).
- There is no `PRINTF` statement. The comparable feature is the `DBMS_OUTPUT.PUT_LINE` statement. In this statement, literal and variable text is separated using the double vertical bar (||).
- A function must have a return value, and a procedure cannot have a return value.
- If you call a function, it must be on the right side of an assignment operator.
- Many PL/SQL keywords cannot be used as variable names.

See Also: *PL/SQL User's Guide and Reference.*

Common Potential Errors

This section presents several kinds of errors you may make in creating a data cartridge.

Signature Mismatches

```
13/19    PLS-00538: subprogram or cursor '<name>' is declared in an object
         type specification and must be defined in the object type body
15/19    PLS-00539: subprogram '<name>' is declared in an object type body
         and must be defined in the object type specification
```

If you see either or both of these messages, you have made an error with the signature for a procedure or function. In other words, you have a mismatch between the function or procedure prototype that you entered in the object specification, and the definition in the object body.

Ensure that parameter orders, parameter spelling (including case), and function returns are identical. Use copy-and-paste to avoid errors in typing.

RPC Time Out

```
ORA-28576: lost RPC connection to external procedure agent
ORA-06512: at "<name>", line <number>
ORA-06512: at "<name>", line <number>
ORA-06512: at line 34
```

This error might occur after you exit the debugger for the DLL. Restart the program outside the debugger.

Package Corruption

```
ERROR at line 1:
ORA-04068: existing state of packages has been discarded
ORA-04063: package body "<name>" has errors
ORA-06508: PL/SQL: could not find program unit being called
ORA-06512: at "<name>", line <number>
ORA-06512: at line <number>
```

This error might occur if you are extending an existing data cartridge; it indicates that the package has been corrupted and must be recompiled.

Before you can perform the recompilation, you must delete all tables and object types that depend upon the package that you will be recompiling. To find the dependents on a Windows NT system, use the Oracle Administrator toolbar. Click the Schema button, log in as *sys\change_on_install*, and find packages and tables that you created. Drop these packages and tables by entering SQL statements of the following form into the SQL*Plus interface:

```
DROP TYPE <type_name>;
DROP TABLE <table_name> CASCADE CONSTRAINTS;
```

The recompilation can then be done using a SQL statement of the following form:

```
ALTER TYPE <type_name> COMPILE BODY;
or
ALTER TYPE <type_name> COMPILE SPECIFICATION;
```

6 Working with Multimedia Datatypes

This chapter includes the following topics:

Overview

Some data cartridges need to handle large amounts of raw binary data, such as graphic images or sound waveforms, or character data, such as text or streams of numbers. Oracle supports large objects (LOBs) to handle these kinds of data.

Internal LOBs are stored in the database tablespaces in way that optimizes space and provides efficient access. Internal LOBs participate in the transactional model of the server. **External** LOBs are stored in operating system files outside the database tablespaces. External LOBs do not participate in transactions.

Internal LOBs can store binary data (BLOBs), single-byte character data (CLOBs), or fixed-width single-byte or multibyte character data (NCLOBs). An NCLOB consists of character data that corresponds to the national character set defined for the Oracle database. Varying-width character data is not supported in Oracle. External LOBs store only binary data (BFILEs). Together, internal and external LOBs provide considerable flexibility in handling large amounts of data.

Data stored in a LOB is called the LOB's **value**. To the Oracle server, a LOB's value is unstructured and cannot be queried. You must unpack and interpret a LOB's value in cartridge-specific ways.

LOBs can be manipulated using the Oracle Call Interface (OCI) or the PL/SQL DBMS_LOB package. You can write functions (including methods on object types that can contain LOBs) to manipulate parts of LOBs. Details on LOBs can be found in the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

DDL for LOBs

LOB definition can involve the CREATE TYPE and the CREATE TABLE statements. For example, the following statement specifies a CLOB within a datatype named *lob_type*:

```
CREATE OR REPLACE TYPE lob_type AS OBJECT (
        id  INTEGER,
        data CLOB );
```

The following statement creates an object table (*lob_table*) in which each row is an instance of *lob_type* data:

```
CREATE TABLE lob_table OF lob_type;
```

The following statement stores LOBs in a regular table, as opposed to an object table as in the preceding statement:

```
CREATE TABLE lob_table1  (
                id  INTEGER,
                b_lob   BLOB,
                c_lob   CLOB,
                nc_lob  NCLOB,
                b_file  BFILE );
```

When creating LOBs in tables, you can set the LOB storage, buffering, and caching properties. See the *Oracle9i SQL Reference* manual and the *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about using LOBs in the following DDL statements:

- CREATE TABLE and ALTER TABLE
  - LOB columns
  - LOB storage clause
  - NOCACHE and NOLOGGING options
- CREATE TYPE and ALTER TYPE
  - BLOB, CLOB and BFILE datatypes

LOB Locators

LOBs can be stored with other row data or separate from row data. Regardless of the storage location, each LOB has a **locator**, which can be viewed as a handle or pointer to the actual location. Selecting a LOB returns the LOB locator instead of the LOB value.

The following PL/SQL code selects the LOB locator for *b_lob* and place it a PL/SQL local variable named *image1*:

```
DECLARE
       image1  BLOB;
       image_no  INTEGER := 101;
BEGIN
       SELECT b_lob  INTO image1 FROM lob_table
                  WHERE key_value = image_no;
             ...
END;
```

When you use an API function to manipulate the LOB value, you refer to the LOB using the locator. The PL/SQL DBMS_LOB package contains useful routines to manipulate LOBs, such as PUT_LINE and GETLENGTH:

```
BEGIN
     DBMS_OUTPUT.PUT_LINE('Size of the Image is: ',
                          DBMS_LOB.GETLENGTH(image1));
END;
```

In the OCI, LOB locators are mapped to LOBLocatorPointers (*OCILobLocator* *).

The OCI LOB interface and the PL/SQL DBMS_LOB package are described briefly in this chapter. The OCI is described in more detail in the *Oracle Call Interface Programmer's Guide*. The DBMS_LOB API is described in the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

For a BFILE, the LOB column has its own distinct locator, which refers to the LOB's value that is stored in an external file in the server's file system. This implies that two rows in a table with a BFILE column may refer to the same file or two distinct files. A BFILE locator variable in a PL/SQL or OCI program behaves like any other automatic variable. With respect to file operations, it behaves like a file descriptor available as part of the standard I/O library of most conventional programming languages.

EMPTY_BLOB and EMPTY_CLOB Functions

You can use the special functions EMPTY_BLOB and EMPTY_CLOB in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. These are available as special functions in Oracle SQL DML, and are not part of the DBMS_LOB package.

Before you can start writing data to an internal LOB using OCI or the DBMS_LOB package, the LOB column must be made non-null, that is, it must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value to empty by using the function EMPTY_BLOB in the VALUES clause of an INSERT statement. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB.

Syntax

```
FUNCTION EMPTY_BLOB() RETURN BLOB;
FUNCTION EMPTY_CLOB() RETURN CLOB;
```

Note: The parentheses are required syntax for both functions.

Parameters

None.

Return Values

`EMPTY_BLOB` returns an empty locator of type `BLOB` and `EMPTY_CLOB` returns an empty locator of type `CLOB`, which can also be used for `NCLOBs`.

Pragma

None.

Exceptions

An exception is raised if you use these functions anywhere but in the `VALUES` clause of a SQL `INSERT` statement or as the source of the `SET` clause in a SQL `UPDATE` statement.

Examples

The following example shows `EMPTY_BLOB` used with SQL DML:

```
INSERT INTO lob_table VALUES (1001, EMPTY_BLOB(), 'abcde', NULL);
UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value = 1001;
INSERT INTO lob_table VALUES (1002, NULL, NULL, NULL);
```

The following example shows the correct and erroneous usage of `EMPTY_BLOB` and `EMPTY_CLOB` in PL/SQL programs:

```
DECLARE
  loba         BLOB;
  lobb         CLOB;
  read_offset  INTEGER;
  read_amount  INTEGER;
  rawbuf       RAW(20);
  charbuf      VARCHAR2(20);
BEGIN
  loba := EMPTY_BLOB();
  read_amount := 10; read_offset := 1;
  -- the following read will fail
  dbms_lob.read(loba, read_amount, read_offset, rawbuf);

  -- the following read will succeed;
  UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value =
         1002 RETURNING c_lob INTO lobb;
dbms_lob.read(lobb, read_amount, read_offset, charbuf);
  dbms_output.put_line('lobb value: ' || charbuf);
```

Using the OCI to Manipulate LOBs

The OCI includes functions that you can use to access data stored in BLOBs, CLOBs, NCLOBs, and BFILEs. These functions are mentioned briefly in Table 6–1. For detailed documentation, including parameters, parameter types, return values, and example code, see the *Oracle Call Interface Programmer's Guide*.

Table 6–1 OCI Functions for Manipulating LOBs

Function	Description
OCILobAppend()	Appends LOB value to another LOB.
OCILobAssign()	Assigns one LOB locator to another.
OCILobCharSetForm()	Returns the character set form of a LOB.
OCILobCharSetId()	Returns the character set ID of a LOB.
OCILobCopy()	Copies a portion of a LOB into another LOB.
OCILobDisableBuffering()	Disables the buffering subsystem use.
OCILobEnableBuffering()	Uses the LOB buffering subsystem for subsequent read and write operations of LOB data.
OCILobErase()	Erases part of a LOB, starting at a specified offset.
OCILobFileClose()	Closes an open BFILE.
OCILobFileCloseAll()	Closes all open BFILEs.
OCILobFileExists()	Tests to see if a BFILE exists.
OCILobFileGetName()	Returns the name of a BFILE.
OCILobFileIsOpen()	Tests to see if a BFILE is open.
OCILobFileOpen()	Opens a BFILE.
OCILobFileSetName()	Sets the name of a BFILE in a locator.
OCILobFlushBuffer()	Flushes changes made to the LOB buffering subsystem to the database (server)
OCILobGetLength()	Returns the length of a LOB or a BFILE.
OCILobIsEqual()	Tests to see if two LOB locators refer to the same LOB.
OCILobLoadFromFile()	Loads BFILE data into an internal LOB.
OCILobLocatorIsInit()	Tests to see if a LOB locator is initialized.
OCILobLocatorSize()	Returns the size of a LOB locator.

Table 6–1 OCI Functions for Manipulating LOBs (Cont.)

Function	Description
`OCILobRead()`	Reads a specified portion of a non-null `LOB` or a `BFILE` into a buffer.
`OCILobTrim()`	Truncates a `LOB`.
`OCILobWrite()`	Writes data from a buffer into a `LOB`, writing over existing data.

Table 6–2 compares the OCI and PL/SQL (`DBMS_LOB` package) interfaces in terms of `LOB` access.

Table 6–2 OCI and PL/SQL (DBMS_LOB) Interfaces Compared

OCI (ociap.h)	PL/SQL DBMS_LOB (dbmslob.sql)
N/A	DBMS_LOB.COMPARE()
N/A	DBMS_LOB.INSTR()
N/A	DBMS_LOB.SUBSTR()
OCILobAppend	DBMS_LOB.APPEND()
OCILobAssign	N/A [use PL/SQL assign operator]
OCILobCharSetForm	N/A
OCILobCharSetId	N/A
OCILobCopy	DBMS_LOB.COPY()
OCILobDisableBuffering	N/A
OCILobEnableBuffering	N/A
OCILobErase	DBMS_LOB.ERASE()
OCILobFileClose	DBMS_LOB.FILECLOSE()
OCILobFileCloseAll	DBMS_LOB.FILECLOSEALL()
OCILobFileExists	DBMS_LOB.FILEEXISTS()
OCILobFileGetName	DBMS_LOB.FILEGETNAME()
OCILobFileIsOpen	DBMS_LOB.FILEISOPEN()
OCILobFileOpen	DBMS_LOB.FILEOPEN()
OCILobFileSetName	N/A (use `BFILENAME` operator)

Table 6–2 OCI and PL/SQL (DBMS_LOB) Interfaces Compared (Cont.)

OCI (ociap.h)	PL/SQL DBMS_LOB (dbmslob.sql)
OCILobFlushBuffer	N/A
OCILobGetLength	DBMS_LOB.GETLENGTH()
OCILobIsEqual	N/A [use PL/SQL equal operator]
OCILobLoadFromFile	DBMS_LOB.LOADFROMFILE()
OCILobLocatorIsInit	N/A [always initialize]
OCILobRead	DBMS_LOB.READ()
OCILobTrim	DBMS_LOB.TRIM()
OCILobWrite	DBMS_LOB.WRITE()

The following example shows a LOB being selected from the database into a locator. This example assumes that the type *lob_type* has two attributes (*id* of type INTEGER and *data* of type CLOB) and that a table (*lob_table*) of this type (*lob_type*) has been created.

```
/*-----------------------------------------------------------------------*/
/* Select lob locators from a CLOB column */
/* We need the 'FOR UPDATE' clause because we need to write to the LOBs. */
/*-----------------------------------------------------------------------*/
static OCIEnv       *envhp;
static OCIServer    *srvhp;
static OCISvcCtx    *svchp;
static OCIError     *errhp;
static OCISession   *authp;
static OCIStmt      *stmthp;
static OCIDefine    *defnp1;
static OCIBind      *bndhp;

sb4 select_locator(int rowind)
{
  sword retval;
  boolean flag;
  int colc = rowind;
  OCILobLocator *clob;
  text  *sqlstmt = (text *)"SELECT DATA FROM LOB_TABLE WHERE ID = :1 FOR
UPDATE";

  if (OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
  {
    (void) printf("FAILED: OCIStmtPrepare() sqlstmt\n");
    return OCI_ERROR;
  }

  if (OCIStmtBindByPos(stmthp, bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
  {
    (void) printf("FAILED: OCIStmtBindByPos()\n");
    return OCI_ERROR;
  }


  if (OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
             (dvoid *) &clob, (sb4) -1, (ub2) SQLT_CLOB,
                (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT))
  {
    (void) printf("FAILED: OCIDefineByPos()\n");
    return OCI_ERROR;
  }

  /* Execute the select and fetch one row */
  if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT))
  {
    (void) printf("FAILED: OCIStmtExecute() sqlstmt\n");
    report_error();
    return OCI_ERROR;
  }

  /* Now test to see if the LOB locator is initialized */
  retval = OCILobLocatorIsInit(envhp, errhp, clob, &flag);
  if ((retval != OCI_SUCCESS) && (retval != OCI_SUCCESS_WITH_INFO))
  {
    (void) printf("Select_Locator --ERROR: OCILobLocatorIsInit(), retval =
%d\n", retval);
    report_error();
    checkerr(errhp, retval);
    return OCI_ERROR;
  }
  if (!flag)
  {
   (void) printf("Select_Locator --ERROR: LOB Locator is not initialized.\n");
    return OCI_ERROR;
  }


  return OCI_SUCCESS;
}
```

A sample program (populate.c) that uses the OCI to populate a CLOB with the contents of a file is included on the disk.

Using DBMS_LOB to Manipulate LOBs

The `DBMS_LOB` package can be used to manipulate `LOBs` from PL/SQL.

The routines that can modify `BLOB`, `CLOB`, and `NCLOB` values are:

- `APPEND()` -- append the contents of the source `LOB` to the destination `LOB`
- `COPY()` -- copy all or part of the source `LOB` to the destination `LOB`
- `ERASE()` -- erase all or part of a `LOB`
- `LOADFROMFILE()` -- load `BFILE` data into an internal `LOB`
- `TRIM()` -- trim the `LOB` value to the specified shorter length
- `WRITE()` -- write data to the `LOB` from a specified offset

The routines that read or examine `LOB` values are:

- `GETLENGTH()` -- get the length of the `LOB` value
- `INSTR()` -- return the matching position of the *nth* occurrence of the pattern in the LOB
- `READ()` -- read data from the `LOB` starting at the specified offset
- `SUBSTR()` -- return part of the `LOB` value starting at the specified offset

The read-only routines specific to `BFILEs` are:

- `FILECLOSE()` -- close the file
- `FILECLOSEALL()` -- close all previously opened files
- `FILEEXISTS()` -- test to see if the file exists on the server
- `FILEGETNAME()` -- get the directory alias and file name

- FILEISOPEN() -- test to see if the file was opened using the input BFILE locators
- FILEOPEN() -- open a file

The following example calls the TRIM procedure to trim a CLOB value to a smaller length is shown in the following example. This example assumes that the type *lob_type* has two attributes (*id* of type INTEGER and *data* of type CLOB) and that a table (*lob_table*) of this type (*lob_type*) has been created.

```
PROCEDURE Trim_Clob IS
        clob_loc  CLOB;
BEGIN
 -- get the LOB Locator
       SELECT data into clob_loc  FROM lob_table
       WHERE id  =  179 FOR UPDATE;
   -- call the TRIM Routine
       DBMS_LOB.TRIM(clob_loc, 834004);
       COMMIT;
END;
```

Because this example deals with CLOB data, the second argument (834004) to DBMS_LOB.TRIM specifies the number of characters. If the example dealt with BLOB data, this argument would be interpreted as the number of bytes.

LOBs in External Procedures

LOB locators can be passed as arguments to an external procedure. The corresponding C routine gets an argument of type OCILobLocator *. For example, a PL/SQL external procedure could be defined as:

```
FUNCTION DS_Findmin(data CLOB) RETURN PLS_INTEGER IS EXTERNAL
                   NAME "c_findmin" LIBRARY DS_Lib LANGUAGE C;
```

When this function is called, it invokes a routine (*c_findmin*) with the signature:

```
int   c_findmin (OCILobLocator *)
```

This routine in a shared library associated with *DS_Lib*. In order to use the pointer OCILobLocator * to get data from the LOB (for example, using OCILobRead()), you must reconnect to the database by performing a callback. External procedures and callbacks are discussed in “Doing Callbacks” on page 5-10.

LOBs and Triggers

You cannot write to a LOB (:old or :new value) in any kind of trigger.

In regular triggers, you can read the :old value but you cannot read the :new value. In INSTEAD OF triggers, you can read the :old and the :new values.

You cannot specify LOB type columns in an OF clause, because BFILE types can be updated without updating the underlying table on which the trigger is defined.

Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns will not fire triggers defined on the table containing the columns or the attributes.

Using Open/Close as Bracketing Operations for Efficient Performance

The Open/Close functions let you indicate the beginning and end of a series of LOB operations so that large-scale operations, such updating indexes, can be performed once the Close function is called. This means that once the Open call is made, the index would not be updated each time the LOB is modified, and that such updating would not resume until the Close call.

You do not have to wrap all LOB operations inside the Open/Close operations, but this function can be very useful for cartridge developers.

For one thing, if the you do not wrap LOB operations inside an Open/Close call, then each modification to the LOB will implicitly open and close the LOB, thereby firing any triggers. But if do you wrap the LOB operations inside a pair of Open/Close operations, then the triggers will not be fired for each LOB modification. Instead, one trigger will be fired at the time the Close call is made. LIkewise, extensible indexes will not be updated until the user calls Close. This means that any extensible indexes on the LOB are not valid between the Open/Close calls.

You need to apply this technology carefully since state, reflecting the changes to the LOB, is not saved between the Open and the Close operations. Once you have called Open, Oracle no longer keeps track of what portions of the LOB value were modified, nor of the old and new values of the LOB that result from any modifications. The LOB value is still updated directly for each OCILob* or DBMS_LOB operation, and the usual read consistency mechanism is still in place. Moreover, you may want extensible indexes on the LOB to be updated as LOB modifications are made because in that case, the extensible LOB indexes are always valid and may be used at any time.

The API enables you to find out if the LOB is "open" or not. In all cases openness is associated with the LOB, not the locator. The locator does not save any information as to whether the LOB to which it refers is open.

Errors and Restrictions Regarding Open/Close Operations

Note that it is an error to commit the transaction before closing all previously opened LOBs. At transaction rollback time, all LOBs that are still open will be discarded, which means that they will not be closed thereby firing the triggers).

Only 32 LOBs may be open at any one time. An error will be returned when the 33rd LOB is opened. Assigning an already opened locator to another locator does not incur a round trip to the server and does not count as opening a new LOB (both locators refer to the same LOB).

It is an error to Open/Close the same LOB twice either with different locators or with the same locator. It is an error to close a LOB that has not been opened.

Example

Assume loc1 is refers to an opened LOB and is assigned to loc2. If loc2 is subsequently used to modify the LOB value, the modification is grouped together with loc1's modifications (that is, there is only one entry in the LOB manager's state, not one for each locator). Once the LOB is closed (through loc1 or loc2), the triggers are fired and all updates made to the LOB through any locator are committed. After the close of the LOB, if the user tries to use either locator to modify the LOB, the operation will be performed as Open/operation/Close. Note that consistent read is still maintained for each locator. This discussion is merely showing that the LOB, not the locator, is opened and closed. No matter how many copies of the locator are made, the triggers for the LOB are fired only once on the first Close call.

For example:

```
open (loc1);
loc2 := loc1;
write (loc1);
write (loc2);
open (loc2);  /* error because the LOB is already open */
close (loc1); /* triggers are fired and all LOB updates made prior to this
                 statement by any locator are incorporated in the extensible
                 index */
write (loc2); /* implicit open, write, implicit close */
```

7

g Building Domain Indexes

This chapter describes extensible indexing, including:

Introduction to Extensible Indexing

What is extensible indexing? Why is it important to you as a cartridge developer? How should you go about implementing it?

To answer these questions we first need to understand the modes of indexing provided by the Oracle, which in turn requires that we first consider the role of indexing in information management systems.

What is Indexing?

The impetus to index data arises because of the need to locate specific information and then to retrieve it as efficiently as possible. If you could keep the entire dataset in main memory (equivalent to a person memorizing a book), there would be no need for indexing. Since this is not possible, and since disk access times are much slower than main memory access times, you are forced to wrestle with the art of indexing.

If you think of the form of indexing with which we are most familiar — the index at the back of a technical book — you will note that every index token has three characteristics which refer to the item being indexed:

- Identity — the token must allow us to identify the item in such a way that it is distinguished from the rest of the mass of the data. But this is not simply a representative relationship. By defining an index item you filter the information, implicitly providing a logical structure for the indexed information.

  This has many implications. For one, it means that the same data can be subject to different indexing schemes. For another, it means that the indexing scheme provides a pathway of access to the information. The index in the back of the book gives you access to the entire range of topics covered in the book. Provided that its structure meets your needs, its presorting of the data means that you do not have to sift through every iota of information.

- Location —the token must allow us to locate the information. In the case of a book, this is a page number, and may also include a chapter designation. This is not very precise since we still have to search the page for the item. In contrast to the normal index, conversation analysis makes use of line numbers because of the need for greater precision in locating the item:

  ```
  10296   HELEN: If you really loved me you wouldn't go to war.
  10297   PARIS: If you really loved me you wouldn't stand in the way of my
                 duty.
  ```

- Storage — the index token has to be located somewhere, and the information that it maps also has to be stored. In the case of books, a page is normally the unit of storage in both cases, but the nature of the storage is different. While the body text is stored as sentences, the index tokens have an altogether different structure.

The upshot is that you can retrieve the information much quicker than if you had to page through the entire book (equivalent to sequential scanning of a file)! However, note that while indexing speeds up retrieval, it slows down inserts because you have to update the index.

Index Structures

An index can be any structure which can be used to represent information that can be used to efficiently evaluate a query.

The Relationship between Logical and Physical Structures

There is no single structure that is optimal for all applications.

- If you want to discover if any `Regions` contain a city named Metropolis, you will deploy an equality operator that will return an exact match (or not).
- If you are interested how many time-periods have power demands between two stipulated numbers, you will use an operator that can process a range of data.

In each case, you will want to organize the data in a different index structure since different queries require that information be indexed in different ways. As we will discuss in the following sections, a Hash structure is best suited for determining exact match, whereas a B-tree is much better suited for range queries.

Moreover, these are not the only kind of queries. What if you want to discover whether Power Station A or B can best service Quadrant 3, or to determine the overlapping coverage zones derived from different distributions of power stations? In these cases, you will want to create operators (`inRangeOf`, `servesArea`, and so on) that meet your specific requirements. Unfortunately, you cannot do this by means of either Hash or B-tree indexes.

The Need for Index Structures that Encompass Unstructured Data

The limitation of Hash and B-tree indexes is important because one criterion that distinguishes cartridges from other database applications is that data often incorporates many different kinds of information. While database systems are accomplished in processing scalar values, they cannot encompass the domain-specific data of interest to cartridge developers. Information in these contexts may be made up of text, images, audio, video — and combinations of these that comprise domain-specific datatypes.

One way to resolve this problem is to create an index that serves as an intermediate structure. This is a logical extension of the basic idea underlying software-based indexing, namely that pointers refer to data (records, pages, files). In this scheme, keywords used to index video may be stored as an index. Going one step further, an intermediate structure may itself be indexed, as you might index abstracts (capsule text descriptions) of films.The advantage of this approach is that it may be easier to construct an index based on textual description of film than it is to index video footage. Employing this strategy you can scan the index without ever referring to the primary data (the film).

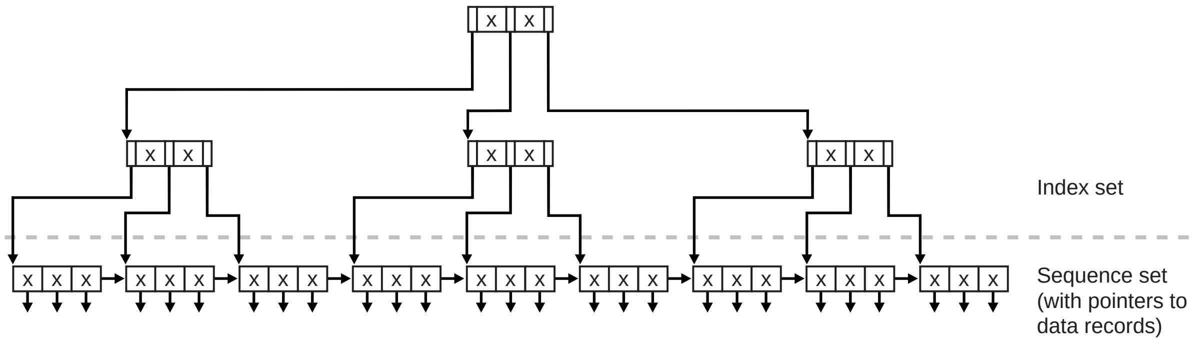
Unfortunately, intermediate structures in which text or scalars are used to represent unstructured data cannot satisfy all requirements. For one thing, they are always slower than direct indexing of the data because they introduce a level of indirection. More importantly, if the task is to analyze the density of bone in x-rays, or to categorize primate gestures, or to record the radio emissions of stars, there is no efficient substitute for direct indexing of unstructured data.

Kinds of Indexes

B-tree

While there is no single kind of index that can satisfy all needs, the B-tree index comes closest to meeting the requirement. Here we describe the Knuth variation in which the index consists of two parts: a sequence set that provides fast sequential access to the data, and an index set that provides direct access to the sequence set.

Figure 7–1 B-tree Index Structure



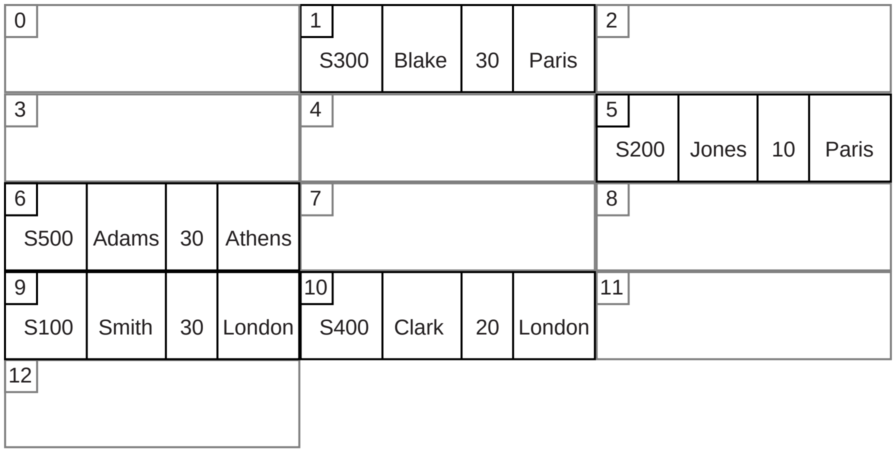
While the nodes of a B-tree will generally not contain the same number of data values, and will usually contain a certain amount of unused space, the B-tree algorithm ensures that it remains balanced (the leaf nodes will all be at the same level).

Hash

Hashing gives fast direct access to a specific stored record based on a given field value. Each record is placed at a location whose address is computed as some function of some field of that record. The same function is used both at the time of insertion and retrieval.

The problem with hashing is that the physical ordering of records has little if any relation to their logical ordering. Also, there may be large unused areas on the disk.

Figure 7–2 Hash Index Structure

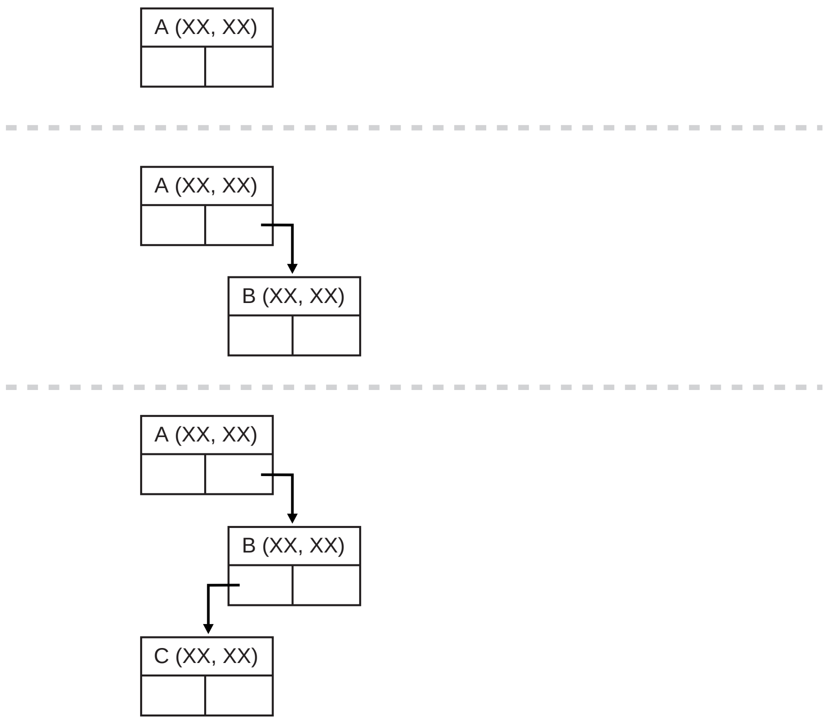


k-d tree

Our sample scenario integrates geographic data with other kinds of data. Insofar as we are interested in points that can be defined with two dimensions (latitude and longitude), such as geographic location of power stations, we can use a variation on the k-d tree known as the 2-d tree.

In this structure, each node is a datatype with fields for information, the two co-ordinates, a left-link and a right-link which can point to two children.

Figure 7–3 2-d Index Structure

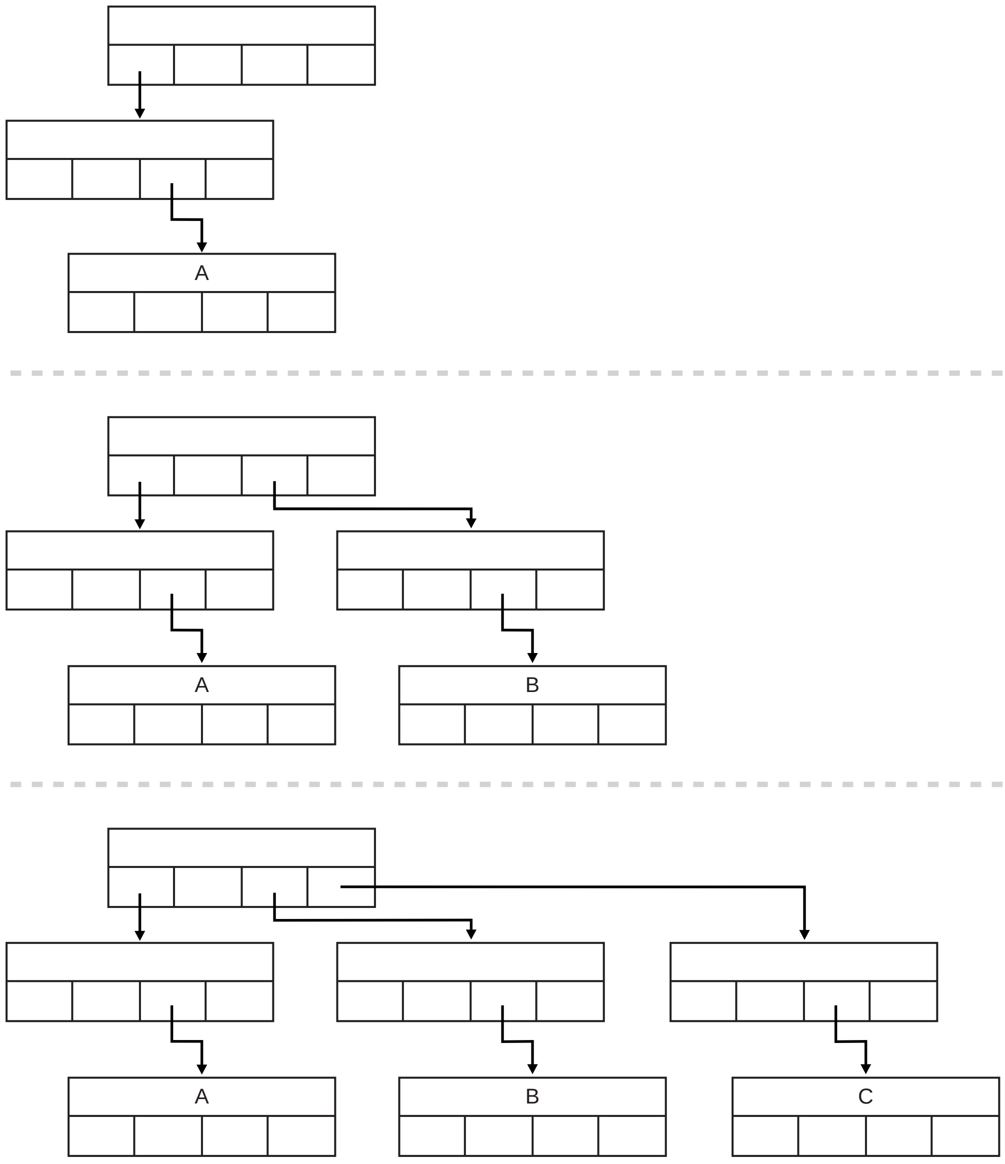


The structure allows for range queries. That is, if the user specifies a point (xx, xx) and a distance, the query will return the set of all points within the specified distance of the point.

2-d trees are very easy to implement. However,the fact that a 2-d tree containing *k* nodes may have a height of *k* means that insertion and querying may be complex.

Point Quadtree

Figure 7–4 Point Quadtree Index Structure



The point quadtree is also used to represent point data in a two dimensional spaces. But these structures divide regions into four parts while 2-d trees divide regions into two. The fields of the record type for this node are comprised of an attribute for information, two co-ordinates, and four compass points (NW, SW, NE, SE) that can therefore point to four children.

Like 2-d trees, point quadtrees are very easy to implement. Also like 2-d trees, the fact that a point quadtree containing *k* nodes may have a height of *k* means that insertion and querying may be complex. Each comparison requires comparisons on at least two co-ordinates. However, in practice the lengths from root to leaf tend to be shorter in point quadtrees.

Why is Extensible Indexing Necessary?

The fact is that Oracle provides a limited number of kinds of indexes, so that if (for instance) you wish to utilize either a k-d tree or the point quadtree, you will have to implement this yourself. As you consider your need to access your data, you need to keep in mind the following restrictions that pertain to the standard kinds of indexes:

Inability to Index Unstructured Data

Oracle's standard modes of indexing do not permit indexing a column that contains `LONG` or `LOB` values.

Inability to Index Attributes of Column Objects

You may not be able to index a column object using Oracle's standard indexing schemes or the elements of a collection type.

Inability to Index Values Derived from Domain-specific Operations

Oracle object types may be compared using either a map function or an order function. If the object utilizes a map function, then you can define a function-based index that can be used implicitly to evaluate relational predicates. However, if an order function is used, you will not be able to use this to construct an index.

Further, you cannot utilize functions in predicates in which the range of the parameters is infinite. Function-based indexes allow you to include a function in a predicate, provided you can precompute the function values for all the rows. Typically the index would store the rowid and the functional value. Queries that apply relational operators to values based on derived values utilize the index.

However, you can use function-based indexes only if the function is so designed that there are a finite number of input combinations. Put another way: you cannot use function-based indexes in cases in which the input parameters do not have a limited cardinality.

The Extensible Indexing API

This SQL-based interface lets you define domain-specific operators and indexing schemes, and integrate these into the Oracle server.

Oracle provides a set of pre-defined operators which include arithmetic operators (+, -, *, /), comparison operators (=, >, <) and logical operators (`NOT`, `AND`, `OR`). These operators take as input one or more arguments (or operands) and return a result. They are represented by special characters (+) or keywords (`AND`).

Like built-in operators, user-defined operators (such as `Contains`) take a set of operands as input and return a result. The implementation of the operator is provided by the user. After a user has defined a new operator, it can be used in SQL statements like any other built-in operator.

For instance, suppose you define a new operator `Contains`, which takes as input a text document and a keyword, and returns `1` if the document contains the specified keyword. You can then write an SQL query as:

```
SELECT * FROM Employees WHERE Contains(resume, 'Oracle and UNIX')=1;
```

Oracle uses indexes to efficiently evaluate some built-in operators. For example, a B-tree index can be used to evaluate the comparison operators =, > and <. Similarly, user-defined domain indexes can be used to efficiently evaluate user-defined operators.

Typical database management systems support a few types of access methods (B+Trees, Hash Index) on some set of data types (numbers, strings, and so on). In recent years, databases are more and more being used to store different types of data, such as text, spatial, image, video and audio. In these complex domains, there is a need for indexing complex data types and also specialized indexing techniques. For instance, R-trees are an efficient method of indexing spatial data. No database server can be built with support for all possible kinds of complex data and indexing. The solution is to provide an extensible server which lets the user define new index types.

The framework to develop new index types is based on the concept of cooperative indexing where an application and the Oracle server cooperate to build and maintain indexes for data types such as text, spatial and On-line-Analytical

Processing (OLAP). The application software, in the form of a cartridge, is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The index structure itself can either be stored in an Oracle database as an Index-Organized Table, or externally as a file.

The extensible indexing framework consists of the following components:

- **Indextype**: A schema object Indextype specifies the routines that manage all aspects of an application-specific index, namely, index definition, index maintenance, and index scan operations. This schema object enables the Oracle Server to establish a user-defined index on a column of a table or attribute of an Object. It encapsulates the set of routines that together manage and access the user-defined index.
- **Domain Index**: Using the Indextype schema object, an application-specific index can be created. Such an index is called a domain index since it is used for indexing data in application-specific domains. A domain index is an instance of an index which is created, managed, and accessed by routines supplied by an indextype. This is in contrast to B-tree indexes maintained by Oracle internally, which are simply referred to as indexes.
- **Operators**: Queries and data manipulation statements can involve application-specific operators, like the `Overlaps` operator in the spatial domain. In general, user-defined operators can be bound to functions. However, operators can also be evaluated using indexes. For instance, the equality operator can be evaluated using a hash index. An indextype provides index-based implementation for the operators listed in the indextype definition.
- **Index-Organized tables**: This feature enables applications to define, build, maintain, and access indexes for complex objects using a table metaphor. To the application, an index is modeled as a table, where each row is an index entry. In addition, this feature extends the current sorted access method to handle indexing content-rich objects by providing improved handling of duplicate index entries. For detailed information on index-organized tables see *Oracle9i Database Administrator's Guide.*

To illustrate the role of each of these components, let us consider a text domain application. Suppose a new indextype `TextIndexType` be defined as part of the text cartridge. It contains routines for managing and accessing the text index. The text index is an inverted index storing the occurrence list for each token in each of the text documents. The text cartridge also defines the `Contains` operator for performing content-based search on textual data. It provides both a functional implementation (a simple number function) and an index implementation (using the text index) for the Contains operator.

Now, let `Employees` be an employee table with a `resume` column containing textual data.

```
CREATE TABLE Employees
(name VARCHAR(128), id INTEGER, resume VARCHAR2(1024));
```

A domain index can be created on resume column as follows:

```
CREATE INDEX ResumeTextIndex ON Employees(resume)
INDEXTYPE IS TextIndexType;
```

The Oracle server invokes the routine corresponding to the create method in the `TextIndexType`, which results in the creation of an index-organized table to store the occurrence list of all tokens in the resumes (essentially, the inverted index data). The inverted index modeled by `ResumeTextIndex` is automatically maintained by invoking routines defined in `TextIndexType`, whenever an `Employees` row is inserted, updated, or deleted.

Content-based search on the resume column can be performed as follows:

```
SELECT * FROM Employees WHERE Contains(resume, ’Oracle and UNIX’)=1;
```

Index-based implementation of the `Contains` operator can take advantage of the previously built inverted index. Specifically, the Oracle server can invoke routines specified in `TextIndexType` to search the domain index for identifying candidate rows, and then do further processing such as filtering, selection, and fetching of rows. Note that the preceding query can also be evaluated using the non-index implementation of the `Contains` operator, if the Oracle server chooses to not use the index defined on resume column. In such a case, the filtering of rows will be done by applying the non-index implementation on each resume instance of the table.

In summary, the extensible indexing interface will

- Allow encapsulating application-specific index management routines as an indextype schema object,
- Support defining a domain index (an application-specific index) on table columns, and
- Provide efficient processing of application-specific operators.

This interface will enable a domain index to operate essentially the same way as any other Oracle Server index, the primary difference being that the Oracle Server will invoke application code specified as part of the indextype to create, drop, truncate, modify, and search a domain index.

It should be noted that an index designer may choose to store the index data in files, rather than in index-organized tables. The SQL interface for extensible indexing makes no restrictions on the location of the index data, only that the application adhere to the protocol for index definition, maintenance and search operations.

Concepts: Extensible Indexing

This section describes the key concepts of the Extensible Indexing Framework.

Overview

For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips and other complex data types that require content-based retrieval (CBR). The essential reason is that complex data types have application specific formats, indexing requirements, and selection predicates. For example, there are many different document encodings (such as ODA, SGML, plain text) and information retrieval (IR) techniques (keyword, full-text boolean, similarity, probabilistic, and so on). To effectively accommodate the large and growing number of complex data objects, the database system must support application specific indexing. The approach that we employ to satisfy this requirement is termed *extensible indexing*.

With Extensible indexing,

- The application defines the structure of the domain index
- The application stores the index data either inside the Oracle database (for example, in the form of index-organized tables) or outside the Oracle database
- The application manages, retrieves and uses the index data to evaluate user queries

In effect, the application controls the structure and semantic content of the domain index. The database system interacts with the application to build, maintain, and employ the domain index. It is highly desirable for the database to handle the physical storage of domain indexes. In the following discussion, we implicitly make the assumption that the index is stored in an index-organized table. Note however, that the extensible indexing paradigm does not impose this requirement. The index could be stored in one or more external files.

To illustrate the notion of extensible indexing, we consider a textual database application with IR functionality. For such applications, document indexing involves parsing the text and inserting the words, or tokens, into an inverted index. Such index entries typically have the following logical form

(token, <docid, data>)

where *token* is the key, *docid* is a unique identifier (such as object identification) for the related document, and *data* is a segment containing IR specific quantities. For example, a probabilistic IR scheme could have a data segment with token frequency and occurrence list attributes. The occurrence list identifies all locations within the related document where the token appears. Assuming an IR scheme such as this, each index entry would be of the form:

(token, <docid, frequency, occlist> ..)

The following sample index entry for the token Archimedes illustrates the associated logical content.

```
(Archimedes, <5, 3, [7 62 225]>, <26, 2, [33, 49]>, ...);
```

In this sample index entry, the token "Archimedes" appears in document 5 at 3 locations(7, 62, and 225), and in document 26 at 2 locations(33 and 49). Note that the index would contain one entry for every document with the word "Archimedes".

IR applications can use domain indexes to locate documents that satisfy some given selection criteria. After consulting the index, the documents of interest are retrieved with the related *docid* values. It should be noted that the occurrence lists are required for queries that contain proximity expressions (for example, the phrase "Oracle Corporation").

When the database system handles the physical storage of domain indexes, applications must be able to:

- Define the format and content of an index. This enables applications to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. With this capability, the application software handles building and maintaining the index structures. Note that this is a significant departure from the "automatic" indexing features provided for simple SQL data types. Also, since an index is modeled as a collection of tuples, in-place updating is directly supported.
- Access and interpret the content of an index. This capability enables the application software to become an integral component of query processing.

That is, the content-related clauses for database queries are handled by application software.

In the following section, we illustrate the extensible indexing framework by building a text domain index.

Example: A Text Indextype

This section presents an example of adding a text indexing scheme to Oracle RDBMS using the extensible indexing framework. It describes:

- Defining a new indexing scheme using text indextype.
- Use of text indextype by the end user to index and operate on textual data.

Text Indextype Designer

'The sequence of steps required to define the Text Indextype are:

- Define and code functions to support functional implementation of operators which would eventually be supported by the text indextype.

The text cartridge intends to support an operator `Contains`, that takes as parameters a text value and a key and returns a number value indicating whether the text contained the key. The functional implementation of this operator is a regular function defined as:

```
CREATE FUNCTION TextContains(Text IN VARCHAR2, Key IN VARCHAR2)
RETURN NUMBER AS
BEGIN
.......
END TextContains;
```

- Create a new operator, and define its specification, namely, the argument and return datatypes, and the functional implementation

```
CREATE OPERATOR Contains
 BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER USING TextContains;
```

- Define a type that implements the index interface `ODCIIndex`. This involves implementing routines for index definition, index maintenance, and index scan operations.

  The index definition routines (`ODCIIndexCreate`, `ODCIIndexAlter`, `ODCIIndexDrop`, `ODCIIndexTruncate`) build the text index when index is created, alter the index information when index is altered, remove the index information when the index is dropped, and truncate the text index when the base table is truncated.

The index maintenance routines (`ODCIIndexInsert`, `ODCIIndexDelete`, `ODCIIndexUpdate`) maintain the text index when the table rows are inserted, deleted, or updated.

The index scan routines (`ODCIIndexStart`, `ODCIIndexFetch`, `ODCIIndexClose`) implement access to the text index to retrieve rows of the base table that satisfy the operator predicate. In this case, the `Contains`(...) =1, whose arguments are passed to the index scan routines. The index scan routines scan the text index and return the qualifying rows to the system.

```
CREATE TYPE TextIndexMethods
(
FUNCTION ODCIIndexCreate(...)
...
);
CREATE TYPE BODY TextIndexMethods
(
...
);
```

- Create the `Text Indextype` schema object. The Indextype definition also specifies all the operators supported by the new indextype and specifies the type that implements the index interface.

```
CREATE INDEXTYPE TextIndexType
FOR Contains(VARCHAR2, VARCHAR2)
USING TextIndexMethods;
```

End User of Text Indextype

Suppose that the text indextype presented in the previous section has been defined in the system. You can define text indexes on text columns and use the associated `Contains` operator to query text data.

Consider the `Employees` table defined as follows:

```
CREATE TABLE Employees
(name VARCHAR2(64), id INTEGER, resume VARCHAR2(2000));
```

A text domain index can be built on the resume column as follows:

```
CREATE INDEX ResumeIndex ON Employees(resume) INDEXTYPE IS TextIndexType;
```

The text data in the resume column can be queried as:

```
SELECT * FROM Employees WHERE Contains(resume, 'Oracle') =1;
```

The query execution will use the text index on `resume` to efficiently evaluate the `Contains` predicate.

The following sections describe the concepts of indextypes, domain indexes and operators in greater detail.

Indextypes

The purpose of an indextype is to enable efficient search and retrieval functions for complex domains such as text, spatial, image, and OLAP using external software. An indextype is analogous to the sorted or bit-mapped indextype that are supplied internally within the Oracle Server. The essential difference is that the implementation for an indextype is provided by application software, as opposed to the Oracle Server internal routines.

Interface A set of routine specifications. It does not refer to a separate schema object but rather a logical set of documented method specifications.

ODCIIndex Interface The set of index definition, maintenance and scan routine specifications.

The interface specifies all the routines which have to be implemented by the index designer. The routines are implemented as type methods.

Creating Indextypes

After the type implementing the `ODCIIndex` interface has been defined, a new indextype can be created by specifying the list of operators supported by the indextype and referring to the type that implements the index interface.

Using the information retrieval example, the DDL statement for defining the new indextype `TextIndexType` which supports the `Contains` operator and whose implementation is provided by the type `TextIndexMethods` (implemented in the previous section) is as follows:

```
CREATE INDEXTYPE TextIndexType
        FOR Contains (VARCHAR2, VARCHAR2)
        USING TextIndexMethods;
```

In addition to the `ODCIIndex` interface routines, the implementation type must always implement the `ODCIGetInterfaces` routine. This function returns the list of names of the interface routines implemented by the type and tells the server the version of these routines. The `ODCIGetInterfaces` routine is invoked by Oracle when `CREATE INDEXTYPE` is executed. If the indextype implements the Oracle9*i* version of the routines, ODCIGetInterfaces must specify 'SYS.ODCIINDEX2' in the OUT parameter. If the indextype implements the Oracle8*i* version of the routines, ODCIGetInterfaces must specify 'SYS.ODCIINDEX1' in the OUT parameter. (The Oracle8*i* routines lack the ODCIEnv parameter added to many of the routines in Oracle9*i*.)

Dropping Indextypes

A corresponding DROP statement is supported to remove the definition of an indextype. For our example, this statement would be of the following form:

```
DROP INDEXTYPE TextIndexType;
```

The default DROP behavior is DROP RESTRICT semantics, that is, if one or more domain indexes exist that uses the indextype then the DROP operation is disallowed. User can override the default behavior with the FORCE option, which drops the indextype and marks dependent domain indexes (if any) invalid. For more details on object dependencies and drop semantics see "Object Dependencies, Drop Semantics, and Validation" on page 7-40.

Commenting on Indextypes

The COMMENT statement can be used to supply information about an indextype or operator. For example:

```
COMMENT ON INDEXTYPE
Ordsys.TextIndexType IS 'implemented by the type TextIndexMethods to support the
Contains operator';
```

Comments on indextypes can be viewed in these data dictionary views:

- USER_INDEXTYPE_COMMENTS
- ALL_INDEXTYPE_COMMENTS
- DBA_INDEXTYPE_COMMENTS

To place a comment on an indextype, the indextype must be in your own schema or you must have the COMMENT ANY INDEXTYPE privilege.

ODCI Index Interface

The ODCIIndex (Oracle Data Cartridge Interface Index) interface consists of the following classes of methods:

- Index Definition methods

- Index Maintenance methods
- Index Scan methods
- Index Metadata method

Index Definition Methods

Index definition methods allow specification of CREATE, ALTER, DROP, and TRUNCATE behaviors.

ODCIIndexCreate

The ODCIIndexCreate procedure is called when a CREATE INDEX statement is issued that references the indextype. Upon invocation, any physical parameters specified as part of the CREATE INDEX... PARAMETERS (...) statement are passed in along with the description of the index.

A typical action of this procedure is to create tables / files to store index data. Further, if the base table is not empty, this routine should build the index for the existing data in the indexed columns.

ODCIIndexAlter

The ODCIIndexAlter procedure is invoked when a domain index is altered using an ALTER INDEX statement. The description of the domain index to be altered is passed in along with any specified physical parameters.

In addition, this procedure is allowed to handle ALTER with REBUILD option, which supports rebuilding of domain index. The precise behavior in these two cases is defined by the person who implements indextype.

The ODCIIndexAlter routine is also invoked when a domain index is renamed using the ALTER INDEX ... RENAME command.

ODCIIndexTruncate

The ODCIIndexTruncate procedure is called when a TRUNCATE statement is issued against a table that contains a column or OBJECT type attribute indexed by the indextype. After this procedure executes, the domain index should be empty.

ODCIIndexDrop

The ODCIIndexDrop procedure is invoked when a domain index is destroyed using a DROP INDEX statement.

Index Maintenance Methods

Index maintenance methods allow specification of index `INSERT`, `UPDATE`, and `DELETE` behaviors.

ODCIIndexInsert

The `ODCIIndexInsert` procedure in the indextype is called when a record is inserted in a table that contains columns or `OBJECT` attributes indexed by the indextype. The new values in the indexed columns are passed in as arguments along with the corresponding row identifier.

ODCIIndexDelete

The `ODCIIndexDelete` procedure in the indextype is called when a record is deleted from a table that contains columns or `OBJECT` attributes indexed by the indextype. The old values in the indexed columns are passed in as arguments along with the corresponding row identifier.

ODCIIndexUpdate

The `ODCIIndexUpdate` procedure in the indextype is called when a record is updated in a table that contains columns or `OBJECT` attributes indexed by the indextype. The old and new values in the indexed columns are passed in as arguments along with the row identifier.

Index Scan Methods

Index scan methods allow specification of an index-based implementation for evaluating predicates containing operators.

An index scan is specified through three routines, `ODCIIndexStart`, `ODCIIndexFetch`, and `ODCIIndexClose`. These perform initialization, fetch rows (or identifiers of rows) that satisfy the predicate, and clean up after all rows are returned.

ODCIIndexStart

`ODCIIndexStart`() is invoked to initialize any data structures and start an index scan. The index related information and the operator related information are passed in as arguments.

A typical action performed when `ODCIIndexStart`() is invoked is to parse and execute SQL statements that query the tables storing the index data. It could also generate some set of result rows to be returned later when `ODCIIndexFetch`() is invoked.

Since the index and operator related information are passed in as arguments to `ODCIIndexStart`() and not to the other index scan routines (`ODCIIndexFetch`() and `ODCIIndexClose`()), any information needed in the later routines must be saved. This is referred to as the **state** that has to be shared among the index scan routines. There are two ways of doing this:

- **Return State**: If the state to be maintained is small, it can be returned back to Oracle RDBMS through an output SELF argument.
- **Return Handle**: If the state to be maintained is large (for example, a subset of the results), cursor-duration memory can be allocated to save the state. In this case, a handle to the memory can be returned to Oracle RDBMS through the output `SELF` parameter.

> **See Also:** The chapter on Cartridge Services in the *Oracle Call Interface Programmer's Guide* for information on memory services and maintaining context

In both cases, Oracle RDBMS will pass the `SELF` value to subsequent `ODCIIndexFetch`() and `ODCIIndexClose`() calls which can then use the value to access the relevant context information.

There are two modes of evaluating the operator predicate to return the result set of rows.

- **Precompute All**: Compute the entire result set in `ODCIIndexStart`(). Iterate over the results returning a row at a time in `ODCIIndexFetch`(). This mode is required for operators involving some sort of ranking over the entire collection. Evaluating such operators would require looking at the entire result set to compute the ranking, relevance, and so on for each candidate row.
- **Incremental Computation**: Compute one result row at a time as part of `ODCIIndexFetch`(). This mode is applicable for operators which can determine the candidate rows one at a time without having to look at the entire result set.

The choice of evaluating modes as well as what gets saved is left to the index designer. In either case, the Oracle RDBMS simply executes the `ODCIIndexStart`() routine as part of processing query containing operators which returns the context as an output `SELF` value.The returned value is passed back to subsequent `ODCIIndexFetch`() and `ODCIIndexClose`() calls.

ODCIIndexFetch

`ODCIIndexFetch`() returns the "next" row identifier of the row that satisfies the operator predicate.The operator predicate is specified in terms of the operator expression (name and arguments) and a lower and upper bound on the operator return values. Thus, a `ODCIIndexFetch`() call returns the row identifier of the rows for which the operator return value falls within the specified bounds. A `NULL` is returned to indicate end of index scan. The fetch method supports returning a batch of rows in each call. The state returned by `ODCIIndexStart`() or a previous call to `ODCIIndexFetch`() is passed in as an argument.

ODCIIndexClose

`ODCIIndexClose`() is invoked when the cursor is closed or reused. In this call the Indextype can perform any clean-ups or other needed functions. The current state is passed in as an argument.

Index Metadata Method

The optional `ODCIIndexGetMetadata` routine, if it is implemented, is called by the export utility to write implementation-specific metadata into the export dump file. This metadata might be policy information, version information, individual user settings, and so on, which are not stored in the system catalogs. The metadata is written to the dump files as anonymous PL/SQL blocks that get executed at import time immediately prior to the creation of the associated index.

Transaction Semantics during Index Method Execution

The index interface routines (with the exception of index definition methods, namely, `ODCIIndexCreate()`, `ODCIIndexAter()`, `ODCIIndexTruncate()`, `ODCIIndexDrop()`) are invoked under the same transaction that triggered these actions. Thus, the changes made by these routines are atomic and are committed or aborted based on the parent transaction. To achieve this, there are certain restrictions on the nature of the actions that can be performed in the different indextype routines.

- Index definition routines have no restrictions.
- Index maintenance routines can only execute DML statements. However, the DML statements cannot update the base table on which the domain index is created.
- Index scan routines can only execute SQL query statements.

For example, if an `INSERT` statement caused the `ODCIIndexInsert()` routine to be invoked, `ODCIIndexInsert()` runs under the same transaction as `INSERT`.

The `ODCIIndexInsert()` routine can execute any number of DML statements (for example, insert into index-organized tables). If the original transaction aborts, all the changes made by the indextype routines are rolled back.

However, if the indextype routines cause changes external to the database (like writing to external files), transaction semantics are not assured.

Transaction Semantics for Index Definition Routines

The index definition routines do not have any restrictions on the nature of actions within them. Consider `ODCIIndexCreate()` to understand this difference. A typical set of actions to be performed in `ODCIIndexCreate()` could be:

1. Create an index-organized table
2. Insert data into the index-organized table
3. Create a secondary index on a column of the index-organized table

To allow `ODCIIndexCreate()` to execute an arbitrary sequence of DDL and DML statements, we consider each statement to be an independent operation. Consequently, the changes made by `ODCIIndexCreate()` are not guaranteed to be atomic. The same is true for other index-definition routines.

Consistency Semantics during Index Method Execution

The index maintenance (and scan routines) execute with the same snapshot as the top level SQL statement performing the DML (or query) operation. This enables the index data processed by the index method to be consistent with the data in the base tables.

Privileges During Index Method Execution

Indextype routines always execute as the owner of the index. To support this, the index access driver will dynamically change user mode to index owner before invoking the indextype routines.

For certain operations, indextype routines may require to store information in tables owned by indextype designer. Indextype implementation must code those actions in a separate routine which will be executed using definer’s privileges. For more information on syntax, see CREATE TYPE in the *Oracle9i SQL Reference.*

Domain Indexes

This section describes the domain index operations and how metadata associated with the domain index can be obtained.

Domain Index Operations

Creating a Domain Index

A domain index can be created on a column of a table just like a B-tree index. However, an indextype must be explicitly specified. For example:

```
CREATE INDEX ResumeTextIndex ON Employees(resume)
INDEXTYPE IS TextIndexType
PARAMETERS (':Language English :Ignore the a an');
```

The `INDEXTYPE` clause specifies the indextype to be used. The `PARAMETERS` clause identifies any parameters for the domain index, specified as a string. This string is passed uninterpreted to the `ODCIIndexCreate` routine for creating the domain index. In the preceding example, the parameters string identifies the language of the text document (thus identifying the lexical analyzer to use) and the list of stop words which are to be ignored while creating the text index.

Altering a Domain Index

A domain index can be altered using `ALTER INDEX` statement. For example:

```
ALTER INDEX ResumeTextIndex PARAMETERS (':Ignore on');
```

The parameter string is passed uninterpreted to `ODCIIndexAlter()` routine, which takes appropriate actions to alter the domain index. In the preceding example, additional stop words to ignore in the text index are specified.

The `ALTER` statement can be used to rename a domain index.

```
ALTER INDEX ResumeTextIndex RENAME TO ResumeTIdx;
```

The `ODCIIndexAlter()` routine is invoked, which takes appropriate actions to rename the domain index.

In addition, the `ALTER` statement can be used to rebuild a domain index.

```
ALTER INDEX ResumeTextIndex REBUILD PARAMETERS (':Ignore of');
```

The same `ODCIIndexAlter()` routine is called but with additional information about the `ALTER` option.

When the end user executes an `ALTER INDEX` *`<domain_index>`* `UPDATE BLOCK REFERENCES` for a domain index on an index-organized table (IOT), `ODCIIndexAlter()` is called with the `AlterIndexUpdBlockRefs` bit set. This gives the cartridge developer the opportunity to update guesses (as to the block locations of rows) stored in the domain index in logical rowids.

Truncating a Domain Index

There is no explicit statement for truncating a domain index. However, when the corresponding table is truncated the truncate procedure specified as part of the indextype is invoked. For example:

```
TRUNCATE TABLE Employees;
```

will result in truncating `ResumeTextIndex` by calling `ODCIIndexTruncate`() routine.

Dropping a Domain Index

To drop an instance of a domain index, the `DROP INDEX` statement is used. For our example, this statement would be of the form:

```
DROP INDEX ResumeTextIndex;
```

This results in calling the `ODCIIndexDrop`() routine and passing information about the index.

Domain Indexes on Index-Organized Tables

Storing rowids in a UROWID column When the base table of a domain index is an index-organized table, and you want to store rowids for the base table in a table of your own, you should store the rowids in a `UROWID` column if you will be testing rowids for equality.

If the rowids are stored in a `VARCHAR` column instead, comparisons for equality of the text of the rowids from the base table and your own table will fail even when the rowids pick out the same row. This is because a domain index on an index-organized table uses logical instead of physical rowids, and, unlike physical rowids, logical rowids for the same row can have different textual representations. (Two logical rowids are equivalent when they have the same primary key, regardless of the guesses stored with them.)

A `UROWID` (univeral rowid) column can contain both physical and logical rowids. Storing rowids for an IOT in a `UROWID` column ensures that the equality operator will succeed on two logical rowids that have the same primary key information but different primary keys.

If you create an index storage table with a rowid column by performing a CREATE TABLE AS SELECT from the IOT base table, then a UROWID column of the correct size is created for you in your index table. If you create a table with a rowid column, then you need to explicitly declare your rowid column to be of type UROWID(x), where x is the size of the UROWID column. The size chosen should be large enough to hold any rowid from the base table and so should be a function of the primary key from the base table. The following query can be used to determine a suitable size for the UROWID column:

```
SELECT (SUM(column_length + 3) + 7)
FROM user_ind_columns ic, user_indexes i
WHERE ic.index_name = i.index_name
AND i.index_type = IOT - TOP
AND ic.table_
name = <base_table>;
```

You can use the IndexOnIOT bit of IndexInfoFlags in the ODCIIndexInfo structure to determine if the base table is an IOT.

Doing an ALTER INDEX REBUILD on index storage tables raises the same issues as doing a CREATE TABLE if you drop your storage tables and re-create them. If, on the other hand, you reuse your storage tables, no additional work should be necessary if your base table is an IOT.

DML on Index Storage Tables If a UROWID column is maintained in the index storage table, then you may need to change the type of the rowid bind variable in DML INSERT, UPDATE, DELETE statements so that it will work for all kinds of rowids. Converting the rowid argument passed in to a character array and then binding it as a SQLT_STR works well for both physical and universal rowids. This strategy may enable you to more easily code your indextype to work with both regular tables and IOTs.

Start, Fetch, and Close Operations on Index Storage Tables If you use an index scan-context structure to pass context between Start, Fetch, and Close, you will need to alter this structure. In particular, if you store the rowid define variable for the query in a buffer in this structure, then you will need to allocate the maximum size for a UROWID in this buffer (3800 bytes for universal rowids in byte format, 5072 for universal rowids in character format) unless you know the size of the primary key of the base table in advance or wish to determine it at run-time. You will also need to store a bit in the context to indicate if the base table is an IOT, since ODCIIndexInfo is not available in Fetch.

As with DML operations, setting up the define variable as a SQLT_STR works well for both physical and universal rowids. When physical rowids are fetched from the index table, you can be sure that their length is 18 characters. Universal rowids, however, may be up to 5072 characters long, and so a string length function must be used to correctly determine the actual length of a fetched universal rowid.

Indexes on Non-Unique Columns All values of a primary key column must be unique, so a domain index defined upon a non-unique column of a table cannot use this column as the primary key of an underlying IOT used to store the index. To work around this, you can add a column in the IOT, holding the index data, to hold a unique sequence number. Whenever a column value is inserted in the table, you should generate a unique sequence number to go with it. The indexed column together with the sequence number can be used as the primary key of the IOT. (Note that the sequence-number column cannot be a UROWID because UROWID columns cannot be part of a primary key for an IOT.) This approach also preserves the fast access to primary key column values that is a major benefit of IOTs.

Domain Index Metadata

For B-tree indexes, users can query the USER_INDEXES view to get index information. To provide similar support for domain indexes, indextype designers can add any domain-specific metadata in the following manner:

- The indextype designer can define one or more tables that will contain this meta information. The key column of this table must be a unique identifier for the index. This unique key could be the index name (schema.index). The remainder of the column definitions are at the discretion of the index designer.
- Views can be created that join the system defined metadata tables with the index meta tables to provide a comprehensive set of information for each instance of a domain index. It is the responsibility of the indextype designer to provide the view definition.

Export/Import of Domain Indexes

Like B-tree and bitmap indexes, domain indexes are exported and subsequently imported when their base tables are exported. However, domain indexes can have implementation-specific metadata associated with them that are not stored in the system catalogs. For example, a text domain index can have associated policy information, a list of irrelevant words, and so on. Export/Import provides a mechanism to opaquely move this metadata from the source platform to target platform.

To move the domain index metadata, the indextype needs to implement the `ODCIIndexGetMetadata` interface routine (see the reference chapters for details). This interface routine gets invoked when a domain index is being exported. The domain index information is passed in as a parameter. It can return any number of anonymous PL/SQL blocks that are written into the dump file and executed on import. If present, these anonymous PL/SQL blocks are executed immediately before the creation of the associated domain index.

Note that the `ODCIIndexGetMetadata` is an optional interface routine. It is needed only if the domain index has extra metadata to be moved.

Moving Domain Indexes Using Transportable Tablespaces

The **transportable tablespaces** feature enables you to move tablespaces from one Oracle database into another. You can use transportable tablespaces to move domain index data as an alternative to exporting and importing it.

Moving data using transportable tablespaces can be much faster than performing either an export/import or unload/load of the data because transporting a tablespace only requires copying datafiles and integrating tablespace structural information. Also, you do not need to rebuild the index afterward as you do when loading or importing.

> **See Also:** *Oracle9i Database Administrator's Guide* for information about using transportable tablespaces

Operators

A user-defined operator is a top-level schema object. It is identified by a name which is in the same namespace as tables, views, types and standalone functions.

Operator Bindings

An operator binding associates an operator with the **signature** of a function that implements the operator. A signature consists of a list of the datatypes of the arguments of the function, in order of occurrence, and the function's return type. Binding an operator to a certain signature enables Oracle to pick out the function to execute when the operator is invoked. An operator can be implemented by more than one function as long as each function has a different signature. For each such function, you must define a corresponding binding.

Thus, any operator has an associated set of one or more bindings. Each binding can be evaluated using a user-defined function of any of these kinds:

- Standalone function
- Package function
- `OBJECT` member method

An operator created in a schema can be evaluated using functions defined in the same or different schemas. The operator bindings can be specified at the time of creating the operator. It is ensured that the signatures of the bindings are unique.

Creating operators

An operator can be created by specifying the operator name and its bindings.

For example, an operator `Contains` can be created in the `Ordsys` schema with two bindings and the corresponding functions that provide the implementation in the Text and Spatial domains.

```
CREATE OPERATOR Ordsys.Contains
BINDING
(VARCHAR2, VARCHAR2) RETURN NUMBER USING text.contains,
(Spatial.Geo, Spatial.Geo) RETURN NUMBER USING Spatial.contains;
```

Note: Although the return data type is specified as part of operator binding declaration, it is not considered to determine the uniqueness of the binding, However, the specified function must have the same argument and return datatypes as the operator binding.

Dropping Operators

An existing operator and all its bindings can be dropped using the `DROP OPERATOR` statement. For example:

```
DROP OPERATOR Contains;
```

The default `DROP` behavior is `DROP RESTRICT` semantics. Namely, if there are any dependent indextypes for any of the operator bindings, then the `DROP` operation is disallowed.

However, users can override the default behavior by using the `FORCE` option. For example,

```
DROP OPERATOR Contains FORCE;
```

drops operator `Contains` and all its bindings and marks any dependent indextype objects invalid

Commenting on Operators

The `COMMENT` statement can be used to supply information about an indextype or operator. For example:

```
COMMENT ON OPERATOR
Ordsys.TextIndexType IS 'a number indicating whether the text contains the key';
```

Comments on operators can be viewed in these views in the data dictionary:

- `USER_OPERATOR_COMMENTS`
- `ALL_OPERATOR_COMMENTS`
- `DBA_OPERATOR_COMMENTS`

To place a comment on an operator, the operator must be in your own schema or you must have the `COMMENT ANY OPERATOR` privilege.

Invoking Operators

Operator Usage

User-defined operators can be invoked anywhere built-in operators can be used, that is, wherever expressions can occur. For example, user-defined operators can be used in the following:

- the select list of a `SELECT` command
- the condition of a `WHERE` clause
- the `ORDER BY` and `GROUP BY` clauses

Operator Execution

When an operator is invoked, the evaluation of the operator is accomplished by executing a function bound to it. The function is selected based on the datatypes of the arguments to the operator. If no function bound to the operator matches the signature with which the operator is invoked (perhaps after some implicit type conversions), an error is raised.

Examples

Consider the operator created with the following statement:

```
CREATE OPERATOR Ordsys.Contains
BINDING
(VARCHAR2, VARCHAR2) RETURN NUMBER
USING text.contains,
(spatial.geo, spatial.geo) RETURN NUMBER
USING spatial.contains;
```

Consider the operator `Contains` being used in the following SQL statements:

```
SELECT * FROM Employee
WHERE Contains(resume, ’Oracle’)=1 AND Contains(location, :bay_area)=1;
```

The invocation of the operator `Contains(resume, ’Oracle’)` is transformed into the execution of the function `text.contains(resume, ’Oracle’)` since the signature of the function matches the datatypes of the operator arguments. Similarly, the invocation of the operator `Contains(location, :bay_area)` is transformed into the execution of the function `spatial.contains(location, :bay_area)`.

The following statement would raise an error since none of the operator bindings satisfy the argument datatypes:

```
SELECT * FROM Employee
WHERE Contains(salary, 10000)=1;
```

Operator Privileges

System privileges for operator schema objects are:

- `CREATE OPERATOR`
- `CREATE ANY OPERATOR`
- `DROP ANY OPERATOR`

See the *Oracle9i SQL Reference* for details.

To use a user-defined operator in an expression, you must own the operator or have `EXECUTE` privilege on it.

Operators and Indextypes

An operator can be optionally supported by one or more user-defined indextypes. An indextype can support one or more operators. This means that a domain index of this indextype can be used in efficiently evaluating these operators. For example, B-tree indexes can be used to evaluate the relational operators like =, < and >. Operators can also be bound to regular functions. For example, an operator Equal can be bound to a function `eq(number, number)` that compares two numbers. The DDL for this would be:

```
CREATE OPERATOR Equal
BINDING(NUMBER, NUMBER) RETURN NUMBER USING eq;
```

Thus, an indextype designer should first design the set of operators to be supported by the indextype. For each of these operators, a functional implementation should be provided.

The list of operators supported by an indextype are specified when the indextype schema object is created (as described previously).

Operators that occur in a `WHERE` clause are evaluated differently than operators occurring elsewhere in a SQL statement. Both kinds of cases are considered in the following sections.

Operators in the WHERE Clause

Operators appearing in the `WHERE` clause can be evaluated efficiently by performing an index scan using the scan methods provided as part of the implementation of an indextype. This involves recognizing operator predicates of a certain form, selection of a domain index, setting up an appropriate index scan, and finally, executing index scan methods. Let's consider each one of these steps in detail.

Operator Predicates

An indextype supports efficient evaluation of operator predicates that can be represented by a range of lower and upper bounds on the operator return values. Specifically, predicates of the form:

```
op(...) relop <value expression>, where relop in {<, <=, =, >=,>}
```

```
op(...) LIKE <value_expression>
```

are possible candidates for index scan-based evaluation.

Use of the operators in any expression, for example

```
op(...) + 2 = 3
```

precludes index scan-based evaluation.

Predicates of the form,

```
op() is NULL
```

are not evaluated using an index scan. They are evaluated using the functional implementation.

Finally, any other operator predicates which can internally be converted into one of the preceding forms by Oracle can also make use of the index scan based evaluation.

Operator Resolution

An index scan-based evaluation of an operator is a possible candidate for predicate evaluation only if the operator occurring in the predicate (as described in the preceding section) operates on a column or object attribute indexed using an indextype. The final decision to choose between the indexed implementation and the functional implementation is made by the optimizer. The optimizer takes into account the selectivity and cost while generating the query execution plan.

For example, consider the query

```
SELECT * FROM Employees WHERE Contains(resume, 'Oracle') = 1;
```

The optimizer can choose to use a domain index in evaluating the `Contains` operator if

- The resume column has an index defined on it
- The index is of type `TextIndexType`
- `TextIndexType` supports the appropriate `Contains`() operator

If any of these conditions do not hold, a complete scan of the `Employees` table is performed and the functional implementation of `Contains` is applied as a post-filter. If these conditions are met, the optimizer uses selectivity and cost functions to compare the cost of index-based evaluation with the full table scan and appropriately generates the execution plan.

Consider a slightly different query,

```
SELECT * FROM Employees WHERE Contains(resume, 'Oracle') =1 AND id =100;
```

In this query, the `Employees` table could be accessed through an index on the `id` column or one on the `resume` column. The optimizer estimates the costs of the two plans and picks the cheaper one, which could be to use the index on `id` and apply the `Contains` operator on the resulting rows. In this case, the functional implementation of `Contains`() is used and the domain index is not used.

Index Scan Setup

If a domain index is selected for the evaluation of an operator predicate, an index scan is set-up. The index scan is performed by the scan methods (`ODCIIndexStart`(), `ODCIIndexFetch`(), `ODCIIndexClose`()) specified as part of the corresponding indextype implementation. The `ODCIIndexStart`() method is invoked with the operator related information including name and arguments and the lower and upper bounds describing the predicate. After the `ODCIIndexStart`() call, a series of fetches are performed to obtain row identifiers of rows satisfying the predicate, and finally the `ODCIIndexClose`() is called when the SQL cursor is destroyed.

Execution Model for Index Scan Methods

The index scan routines must be implemented with an understanding of how the routines' invocations are ordered and how multiple sets of invocations can be interleaved.

As an example, consider the following query:

```
SELECT * FROM Emp1, Emp2 WHERE
Contains(Emp1.resume, 'Oracle') =1 AND Contains(Emp2.resume, 'Unix') =1
AND Emp1.id = Emp2.id;
```

If the optimizer decides to use the domain indexes on the resume columns of both tables, the indextype routines may be invoked in the following sequence:

```
start(ctx1, ...); /* corr. to Contains(Emp1.resume, 'Oracle') */
start(ctx2, ...); /* corr. to Contains(Emp2.resume, 'Unix');
fetch(ctx1, ...);
fetch(ctx2, ...);
fetch(ctx1, ...);
...
close(ctx1);
close(ctx2);
```

Thus, the same indextype routine may be invoked but for different instances of operators. At any time, many operators are being evaluated through the same indextype routines. In case of a routine that does not need to maintain any state across calls because all the information is obtained through its parameters (as with the create routine), this is not a problem. However, in case of routines needing to maintain state across calls (like the fetch routine, which needs to know which row to return next), the state should be maintained in the `SELF` parameter that is passed in to each call. The `SELF` parameter (which is an instance of the implementation type) can be used to store either the entire state (if it is not too big) or a handle to the cursor-duration memory that stores the state.

Operators Outside the WHERE Clause

Using a Functional Implementation

Operators occurring in expressions other than in the `WHERE` clause are evaluated using the functional implementation. For example,

```
SELECT Contains(resume, 'Oracle') FROM Employee;
```

would be executed by scanning the `Employee` table and invoking the functional implementation for `Contains` on each instance of resume. The function is invoked by passing it the actual value of the resume (text data) in the current row. Note that this function would not make use of any domain indexes that may have been built on the resume column.

However, it is possible to have a functional implementation for an operator that makes use of a domain index. The following sections discuss how functions that use domain indexes can be written and how they are invoked by the system.

Creating Index-based Functional Implementation

For many domain-specific operators, such as `Contains`, the functional implementation can work in two ways:

1. If the operator is operating on a column (or `OBJECT` attribute) that has a domain index of a particular indextype, the function can evaluate the operator by looking at the index data rather than the actual argument value.

   For example, when `Contains(resume, 'Oracle')` is invoked on a particular row of the `Employee` table, it is easier for the function to look up the text domain index defined on the resume column and evaluate the operator based on the row identifier for the row containing the resume - rather than work on the resume text data argument.

2. If the operator is operating on a column that does not have an appropriate domain index defined on it or if the operator is invoked with literal values (non-columns), the functional implementation evaluates the operator based on only the argument values. This is the default behavior for all operator bindings.

To achieve both the behaviors of (1) and (2), the functional implementation is provided using a regular function which has three additional arguments—that is, additional to the original arguments to the operator. The additional arguments are:

- Index context—containing domain index information and the row identifier of the row on which the operator is being evaluated
- Scan context—a context value to share state with subsequent invocations of the same operator (operating on other rows of the table)

- Scan flag—indicates whether the current call is the last invocation during which all clean up operations should be done

For example, the index-based functional implementation for the `Contains` operator is provided by the following function.

```
CREATE FUNCTION TextContains (Text IN VARCHAR2, Key IN VARCHAR2,
indexctx IN ODCIIndexCtx, scanctx IN OUT TextIndexMethods, scanflg IN NUMBER)
RETURN NUMBER AS
BEGIN
.......
END TextContains;
```

The `Contains` operator is bound to the functional implementation as follows:

```
CREATE OPERATOR Contains
BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER
WITH INDEX CONTEXT, SCAN CONTEXT TextIndexMethods
USING TextContains;
```

The `WITH INDEX CONTEXT` clause specifies that the functional implementation can make use of any applicable domain indexes. The `SCAN CONTEXT` specifies the datatype of the scan context argument. It must be the same as the implementation type of the relevant indextype that supports this operator.

Operator Resolution

Oracle invokes the functional implementation for the operator if the operator appears outside the `WHERE` clause. If the functional implementation is index-based (that is, defined to use an indextype), the additional index information is passed in as arguments only if the operator's first argument is a column (or object attribute) with a domain index of the appropriate indextype defined on it.

For example, in the query

```
SELECT Contains(resume, 'Oracle & Unix') FROM Employees;
```

the Operator `Contains` is evaluated using the index-based functional implementation by passing the index information about the domain index on the `resume` column instead of the resume data.

Operator Execution

To execute the index-based functional implementation, Oracle RDBMS sets up the arguments in the following manner:

- The initial set of arguments are the same as those specified by the user for the operator.
- If the first argument is not a column, the ODCIIndexCtx attributes are set to NULL.
- If the first argument is a column, the ODCIIndexCtx attributes are set up as follows.
  - If there are no applicable domain indexes, the ODCIIndexInfo attribute is set to NULL, else it is set up with the information about the domain index.
  - The rowid attribute holds the row identifier of the row being operated on.
- The scan context is passed as NULL to the first invocation of the operator. Since it is an IN/OUT parameter, the return value from the first invocation is passed in to the second invocation and so on.
- The scan flag is set to RegularCall for all normal invocations of the operator. After the last invocation, the functional implementation is invoked once more, at which time any cleanup actions can be performed. During this call, the scan flag is set to CleanupCall and all other arguments except the scan context are set to NULL.

When index information is passed in, the implementation can compute the operator value by doing a domain index lookup using the row identifier as key. The index metadata is used to identify the index structures associated with the domain index. The scan context is typically used to share state with the subsequent invocations of the same operator.

Ancillary Data

Apart from filtering rows, an operator occurring in the WHERE clause might need to support returning ancillary data. Ancillary data is modeled as an operator (or multiple operators) with a single literal number argument. It has a functional implementation that has access to state generated by the index scan-based implementation of the primary operator occurring in the WHERE clause.

For example, in the following query,

```
SELECT Score(1) FROM Employees
WHERE Contains(resume, 'OCI & UNIX', 1) =1;
```

Contains is the primary operator and can be evaluated using an index scan which, in addition to determining the rows that satisfy the predicate, also computes a score value for each row. The functional implementation for the Score operator simply accesses the state generated by the index scan to obtain the score for a given row identified by its row identifier. The literal argument `1` associates the ancillary operator `Score` to the corresponding primary operator `Contains` which generates the ancillary data.

In summary, ancillary data is modeled as independent operator(s) and is invoked by the user with a single number argument that ties it with the corresponding primary operator. Its functional implementation makes use of either the domain index or the state generated by the primary operator occurring in the `WHERE` clause. The functional implementation is invoked with extra arguments: the index context containing the domain index information, and the scan context which provides access to the state generated by the primary operator. The following sections discuss how operators modeling ancillary data are defined and invoked.

Creating Operator Binding that Computes Ancillary Data

An indextype designer needs to specify that an operator binding computes ancillary data. Such a binding is referred to as a *primary* binding. For example, a primary binding for Contains can be defined as follows:

```
CREATE OPERATOR Contains
BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER
WITH INDEX CONTEXT, SCAN TextIndexMethods COMPUTE ANCILLARY DATA
USING TextContains;
```

This definition registers two bindings for `Contains`, namely:

- `CONTAINS(VARCHAR2, VARCHAR2)`—This can be used as before.
- `CONTAINS(VARCHAR2, VARCHAR2, NUMBER)`—When ancillary data is required elsewhere in SQL query, the operator can be invoked with the preceding signature. The `NUMBER` argument is used to associate the corresponding ancillary operator binding.

However, the indextype designer needs to define a single functional implementation:

```
TextContains(VARCHAR2, VARCHAR2, ODCIIndexCtx, TextIndexMethods, NUMBER).
```

Creating Operator Binding that Models Ancillary Data

An indextype designer has to implement the functional implementation for ancillary data operators in a manner similar to the index-based functional implementation. As discussed earlier, the function takes extra arguments. After the function is defined, the indextype designer can bind it to the operator with an additional ANCILLARY TO attribute, which indicates that the functional implementation needs to share state with the *primary* operator binding. The binding that is used for modeling ancillary data is referred to as the *ancillary* operator binding.

For example, let the TextScore() function contain code to evaluate the Score ancillary operator.

```
CREATE FUNCTION TextScore (Text IN VARCHAR2, Key IN VARCHAR2,
indexctx IN ODCIIndexCtx, scanctx IN OUT TextIndexMethods, scanflg IN NUMBER)
RETURN NUMBER AS
BEGIN
.......
END TextScore;
```

An ancillary operator binding can be created as follows:

```
CREATE OPERATOR Score
BINDING (NUMBER) RETURN NUMBER
ANCILLARY TO Contains(VARCHAR2, VARCHAR2)
USING TextScore;
```

- The ANCILLARY TO clause specifies that it shares state with the implementation of corresponding primary operator binding CONTAINS(VARCHAR2, VARCHAR2).
- Note that the functional implementation for the ancillary operator binding must have the same signature as the functional implementation for the primary operator binding.
- The ancillary operator binding is invoked with a single literal number argument, such as Score(1), Score(2), and so on.

Operator Resolution

The operators corresponding to ancillary data are invoked by the user with a single number argument.

Note: The number argument must be a literal in both the ancillary operation and the primary operator invocation. This is required so that the operator association can be done at the query compilation time.

The corresponding primary operator invocation in the query is determined by matching it with the number passed in as the last argument to the primary operator. After the matching primary operator invocation is found (it is an error to find zero or more than one matching primary operator invocation):

- The arguments to the primary operator are also made operands to the ancillary operator.
- The ancillary and primary operator executions are passed in the same scan context.

For example, consider the query

```
SELECT Score(1) FROM Employees
WHERE Contains(resume, ' Oracle & Unix', 1) =1;
```

The invocation of `Score` is determined to be ancillary to `Contains` based on the number argument `1`, and the functional implementation for `Score` gets the following operands: `(resume`, `'Oracle&Unix'`, `indexctx`, `scanctx`, `scanflg)`, where `scanctx` is shared with the invocation of `Contains`.

Operator Execution

The execution involves using an index scan to process the `Contains` operator. For each of the rows returned by the `fetch`() call of the index scan, the functional implementation of `Score` is invoked by passing it the `ODCIIndexCtx` argument, which contains the index information, row identifier, and a handle to the index scan state. The functional implementation can use the handle to the index scan state to compute the score.

Object Dependencies, Drop Semantics, and Validation

Dependencies

The dependencies among various objects are as follows:

- **Functions, Packages, and Object Types:** Referenced by Operators and Indextypes.
- **Operators:** Referenced by Indextypes, DML and Query SQL Statements.
- **Indextypes:** Referenced by Domain Indexes.
- **Domain Indexes:** Referenced (used implicitly) by DML and Query SQL Statements

Thus, the order in which these objects must be created, or their definitions exported for future Import are:

- Functions, Packages, and Object Types, followed by Operators, followed by Indextypes.

Drop Semantics

The drop behavior for an object is as follows:

- **RESTRICT semantics:** If there are any dependent objects the drop operation is disallowed.
- **FORCE semantics:** The object is dropped even in the presence of dependent objects and the dependent objects if any are recursively marked invalid.

The following table shows the default and explicit drop options supported for operators and indextypes. The other schema objects are included for completeness and the corresponding drop behavior already available in Oracle.

Schema Object	Default Drop Behavior	Explicit Options Supported
Function	FORCE	None
Package	FORCE	None
Object Types	RESTRICT	FORCE
Operator	RESTRICT	FORCE
Indextype	RESTRICT	FORCE

Object Validation

Invalid objects are automatically revalidated, if possible, the next time they are referenced.

Privileges

- To create an operator and its bindings, you must have EXECUTE privilege on the function, operator, package, or the type referenced in addition to CREATE OPERATOR or CREATE ANY OPERATOR privilege.
- To create an indextype, you must have EXECUTE privilege on the type that implements the indextype in addition to CREATE INDEXTYPE or CREATE ANY INDEXTYPE privilege. Also, you must have EXECUTE privileges on the operators that the indextype supports.

- To alter an indextype in your own schema, you must have `CREATE INDEXTYPE` system privilege.
- To alter an indextype or operator in another user's schema, you must have the `ALTER ANY INDEXTYPE` or `ALTER ANY OPERATOR` system privilege.
- To create a domain index, you must have `EXECUTE` privilege on the indextype in addition to `CREATE INDEX` or `CREATE ANY INDEX` privileges.
- To alter a domain index, you must have `EXECUTE` privilege on the indextype.
- To use the operators in queries or DML statements, you must have `EXECUTE` privilege on the operator and the associated function / package / type.
- To change the implementation type, you must have `EXECUTE` privilege on the new implementation type.

Partitioned Domain Indexes

A domain index can be built to have discrete index partitions that correspond to the partitions of a range-partitioned table. Such an index is called a **local domain index**, as opposed to a *global* domain index, which has no index partitions. The term *local domain index* refers to a partitioned index as a whole, not to the partitions that comprise a local domain index.

A local domain index is equipartitioned with the underlying table: all keys in a local domain index refer only to rows stored in its corresponding table partition; none refer to rows in other partitions.

Currently, local domain indexes can be created only for range-partitioned tables. Local domain indexes cannot be built for hash-partitioned tables or IOTs.

A local (as opposed to a global) domain index can index only a single column; it cannot index an expression.

You provide for using local domain indexes in the indextype, with the `CREATE INDEXTYPE` statement. For example:

```
CREATE INDEXTYPE TextIndexType
  FOR Contains (VARCHAR2, VARCHAR2)
  USING TextIndexMethods
  WITH LOCAL RANGE PARTITION;
```

This statement specifies that the implementation type `TextIndexType` is capable of creating / maintaining local domain indexes. The clause `WITH LOCAL RANGE PARTITION` specifies the partitioning method for the base table.

The CREATE INDEX statement creates and partitions the index. Here is the syntax:

```
CREATE INDEX [schema.]index
  ON [schema.]table [t.alias] (indexed_column)
  INDEXTYPE IS indextype
  [LOCAL [PARTITION [partition [PARAMETERS ('string')]]] [...] ]
  [PARAMETERS ('string')];
```

Note: The given syntax for CREATE INDEX differs from the syntax shown in *Oracle9i SQL Reference*, which omits the LOCAL [PARTITION] clause. Use this syntax to create a local domain index.

The LOCAL [PARTITION] clause indicates that the index is a local index on a partitioned table. You can specify partition names or allow Oracle to generate them.

In the PARAMETERS clause, specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the default parameters for the index partitions. If you specify this clause as part of the LOCAL [PARTITION] clause, you override any default parameters with parameters for the individual partition. The LOCAL [PARTITION] clause can specify multiple partitions.

Once the domain index is created, Oracle invokes the appropriate ODCI routine. If the routine does not return successfully, the domain index is marked FAILED. The only operations supported on an failed domain index are DROP INDEX and (for non-local indexes) REBUILD INDEX.

The following example creates local domain index ResumeIndex:

```
CREATE INDEX ResumeIndex ON Employees(Resume)
  INDEXTYPE IS TextIndexType LOCAL;
```

There are these restrictions on creating a local domain index:

- The *index_expr* list can specify only a single column.
- You cannot specify a bitmap or unique domain index.

Dropping a Local Domain Index

A specified index partition cannot be dropped explicitly. To drop a local index partition, the entire local domain index must be dropped:

```
DROP INDEX ResumeIndex;
```

Altering a Local Domain Index

The `ALTER INDEX` statement can be used to perform the following operations on a local domain index:

- Rename the top level index
- Modify the default parameter string for all the index objects
- Modify the parameter string associated with a specific partition
- Rename an index partition
- Rebuild an index partition

The `ALTER INDEXTYPE` statement enables you to change properties and the implementation type of an indextype without having to drop and re-create the indextype and then rebuild all dependent indexes.

> **See Also:** *Oracle9i SQL Reference* for complete syntax of the SQL statements mentioned in this section

Summary of Index States

Like a domain index, a partition of a local domain index can be in one or more of several states:

State	Description
`IN_PROGRESS`	The index or the index partition is in this state before and during the execution of the `ODCIndex` DDL interface routines. The state is generally transitional and temporary. However, if the routine ends prematurely, the index could remain marked `IN_PROGRESS`.
`FAILED`	If the `ODCIIndex` interface routine doing DDL operations on the index returns an error, the index or index partition is marked `FAILED`.

State	Description
UNUSABLE	Same as for regular indexes: An index on a partitioned table is marked UNUSABLE as a result of certain partition maintenance operations. Note that, for partitioned indexes, UNUSABLE is associated only with an index partition, not with the index as a whole.
INVALID or VALID	An index gets marked INVALID if an object that the index directly or indirectly depends upon is dropped or invalidated. This property is associated only with an index, never with an index partition.

DML Operations with Local Domain Indexes

DML operations cannot be performed on the underlying table if an index partition of a local domain index is in any of these states: IN_PROGRESS, FAILED, or UNUSABLE.

Table Operations That Affect Indexes

The following tables list operations that can be performed on the underlying table of an index and describe the effect, if any, on the index.

Table Operation	Description
DROP table	Drops the table. Drops all the indexes and their corresponding partitions
TRUNCATE table	Truncates the table. Truncates all the indexes and the index partitions

ALTER TABLE Operation	Description
Base table operations that do not involve partition maintenance	
Modify Partition Unusable local indexes	Marks the local index partition associated with the table partition as UNUSABLE
Modify Partition Rebuild Unusable local indexes	Rebuilds the local index partitions that are marked UNUSABLE and are associated with this table partition
Add Partition	Adds a new table partition. Also adds a new local index partition.

ALTER TABLE Operation	Description
Drop Partition	Drops a range table partition. Also drops the associated local index partition
Truncate Partition	Truncate the table partition. Also truncates the associated local index partition
Base table operations that involve partition maintenance	
Move Partition	Moves the base table partition to another tablespace. Corresponding local index partitions are marked `UNUSABLE`
Split Partition	Splits a table partition into two partitions. Corresponding local index partition is also split. If the resulting partitions are non-empty, the index partitions are marked `UNUSABLE`
Merge Partition	Merges two table partitions into one partition. Corresponding local index partitions should also merge. If the resulting partition contains data, the index partition is marked `UNUSABLE`
Exchange Partition Excluding Indexes	Exchanges a table partition with a non-partitioned table. Local index partitions and global indexes are marked `UNUSABLE`
Exchange Partition Including Indexes	Exchanges a table partition with a non-partitioned table. Local index partition is exchanged with global index on the non-partitioned table. Index partitions remain `USABLE`

ODCIIndex Interfaces for Partitioning Domain Indexes

The set of `ODCIIndex` interfaces that needs to be implemented for a domain index depends on whether the index is to be partitioned and, if so, in what way. There are two possibilities:

- Non-partitioned domain index
- Local range-partitioned indexes

The `ODCIIndex` interfaces that must be implemented for each option are listed in the following sections. Those in the first group must be implemented for any domain index, partitioned or not. Those in the other group need be implemented only to provide support for local range-partitioned indexes.

ODCIIndex Interfaces Required for any Domain Index

`ODCIIndexGetInterface()`

`ODCIIndexAlter()`

`ODCIIndexCreate()`

`ODCIIndexDrop()`

`ODCIIndexTruncate()`

`ODCIIndexInsert()`

`ODCIIndexDelete()`

`ODCIIndexUpdate()`

`ODCIIndexStart()`

`ODCIIndexFetch()`

`ODCIIndexClose()`

ODCIIndex Interfaces Required for Local Range-Partitioned Indexes

`ODCIIndexExchangePartition()`

`ODCIIndexMergePartition()`

`ODCIIndexSplitPartition()`

Domain Indexes and SQL*Loader

SQL*Loader conventional path loads are supported for tables on which domain indexes are defined, but direct path loads are not. To do a direct path load, first drop the domain index, do the direct path load in SQL*Loader, and then re-create the domain indexes.

8

Query Optimization

This chapter describes query optimization, including:

Overview

Query Optimization is the process of choosing the most efficient way to execute a SQL statement. When the *cost-based optimizer* was offered for the first time with Oracle7, Oracle supported only standard relational data. The introduction of objects extended the supported datatypes and functions. The *Extensible Indexing* feature discussed in the previous chapter, introduces user-defined access methods.

See Also:

- *Oracle9i Database Concepts* for an introduction to optimization
- *Oracle9i Database Performance Guide and Reference* for information about using hints in SQL statements

The extensible optimizer feature allows authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions that are used by the optimizer in choosing a query plan. The optimizer cost model is extended to integrate information supplied by the user to assess CPU and the I/O cost, where CPU cost is the number of machine instructions used, and I/O cost is the number of data blocks fetched.

Specifically, you now can:

- Associate cost functions and default costs with domain indexes (partitioned or unpartitioned), indextypes, packages, and standalone functions. The optimizer can obtain the cost of scanning a single partition of a domain index, multiple domain index partitions, or an entire index.
- Associate selectivity functions and default selectivity with methods of object types, package functions, and standalone functions. The optimizer can estimate user-defined selectivity for a single partition, multiple partitions, or the entire table involved in a query.
- Associate statistics collection functions with domain indexes and columns of tables. The optimizer can collect user-defined statistics at both the partition level and the object level for a domain index or a table.
- Order predicates with functions based on cost.
- Select a user-defined access method (domain index) for a table based on access cost.
- Use the `DBMS_STATS` package or the `ANALYZE` command to invoke user-defined statistics collection and deletion functions.

- Use new data dictionary views to include information about the statistics collection, cost, or selectivity functions associated with columns, domain indexes, indextypes or functions.
- Add a hint to preserve the order of evaluation for function predicates.

Please note that only the *cost-based* optimizer has been enhanced; Oracle has not altered the operation of the *rule-based* optimizer.

The optimizer generates an execution plan for SQL queries and DML statements—`SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. For simplicity, we describe the generation of an execution plan in terms of a `SELECT` statement, but the process for DML statements is similar.

An execution plan includes an *access method* for each table in the `FROM` clause, and an ordering, called the *join order*, of the tables in the `FROM` clause. System-defined access methods include indexes, hash clusters, and table scans. The optimizer chooses a plan by generating a set of join orders, or permutations, by computing the cost of each, and then by selecting the process with the lowest cost. For each table in the join order, the optimizer computes the cost of each possible access method and join method and chooses the one with the lowest cost. The cost of the join order is the sum of the access method and join method costs. The costs are calculated using algorithms which together comprise the *cost model*. The cost model includes varying level of detail about the physical environment in which the query is executed.

The optimizer uses statistics about the objects referenced in the query to compute the selectivity and costs. The statistics are gathered using the `ANALYZE` command. The selectivity of a predicate is the fraction of rows in a table that is chosen by the predicate. It is a number between 0 and 1.

Note: Oracle Corporation recommends that you use the `DBMS_STATS` package instead of the `ANALYZE` command to collect optimizer statistics. In a future release, functionality to collect optimizer statistics will be removed from `ANALYZE`.

See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about `DBMS_STATS`.

The *Extensible Indexing* feature allows users to define new operators, indextypes, and domain indexes. For user-defined operators and domain indexes, the *Extensible Optimizer* feature enables you to control the three main components used by the optimizer to select an execution plan:

- Statistics
- Selectivity
- Cost

In the following sections, we describe each of these components in greater detail.

Statistics

Statistics are collected using the SQL ANALYZE statement. Statistics can be collected for tables and indexes. In general, the more accurate the statistics, the better the execution plan generated by the optimizer. We call the statistics generated by the current ANALYZE command *standard statistics*. However, the standard ANALYZE statement cannot generate statistics on a domain index because the database does not know the index storage structure.

Note: Oracle Corporation recommends that you use the DBMS_STATS package instead of the ANALYZE command to collect optimizer statistics. In a future release, functionality to collect optimizer statistics will be removed from ANALYZE.

See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about DBMS_STATS.

User-Defined Statistics

The Extensible Optimizer feature lets you define *statistics collection* functions for domain indexes and columns as well as for partitions of a domain index or table. The extension to the ANALYZE command has the effect that whenever a domain index is analyzed, a call is made to the user-specified statistics collection function. The database does not know the representation and meaning of the user-collected statistics.

In addition to domain indexes, Oracle supports user-defined statistics collection functions for individual columns of a table, and for user-defined datatypes. In the former case, whenever a column is analyzed, the user-defined statistics collection function is called to collect statistics in addition to any standard statistics that the database collects. If a statistics collection function exists for a datatype, it is called for each column of the table being analyzed that has the required type.

Thus, the *Extensible Optimizer* feature extends ANALYZE to allow user-defined statistics collection functions for domain indexes, indextypes, datatypes, individual table columns, and partitions.

The cost of evaluating a user-defined function depends on the algorithm and the statistical properties of its arguments. It is not practical to store statistics for all possible combinations of columns that could be used as arguments for all functions. Therefore, Oracle maintains only statistics on individual columns. It is also possible that function costs depend on the different statistical properties of each argument. Every column could require statistics for every argument position of every applicable function. Oracle does not support such a proliferation of statistics and cost functions because it would decrease performance.

A user-defined function to drop statistics is required whenever there is a user-defined statistics collection function; it is called by `ANALYZE DELETE`.

User-Defined Statistics for Partitioned Objects

Since domain indexes cannot be partitioned in Oracle9*i*, a user-defined statistics collection function collects only global statistics on the non-partitioned index.

When an `ANALYZE` command specifies a list of partitions, the information is not passed to user-defined statistics collection functions.

Selectivity

The optimizer uses statistics to calculate the selectivity of predicates. The selectivity is the fraction of rows in a table or partition that is chosen by the predicate. It is a number between 0 and 1. The selectivity of a predicate is used to estimate the cost of a particular access method; it is also used to determine the optimal join order. A poor choice of join order by the optimizer could result in a very expensive execution plan.

Currently, the optimizer uses a standard algorithm to estimate the selectivity of selection and join predicates. However, the algorithm does not always work well in cases in which predicates contain functions or type methods. In addition, predicates can contain user-defined operators about which the optimizer does not have any information. In that case the optimizer cannot compute an accurate selectivity.

User-Defined Selectivity

For greater control over the optimizer's selectivity estimation, this feature lets you specify user-defined selectivity functions for predicates containing user-defined operators, standalone functions, package functions, or type methods. The user-defined selectivity function is called by the optimizer whenever it encounters a predicate with one of the following forms:

```
operator(...) relational_operator <constant>
```

```
<constant> relational_operator operator(...)

operator(...) LIKE <constant>
```

where

- `operator(...)` is a user-defined operator, standalone function, package function, or type method,
- `relational_operator` is one of `{<, <=, =, >=, >}`, and
- `<constant>` is a constant value expression or bind variable.

For such cases, users can define selectivity functions associated with `operator(...)`. The arguments to `operator` can be columns, constants, bind variables, or attribute references. When optimizer encounters such a predicate, it calls the user-defined selectivity function and passes the entire predicate as an argument (including the operator, function, or type method and its arguments, the relational operator `relational_operator`, and the constant expression or bind variable). The return value of the user-defined selectivity function must be expressed as a percent, and be between 0 and 100 inclusive; the optimizer ignores values outside this range.

Wherever possible, the optimizer uses user-defined selectivity values. However, this is not possible in the following cases:

- The user-defined selectivity function returns an invalid value (less than 0 or greater than 100)
- There is no user-defined selectivity function defined for the operator, function, or method in the predicate
- The predicate does not have one of the preceding forms: for example, `operator(...) + 3 relational_operator <constant>`

In each of these cases, the optimizer uses heuristics to estimate the selectivity.

Cost

The optimizer estimates the cost of various access paths to choose an optimal plan. For example, it computes the CPU and I/O cost of using an index and a full table scan to choose between the two. However, with regard to domain indexes, the optimizer does not know the internal storage structure of the index, and so it cannot compute a good estimate of the cost of a domain index.

User-Defined Cost

For greater flexibility, the cost model has been extended to let you define costs for domain indexes, index partitions, and user-defined standalone functions, package functions, and type methods. The user-defined costs can be in the form of default costs that the optimizer looks up, or they can be full-fledged cost functions which the optimizer calls to compute the cost.

Like user-defined selectivity statistics, user-defined cost statistics are optional. If no user-defined cost is available, the optimizer uses heuristics to compute an estimate. However, in the absence of sufficient useful information about the storage structures in user-defined domain indexes and functions, such estimates can be very inaccurate and result in the choice of a sub-optimal execution plan.

User-defined cost functions for domain indexes are called by the optimizer only if a domain index is a valid access path for a user-defined operator (for details regarding when this is true, see the discussion of *user-defined indexing* in the previous chapter). User-defined cost functions for functions, methods and domain indexes are only called when a predicate has one of the following forms:

```
operator(...) relational_operator <constant>

<constant> relational_operator operator(...)

operator(...) LIKE <constant>
```

where

- `operator(...)` is a user-defined operator, standalone function, package function, or type method,
- `relational_operator` is one of {`<, <=, =, >=, >`}, and
- `<constant>` is a constant value expression or bind variable.

This is, of course, identical to the conditions for user-defined selectivity functions.

User-defined cost functions can return three cost values, each value representing the cost of a *single* execution of a function or domain index implementation:

- `CPU` — the number of machine cycles executed by the function or domain index implementation. This does not include the overhead of invoking the function.
- `I/O` — the number of data blocks read by the function or domain index implementation. For a domain index, this does not include accesses to the Oracle table. The multiblock I/O factor is not passed to the user-defined cost functions.

- NETWORK — the number of data blocks transmitted. This is valid for distributed queries as well as functions and domain index implementations. For Oracle this cost component is not used and is ignored; however, as described in the following sections, the user is required to stipulate a value so that backward compatibility is facilitated when this feature is introduced.

The optimizer computes a composite cost from these cost values.

The package DBMS_ODCI contains a function estimate_cpu_units to help get the CPU and I/O cost from input consisting of the elapsed time of a user function. estimate_cpu_units measures CPU units by multiplying the elapsed time by the processor speed of the machine and returns the approximate number of CPU instructions associated with the user function. (For a multiprocessor machine, estimate_cpu_units considers the speed of a single processor.)

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about package DBMS_ODCI

Optimizer Parameters

The cost of a query is a function of the cost values discussed in the preceding section. The settings of optimizer initialization parameters determine which cost to minimize. If optimizer_mode is first_rows, the resource cost of returning a single row is minimized, and the optimizer mode is passed to user-defined cost functions. Otherwise, the resource cost of returning all rows is minimized.

Defining Statistics, Selectivity, and Cost Functions

You can compute and store user-defined statistics for domain indexes and columns. These statistics are in addition to the standard statistics that are already collected by ANALYZE. User-defined selectivity and cost functions for functions and domain indexes can use both standard and user-defined statistics in their computation. The internal representation of these statistics need not be known to Oracle, but you must provide methods for their collection. You are solely responsible for defining the representation of such statistics and for maintaining them. Note that user-collected statistics are used only by user-defined selectivity and cost functions; the optimizer uses only its standard statistics.

User-defined statistics collection, selectivity, and cost functions must be defined in a user-defined type. Depending on the functionality you want it to support, this type must implement as methods some or all of the functions defined in the system interface ODCIStats (**O**racle **D**ata **C**artridge **I**nterface **Stat**istic**s**), described in Chapter 18.

The following example shows a type definition (or the outline of one) that implements all the functions in the `ODCIStats` interface.

```
CREATE TYPE my_statistics AS OBJECT (

   -- Function to get current interface
   FUNCTION ODCIGetInterfaces(ifclist OUT ODCIObjectList) RETURN NUMBER,

   -- User-defined statistics functions
   FUNCTION ODCIStatsCollect(col ODCIColInfo, options ODCIStatsOptions,
      statistics OUT RAW, env ODCIEnv) RETURN NUMBER,
   FUNCTION ODCIStatsCollect(ia ODCIIndexInfo, options ODCIStatsOptions,
      statistics OUT RAW, env ODCIEnv) RETURN NUMBER,
   FUNCTION ODCIStatsDelete(col ODCIColInfo, statistics OUT RAW, env ODCIEnv)
      RETURN NUMBER,
   FUNCTION ODCIStatsDelete(ia ODCIIndexInfo, statistics OUT RAW, env ODCIEnv)
      RETURN NUMBER,

   -- User-defined selectivity function
   FUNCTION ODCIStatsSelectivity(pred ODCIPredInfo, sel OUT NUMBER, args
      ODCIArgDescList, start <function_return_type>,
      stop <function_return_type>, <list of function arguments>,
      env ODCIEnv) RETURN NUMBER,

   -- User-defined cost function for functions and type methods
   FUNCTION ODCIStatsFunctionCost(func ODCIFuncInfo, cost OUT ODCICost,
      args ODCIArgDescList, <list of function arguments>) RETURN NUMBER,

   -- User-defined cost function for domain indexes
   FUNCTION ODCIStatsIndexCost(ia ODCIIndexInfo, sel NUMBER,
      cost OUT ODCICost, qi ODCIQueryInfo, pred ODCIPredInfo,
      args ODCIArgDescList, start <operator_return_type>,
      stop <operator_return_type>, <list of operator value arguments>,
      env ODCIEnv) RETURN NUMBER
)
```

The object type that you define, referred to as a *statistics type*, need not implement all the functions from `ODCIStats`. User-defined statistics collection, selectivity, and cost functions are optional, so a statistics type may contain only a subset of the functions in `ODCIStats`. Table 8–1 lists the type methods and default statistics associated with different kinds of schema objects.

Table 8–1 Statistics Type Methods and Default Statistics for Various Schema Objects

ASSOCIATE STATISTICS WITH	Statistics Type Methods Used	Default Statistics Used
column	ODCIStatsCollect, ODCIStatsDelete	
object type	ODCIStatsCollect, ODCIStatsDelete, ODCIStatsFunctionCost, ODCIStatsSelectivity	cost, selectivity
function	ODCIStatsFunctionCost, ODCIStatsSelectivity	cost, selectivity
package	ODCIStatsFunctionCost, ODCIStatsSelectivity	cost, selectivity
index	ODCIStatsCollect, ODCIStatsDelete, ODCIIndexCost	cost
indextype	ODCIStatsCollect, ODCIStatsDelete, ODCIIndexCost	cost

The types of the parameters of statistics type methods are system-defined ODCI (**O**racle **D**ata **C**artridge **I**nterface) datatypes. These are described in Chapter 17 and Chapter 18.

The selectivity and cost functions must not change any database or package state. Consequently, no SQL DDL or DML operations are permitted in the selectivity and cost functions. If such operations are present, the functions will not be called by the optimizer.

User-Defined Statistics Functions

There are two user-defined statistics collection functions, one for collecting statistics and the other for deleting them.

The first, `ODCIStatsCollect`, is used to collect user-defined statistics; its interface depends on whether a column or domain index is being analyzed. It is called when analyzing a column of a table or a domain index and takes two parameters:

- `col` for the column being analyzed, or

  `ia` for the domain index being analyzed;
- `options` for options specified in the ANALYZE command (for example, the sample size when ANALYZE ESTIMATE is used).

As mentioned, the database does not interpret statistics collected by `ODCIStatsCollect`. You can store output in a user-managed format or in a dictionary table (described in the *Extensible Optimizer* reference chapter) provided for the purpose. The statistics collected by the ODCIStatsCollect functions are returned in the output parameter, statistics, as a RAW datatype.

When an ANALYZE DELETE command is issued, user-collected statistics are deleted by calling the ODCIStatsDelete function whose interface depends on whether the statistics for a column or domain index are being dropped. It takes a single parameter: col, for the column whose user-defined statistics need to be deleted, or ia, for the domain index whose statistics are to be deleted.

If a user-defined ODCIStatsCollect function is present in a statistics type, the corresponding ODCIStatsDelete function must also be present.

The return values of the ODCIStatsCollect and ODCIStatsDelete functions must be Success (indicating success), Error (indicating an error), or Warning (indicating a warning); these return values are defined in a system package ODCIConst (described in the *Extensible Optimizer* reference).

User-Defined Selectivity Functions

You will recall that user-defined selectivity functions are used only for predicates of the following forms:

```
operator(...) relational_operator <constant>

<constant> relational_operator operator(...)

operator(...) LIKE <constant>
```

A user-defined selectivity function, ODCIStatsSelectivity, takes five sets of input parameters that describe the predicate:

- pred describing the function operator and the relational operator relational_operator;
- args describing the start and stop values (that is, <constant>) of the function and the actual arguments to the function (operator());
- start whose datatype is the same as that of the function’s return value, describing the start value of the function;
- stop whose datatype is the same as that of the function’s return value, describing the stop value of the function;
- and a list of function arguments whose number, position, and type must match the arguments of the function operator.

The computed selectivity is returned in the output parameter `sel` as a number between `0` and `100` (inclusive) that represents a percentage. The optimizer ignores numbers less than `0` or greater than `100` as invalid values.

The return value of the `ODCIStatsSelectivity` function must be

- `Success` indicating success, or
- `Error` indicating an error, or
- `Warning` indicating a warning.

As an example, consider a function `myFunction` defined as follows:

```
myFunction (a NUMBER, b VARCHAR2(10)) return NUMBER
```

A user-defined selectivity function for the function `myFunction` would be as follows:

```
ODCIStatsSelectivity(pred ODCIPredInfo, sel OUT NUMBER, args ODCIArgDescList,
   start NUMBER, stop NUMBER, a NUMBER, b VARCHAR2(10), env ODCIEnv)
   return NUMBER
```

If the function `myFunction` is called with literal arguments, for example,

```
myFunction (2, ’TEST’) > 5
```

then the selectivity function is called as follows:

```
ODCIStatsSelectivity(<ODCIPredInfo constructor>, sel,
   <ODCIArgDescList constructor>, 5, NULL, 2, ’TEST’, <ODCIEnv flag>)
```

If, on the other hand, the function `myFunction` is called with some non-literals—for example:

```
myFunction(Test_tab.col_a, ’TEST’)> 5
```

where `col_a` is a column in table `Test_tab`, then the selectivity function is called as follows:

```
ODCIStatsSelectivity(<ODCIPredInfo constructor>, sel,
   <ODCIArgDescList constructor>, 5, NULL, NULL, ’TEST’, <ODCIEnv flag>)
```

In other words, the start, stop, and function argument values are passed to the selectivity function only if they are literals; otherwise they are `NULL`. The `ODCIArgDescList` descriptor describes all its following arguments.

User-Defined Cost Functions for Functions

As already mentioned, user-defined cost functions are only used for predicates of the following forms:

```
operator(...) relational_operator <constant>

<constant> relational_operator operator(...)

operator(...) LIKE <constant>
```

You can define a function, `ODCIStatsFunctionCost`, for computing the cost of standalone functions, package functions, or type methods. This function takes three sets of input parameters describing the predicate:

- `func` describing the function `operator`;
- `args` describing the actual arguments to the function `operator`;
- and a list of function arguments whose number, position, and type must match the arguments of the function `operator`.

The `ODCIStatsFunctionCost` function returns its computed cost in the `cost` parameter. As mentioned, the returned cost can have two components — CPU and I/O — which are combined by the optimizer to compute a composite cost. The costs returned by user-defined cost functions must be positive whole numbers. Invalid values are ignored by the optimizer.

The return value of the `ODCIStatsFunctionCost` function must be

- `Success` indicating success, or
- `Error` indicating an error, or
- `Warning` indicating a warning.

Consider a function `myFunction` defined as follows:

```
myFunction (a NUMBER, b VARCHAR2(10)) return NUMBER
```

A user-defined cost function for the function `myFunction` would be coded as follows:

```
ODCIStatsFunctionCost(func ODCIFuncInfo, cost OUT ODCICost,
   args ODCIArgDescList, a NUMBER, b VARCHAR2(10), env ODCIEnv) return NUMBER
```

If the function `myFunction` is called with literal arguments—for example,

```
myFunction(2, ’TEST’) > 5,
```

then the cost function is called as follows:

```
ODCIStatsFunctionCost(<ODCIFuncInfo constructor>, cost,
   <ODCIArgDescList constructor>, 2, 'TEST', <ODCIEnv flag>)
```

If, on the other hand, the function `myFunction` is called with some non-literals—for example,

```
myFunction(Test_tab.col_a, 'TEST') > 5
```

where `col_a` is a column in table `Test_tab`, then the cost function is called as follows:

```
ODCIStatsFunctionCost(<ODCIFuncInfo constructor>, cost,
   <ODCIArgDescList constructor>, NULL, 'TEST', <ODCIEnv flag>)
```

In other words, function argument values are passed to the cost function only if they are literals; otherwise they are `NULL`. The `ODCIArgDescList` descriptor describes all its following arguments.

User-Defined Cost Functions for Domain Indexes

User-defined cost functions for domain indexes are used for the same type of predicates mentioned previously, except that `operator` must be a user-defined operator for which a valid domain index access path exists.

The `ODCIStatsIndexCost` function takes these sets of parameters:

- `ia` describing the domain index
- `sel` representing the user-computed selectivity of the predicate
- `cost` giving the computed cost
- `qi` containing additional information about the query
- `pred` describing the predicate
- `args` describing the start and stop values (that is, <`constant`>) of the operator and the actual arguments to the operator `operator`
- `start`, whose datatype is the same as that of the operator's return value, describing the start value of the operator
- `stop` whose datatype is the same as that of the operator's return value, describing the stop value of the operator

- a list of operator value arguments whose number, position, and type must match the arguments of the operator `operator`. The value arguments of an operator are the arguments excluding the first argument.
- `env`, an environment flag set by the server to indicate which call is being made in cases where multiple calls are made to the same routine. The flag is reserved for future use; currently it is always set to `0`.

The computed cost of the domain index is returned in the output parameter, `cost`.

`ODCIStatsIndexCost` returns

- `Success` indicating success, or
- `Error` indicating an error, or
- `Warning` indicating a warning.

Consider an operator

```
Contains(a_string VARCHAR2(2000), b_string VARCHAR2(10))
```

that returns 1 or 0 depending on whether or not the string `b_string` is contained in the string `a_string`. Further, assume that the operator is implemented by a domain index. A user-defined index cost function for this domain index would be coded as follows:

```
ODCIStatsIndexCost(ia ODCIIndexInfo, sel NUMBER, cost OUT ODCICost,
   qi ODCIQueryInfo, pred ODCIPredInfo, args ODCIArgDescList,
   start NUMBER, stop NUMBER, b_string VARCHAR2(10), env ODCIEnv) return NUMBER
```

Note that the first argument, `a_string`, of `Contains` does not appear as a parameter of `ODCIStatsIndexCost`. This is because the first argument to an operator must be a column for the domain index to be used, and this column information is passed in through the `ODCIIndexInfo` parameter. Only the operator arguments after the first (the "value" arguments) must appear as parameters to the `ODCIStatsIndexCost` function.

If the operator is called—for example:

```
Contains(Test_tab.col_c,'TEST') <= 1
```

then the index cost function is called as follows:

```
ODCIStatsIndexCost(<ODCIIndexInfo constructor>, sel, cost,
   <ODCIQueryInfo constructor>, <ODCIPredInfo constructor>,
   <ODCIArgDescList constructor>, NULL, 1, 'TEST', <ODCIEnv flag>)
```

In other words, the start, stop, and operator argument values are passed to the index cost function only if they are literals; otherwise they are `NULL`. The `ODCIArgDescList` descriptor describes all its following arguments.

Using User-Defined Statistics, Selectivity, and Cost

Statistics types act as interfaces for user-defined functions that influence the choice of an execution plan by the optimizer. However, for the optimizer to be able to use a statistics type requires a mechanism to bind the statistics type to a database object (column, standalone function, object type, index, indextype or package; you cannot associate a statistics type with a partition of a table or a partition of a domain index). Creating this association is the job of the `ASSOCIATE STATISTICS` command. The following sections describe this command in more detail.

User-Defined Statistics

User-defined statistics functions are relevant for columns (both standard SQL datatypes and object types) and domain indexes. The functions `ODCIStatsSelectivity`, `ODCIStatsFunctionCost`, and `ODCIStatsIndexCost` are not used for user-defined statistics, so statistics types used only to collect user-defined statistics need not implement these functions. The following sections describe how column and index user-defined statistics are collected.

User-collected statistics can either be stored in some predefined dictionary tables or users could create their own tables. The latter approach requires that privileges on these tables be administered properly, backup and restoration of these tables be done along with other dictionary tables, and point-in-time recovery considerations be resolved.

Statistics are stored in a predefined system table for use by user-defined selectivity and cost functions. Three system views of this table are available:

- DBA_USTATS
- ALL_USTATS
- USER_USTATS

See Also: *Oracle9i Database Reference* for information about the *_USTATS views

Column Statistics

Consider a table Test_tab defined as follows:

```
CREATE TABLE Test_tab (
   col_a    NUMBER,
   col_b    typ1,
   col_c    VARCHAR2(2000)
)
```

where typ1 is an object type. Suppose that stat is a statistics type that implements ODCIStatsCollect and ODCIStatsDelete functions. User-defined statistics are collected by the ANALYZE command for the column col_b if we bind a statistics type with the column as follows:

```
ASSOCIATE STATISTICS WITH COLUMNS Test_tab.col_b USING stat
```

A list of columns can be associated with the statistics type stat. Note that Oracle supports only associations with top-level columns, not attributes of object types; if you wish, the ODCIStatsCollect function can collect individual attribute statistics by traversing the column.

Another way to collect user-defined statistics is to declare an association with a datatype as follows:

```
ASSOCIATE STATISTICS WITH TYPES typ1 USING stat_typ1
```

which declares stat_typ1 as the statistics type for the type typ1. When the table Test_tab is analyzed with this association, user-defined statistics are collected for the column col_b using the ODCIStatsCollect function of statistics type stat_typ1.

Individual column associations always have precedence over associations with types. Thus, in the preceding example, if both ASSOCIATE STATISTICS commands are issued, ANALYZE would use the statistics type stat (and *not* stat_typ1) to collect user-defined statistics for column col_b. It is also important to note that standard statistics, if possible, are collected along with user-defined statistics.

User-defined statistics are deleted using the ODCIStatsDelete function from the same statistics type that was used to collect the statistics.

Associations defined by the ASSOCIATE STATISTICS command are stored in a dictionary table called ASSOCIATION$.

Only user-defined datatypes can have statistics types associated with them; you cannot declare associations for standard SQL datatypes.

Domain Index Statistics

A domain index has an indextype. A statistics type for a domain index is defined by associating it either with the index or its indextype. Consider the following example using the table `Test_tab` we defined earlier:

```
CREATE INDEX Test_indx ON Test_tab(col_a)
INDEXTYPE IS indtype PARAMETERS('example');

CREATE OPERATOR userOp BINDING (NUMBER) RETURN NUMBER
USING userOp_func;

CREATE INDEXTYPE indtype
FOR userOp(NUMBER)
USING imptype;
```

Here, `indtype` is the indextype, `userOp` is a user-defined operator supported by `indtype`, `userOp_func` is the functional implementation of `userOp`, and `imptype` is the implementation type of the indextype `indtype`.

A statistics type `stat_Test_indx` can be associated with the index `Test_indx` as follows:

```
ASSOCIATE STATISTICS WITH INDEXES Test_indx USING stat_Test_indx
```

When the domain index `Test_indx` is analyzed, user-defined statistics for the index are collected by calling the `ODCIStatsCollect` function of `stat_Test_indx`.

If a statistics type association is not defined for a specific index, Oracle looks for a statistics type association for the indextype of the index. In the preceding example, a statistics type `stat_indtype` can be associated with the indextype `indtype` as follows:

```
ASSOCIATE STATISTICS WITH INDEXTYPES indtype USING stat_indtype
```

When the domain index `Test_indx` is analyzed and no statistics type association has been defined for the index `Test_indx`, then user-defined statistics for the index are collected by calling the `ODCIStatsCollect` function of `stat_indtype`.

Thus, individual domain index associations always have precedence over associations with the corresponding indextypes.

Domain index statistics are dropped using the `ODCIStatsDelete` function from the same statistics type that was used to collect the statistics.

User-Defined Selectivity

Selectivity functions are used by the optimizer to compute the selectivity of predicates in a query. The predicates must have one of the appropriate forms and can contain user-defined operators, standalone functions, package functions, or type methods. Selectivity computation for each is described in the following sections.

User-defined Operators

Consider the example laid out earlier, and suppose that the following association is declared:

```
ASSOCIATE STATISTICS WITH FUNCTIONS userOp_func USING stat_userOp_func
```

Now, if the following predicate

```
userOp(Test_tab.col_a) = 1
```

is encountered, the optimizer calls the `ODCIStatsSelectivity` function (if present) in the statistics type `stat_userOp_func` that is associated with the functional implementation of the `userOp_func` of the `userOp` operator.

Standalone Functions

If the association

```
ASSOCIATE STATISTICS WITH FUNCTIONS myFunction USING stat_MyFunction
```

is declared for a standalone function `myFunction`, then the optimizer calls the `ODCIStatsSelectivity` function (if present) in the statistics type `stat_myFunction` for the following predicate (for instance):

```
myFunction(Test_tab.col_a, ’TEST’) = 1.
```

Package Functions

If the association

```
ASSOCIATE STATISTICS WITH PACKAGES Demo_pack USING stat_Demo_pack
```

is declared for a package `Demo_pack`, then the optimizer calls the `ODCIStatsSelectivity` function (if present) in the statistics type `stat_Demo_pack` for the following predicate (for instance):

```
Demo_pack.myDemoPackFunction(Test_tab.col_a, ’TEST’) = 1
```

where `myDemoPackFunction` is a function in `Demo_pack`.

Type Methods

If the association

```
ASSOCIATE STATISTICS WITH TYPES Example_typ USING stat_Example_typ
```

is declared for a type `Example_typ`, then the optimizer calls the `ODCIStatsSelectivity` function (if present) in the statistics type `stat_Example_typ` for the following predicate (for instance):

```
myExampleTypMethod(Test_tab.col_b) = 1
```

where `myExampleTypMethod` is a method in `Example_typ`.

Default Selectivity

An alternative to selectivity functions is user-defined *default selectivity*. The default selectivity is a value (between 0% and 100%) that is looked up by the optimizer instead of calling a selectivity function. Default selectivities can be used for predicates with user-defined operators, standalone functions, package functions, or type methods.

The following command:

```
ASSOCIATE STATISTICS WITH FUNCTIONS myFunction DEFAULT SELECTIVITY 20
```

declares that the following predicate, for instance,

```
myFunction(Test_tab.col_a) = 1
```

always has a selectivity of 20 percent (or 0.2) regardless of the parameters of `myFunction`, or the comparison operator "=", or the constant "1". The optimizer uses this default selectivity instead of calling a selectivity function.

An association can be declared using either a statistics type or a default selectivity, but not both. Thus, the following statement is illegal:

```
ASSOCIATE STATISTICS WITH FUNCTIONS myFunction USING stat_myFunction
   DEFAULT SELECTIVITY 20
```

The following are some more examples of default selectivity declarations:

```
ASSOCIATE STATISTICS WITH PACKAGES Demo_pack DEFAULT SELECTIVITY 20
ASSOCIATE STATISTICS WITH TYPES Example_typ DEFAULT SELECTIVITY 20
```

User-Defined Cost

The optimizer uses user-defined cost functions to compute the cost of predicates in a query. The predicates must have one of the forms listed earlier and can contain user-defined operators, standalone functions, package functions, or type methods. In addition, user-defined cost functions are also used to compute the cost of domain indexes. Cost computation for each is described in the following sections.

User-defined Operators

Consider the example outlined in the preceding section, and suppose that the following associations are declared:

```
ASSOCIATE STATISTICS WITH INDEXES Test_indx USING stat_Test_indx
ASSOCIATE STATISTICS WITH FUNCTIONS userOp USING stat_userOp_func
```

Consider the following predicate:

```
userOp(Test_tab.col_a) = 1.
```

If the domain index `Test_indx` implementing `userOp` is being evaluated, the optimizer calls the `ODCIStatsIndexCost` function (if present) in the statistics type `stat_Test_indx`. If the domain index is not used, however, the optimizer calls the `ODCIStatsFunctionCost` function (if present) in the statistics type `stat_userOp` to compute the cost of the functional implementation of the operator `userOp`.

Standalone Functions

If the association

```
ASSOCIATE STATISTICS WITH FUNCTIONS myFunction USING stat_myFunction
```

is declared for a standalone function `myFunction`, then the optimizer calls the `ODCIStatsFunctionCost` function (if present) in the statistics type `stat_myFunction` for the following predicate (for instance):

```
myFunction(Test_tab.col_a, 'TEST') = 1
```

User-defined function costs do not influence the choice of access methods; they are only used for ordering predicates (described in the *Extensible Optimizer* reference).

Package Functions

If the association

```
ASSOCIATE STATISTICS WITH PACKAGES Demo_pack USING stat_Demo_pack;
```

is declared for a package `Demo_pack`, then the optimizer calls the `ODCIStatsFunctionCost` function (if present) in the statistics type `stat_Demo_pack` for the following predicate (for instance):

```
Demo_pack.myDemoPackFunction(Test_tab.col_a) = 1
```

where `myDemoPackFunction` is a function in `Demo_pack`.

Type Methods

If the association

```
ASSOCIATE STATISTICS WITH TYPES Example_typ USING stat_Example_typ;
```

is declared for a type `Example_typ`, then the optimizer calls the `ODCIStatsFunctionCost` function (if present) in the statistics type `stat_Example_typ` for the following predicate:

```
myExampleTypMethod(Test_tab.col_b) = 1
```

where `myExampleTypMethod` is a method in `Example_typ`.

Default Cost

Like default selectivity, default costs can be used for predicates with user-defined operators, standalone functions, package functions, or type methods. So, the following command

```
ASSOCIATE STATISTICS WITH INDEXES Test_indx DEFAULT COST (100, 5, 0)
```

declares that using the domain index `Test_indx` to implement the following predicate (to select one example)

```
userOp(Test_tab.col_a) = 1
```

always has a CPU cost of 100, I/O of 5, and network of 0 (the network cost is ignored in Oracle) regardless of the parameters of `userOp`, the comparison operator "=", or the constant "1". The optimizer uses this default cost instead of calling an `ODCIStatsIndexCost` cost function.

You can declare an association using either a statistics type or a default cost but not both. Thus, the following statement is illegal:

```
ASSOCIATE STATISTICS WITH INDEXES Test_indx USING stat_Test_indx
   DEFAULT COST (100, 5, 0)
```

The following are some more examples of default cost declarations:

```
ASSOCIATE STATISTICS WITH FUNCTIONS myFunction DEFAULT COST (100, 5, 0)
ASSOCIATE STATISTICS WITH PACKAGES Demo_pack DEFAULT COST (100, 5, 0)
ASSOCIATE STATISTICS WITH TYPES Example_typ DEFAULT COST (100, 5, 0)
ASSOCIATE STATISTICS WITH INDEXTYPES indtype DEFAULT COST (100, 5, 0)
```

Declaring a NULL Association for an Index or Column

An association of a statistics type defined for an indextype or object type is inherited by index instances of that indextype and by columns of that object type. An inherited association can be overridden by explicitly defining a different association for an index instance or column, but there may be occasions when you would prefer an index or column not to have any association at all. For example, for a particular query the benefit of a better plan may not outweigh the additional compilation time incurred by invoking the cost or selectivity functions. For cases like this, you can use the `ASSOCIATE` command to declare a `NULL` association for a column or index.

```
ASSOCIATE STATISTICS WITH COLUMNS <columns> NULL;
ASSOCIATE STATISTICS WITH INDEXES <indexes> NULL;
```

If the `NULL` association is specified, the schema object does not inherit any statistics type from the column type or the indextype. A `NULL` association also precludes default values.

How Statistics Are Affected by DDL Operations

Partition-level and schema object-level aggregate statistics are affected by DDL operations in the same way as standard statistics. Table 8–2 summarizes the effects.

Table 8–2 Effects of DDL on Statistics

Operation	Effect on Partition Statistics	Effect on Global Statistics
ADD PARTITION	None	No Action
DROP PARTITION	Statistics deleted	Statistics recalculated
SPLIT PARTITION	Statistics deleted	None
MERGE PARTITION	Statistics deleted	None
TRUNCATE PARTITION	Statistics deleted	None

Table 8–2 Effects of DDL on Statistics

Operation	Effect on Partition Statistics	Effect on Global Statistics
EXCHANGE PARTITION	Statistics deleted	Statistics recalculated
REBUILD PARTITION	None	None
MOVE PARTITION	None	None
RENAME PARTITION	None	None

If statistics for any partition are deleted, aggregate statistics for that object are deleted, and the aggregate statistics for the table or the index are recalculated.

If an existing partition is exchanged, or dropped with an `ALTER TABLE DROP PARTITION` statement, and the `_minimal_stats_aggregation` parameter is set to `FALSE`, the statistics for that partition are deleted, and the aggregate statistics of the table or index are recalculated.

Predicate Ordering

In the absence of an `ORDERED_PREDICATES` hint (discussed on page 18-3), predicates (except those used for index keys) are evaluated in the order specified by the following rules:

- Predicates without any user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the `WHERE` clause.
- Predicates with user-defined functions and type methods which have user-computed costs are evaluated in increasing order of their cost.
- Predicates with user-defined functions and type methods that have no user-computed cost are evaluated next, in the order specified in the `WHERE` clause.
- Predicates not specified in the `WHERE` clause (for example, predicates transitively generated by the optimizer) are evaluated next.
- Predicates with subqueries are evaluated last in the order specified in the `WHERE` clause.

Dependency Model

The dependency model reflects the actions that are taken when you issue any of the SQL commands described in Table 8–3.

Table 8–3 Dependency Model for DDLs

Command	Action
DROP statistics_type	if an association is defined with statistics_type, the command fails, otherwise the type is dropped
DROP statistics_type FORCE	calls DISASSOCIATE FORCE for all objects associated with the statistics_type; drops statistics_type
DROP object	calls DISASSOCIATE, drops object_type if DISASSOCIATE succeeds
ALTER TABLE DROP COLUMN	if association is present for the column, this calls DISASSOCIATE FORCE with column; if no entry in ASSOCIATION$ but there are entries in type USATS$, then ODCIStatsDelete for the columns is invoked
DISASSOCIATE	if user-defined statistics collected with the statistics type are present, the command fails
DISASSOCIATE FORCE	deletes the entry in ASSOCIATION$ and calls ODCIStatsDelete
ANALYZE TABLE DELETE STATISICS	the ODCIStatsDelete function is invoked; if any errors are raised, ANALYZE fails and the error is reported
ASSOCIATE	if an association or user-defined statistics are present for the associated object, the command fails

Restrictions and Suggestions

A statistics type is an ordinary object type. Since an object type must have at least one attribute, a statistics type also must have at least one attribute. This will be a dummy attribute, however, since it will never be set or accessed.

Parallel Query

In Oracle9*i* domain indexes are non-partitioned and serial. The optimizer computes the composite cost of a domain index access path assuming a serial execution.

Distributed Execution

Oracle’s distributed implementation does not support adding functions to the remote capabilities list. All functions referencing remote tables are executed as filters. The placement of the filters occurs outside the optimizer. The cost model reflects this implementation and does not attempt to optimize placement of these predicates.

Since predicates are not shipped to the remote site, you cannot use domain indexes on remote tables. Therefore, the `DESCRIBE` protocol is unchanged, and remote domain indexes are not visible from the local site.

Performance

The cost of execution of the queries remains the same with the extensible optimizer if the same plan is chosen. If a different plan is chosen, the execution time should be better assuming that the user-defined cost, selectivity, and statistics collection functions are accurate. In light of this, you are strongly encouraged to provide statistics collection, selectivity, and cost functions for user-defined structures because the optimizer defaults can be inaccurate and lead to an expensive execution plan.

9

Using Cartridge Services

This chapter describes how to use cartridge services, including:

Cartridge Services — Introduction

This chapter describes a set of services that will help you create data cartridges in the Oracle Extensibility framework.

Using Oracle Cartridge Services offers you these advantages:

Portability

Oracle Cartridge Services offers you the flexibility to work across different machine architectures

Flexibility Within Oracle Environments

Another type of flexibility is offered to you in terms of the fact that all cartridge services will work with your Oracle Database irrespective of the configuration of operations that has been purchased by your client.

Language Independence

The use of the Globalization Support services lets you internationalize your cartridge. Language independence means that you can have different instances of your cartridge operating in different language environments.

Tight Integration with the Server

Various cartridge services have been designed to facilitate access with Oracle ORDBMS. This offers far superior performance to client -side programs attempting to perform the same operations.

Guaranteed Compatibility

Oracle is a rapidly evolving technology and it is likely that your clients might be operating with different releases of Oracle. The cartridge services will operate with all versions of Oracle database.

Integration of Different Cartridges

The integration of cartridge services lets you produce a uniform integration of different data cartridges.

The following sections provide a brief introduction to the set of services that you can use as part of your data cartridge. The APIs that describe these interfaces are described in Chapter 9, "Using Cartridge Services"

Cartridge Handle

Cartridge services require various handles that are encapsulated inside two types of OCI handles -

- **Environment handle** (`OCIEnv` or `OCI_HTYPE_ENV`).

  Various cartridge services are required at the process level when no session is available. The `OCIInitialize`() should use the `OCI_OBJECT` option for cartridge service.

- **User Session handle** (`OCISession` or `OCI_HTYPE_SESSION`).

  In a callout, the services can be used when the handle is allocated even without opening a connection back to the database.

All cartridge service calls take a `dvoid *` OCI handle as one of the arguments that may be either an environment or a session handle. While most service calls are allowed with either of the handles, certain calls may not be valid with one of the handles. For example, it may be an error to allocate `OCI_DURATION_SESSION` with an environment handle. An error will typically be returned in an error handle.

Client Side Usage

Most of the cartridge service can also be used on the client side code. Refer to individual services for restrictions. To use cartridge service on the client side, the OCI environment has to be initialized with `OCI_OBJECT` option. This is automatically effected in a cartridge.

Cartridge Side Usage

Most of the services listed in this document can be used in developing a database cartridge, but please refer to documentation of each individual service for restrictions. New service calls are available to obtain the session handle in a callout. The session handle is available without opening a connection back to the server.

Service Calls

Before using any service, the OCI environment handle must be initialized. All the services take an OCI environment (or user_session) handle as an argument. Errors are returned in an OCI error handle. The sub handles required for various service calls are not allocated along with the OCI environment handle. Services which need to initialize an environment provide methods to initialize it.

The following example demonstrates the initialization of these handles:

```
{
OCIEnv *envhp;
OCIError *errhp;
(void) OCIInitialize(OCI_OBJECT, (dvoid *)0, 0, 0, 0);
(void) OCIEnvInit(&envhp, OCI_OBJECT, (size_t)0, (dvoid **)0);
(void) OCIHandleAlloc((dvoid *)envhp, (dvoid **)errhp,   OCI_HTYPE_ERROR, (size_
t)0, (dvoid **)0);
/* ... use the handles ... */
(void) OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
}
```

Error Handling

Routines that return errors will generally return `OCI_SUCCESS` or `OCI_ERROR`. Some routines may return `OCI_SUCCESS_WITH_INFO`, `OCI_INVALID_HANDLE`, or `OCI_NO_DATA`. If `OCI_ERROR` or `OCI_SUCCESS_WITH_INFO` is returned, then an error code, an error facility, and possibly an error message can be retrieved by calling `OCIErrorGet`:

```
{
OCIError *errhp;
ub4 errcode;
text buffer[512];
(void) OCIErrorGet((dvoid *)errhp, 1, (text *)NULL, &errcode, buffer,
                   sizeof(buffer), OCI_HTYPE_ERROR);
}
```

Memory Services

Memory management is one of the services that is required by cartridge developers.

The memory service allows the client to allocate or free memory chunks. Each memory chunk is associated with a duration. This allows clients to automatically free all memory associated with a duration (at the end of the duration). The duration determines the heap that is used to allocate the memory. The memory service predefines three kinds of durations: call (`OCI_DURATION_CALL`), statement (`OCI_DURATION_STATEMENT`) and session (`OCI_DURATION_SESSION`).

The client can also create a user duration. The client has to explicitly start and terminate a user duration. Thus, the client can control the ’length’ of a user duration. Like the predefined durations, a user duration can be used to specify the allocation duration (for example, memory chunks are freed at the end of the user duration).

Each user duration has a parent duration. A user duration terminates implicitly when its parent duration terminates. A parent duration can be call, statement, transaction, session or any other user duration. Memory allocated in the user duration comes from the heap of its parent duration.

The Oracle RDBMS memory manager already supports a variety of memory models. Currently callouts support memory for the duration of that callout. With the extension of row sources to support external indexing, there is a need for memory of durations greater than a callout.

The following functionality is supported:

- Allocate (permanent and freeable) memory of following durations
  - call to agent process
  - statement
  - session
  - shared attributes (metadata) for cartridges
- Ability to re-allocate memory
- Ability to create a subduration memory, a sub heap which gets freed up when the parent heap gets freed up. Memory for this sub heap can be allocated and freed.
- Ability to specify zeroed memory
- Ability to allocate large contiguous memory

Maintaining Context

Context management allows the clients to store values across calls. Cartridge services provide a mechanism for saving and restoring context.

Most operating systems which support threads have the concept of thread context. Threads can store thread specific data in this context (or state) and retrieve it at any point. This provides a notion of thread global variable. Typically a pointer which points to the root of a structure is stored in the context.

When the row source mechanism is externalized, you will need a mechanism to maintain state between multiple calls to the same row source.

There is a need to maintain session, statement and process states. Session state includes information about multiple statements that are open, message files based on sessions' Globalization Support settings, and so on. Process state includes shared metadata (including systemwide metadata), message files, and so on. Depending on whether the cartridge application is truly multi threaded, information sharing can be at a process level or system level.

Since a user can be using multiple cartridges at any time, the state must be maintained for each cartridge. This is done by requiring the user to supply a key for each duration.

Durations

There are various predefined types of durations supported on memory and context management calls. An additional parameter in all these calls is a context.

- `OCI_DURATION_CALL`. The duration of this operation is that of a callout.
- `OCI_DURATION_STATEMENT`. The duration of this operation is the external row source.
- `OCI_DURATION_SESSION`. The duration of this operation is the user session.
- `OCI_DURATION_PROCESS`. The duration of this is agent process.

Globalization Support

To support multilingual application, Globalization Support functionality is required for cartridges and callouts. `NLSRTL` is a multiplatform, multilingual library current used in RDBMS and provides consistent Globalization Support behavior to all Oracle products.

Globalization Support basic services will provide the following language and cultural sensitive functionality:

- Locale information retrieval.
- String manipulation in the format of multibyte and wide-char.
- Character set conversion including Unicode support.
- Messaging mechanism.

Globalization Support Language Information Retrieval

An Oracle locale consists of language, territory and character set definitions. The locale determines conventions such as native day and month names; and date, time, number, and currency formats. An internationalized application will obey a user's locale setting and cultural convention. For example, in a German locale setting, users will expect to see day and month names in German spelling. The following interface provides a simple way to retrieve local sensitive information.

String Manipulation

Two types of data structure are supported for string manipulation: multibyte string and wide char string. Multibyte string is in native Oracle character set encoding, and functions operated on it take the string as a whole unit. Wide char string function provides more flexibility in string manipulation and supports character-based and string-based operations.

The wide char data type we use here is Oracle-specific and not to be confused with the `wchar_t` defined by the ANSI/ISO C standard. The Oracle wide char is always 4 bytes in all the platforms, while `wchar_t` is dependent on the implementation and platform. The idea of Oracle wide char is to normalize multibyte characters to have a fixed-width for easy processing. Round-trip conversion between Oracle wide char and native character set is guaranteed.

The string manipulation can be classified into the following categories:

- Conversion of string between multibyte and wide char.
- Character classifications.
- Case conversion.
- Display length calculation.
- General string manipulation, such as compare, concatenation and searching.

Parameter Manager Interface

The parameter manager provides a set of routines to process parameters from a file or a string. Routines are provided to process the input and to obtain key and value pairs. These key and value pairs are stored in memory and routines are provided which can access the values of the stored parameters.

The input processing routines match the contents of the file or the string against an existing grammar and compare the key names found in the input against the list of known keys that the user has registered. The behavior of the input processing routines can be configured depending on the bits that are set in the flag argument.

The parameters can be retrieved either one at a time or all at once by calling a function that iterates over the stored parameters.

Input Processing

Parameters consist of a key, or parameter name, type, and a value and must be specified in the following format:

```
key = value
```

Parameters can optionally accept lists of values which may be surrounded by parentheses. The following two formats are acceptable for specifying a value list:

```
key = (value1 value2 ... valuen)
key = value1 value2 ... valuen
```

A value can be a string, integer, `OCINumber`, or Boolean. A boolean value starting with '`y`' or '`t`' maps to `TRUE` and a boolean value starting with '`n`' or '`f`' maps to `FALSE`. The matching for boolean values is case insensitive.

The parameter manager views certain characters as “special characters” which are not parsed literally. The special characters and their meanings are indicated in Table 9–1.

Table 9–1 Special Characters and their Meanings

Character	Meaning
#	Comment (only for files)
(	Start a list of values
)	End a list of values
"	Start or end of quoted string
'	Start or end of quoted string
=	Separator of keyword and value
\	Escape character

If a special character must be treated literally, then it must either be prefaced by the escape character or the entire string must be surrounded by single or double quotes.

A key string can contain alphanumeric characters only. A value can contain any characters. However, the value cannot contain special characters unless they are quoted or escaped.

Parameter Manager Behavior Flag

The routines to process a file or a string take a behavior flag that can alter certain default characteristics of the parameter manager. The following bits can be set in the flag to produce the new behavior:

- `OCI_EXTRACT_CASE_SENSITIVE`. All comparisons are case sensitive. The default is to use case insensitive comparisons.
- `OCI_EXTRACT_UNIQUE_ABBREVS`. Unique abbreviations are allowed for keys. The default is that unique abbreviations are not allowed.
- `OCI_EXTRACT_APPEND_VALUES`. If a value or values are already stored for a particular key, then any new values for this key should be appended. The default is to return an error.

Key Registration

Before invoking the input processing routines (`OCIExtractFromFile` or `OCIExtractFromString`), all of the keys must be registered by calling `OCIExtractSetNumKeys` followed by `OCIExtractSetKey`. `OCIExtractSetKey` requires the following information for each key:

- Name of the key
- Type of the key (`integer`, `string`, `boolean`, `OCINumber`)
- `OCI_EXTRACT_MULTIPLE` is set for the flag value if multiple values are allowed (default: only one value allowed)
- Default value to be used for the key (may be `NULL`)
- Range of allowable integer values given by the starting and ending value, inclusive (may be `NULL`)
- List of allowable string values (may be `NULL`)

Parameter Storage and Retrieval

The results of processing the input into a set of keys and values are stored. The validity of the parameters is checked before storing the parameters in memory. The values are checked to see if they are of the proper type. In addition, if you wish, the values can be checked to see if they fall within a certain range of integer values or are members of a list of enumerated string values. Also, if you do not specify that a key can accept multiple values, then an error will be returned if a key is specified more than once in a particular input source. Also, an error will be returned if the key is unknown.

After the processing is completed, the value(s) for a particular key can be queried. Separate routines are available to retrieve a string value, an integer value, an `OCINumber` value, and a boolean value.

It is possible to retrieve all parameters at once. The function `OCIExtractToList` must first be called to generate a list of parameters that is created from the parameter structures stored in memory. `OCIExtractToList` will return the number of unique keys stored in memory, and then `OCIExtractFromList` can be called to return the list of values associated with each key.

Parameter Manager Context

The parameter manager maintains its own context within the `OCI` environment handle. This context stores all the processed parameter information and some internal information. It must be initialized with a call to `OCIExtractInit` and cleaned up with a call to `OCIExtractTerm`.

File I/O

The OCI file I/O package is designed to make it easier for you to write portable code that interacts with the file system by providing a consistent view of file I/O across multiple platforms.

You need to be aware of two issues when using this package in a data cartridge environment. The first issue is that this package does not provide any security when opening files for writing or when creating new files in a directory other than the security provided by the operating system protections on the file and directory. The second issue is that this package will not support the use of file descriptors across calls in a multithreaded server environment.

String Formatting

The OCI string formatting package facilitates writing portable code that handles string manipulation by means of the `OCIFormatString` routine. This is an improved and portable version of `sprintf` that incorporates additional functionality and error checking that the standard `sprintf` does not. This additional functionality includes:

- Arbitrary argument selection.
- Variable width and precision specification.
- Length checking of the buffer.
- Oracle Globalization Support for internationalization.

Part III

Advanced Topics

10

Design Considerations

This chapter describes various design considerations, including:

Designing the Types

Structured and Unstructured Data

Structured data is data that can be represented to Oracle in the form of an object type. Unstructured data—that is, data of type `RAW` or `BLOB`—cannot be interpreted by Oracle. The choice whether to model cartridge data as structured or unstructured depends on the following considerations:

1. Structured data can be shared by different applications since the structure is published in Oracle.
2. Structured types provide strong type checking whereas unstructured data does not.
3. Structured data is easily queried whereas unstructured data is not. One has to publish user-defined functions to facilitate querying the unstructured data.
4. Constraints are easily supported on structured data but not on unstructured data.
5. Indexes are easily supported on structured data, whereas, on unstructured data, indexes on user-defined functions would need to be created, or extensible indexes would need to be defined.
6. Structured data needs to be marshalled by Oracle to be retrieved to client as a value, whereas, unstructured data is easily retrievable as a value.

Using Nested Tables or VARRAYs

Unlike nested tables, `VARRAY`s are an *ordered* set of items. Physically, `VARRAY`s are stored as `RAW` or `LOB` columns, whereas nested tables are stored in tables. Following are some considerations to weigh in choosing which sort of collection better suits your purpose.

Nested Tables

- Nested tables can be queryed since elements are represented as rows.
- Indexes may be created on columns of nested tables for faster searches.
- Constraints may be specified for nested tables.
- Clustering of nested elements belonging to a common parent row is possible when the storage table is specified as an Index Organized Table, furthermore,

specifying key compression reduces the overhead of the system assigned `NESTED_TABLE_ID` values.

- When stored as a Heap Organized Table, creating an index on the `NESTED_TABLE_ID` column enhances retrieval of nested tables.
- Retrieving the nested table as a value for a given parent incurs the overhead of selecting and marshalling the individual rows to form the collection value.
- Even though parent tables may be partitioned, storage tables corresponding to their nested tables cannot be partitioned.

VARRAYs

- A VARRAYs is better suited for retrieval as a value since that is s how it is stored.
- Support for indexing, specification of constraints on VARRAYs is not available.
- Querying of VARRAYs is sub-optimal since rows have to materialized from collection value.
- Partitioning of VARRAYs stored as `LOB`s is permitted when the parent table is partitioned.

Based on the preceding implications, if the ability to query of update individual collection elements is important, then nested tables are a better choice to model your collection data. On the other hand, if your application is requires fetching the entire collection as a whole and then operating on it, modeling the collection data as a VARRAY will yield better retrieval performance.

Choosing a Language in Which to Write Methods

When writing methods for object types, you have the choice of implementing them in PL/SQL, C/C++, or Java. PL/SQL and Java methods run in the address space of the server. C/C++ methods are dispatched as external procedures and run outside the address space of the server.

The best implementation choice varies with the situation. Here are some guidelines:

1. A callout involving C or C++ is generally fastest if the processing is substantially CPU-bound. However, callouts incur the cost of dispatch, and if the amount of processing in C/C++ is not large then the cost of dispatch does not amortize very well.
2. PL/SQL tends to offer the best price-performance for methods that are not computation-intensive. The other implementation options are typically favored

over PL/SQL if you have a large body of code already implemented in another language that you want to use a part of the data cartridge

3. Java is a relatively open implementation choice. The interpreted nature of Java implies that for high performance applications, some sort of compilation of methods written in Java will be needed.

Invokers Rights — Why, When, How

Until release 8.1.5, stored procedures and SQL methods could only execute with the privileges of the definer. Such definer-rights routines are bound to the schema in which they reside, and this remains the default. Under this condition, a routine executes with the rights of the definer of the function, not the user invoking it. However, this is a limitation if the function statically or dynamically issues SQL statements.

For example, if the function had a static cursor that performs a SELECT from USER_TABLES, the USER_TABLES it would retrieve would be that of the definer irrespective of which user was using the function. For the function to be used against data not owned by the definer, explicit GRANTs had to be issued from the owner to the definer, or the function needed to be defined in the same schema where the data resided. The former course creates security and administration problems; the latter forces the function to be redefined in each schema that needs to use it.

The invoker-rights mechanism permits a function to execute with the privileges of the invoker. This permits cartridges to live within a schema dedicated to the cartridge and to be used by other schemas without requiring privileges be granted to operate on objects in the schema where the cartridge resides.

Callouts

When to Callout

You should consider utilizing callouts in the following circumstances:

- When it would be impractical or impossible to code the algorithm you require in SQL.
- When the performance gains of a compiled language (such as C or C++) outweigh the extproc callout overhead
- When you wish to leverage existing 3GL code

When to Callback

You should consider utilizing callbacks in the following circumstances:

- When you need data that was not passed as an argument to the call out.
- When it isn't practical to pass the data to the call out (for example, the number and size of the parameters exceeds that which is allowed or performs well).

Consider making a single callout which does multiple callbacks rather than multiple callouts (for example, instead of a factorial callout which takes a single number and computes a the factorial for it, consider making a callout which takes a VARRAY and repeatedly calls back to get next number to compute the factorial for. You always do performance testing to see at what at point the multi-call back approach out-performs the multi-callout approach.

Callouts and LOB

- It may be to your advantage to code your callout so that it is independent of `LOB` type (`BFILE`/`BLOB`).
- The PL/SQL layer of your cartridge can "open" your `BFILE` so that no `BFILE`-specific logic is required in your callout (other than error recovery from OCILob calls that do not operate on `BFILEs`).
- With the advent of temporary `LOB`s in Oracle8*i* release 8.1.5, you need to be aware of the deep copy that can occur when assignments and calls are done with temporary `LOB`s. Use "`NOCOPY`" (`BY REFERENCE`) on `BLOB` parameters as appropriate.

Saving and Passing State

External procedures under Oracle 8.0 have a "state-less" model. All Statement handles opened during the invocation of an external procedure are closed implicitly at the end of the call.

In Oracle9*i*, we allow "state" (OCI Statement handles and so forth, and associated state in the DBMS) to be saved and used across invocations of external procedures in a session. B y default cartridges are still stateless, however, `OCIMemory` services and `OCIContext` services can be used with `OCI_DURATION_SESSION` or other appropriate duration to save state. Statement handles created in one external procedure invocation can get re-used in another. The Data Cartridge developer needs to explicitly free these handles. It is recommended that this is done as soon as the statement handle is no longer needed. All state maintained for the statement in the OCI handles and in the DBMS would get freed as a result. This should help in improving the scalability of the Data Cartridge.

Designing Indexes

Influencing Index Performance

It is wrong to assume that creating domain index is always the best course. If, after careful consideration, you determine that you need to create domain index, you should keep the following factors in mind. For one, if the domain index is complex, the functional implementation will work better

- When the data size is small
- When the result is a large percentage of the total data size.

Judicious use of the extensible optimizer can lead to good performance.

Influencing Index Performance

Naming of internal components can be an issue. Naming of internal data objects for a domain index implementation and are typically based on names you provide for table and indexes. The problem is that the derived names for the internal objects should not conflict with any other user defined object or system object. You may have to develop some policy that restricts names, or implement some metadata management scheme to avoid errors during `DROP`, `CREATE`, and so on.

When to Use IOTs

You can create only one index on IOTs in 8.0.x releases. However, if most of your data is in the index, using an IOT is more efficient than storing your data in both a table and an additional index.

You can create secondary indexes on IOTs in Oracle9*i*. These offer a big advantage if you are accessing the data multiple ways.

Can Index Structures Be Stored in LOBs

Index structures can be stored in `LOB`s but take care to tune the `LOB` for best performance. If you are accessing a particular `LOB` frequently, create your table with the `CACHE` option and place the `LOB` index in a separate tablespace. If you are updating a `LOB` frequently, `TURN OFF LOGGING` and read / write in multiples of

CHUNK size. If you are accessing a particular portion of a LOB frequently, buffer your reads / writes using LOB buffering or your own buffering scheme.

External Index Structures

With the extensible indexing framework, the meaning and representation of a user-defined index is left to the cartridge developer. We do provide basic index implementations such as IOTs. In certain cases, binary or character LOBs can also be used to store complex index structures. IOTs, BLOBs and CLOBs all live within the database. In addition to them, you may also store a user-defined index as a structure external to the database, say in a BFILE.

The external index structure gives you the most flexibility in representing your index. An external index structure is particularly useful if you have already invested in the development of in-memory indexing structures. For example, an operating system file may store index data, which is read into a memory mapped file at run time. Such a case can be handled as a BFILE in the external index routines.

External index structures may also provide superior performance, although, this gain comes at some cost. Index structures external to the database do not participate in the transaction semantics of the database, which, in the case of index structures inside the database, make data and concomitant index updates atomic. This means that if an update to the data causes an update for the external index to be invoked through the extensible indexing interface, any failures may cause the data updates to be rolled back but not the index updates. The database can only roll back what is internal to it: external index structures cannot be rolled back in synchronization with a database rollback.

External index structures are perhaps most useful for read-only access. Their semantics become complex if updates to data are involved.

Multi-Row Fetch

```
ODCIIndexFetch(self IN [OUT] <impltype>, nrows IN NUMBER, rids OUT ODCIRidList)
RETURN NUMBER
```

When the ODCIIndexFetch routine is called, the ROWIDs of all the rows that satisfy the operator predicate are returned. The maximum number of rows that can be returned by the ODCIIndexFetch routine is nrows (nrows being an argument to the ODCIIndexFetch routine). The value of nrows is decided by Oracle based on some internal factors. If you have a better idea of the number of rows that ought to be returned to achieve optimal query performance, you can determine that this number of rows is returned in the `ODCIRidList` VARRAY instead of `nrows`. Note that the number of values in the `ODCIRidList` has to be less than or equal to `nrows`.

You, as cartridge designer, are in the best position to make a judgement regarding the number of rows to be returned. For example, if in the index the number of (say 1500) rowids are stored together and `nrows` = 2000, then it may be optimal to return 1500 rows in lieu of 2000 rows. Otherwise the user would have to retrieve 3000 rowids, return 2000 if them and note which 1000 rowids were not returned.

If you not have any specific optimization in mind, you can use the value of `nrows` to determine the number of rows to be returned. Currently the value of `nrows` has been set to 2000.

Anyone implementing indexes which use callouts should use *multirow fetch* to fetch the largest number of rows back to the server. This offsets the cost of making the callout.

Designing Operators

Functional and Index Implementations

All indexes should contain an indexed and functional implementation of the operator, in case the optimizer chooses not to use the indexed implementation. You can, however, use the indexing structures to produce the functional result.

Talking to the Optimizer

Weighing Cost and Selectivity

Estimating Cost

In Oracle9*i* only the CPU and I/O costs are considered.

Cost for functions The cost of executing a C function can be determined using common profilers or tools. For SQL queries, an explain plan of the query would give a rough estimate of the cost of the query. In addition the `tkprof` utility can be used to gather information about the CPU and the I/O cost involved in the operation. The cost of executing a callout could also be determined by using it in a SQL query which "selects from dual" and then estimating its cost from the `tkprof` utility.

Cost for Indexes The cost of the index is a function of the selectivity of the predicate (which is passed as an argument to the cost function) * the total number of data blocks in the index structures. Hence the index cost function should be one which increases with the increase in selectivity of the predicate. With a selectivity of 100%, the cost of accessing the index should be the cost of accessing all the data in all the structures that comprise the domain index.

The total cost of accessing the index is the cost of performing the `ODCIIndexStart`, N * `ODCIIndexFetch` and `ODCIIndexClose` operators, where N is the number of times the `ODCIIndexFetch` routine will be called based on the selectivity of the predicate. The cost of `ODCIIndexStart`, `ODCIIndexFetch` and `ODCIIndexClose` functions can be determined as discussed in the previous section.

Estimating Selectivity

Selectivity for Functions The selectivity of a predicate is the percentage of rows returned by the predicate divided by the total number of rows in the table(s).

The selectivity function should use the statistics collected for the table to determine what percentage of rows of the table will be returned by the predicate with the given list of arguments. For example, to compute the selectivity of a predicate `IMAGE_GREATER_THAN` (`Image SelectedImage`) which determines the images that are greater than the `Image SelectedImage`, a histogram of the sizes of the images in the database can be a useful statistics to compute the selectivity.

Collecting Statistics

Statistics can affect the calculation of selectivity for predicates and also the cost of domain indexes.

Statistics for Tables The statistics collected for a table can affect the computation of selectivity of a predicate. So statistics that can help the user make a better judgement about the selectivity of a predicate should be collected for a table/column. Knowing the predicates that would operate on the data will be helpful to determine what statistics would be good to collect.

Some example of statistics that can be useful in spatial domain for example could be the average/min/max number of elements in a `VARRAY` that contains the nodes of the spatial objects.

Note that standard statistics are collected in addition to the user defined statistics when the `ANALYZE` command is invoked.

Statistics for Indexes When a domain index is analyzed statistics for the underlying objects which constitute the domain index should be analyzed. For example if the domain index is comprised of tables, the statistics collection function should ANALYZE the tables when the domain index is analyzed. The cost of accessing the domain index can be influenced by the statistics that have been collected for the index. For example the cost of accessing a domain index could be approximated to the selectivity * the total number of data blocks (in the various tables) being accessed when the domain index is accessed.

Note: Oracle Corporation recommends that you use the DBMS_STATS package instead of the ANALYZE command to collect optimizer statistics. In a future release, functionality to collect optimizer statistics will be removed from ANALYZE.

See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about DBMS_STATS.

To accurately define cost, selectivity and statistics functions, a good understanding of the domain is required. The preceding guidelines are meant to help you understand some of the issues you need to take into account while working on the cost, selectivity and statistics functions. In general it may be a good idea to start of by using the default cost and selectivity and observe how the queries of interest behave.

Design for maintenance

- Carefully design your object types and methods. Object types are difficult to upgrade once they are in use by applications.
- Use OIDs in all of your object types so users can import/export data easily across databases.
- It is easy to add a method to a type, but hard to remove it.
- You are likely get more use out of the cartridge and the existing tool stack if you support functions against a traditional relational model in addition to an object model.
- Expose significant and frequently used data from your complex objects in object types as attributes so that you can build an index on them.
- If your cartridge maintains a large number of objects, views, tables, and so on, consider making a metadata table to maintain the relationships among the

objects for the user. This will ease the complexity of developing and maintaining the cartridge when it is in use.

How to Make Your Cartridge Extensible

- Keep your interface simple, and document it thoroughly.
- Use OO concepts appropriately.
- Ensure that your methods do not have side affects

How to Make Your Cartridge Installable

- Include a `README` with your cartridge to tell users how to install the cartridge
- Make the cartridge installable in one step in the database, if possible:

  ```
  sqlplus @imginst
  ```

- Tell users how to start the `listener` if you are using callouts.
- Tell users how to setup `extproc`. Most users have never heard of `extproc` and many users have never set up a listener. This is the primary problem when deploying cartridges.
- Using the software packager, you can easily create custom SQL install scripts by using the `’instantiate_file’` action. This is a great feature that enables you to substitute variables in your files when they are installed and it leaves your user with scripts and files that are customized for their machine.

Miscellaneous

How to Write Portable Cartridge Code

You should:

- Use the datatypes in oratypes.h
- Use OCI calls where ever possible.
- Use the switches which enforce ANSI C conformance when possible
- Use ANSI C function prototypes

- Build and test on your target platforms as early in your development cycle as possible (flush out platform specific code and allow as much time to redesign as possible).

You should avoid:

- Storing endian (big / little) specific data
- Storing floating point data (IEEE / VAX / other)
- Operating System-specific calls (if they can't be avoided, isolate them in a layer specific to the operating system; however, if the calls you require are not in the OCI, and also are not in POSIX, then you are likely to encounter intractable problems)
- `int` <-> `size_t` implicit casts on a 64 bit platform

11 User-Defined Aggregate Functions

Oracle provides a number of pre-defined aggregate functions such as `MAX`, `MIN`, `SUM` for performing operations on a set of rows. These pre-defined aggregate functions can be used only with scalar data. However, you can define your own custom implementations of these functions, or define entirely new aggregate functions, to use with complex data—for example, with multimedia data stored using object types, opaque types, and LOBs.

User-defined aggregate functions are used in SQL DML statements just like Oracle's own built-in aggregates. Once such functions are registered with the server, Oracle simply invokes the aggregation routines that you supplied instead of the native ones.

User-defined aggregates can be used with scalar data too. For example, it may be worthwhile to implement special aggregate functions for working with complex statistical data associated with financial or scientific applications.

User-defined aggregates are a feature of the Extensibility Framework. You implement them using `ODCIAggregate` interface routines. This chapter explains how.

This chapter contains these major sections:

- "The ODCIAggregate Interface: Overview"
- "Creating a User-Defined Aggregate"
- "Using a User-Defined Aggregate"
- "Parallel Evaluation of User-Defined Aggregates"
- "User-Defined Aggregates and Materialized Views"
- "User-Defined Aggregates and Analytic Functions"
- "Example: Creating and Using a User-Defined Aggregate"

See Also: Chapter 19, "Reference: User-Defined Aggregates Interface" for a description of the `ODCIAggregate` interface

The ODCIAggregate Interface: Overview

You create a user-defined aggregate function by implementing a set of routines collectively referred to as the `ODCIAggregate` routines. You implement the routines as methods within an object type, so the implementation can be in any Oracle-supported language for type methods, such as PL/SQL, C/C++ or Java. When the object type is defined and the routines are implemented in the type body, you use the `CREATE FUNCTION` statement to create the aggregate function.

Each of the four `ODCIAggregate` routines required to define a user-defined aggregate function codifies one of the internal operations that any aggregate function performs, namely:

- Initialize
- Iterate
- Merge
- Terminate

For example, consider the aggregate function `AVG()` in the following statement:

```
SELECT AVG(T.Sales)
FROM AnnualSales T
GROUP BY T.State;
```

To perform its computation, the aggregate function `AVG()` goes through steps like these:

1. **Initialize**: Initializes the computation:

   ```
   runningSum = 0; runningCount = 0;
   ```

2. **Iterate**: Processes each successive input value:

   ```
   runningSum += inputval; runningCount++;
   ```

3. **Terminate**: Computes the result:

   ```
   return (runningSum/runningCount);
   ```

In this example, the *Initialize* step initializes the aggregation context—the rows over which aggregation is performed. The *Iterate* step updates the context, and the *Terminate* step uses the context to return the resultant aggregate value.

If `AVG()` were a user-defined function, the object type that embodies it would implement a method for a corresponding `ODCIAggregate` routine for each of these steps. The variables `runningSum` and `runningCount`, which determine the state of the aggregation in the example, would be attributes of that object type.

Sometimes a fourth step may be necessary to merge two aggregation contexts and create a new context:

4. **Merge**: Combine the two aggregation contexts and return a single context:

   ```
   runningSum = runningSum1 + runningSum2;
   runningCount = runningCount1 + runningCount2
   ```

This operation combines the results of aggregation over subsets in order to obtain the aggregate over the entire set. This extra step can be required during either serial or parallel evaluation of an aggregate. If needed, it is performed before the *Terminate* step.

The four `ODCIAggregate` routines corresponding to the preceding steps are:

Routine	Description
`ODCIAggregateInitialize`	This routine is invoked by Oracle to initialize the computation of the user-defined aggregate. The initialized aggrgegation context is passed back to Oracle as an object type instance.
`ODCIAggregateIterate`	This routine is repeatedly invoked by Oracle. On each invocation, a new value (or a set of new values) is passed as input. The current aggregation context is also passed in. The routine processes the new value(s) and returns the updated aggregation context back to Oracle. This routine is invoked for every non-`NULL` value in the underlying group. (`NULL` values are ignored during aggregation and are not passed to the routine.)
`ODCIAggregateMerge`	This routine is invoked by Oracle to combine two aggregation contexts. This routine takes the two contexts as inputs, combines them, and returns a single aggregation context.
`ODCIAggregateTerminate`	This routine is invoked by Oracle as the final step of aggregation. The routine takes the aggregation context as input and returns the resulting aggregate value.

Creating a User-Defined Aggregate

The process of creating a user-defined aggregate function has two steps. Here is an overview of the steps, using the `SpatialUnion()` aggregate function defined by the spatial cartridge. The function computes the bounding geometry over a set of input geometries.

Step 1: Implement the ODCIAggregate interface

The `ODCIAggregate` routines are implemented as methods within an object type `SpatialUnionRoutines`. The actual implementation could be in any Oracle-supported language for type methods, such as PL/SQL, C/C++ or Java.

```
CREATE TYPE SpatialUnionRoutines(
STATIC FUNCTION ODCIAggregateInitialize( ... ) ...,
MEMBER FUNCTION ODCIAggregateIterate(...) ... ,
MEMBER FUNCTION ODCIAggregateMerge(...) ...,
MEMBER FUNCTION ODCIAggregateTerminate(...)
);

CREATE TYPE BODY SpatialUnionRoutines IS
...
END;
```

Step 2: Create the User-Defined Aggregate

This step creates the `SpatialUnion()` aggregate function by specifying its signature and the object type that implements the `ODCIAggregate` interface.

```
CREATE FUNCTION SpatialUnion(x Geometry) RETURN Geometry
AGGREGATE USING SpatialUnionRoutines;
```

Using a User-Defined Aggregate

User-defined aggregates can be used just like built-in aggregate functions in SQL DML and query statements. They can appear in the `SELECT` list, `ORDER BY` clause, or as part of the predicate in the `HAVING` clause.

For example, the following query can be used to compute state boundaries by aggregating the geometries of all counties belonging to the same state:

```
SELECT SpatialUnion(geometry)
FROM counties
GROUP BY state
```

User-defined aggregates can be used in the `HAVING` clause to eliminate groups from the output based on the results of the aggregate function. In the following example, `MyUDAG()` is a user-defined aggregate:

```
SELECT groupcol, MyUDAG(col)
FROM tab
```

```
GROUP BY groupcol
HAVING MyUDAG(col) > 100
ORDER BY MyUDAG(col);
```

User-defined aggregates can take `DISTINCT` or `ALL` (default) options on the input parameter. `DISTINCT` causes duplicate values to be ignored while computing an aggregate.

The `SELECT` statement containing a user-defined aggregate can also include `GROUP BY` extensions such as `ROLLUP`, `CUBE` and grouping sets. For example:

```
SELECT ..., MyUDAG(col)
FROM tab
GROUP BY ROLLUP(gcol1, gcol2);
```

The `ODCIAggregateMerge` interface is invoked to compute superaggregate values in such rollup operations.

> **See Also:** *Oracle9i Data Warehousing Guide* for information about `GROUP BY` extensions such as `ROLLUP`, `CUBE` and grouping sets

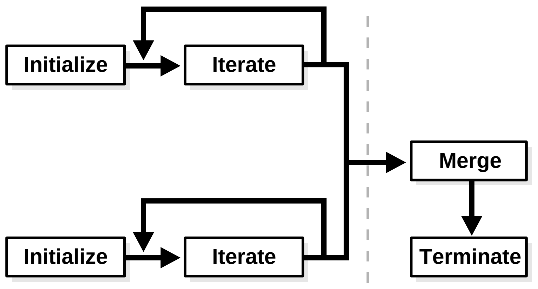
Parallel Evaluation of User-Defined Aggregates

Like built-in aggregate functions, user-defined aggregates can be evaluated in parallel. However, the aggregate function must be declared to be parallel-enabled, as follows:

```
CREATE FUNCTION MyUDAG(...) RETURN ...
PARALLEL_ENABLE AGGREGATE USING MyAggrRoutines;
```

The aggregation contexts generated by aggregating subsets of the rows within the parallel slaves are sent back to the next parallel step (either the query coordinator or the next slave set), which then invokes the Merge routine to merge the aggregation contexts and, finally, invokes the Terminate routine to obtain the aggregate value.

The sequence of calls in this scenario is as follows:



Handling Large Aggregation Contexts

When the implementation type methods are implemented in an external language (such as C or Java), the aggregation context must be passed back and forth between the Oracle server process and the external function's language environment each time an implementation type method is called.

Passing a large aggregation context can have an adverse effect on performance. To avoid this, you can store the aggregation context in external memory, allocated in the external function's execution environment, and pass just a reference or key to the context instead of the context itself. The key should be stored in the implementation type instance (the `self`); you can then pass the key between the Oracle server and the external function.

Passing a key to the context instead of the context itself keeps the implementation type instance small so that it can be transferred quickly. Another advantage of this strategy is that the memory used to hold the aggregation context is allocated in the function's execution environment (for example, `extproc`), and not in the Oracle server.

Usually you should allocate the memory to hold the aggregation context in `ODCIAggregateInitialize` and store the reference to it in the implementation type instance. In subsequent calls, the external memory and the aggregation context that it contains can be accessed using the reference. The external memory should usually be freed in `ODCIAggregateTerminate`. `ODCIAggregateMerge` should free the external memory used to store the merged context (the second argument of `ODCIAggregateMerge`) after the merge is done.

External Context and Parallel Aggregation

With parallel execution of queries with user-defined aggregates, the entire aggregation context comprising all partial aggregates computed by slave processes must sometimes be transmitted to another slave or to the master process. You can implement the optional routine `ODCIAggregateWrapContext` to collect all the partial aggregates. If a user-defined aggregate is being evaluated in parallel, and `ODCIAggregateWrapContext` is defined, Oracle invokes the routine to copy all external context references into the implementation type instance.

The `ODCIAggregateWrapContext` method should copy the aggregation context from external memory to the implementation type instance and free the external memory. To support `ODCIAggregateWrapContext`, the implementation type must contain attributes to hold the aggregation context and another attribute to hold the key that identifies the external memory.

When the aggregation context is stored externally, the key attribute of the implementation type should contain the reference identifying the external memory, and the remaining attributes of the implementation type should be `NULL`. After `ODCIAggregateWrapContext` is called, the key attribute should be `NULL`, and the other attributes should hold the actual aggregation context.

The following example shows an aggregation context type that contains references to external memory and is also able to store the entire context when needed.

```
CREATE TYPE MyAggrRoutines AS OBJECT
(
-- The 4 byte key that is used to look up the external context.
-- When NULL, it implies that the entire context value is self-contained:
-- the context value is held by the rest of the attributes in this object.
key RAW(4),
-- The following attributes correspond to the actual aggregation context. If
-- the key value is non-null, these attributes are all NULL. However, when
-- the context object is self-contained (for example, after a call to
-- ODCIAggregateWrapContext), these attributes hold the context value.
ctxval GeometrySet,
ctxval2 ...
);
```

Each of the implementation type's member methods should begin by checking whether the context is inline (contained in the implementation type instance) or in external memory. If the context is inline (for example, because it was sent from another parallel slave), it should be copied to external memory so that it can be passed by reference.

Implementation of `ODCIAggregateWrapContext` is optional. It should be implemented only when external memory is used to hold the aggregation context, and the user-defined aggregate is evaluated in parallel (that is, declared as `PARALLEL_ENABLE`). If the user-defined aggregate is not evaluated in parallel, `ODCIAggregateWrapContext` is not needed.

If the `ODCIAggregateWrapContext` method is not defined, Oracle assumes that the aggregation context is not stored externally and does not try to call the method.

External Context and User-Defined Analytic Functions

When user-defined aggregates are used as analytic functions, the aggregation context can be reused from one window to the next. In these cases, the flag argument of the `ODCIAggregateTerminate` function has its `ODCI_AGGREGATE_REUSE_CTX` bit set to indicate that the external memory holding the aggregation context should not be freed. Also, the `ODCIAggregateInitialize` method is passed the implementation type instance of the previous window, so you can access and just re-initialize the external memory allocated previously instead of having to allocate memory again.

Summary of Steps to Support External Context

1. `ODCIAggregateInitialize` - If the implementation type instance passed is not null, use the previously allocated external memory (instead of allocating external memory) and reinitialize the aggregation context.
2. `ODCIAggregateTerminate` - Free external memory only if the bit `ODCI_AGGREGATE_REUSE_CTX` of the flag argument is not set.
3. `ODCIAggregateMerge` - Free external memory associated with the merged aggregation context.
4. `ODCIAggregateWrapContext` - Copy the aggregation context from the external memory into the implementation type instance and free the external memory.
5. All member methods - First determine if the context is stored externally or inline. If the context is inline, allocate external memory and copy the context there.

User-Defined Aggregates and Materialized Views

A materialized view definition can contain user-defined aggregates as well as built-in aggregate operators. For example :

```
CREATE MATERIALIZED VIEW MyMV AS
SELECT gcols, MyUDAG(c1) FROM tab GROUP BY (gcols);
```

For the materialized view to be enabled for query rewrite, the user-defined aggregates in the materialized view must be declared as `DETERMINISTIC`. For example:

```
CREATE FUNCTION MyUDAG(x NUMBER) RETURN NUMBER
DETERMINISTIC
AGGREGATE USING MyImplType;

CREATE MATERIALIZED VIEW MyMV
ENABLE QUERY REWRITE AS
SELECT gcols, MyUDAG(c1) FROM tab GROUP BY (gcols);
```

If a user-defined aggregate is dropped or re-created, all dependent materialized views are marked invalid.

See Also: *Oracle9i Data Warehousing Guide* for information about materialized views

User-Defined Aggregates and Analytic Functions

Analytic functions (formerly called window, or windowing functions) enable you to compute various cumulative, moving, and centered aggregates over a set of rows called a **window**. The syntax provides for defining the window. For each row in a table, analytic functions return a value computed on the other rows contained in the given row’s window. These functions provide access to more than one row of a table without a self-join.

User-defined aggregates can be used as analytic functions. For example:

```
SELECT Account_number, Trans_date, Trans_amount,
  MyAVG (Trans_amount) OVER
    (PARTITION BY Account_number ORDER BY Trans_date
      RANGE INTERVAL '7' DAY PRECEDING) AS mavg_7day
FROM Ledger;
```

Reusing the Aggregation Context for Analytic Functions

When a user-defined aggregate is used as an analytic function, the aggregate is calculated for each row's corresponding window. Generally, each successive window contains largely the same set of rows, such that the new aggregation context (the new window) differs by only a few rows from the old aggregation context (the previous window). You can implement an optional routine—`ODCIAggregateDelete`—that enables Oracle to more efficiently reuse the aggregation context. If the aggregation context cannot be reused, all the rows it contains must be reiterated to rebuild it.

To reuse the aggregation context, any new rows that were not in the old context must be iterated over to add them, and any rows from the old context that do not belong in the new context must be removed.

The optional routine `ODCIAggregateDelete` removes from the aggregation context rows from the previous context that are not in the new (current) window. Oracle calls this routine for each row that must be removed. For each row that must be added, Oracle calls `ODCIAggregateIterate`.

If the new aggregation context is a superset of the old one—in other words, contains all the rows from the old context, such that none need to be deleted—then Oracle reuses the old context even if `ODCIAggregateDelete` is not implemented.

See Also:

- "Handling Large Aggregation Contexts" on page 11-7 for information about storing the aggregation context externally
- *Oracle9i Data Warehousing Guide* for information about analytic functions

Example: Creating and Using a User-Defined Aggregate

This example illustrates creating a simple user-defined aggregate function `SecondMax()` that returns the second-largest value in a set of numbers.

Creating SecondMax()

1. Implement the type `SecondMaxImpl` to contain the `ODCIAggregate` routines.

```
create type SecondMaxImpl as object
(
  max NUMBER, -- highest value seen so far
  secmax NUMBER, -- second highest value seen so far
  static function ODCIAggregateInitialize(sctx IN OUT SecondMaxImpl)
    return number,
  member function ODCIAggregateIterate(self IN OUT SecondMaxImpl,
    value IN number) return number,
  member function ODCIAggregateTerminate(self IN SecondMaxImpl,
    returnValue OUT number, flags IN number) return number,
  member function ODCIAggregateMerge(self IN OUT SecondMaxImpl,
    ctx2 IN SecondMaxImpl) return number
);
/
```

2. Implement the type body for `SecondMaxImpl`.

```
create or replace type body SecondMaxImpl is
static function ODCIAggregateInitialize(sctx IN OUT SecondMaxImpl)
return number is
begin
  sctx := SecondMaxImpl(0, 0);
  return ODCIConst.Success;
end;

member function ODCIAggregateIterate(self IN OUT SecondMaxImpl, value IN number)
return number is
begin
  if value > self.max then
    self.secmax := self.max;
    self.max := value;
  elsif value > self.secmax then
    self.secmax := value;
  end if;
  return ODCIConst.Success;
end;

member function ODCIAggregateTerminate(self IN SecondMaxImpl, returnValue OUT
number, flags IN number) return number is
begin
  returnValue := self.secmax;
  return ODCIConst.Success;
end;

member function ODCIAggregateMerge(self IN OUT SecondMaxImpl, ctx2 IN
SecondMaxImpl) return number is
begin
  if ctx2.max > self.max then
    if ctx2.secmax > self.secmax then
      self.secmax := ctx2.secmax;
    else
      self.secmax := self.max;
    end if;
    self.max := ctx2.max;
  elsif ctx2.max > self.secmax then
```

```
    self.secmax := ctx2.max;
  end if;
  return ODCIConst.Success;
end;
end;
/
```

3. Create the user-defined aggregate.

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

Using SecondMax()

```
SELECT SecondMax(salary), department_id
  FROM employees
  GROUP BY department_id
  HAVING SecondMax(salary) > 9000;
```

12

Pipelined and Parallel Table Functions

This chapter describes table functions. It also explains the generic datatypes ANYTYPE, ANYDATA, and ANYDATASET, which are likely to be used with table functions.

Major topics covered are:

- Overview of Table Functions
- Pipelined Table Functions
- Parallel Table Functions
- Input Data Streaming for Table Functions
- Transient and Generic Types

Overview

Table functions are functions that produce a collection of rows (either a nested table or a varray) that can be queried like a physical database table. You use a table function like the name of a database table, in the `FROM` clause of a query.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type or a `REF CURSOR`.

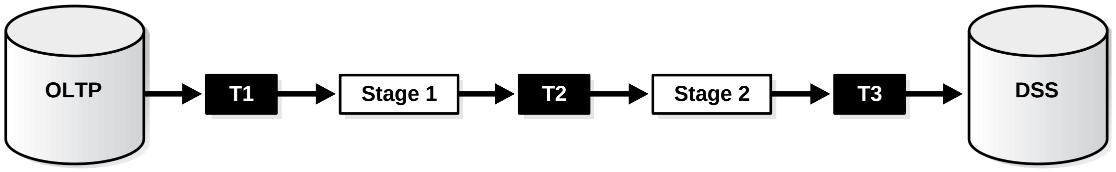
Execution of a table function can be parallelized, and returned rows can be streamed directly to the next process without intermediate staging. Rows from a collection returned by a table function can also be **pipelined**—that is, iteratively returned as they are produced instead of in a batch after all processing of the table function's input is completed.

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling multithreaded, concurrent execution of table functions
- By eliminating intermediate staging between processes
- By improving query response time: With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache does not need to materialize the entire collection.
- By iteratively providing result rows from the collection returned by a table function as the rows are produced instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection

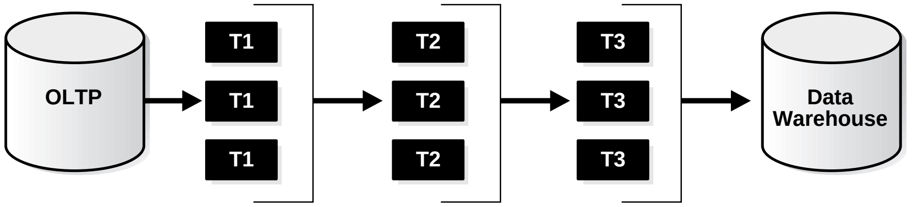
Figure 12–1 shows a typical data-processing scenario in which data goes through several (in this case, three) transformations, implemented by table functions, before finally being loaded into a database. In this scenario, the table functions are not parallelized, and the entire result collection must be staged after each transformation.

Figure 12–1 Typical Data Processing with Unparallized, Unpipelined Table Functions



By contrast, Figure 12–2 shows how streaming and parallel execution can streamline the same scenario.

Figure 12–2 Data Processing Using Pipelining and Parallel Execution



Concepts

Table Functions

Table functions return a collection type instance and can be queried like a table by calling the function in the `FROM` clause of a query. Table functions use the `TABLE` keyword.

The following example shows a table function `GetBooks` that takes a `CLOB` as input and returns an instance of the collection type `BookSet_t`. The `CLOB` column stores a catalog listing of books in some format (either proprietary or following a standard such as XML). The table function returns all the catalogs and their corresponding book listings.

The collection type `BookSet_t` is defined as:

```
CREATE TYPE Book_t AS OBJECT
( name VARCHAR2(100),
  author VARCHAR2(30),
  abstract VARCHAR2(1000));
```

```
CREATE TYPE BookSet_t AS TABLE OF Book_t;
```

The CLOBs are stored in a table `Catalogs`:

```
CREATE TABLE Catalogs
( name VARCHAR2(30),
  cat CLOB);
```

Function `GetBooks` is defined as follows:

```
CREATE FUNCTION GetBooks(a CLOB) RETURN BookSet_t;
```

The following query returns all the catalogs and their corresponding book listings.

```
SELECT c.name, Book.name, Book.author, Book.abstract
  FROM Catalogs c, TABLE(GetBooks(c.cat)) Book;
```

Pipelined Table Functions

Data is said to be **pipelined** if it is consumed by a consumer (transformation) as soon as the producer (transformation) produces it, without being staged in tables or a cache before being input to the next transformation.

Pipelining enables a table function to return rows faster and can reduce the memory required to cache a table function's results.

A pipelined table function can return the table function's result collection in subsets. The returned collection behaves like a stream that can be fetched from on demand. This makes it possible to use a table function like a virtual table.

Pipelined table functions can be implemented in two ways:

- **Native PL/SQL approach:** The consumer and producers can run on separate execution threads (either in the same or different process context) and communicate through a pipe or queuing mechanism. This approach is similar to co-routine execution.
- **Interface approach:** The consumer and producers run on the same execution thread. Producer explicitly returns the control back to the consumer after producing a set of results. In addition, the producer caches the current state so that it can resume where it left off when the consumer invokes it again.

The interface approach requires you to implement a set of well-defined interfaces in a procedural language.

The co-routine execution model provides a simpler, native PL/SQL mechanism for implementing pipelined table functions, but this model cannot be used for table functions written in C or Java. The interface approach, on the other hand, can. The interface approach requires the producer to save the current state information in a "context" object before returning so that this state can be restored on the next invocation.

In the rest of this chapter, the term *table function* is used to refer to a *pipelined* table function—that is, a table function that returns a collection in an iterative, pipelined way.

Pipelined Table Functions with REF CURSOR Arguments

A pipelined table function can accept any argument that regular functions accept. A table function that accepts a `REF CURSOR` as an argument can serve as a transformation function. That is, it can use the `REF CURSOR` to fetch the input rows, perform some transformation on them, and then pipeline the results out (using either the interface approach or the native PL/SQL approach).

For example, the following code sketches the declarations that define a `StockPivot` function. This function converts a row of the type (`Ticker, OpenPrice, ClosePrice`) into two rows of the form (`Ticker, PriceType, Price`). Calling `StockPivot` for the row (`"ORCL", 41, 42`) generates two rows: (`"ORCL", "O", 41`) and (`"ORCL", "C", 42`).

Input data for the table function might come from a source such as table `StockTable`:

```
CREATE TABLE StockTable (
  ticker VARCHAR(4),
  open_price NUMBER,
  close_price NUMBER
);
```

Here are the declarations. See Appendix A for a complete implementation of this table function using the interface approach, in both C and Java.

```
-- Create the types for the table function's output collection
-- and collection elements

CREATE TYPE TickerType AS OBJECT
(
  ticker VARCHAR2(4),
  PriceType VARCHAR2(1),
  price NUMBER
);

CREATE TYPE TickerTypeSet AS TABLE OF TickerType;

-- Define the ref cursor type

CREATE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
/

-- Create the table function

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED ... ;
/
```

Here is an example of a query that uses the `StockPivot` table function:

```
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

In the preceding query, the pipelined table function `StockPivot` fetches rows from the `CURSOR` subquery `SELECT * FROM StockTable`, performs the transformation, and pipelines the results back to the user as a table. The function produces two output rows (collection elements) for each input row.

Note that when a `CURSOR` subquery is passed from SQL to a `REF CURSOR` function argument as in the preceding example, the referenced cursor is already open when the function begins executing.

Errors and Restrictions

- The following cursor operations are not allowed for `REF CURSOR` variables based on table functions:
  - `SELECT FOR UPDATE`
  - `WHERE CURRENT OF`

Parallel Execution of Table Functions

With parallel execution of a function that appears in the `SELECT` list, execution of the function is pushed down to and conducted by multiple slave **scan** processes. These each execute the function on a segment of the function's input data.

For example, the query

```
SELECT f(col1) FROM tab;
```

is parallelized if `f` is a pure function. The SQL executed by a slave scan process is similar to:

```
SELECT f(col1) FROM tab WHERE ROWID BETWEEN :b1 AND :b2;
```

Each slave scan operates on a range of rowids and applies function `f` to each contained row. Function `f` is then executed by the scan processes; it does not run independently of them.

Unlike a function that appears in the `SELECT` list, a table function is called in the `FROM` clause and returns a collection. This affects the way that table function input data is partitioned among slave scans because the partitioning approach must be appropriate for the operation that the table function performs. (For example, an `ORDER BY` operation requires input to be range-partitioned, whereas a `GROUP BY` operation requires input to be hash partitioned.)

A table function itself specifies in its declaration the partitioning approach that is appropriate for it. (See "Input Data Partitioning" on page 12-21.) The function is then executed in a two-stage operation. First, one set of slave processes partitions the data as directed in the function's declaration; then a second set of slave scans executes the table function in parallel on the partitioned data.

For example, the table function in the following query has a `REF CURSOR` parameter:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
```

The scan is performed by one set of slave processes, which redistributes the rows (based on the partitioning method specified in the function declaration) to a second set of slave processes that actually executes function `f` in parallel.

Pipelined Table Functions

Implementation Choices for Pipelined Table Functions

As noted previously, two approaches are supported for implementing pipelined table functions: the interface approach and the PL/SQL approach.

The interface approach requires the user to supply a type that implements a predefined Oracle interface consisting of start, fetch, and close operations. The type is associated with the table function when the table function is created. During query execution, the `fetch` method is invoked repeatedly to iteratively retrieve the results. With the interface approach, the methods of the implementation type associated with the table function can be implemented in any of the supported internal or external languages (including PL/SQL, C/C++, and Java).

With the PL/SQL approach, a single PL/SQL function includes a special instruction to pipeline results (single elements of the collection) out of the function instead of returning the whole collection as a single value. The native PL/SQL approach is simpler to implement because it requires writing only one PL/SQL function.

The approach used to implement pipelined table functions does not affect the way they are used. Pipelined table functions are used in SQL statements in exactly the same way regardless of the approach used to implement them.

Declarations of Pipelined Table Functions

You declare a pipelined table function by specifying the `PIPELINED` keyword. This keyword indicates that the function will return rows iteratively. The return type of the pipelined table function must be a collection type (a nested table or a varray).

The following example shows declarations of pipelined table functions implemented using the interface approach. The interface routines for functions `GetBooks` and `StockPivot` have been implemented in the types `BookMethods` and `StockPivotImpl`, respectively.

```
CREATE FUNCTION GetBooks(cat CLOB) RETURN BookSet_t PIPELINED USING BookMethods;

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t)
  RETURN TickerTypeSet PIPELINED USING StockPivotImpl;
```

The following examples show declarations of the same table functions implemented using the native PL/SQL approach:

```
CREATE FUNCTION GetBooks(cat CLOB) RETURN BookSet_t PIPELINED IS ...;

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED IS...;
```

Implementing the Native PL/SQL Approach

In PL/SQL, the `PIPE ROW` statement causes a table function to pipe a row and continue processing. The statement enables a PL/SQL table function to return rows as soon as they are produced. (For performance, the PL/SQL runtime system provides the rows to the consumer in batches.) For example:

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
```

```
PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
    -- first row
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
    PIPE ROW(out_rec);
    -- second row
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
  RETURN;
END;
/
```

In the example, the `PIPE ROW(out_rec)` statement pipelines data out of the PL/SQL table function.

The `PIPE ROW` statement may be used only in the body of pipelined table functions; an error is raised if it is used anywhere else. The `PIPE ROW` statement can be omitted for a pipelined table function that returns no rows.

A pipelined table function must have a `RETURN` statement that does not return a value. The `RETURN` statement transfers the control back to the consumer and ensures that the next fetch gets a `NO_DATA_FOUND` exception.

Pipelining Between PL/SQL Table Functions

With serial execution, results are pipelined from one PL/SQL table function to another using an approach similar to co-routine execution. For example, the following statement pipelines results from function `g` to function `f`:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

Parallel execution works similarly except that each function executes in a different process (or set of processes).

Implementing the Interface Approach

To use the interface approach, you must define an **implementation type** that implements the ODCITable interface. This interface consists of start, fetch and close routines (and an optional describe method discussed later) whose signatures are specified by Oracle and which you implement as methods of the type.

Oracle invokes the methods to perform the following steps in the execution of a query containing a table function:

1. **Start**: Initialize the **scan context** parameter. This is then used during the second phase.
2. **Fetch**: Produce a subset of the rows in the result collection. This routine is invoked as many times as necessary to return the entire collection.
3. **Close**: Clean up (for example, release memory) after the last fetch.

Scan Context

In order for the fetch method to produce the *next* set of rows, a table function needs to be able to maintain context between successive invocations of the interface routines to fetch another set of rows. This context, called the **scan context**, is defined by the attributes of the implementation type. A table function preserves the scan context by modeling it in an object instance of the implementation type.

Start Routine

The start routine ODCITableStart is the first routine that is invoked to begin retrieving rows from a table function. This routine typically performs the setup needed for the scan. The scan context is created (as an object instance sctx) and returned to Oracle. The signature of the method is:

```
STATIC FUNCTION ODCITableStart(sctx OUT <imptype>, <args>)
RETURN NUMBER;
```

The arguments to the table function, specified by the user in the SELECT statement, are passed in as parameters to this routine.

Note that any REF CURSOR arguments of a table function must be declared as SYS_REFCURSOR type in the declaration of the ODCITableStart method: ordinary REF CURSOR types cannot be used as formal argument types in ODCITableStart. Ordinary REF CURSOR types can only be declared in a package, and types defined in a package cannot be used as formal argument types in a type method. To use a REF CURSOR type in ODCITableStart, you must use the system-defined SYS_REFCURSOR type.

Fetch Routine

The fetch routine ODCITableFetch is invoked one or more times by Oracle to retrieve all the rows in the table function's result set. The scan context is passed in as a parameter. This routine returns the next subset of one or more rows.

The fetch routine is called by Oracle repeatedly until all the rows have been returned by the table function. Returning more rows in each invocation of fetch() reduces the number of fetch calls that need to be made and thus improves performance. The table function should return a null collection to indicate that all rows have been returned. The signature of the fetch routine is:

```
MEMBER FUNCTION ODCITableFetch(self IN OUT <imptype>, nrows IN NUMBER,
    rws OUT <coll-type>) RETURN NUMBER;
```

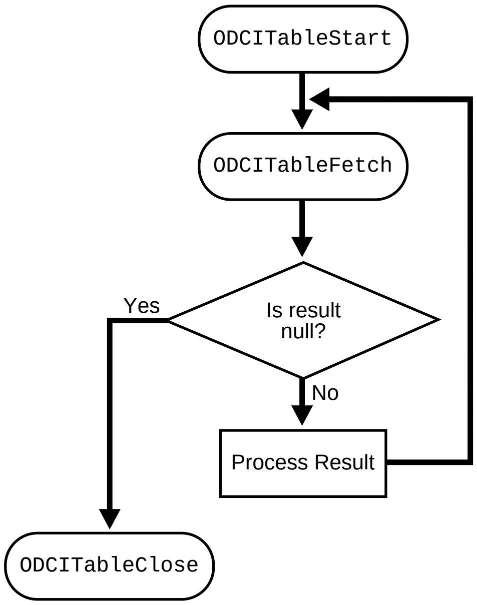
The nrows parameter indicates the number of rows that are required to satisfy the current OCI call. For example, if the current OCI call is an OCIStmtFetch that requested 100 rows, and 20 rows have aready been returned, then the nrows parameter will be equal to 80. The fetch function is allowed to return a different number of rows. The main purpose of this parameter is to prevent ODCITableFetch from returning more rows than actually required. If ODCITableFetch returns more rows than the value of this parameter, the rows are cached and returned in subsequent OCIStmtFetch calls, or they are discarded if the OCI statement handle is closed before they are all fetched.

Close Routine

The close routine ODCITableClose is invoked by Oracle after the last fetch invocation. The scan context is passed in as a parameter. This routine performs the necessary cleanup operations. The signature of the close routine is:

```
MEMBER FUNCTION ODCITableClose(self IN <imptype>)
RETURN NUMBER;
```

Figure 12–3 Flowchart of Table Function Row Source Execution



Example: Pipelined Table Functions: Interface Approach

Two complete implementations of the `StockPivot` table function are given in Appendix A. Both use the interface approach. One implements the `ODCITable` interface in C and one in Java.

Describe Routine

Sometimes it is not possible to define the structure of the return type from the table function statically. For example, the shape of the rows may be different in different queries and may depend on the actual arguments with which the table function is invoked. Such table functions can be declared to return `AnyDataSet`. `AnyDataSet` is a generic collection type. It can be used to model any collection (of any element type) and has an associated set of APIs (both PL/SQL and C) that enable you to construct `AnyDataSet` instances and access the elements.

The following example shows a table function declared to return an `AnyDataSet` collection whose structure is not fixed at function creation time:

```
CREATE FUNCTION AnyDocuments(VARCHAR2) RETURN ANYDATASET
PIPELINED USING DocumentMethods;
```

You can implement a describe interface to find out the format of the elements in the result collection when the format depends on the actual parameters to the table function. The routine, `ODCITableDescribe`, is invoked by Oracle at query compilation time to retrieve the specific type information. Typically, the routine uses the user arguments to figure out the shape of the return rows. The format of elements in the returned collection is conveyed to Oracle by returning an instance of `AnyType`.

The `AnyType` instance specifies the actual structure of the returned rows in the context of the specific query. Like `AnyDataSet`, `AnyType` has an associated set of PL/SQL and C interfaces with which to construct and access the metadata information.

> **See Also:** "Transient and Generic Types" on page 12-27 for information on `AnyDataSet` and `AnyType`

The signature of the describe routine is as follows:

```
STATIC FUNCTION ODCITableDescribe(rtype OUT ANYTYPE, <args>)
                RETURN NUMBER;
```

For example, suppose that the following query of the `AnyDocuments` function could return information on either books or magazines.

```
SELECT * FROM
  TABLE(AnyDocuments('http://.../documents.xml')) x
  WHERE x.Abstract like '%internet%';
```

The following sample implementation of the `ODCITableDescribe` method consults the DTD of the XML documents at the specified location to return the appropriate `AnyType` value (book or magazine). The `AnyType` instance is constructed by invoking the constructor APIs with the field name and datatype information.

```
CREATE TYPE Mag_t AS OBJECT
(   name VARCHAR2(100),
    publisher VARCHAR2(30),
    abstract VARCHAR2(1000)
);

STATIC FUNCTION ODCITableDescribe(rtype OUT ANYTYPE,
                                        url VARCHAR2)
IS BEGIN
    Contact specified web server and retrieve document...
    Check XML doc schema to determine if books or mags...
    IF books THEN
        rtype=AnyType.AnyTypeGetPersistent('SYS','BOOK_T');
    ELSE
        rtype=AnyType.AnyTypeGetPersistent('SYS','MAG_T');
    END IF;
END;
```

When Oracle invokes the describe method, it uses the type information (returned in the `AnyType OUT` argument) to resolve references in the command line, such as the reference to the `x.Abstract` attribute in the preceding query. This functionality is applicable only when the returned type is a named type (and therefore has named attributes).

Another feature of `ODCITableDescribe` is its ability to describe `SELECT` list parameters (for example, using OCI interfaces) when executing a `SELECT *` query. The information retrieved reflects one `SELECT` list item for each top-level attribute of the type returned by `ODCITableDescribe`.

Since the `ODCITableDescribe` method is called at compile time, the table function should have at least one argument which has a value at compile time (for example, a constant). By using the table function with different arguments, you can get different return types from the function. For example:

```
-- Issue a query for books
SELECT x.Name, x.Author
FROM TABLE(AnyDocuments('Books.xml')) x;

-- Issue a query for magazines
SELECT x.Name, x.Publisher
FROM TABLE(AnyDocuments('Magazines.xml')) x;
```

The describe functionality is available only if the table function is implemented using the interface approach. A native PL/SQL implementation of a table function that returns `ANYDATASET` will return rows whose structure is opaque to the server.

Querying Table Functions

Pipelined table functions are used in the `FROM` clause of `SELECT` statements in the same way regardless of whether they are implemented using the native PL/SQL or the interface approach. The result rows are retrieved by Oracle iteratively from the table function implementation. For example:

```
SELECT x.Ticker, x.Price
FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable))) x
WHERE x.PriceType='C';
```

Note: A table function returns a collection. In some cases, such as inside a PL/SQL block, you may need a CAST operator around the table function.

Multiple Calls to Table Functions

Multiple invocations of a table function, either within the same query or in separate queries result in multiple executions of the underlying implementation. That is, in general, there is no buffering or reuse of rows.

For example,

```
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
  WHERE t1.id = t2.id;

SELECT * FROM TABLE(f());

SELECT * FROM TABLE(f());
```

However, if the output of a table function is determined solely by the values passed into it as arguments, such that the function always produces exactly the same result value for each respective combination of values passed in, you can declare the function DETERMINISTIC, and Oracle will automatically buffer rows for it. Note, though, that the database has no way of knowing whether a function marked DETERMINISTIC really *is* DETERMINISTIC, and if one is not, results will be unpredictable.

PL/SQL

PL/SQL REF CURSOR variables can be defined for queries over table functions. For example:

```
OPEN c FOR SELECT * FROM TABLE(f(...));
```

Cursors over table functions have the same fetch semantics as ordinary cursors. REF CURSOR assignments based on table functions do not have a special semantics.

However, the SQL optimizer will not optimize across PL/SQL statements. For example:

```
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
    SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
```

```
END;
```

will not execute as well as:

```
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
```

This is so even ignoring the overhead associated with executing two SQL statements and assuming that the results can be pipelined between the two statements.

Performing DML Operations Inside Table Functions

A table function must be declared with the autonomous transaction pragma in order for the function to execute DML statements. This pragma causes the function to execute in an autonomous transaction not shared by other processes.

Use the following syntax to declare a table function with the autonomous transaction pragma:

```
CREATE FUNCTION f(p SYS_REFCURSOR) return CollType PIPELINED IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN ... END;
```

During parallel execution, each instance of the table function creates an independent transaction.

Performing DML Operations on Table Functions

Table functions cannot be the target table in `UPDATE`, `INSERT`, or `DELETE` statements. For example, the following statements will raise an error:

```
UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
INSERT INTO f(...) VALUES ('any', 'thing');
```

However, you can create a view over a table function and use `INSTEAD OF` triggers to update it. For example:

```
CREATE VIEW BookTable AS
  SELECT x.Name, x.Author
  FROM TABLE(GetBooks('data.txt')) x;
```

The following `INSTEAD OF` trigger is fired when the user inserts a row into the `BookTable` view:

```
CREATE TRIGGER BookTable_insert
```

```
INSTEAD OF INSERT ON BookTable
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
  ...
END;
INSERT INTO BookTable VALUES (...);
```

`INSTEAD OF` triggers can be defined for all DML operations on a view built on a table function.

Handling Exceptions in Table Functions

Exception handling in table functions works just as it does with ordinary user-defined functions.

Some languages, such as C and Java, provide a mechanism for user-supplied exception handling. If an exception raised within a table function is handled, the table function executes the exception handler and continues processing. Exiting the exception handler takes control to the enclosing scope. If the exception is cleared, execution proceeds normally.

An unhandled exception in a table function causes the parent transaction to roll back.

Parallel Table Functions

For a table function to be executed in parallel, it must have a partitioned input parameter. Parallelism is turned on for a table function if, and only if, both the following conditions are met:

- The function has a `PARALLEL_ENABLE` clause in its declaration
- Exactly one `REF CURSOR` is specified with a `PARTITION BY` clause

  If the `PARTITION BY` clause is not specified for any input `REF CURSOR` as part of the `PARALLEL_ENABLE` clause, the SQL compiler cannot determine how to partition the data correctly.

Inputting Data with Cursor Variables

You can pass a set of rows to a PL/SQL function in a `REF CURSOR` parameter. For example:

```
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

Results of a subquery can be passed to a function directly:

```
SELECT * FROM TABLE(f(CURSOR(SELECT empno FROM tab)));
```

In the preceding example, the `CURSOR` keyword is required to indicate that the results of a subquery should be passed as a `REF CURSOR` parameter.

Using Multiple REF CURSOR Input Variables

PL/SQL functions can accept multiple `REF CURSOR` input variables:

```
CREATE FUNCTION g(p1 pkg.refcur_t1, p2 pkg.refcur_t2) RETURN...
  PIPELINED ... ;
```

Function `g` can be invoked as follows:

```
SELECT * FROM TABLE(g(CURSOR(SELECT empno FROM tab),
  CURSOR(SELECT * FROM emp));
```

You can pass table function return values to other table functions by creating a `REF CURSOR` that iterates over the returned data:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g(...)))));
```

Explicitly Opening a REF CURSOR for a Query

You can explicitly open a `REF CURSOR` for a query and pass it as a parameter to a table function:

```
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(...));
  -- Must return a single row result set.
  SELECT * INTO rec FROM TABLE(g(r));
END;
```

PL/SQL REF CURSOR Arguments to Java and C/C++ Functions

Note: Support for passing `REF CURSOR` arguments to C and Java external functions is not provided in the initial Oracle9*i* release. Until support is provided, examples that show this feature will not work.

Parallel and pipelined table functions can be written in C/C++ and Java as well as PL/SQL. Unlike PL/SQL, C/C++ and Java do not support the `REF CURSOR` type, but you can still pass a `REF CURSOR` argument to C/C++ and Java functions.

If a table function is implemented as a C callout, then an `IN REF CURSOR` argument passed to the callout is automatically available as an executed OCI statement handle. You can use this handle like any other executed statement handle.

A `REF CURSOR` argument to a callout passed as an `IN OUT` parameter is converted to an executed statement handle on the way in to the callout, and the statement handle is converted back to a `REF CURSOR` on the way out. (The inbound and outbound statement handles may be different.)

If a `REF CURSOR` type is used as an `OUT` argument or a return type to a callout, then the callout must return the statement handle, which will be converted to a `REF CURSOR` for the caller.

The following code shows a sample callout.

```
CREATE OR replace PACKAGE p1 AS
  TYPE rc IS REF cursor;
  END;

CREATE OR REPLACE LIBRARY MYLIB AS 'mylib.so';

CREATE OR REPLACE FUNCTION MyCallout (stmthp p1.rc)
  RETURN binary_integer AS LANGUAGE C LIBRARY MYLIB
  WITH CONTEXT
  PARAMETERS (context, stmthp ocirefcursor, RETURN sb4);

sb4 MyCallout (OCIExtProcContext *ctx, OCIStmt ** stmthp)
  OCIEnv *envhp;               /* env. handle */
  OCISvcCtx *svchp;            /* service handle */
  OCIError *errhp;             /* error handle */
  OCISession *usrhp;           /* user handle */

  int errnum = 29400;          /* choose some oracle error number */
  char errmsg[512];            /* error message buffer */
  size_t errmsglen;            /* Length of error message */
  OCIDefine *defn1p = (OCIDefine *) 0;
  OCINumber *val=(OCINumber *)0;

  OCINumber *rval = (OCINumber *)0;
  sword status =  0;
  double num=0;
  val = (OCINumber*) OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
  /* Get OCI handles */
  if (GetHandles(ctx, &envhp, &svchp, &errhp, &usrhp,&rval))
    return -1;
  /* Define the fetch buffer */
  psdro_checkerr(NULL, errhp, OCIDefineByPos(*stmthp, &defn1p, errhp, (ub4) 1,
                                          (dvoid *) &num, (sb4) sizeof(num),
                                          SQLT_FLT, (dvoid *) 0, (ub2 *)0,
                                          (ub2 *)0, (ub4) OCI_DEFAULT));

  /* Fetch loop */
  while ((status = OCIStmtFetch(*stmthp, errhp, (ub4) 1,  (ub4) OCI_FETCH_NEXT,
                              (ub4) OCI_DEFAULT)) == OCI_SUCCESS ||
         status == OCI_SUCCESS_WITH_INFO)
  {
    printf("val=%lf\n",num);
  }
  return 0;
}
```

If the function is written as a Java callout, the `IN REF CURSOR` argument is automatically converted to an instance of the Java `ResultSet` class.

For a callout implemented in Java, `IN REF CURSOR` to `ResultSet` mapping is available only if you use a FAT JDBC driver based on OCI. This mapping is not available for a thin JDBC driver. As with an executed statement handle in a C callout, when a `REF CURSOR` is either an `IN OUT` argument, an `OUT` argument, or a return type for the function, a Java `ResultSet` is converted back to a PL/SQL `REF CURSOR` on its way out to the caller.

A predefined weak `REF CURSOR` type `SYS_REFCURSOR` is also supported. With `SYS_REFCURSOR`, you do not need to first create a `REF CURSOR` type in a package before you can use it. This weak `REF CURSOR` type can be used in the `ODCITableStart` method, which, as a type method, cannot accept a package type.

To use a strong `REF CURSOR` type, you still must create a PL/SQL package and declare a strong `REF CURSOR` type in it. Also, if you are using a strong `REF CURSOR` type as an argument to a table function, then the actual type of the `REF CURSOR` argument must match the column type, or an error is generated.

To partion a weak `REF CURSOR` argument, you must partition by `ANY`: a weak `REF CURSOR` argument cannot be partitioned by `RANGE` or `HASH`). Oracle recommends that you not use weak `REF CURSOR` arguments to table functions.

Input Data Partitioning

The table function declaration can specify data partitioning for exactly one REF CURSOR parameter. The syntax to do this is as follows:

```
CREATE FUNCTION f(p <ref cursor type>) RETURN rec_tab_type PIPELINED
  PARALLEL_ENABLE(PARTITION p BY [{HASH | RANGE} (<column list>) | ANY ]) IS
BEGIN ... END;
```

The PARTITION...BY phrase in the PARALLEL_ENABLE clause specifies which one of the input cursors to partition and what columns to use for partitioning.

When explicit column names are specified in the column list, the partitioning method can be RANGE or HASH. The input rows will be hash- or range-partitioned on the columns specified.

The ANY keyword enables you to indicate that the function behavior is independent of the partitioning of the input data. When this keyword is used, the runtime system randomly partitions the data among the slaves. This keyword is appropriate for use with functions that take in one row, manipulate its columns, and generate output row(s) based on the columns of this row only.

For example, the pivot-like function StockPivot shown takes as input a row of the type:

```
(Ticker varchar(4), OpenPrice number, ClosePrice number)
```

and generates rows of the type:

```
(Ticker varchar(4), PriceType varchar(1), Price number).
```

So the row ("ORCL", 41, 42) generates two rows ("ORCL", "O", 41) and ("ORCL", "C", 42).

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN rec_tab_type PIPELINED
    PARALLEL_ENABLE(PARTITION p BY ANY) IS
  ret_rec rec_type;
BEGIN
  FOR rec IN p LOOP
    ret_rec.Ticker := rec.Ticker;
    ret_rec.PriceType := "O";
    ret_rec.Price := rec.OpenPrice;
    PIPE ROW(ret_rec);

    ret_rec.Ticker := rec.Ticker;   -- Redundant; not required
    ret_rec.PriceType := "C";
    ret_rec.Price := rec.ClosePrice;
    push ret_rec;
  END LOOP;
  RETURN;
END;
```

The function `f` can be used to generate another table from `Stocks` table in the following manner:

```
INSERT INTO AlternateStockTable
  SELECT * FROM
  TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

If the `StockTable` is scanned in parallel and partitioned on `OpenPrice`, then the function `StockPivot` is combined with the data-flow operator doing the scan of `StockTable` and thus sees the same partitioning.

If, on the other hand, the `StockTable` is not partitioned, and the scan on it does not execute in parallel, the insert into `AlternateStockTable` also runs sequentially. Here is a slightly more complex example:

```
INSERT INTO AlternateStockTable
  SELECT *
    FROM TABLE(f(CURSOR(SELECT * FROM Stocks))),
         TABLE(g(CURSOR( ... )))
    WHERE <join condition>;
```

where `g` is defined to be:

```
CREATE FUNCTION g(p refcur_pkg.refcur_t) RETURN ... PIPELINED
  PARALLEL_ENABLE (PARTITION p BY ANY)
BEGIN ... END;
```

If function `g` runs in parallel and is partitioned by `ANY`, then the parallel insert can belong in the same data-flow operator as `g`.

Whenever the `ANY` keyword is specified, the data is partitioned randomly among the slaves. This effectively means that the function is executed in the same slave set which does the scan associated with the input parameter.

No redistribution or repartitioning of the data is required here. In the case when the cursor `p` itself is not parallelized, the incoming data is randomly partitioned on the columns in the column list. The round-robin table queue is used for this partitioning.

Parallel Execution of Leaf-level Table Functions

To use parallel execution with a leaf-level table function—that is, a function to perform a unitary operation that does not involve a `REF CURSOR`—arrange things so as to create a need for a `REF CURSOR`.

For example, suppose that you want a function to read a set of external files in parallel and return the records they contain. To provide work for a `REF CURSOR`, you might first create a table and populate it with the filenames. A `REF CURSOR` over this table can then be passed as a parameter to the table function (`readfiles`). The following code shows how this might be done:

```
CREATE TABLE filetab(filename VARCHAR(20));

INSERT INTO filetab VALUES('file0');
INSERT INTO filetab VALUES('file1');
.
.
.
INSERT INTO filetab VALUES('fileN');

SELECT * FROM
    TABLE(readfiles(CURSOR(SELECT filename FROM filetab)));

CREATE FUNCTION readfiles(p pkg.rc_t) RETURN coll_type
  PARALLEL_ENABLE(PARTITION p BY ANY) IS
  ret_rec rec_type;
BEGIN
  FOR rec IN p LOOP
    done := FALSE;
    WHILE (done = FALSE) LOOP
         done := readfilerecord(rec.filename, ret_rec);
         PIPE ROW(ret_rec);
    END LOOP;
  END LOOP;
  RETURN;
END;
```

Input Data Streaming for Table Functions

The way in which a table function orders or clusters rows that it fetches from cursor arguments is called **data streaming**. A function can stream its input data in any of the following ways:

- Place no restriction on the ordering of the incoming rows
- Order them on a particular key column or columns
- Cluster them on a particular key

Clustering causes rows that have the same key values to appear together but does not otherwise do any ordering of rows.

You control the behavior of the input stream using the following syntax:

```
FUNCTION f(p <ref cursor type>) RETURN tab_rec_type [PIPELINED]
         {[ORDER | CLUSTER] BY <column list>}
         PARALLEL_ENABLE({PARTITION p BY
           [ANY | (HASH | RANGE) <column list>]} )
IS
BEGIN ... END;
```

Input streaming may be specified for either sequential or parallel execution of a function.

If an `ORDER BY` or `CLUSTER BY` clause is not specified, rows are input in a (random) order.

> **Note:** The semantics of `ORDER BY` are different for parallel execution from the semantics of the `ORDER BY` clause in a SQL statement. In a SQL statement, the `ORDER BY` clause globally orders the entire data set. In a table function, the `ORDER BY` clause orders the respective rows local to each instance of the table function running on a slave.

The following example illustrates the syntax for ordering the input stream. In the example, function `f` takes in rows of the kind `(Region, Sales)` and returns rows of the form `(Region, AvgSales)`, showing average sales for each region.

```
CREATE FUNCTION f(p <ref cursor type>) RETURN tab_rec_type PIPELINED
  CLUSTER BY Region
  PARALLEL_ENABLE(PARTITION p BY Region) IS
  ret_rec rec_type;
  cnt number;
  sum number;
BEGIN
  FOR rec IN p LOOP
    IF (first rec in the group) THEN
        cnt := 1;
        sum := rec.Sales;
    ELSIF (last rec in the group) THEN
      IF (cnt <> 0) THEN
        ret_rec.Region := rec.Region;
          ret_rec.AvgSales := sum/cnt;
          PIPE ROW(ret_rec);
        END IF;
    ELSE
       cnt := cnt + 1;
      sum := sum + rec.Sales;
    END IF;
  END LOOP;
  RETURN;
END;
```

Parallel Execution: Partitioning and Clustering

Partitioning and clustering are easily confused, but they do different things. For example, sometimes partitioning can be sufficient without clustering in parallel execution.

Consider a function `SmallAggr` that performs in-memory aggregation of salary for each `department_id`, where `department_id` can be either `1`, `2`, or `3`. The input rows to the function can be partitioned by `HASH` on `department_id` such that all rows with `department_id` equal to `1` go to one slave, all rows with `department_id` equal to `2` go to another slave, and so on.

The input rows do not need to be clustered on `department_id` to perform the aggregation in the function. Each slave could have a 1x3 array `SmallSum[1..3]` in which the aggregate sum for each `department_id` is added in memory into `SmallSum[department_id]`. On the other hand, if the number of unique values of `department_id` were very large, you would want to use clustering to compute department aggregates and write them to disk one `department_id` at a time.

Parallelizing Creation of a Domain Index

Creating a domain index can be a lengthy process because of the large amount of data that a domain index typically handles. You can exploit the parallel-processing capabilities of table functions to alleviate this bottleneck. This section shows how you can use table functions to create domain indexes in parallel.

Typically, the `ODCIIndexCreate` routine does the following steps:

- Creates table(s) for storing the index data
- Fetches the relevant data (typically, keycols and rowid) from the base table, transforms it, and inserts relevant transformed data into the table created for storing the index data.
- Builds secondary indexes on the tables that store the index data, for faster access during query.

The second step mentioned—fetching relevant data and inserting it into the index data table—is the bottleneck in creating domain indexes. You can speed up this step by encapsulating these operations in a parallel table function and invoking the function from the `ODCIIndexCreate` function.

For example, a table function `IndexLoad()` might be defined to do this as follows:

```
CREATE FUNCTION IndexLoad(ia ODCIIndexInfo, parms VARCHAR2,
                          p refcur-type)
RETURN status_code_type
PARALLEL_ENABLE(PARTITION p BY ANY)
PRAGMA AUTONOMOUS_TRANSACTION
IS
BEGIN
  FOR rec IN p LOOP
    - process each rec and determine the index entry
    - derive name of index storage table from parameter ia
    - insert into table created in ODCIIndexCreate
  END LOOP;
  COMMIT; -- explicitly commit the autonomous txn
  RETURN ODCIConst.Success;
END;
```

where `p` is a cursor of the form:

```
SELECT /*+ PARALLEL (<base_table>, <par_degree>) */ <keycols> ,rowid
  FROM <base_table>
```

The `<par_degree>` value can be explicitly specified; otherwise, it is derived from the parallel degree of the base table.

Another function, like the function `IndexMerge()` defined in the following example, is needed as well to merge the results from the several instances of `IndexLoad()`.

```
CREATE FUNCTION IndexMerge(p refcur-type)
RETURN NUMBER
IS
```

```
BEGIN
  FOR rec IN p LOOP
    IF (rec != ODCIConst.Success)
      RETURN Error;
  END LOOP;
  RETURN Success;
END;
```

Now the steps in ODCIIndexCreate would be:

- Create metadata structures for the index (that is, tables to store the index data)
- Explicitly commit the transaction so that the `IndexLoad()` function can see the committed data
- Invoke `IndexLoad()` in parallel:

  ```
  status := ODCIIndexMerge(CURSOR(SELECT * FROM TABLE(
                ODCIIndexLoad(ia, parms,
                              CURSOR(SELECT <key_cols>, ROWID
                                     FROM <basetable>)
  ))))
  ```

  (Note that the cursor definition for the `IndexLoad()` function is merely a typical example; you are free to define your own form of cursor.)
- Create secondary index structures.

Transient and Generic Types

Oracle has three special SQL datatypes that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create **anonymous** (that is, unnamed) types, including anonymous collection types.

The three SQL types are implemented as **opaque types**. In other words, the internal structure of these types is not known to the database: their data can be queried only by implementing functions (typically 3GL routines) for the purpose. Oracle provides both an OCI and a PL/SQL API for implementing such functions.

The three generic SQL types are:

Type	Description
`SYS.ANYTYPE`	A type description type. A `SYS.ANYTYPE` can contain a type description of any SQL type, named or unnamed, including object types and collection types. An `ANYTYPE` can contain a type description of a persistent type, but an `ANYTYPE` itself is **transient**: in other words, the value in an `ANYTYPE` itself is not automatically stored in the database. To create a persistent type, use a `CREATE TYPE` statement from SQL.
`SYS.ANYDATA`	A **self-describing** data instance type. A `SYS.ANYDATA` contains an instance of a given type, with data, plus a description of the type. In this sense, a `SYS.ANYDATA` is self-describing. An `ANYDATA` can be persistently stored in the database.
`SYS.ANYDATASET`	A self-describing data set type. A `SYS.ANYDATASET` type contains a description of a given type plus a set of data instances of that type. An `ANYDATASET` can be persistently stored in the database.

Each of these three types can be used with any built-in type native to the database as well as with object types and collection types, both named and unnamed. The types provide a generic way to work dynamically with type descriptions, lone instances, and sets of instances of other types. Using the APIs, you can create a transient `ANYTYPE` description of any kind of type. Similarly, you can create or convert (cast) a data value of any SQL type to an `ANYDATA` and can convert an `ANYDATA` (back) to a SQL type. And similarly again with sets of values and `ANYDATASET`.

The generic types simplify working with stored procedures. You can use the generic types to encapsulate descriptions and data of standard types and pass the encapsulated information into parameters of the generic types. In the body of the procedure, you can detail how to handle the encapsulated data and type descriptions of whatever type.

You can also store encapsulated data of a variety of underlying types in one table column of type `ANYDATA` or `ANYDATASET`. For example, you can use `ANYDATA` with Advanced Queuing to model queues of heterogenous types of data. You can query the data of the underlying datatypes like any other data.

Corresponding to the three generic SQL types are three OCI types that model them. Each has a set of functions for creating and accessing the respective type:

- `OCIType`, corresponding to `SYS.ANYTYPE`
- `OCIAnyData`, corresponding to `SYS.ANYDATA`
- `OCIAnyDataSet`, corresponding to `SYS.ANYDATASET`

See Also: *Oracle Call Interface Programmer's Guide* for the `OCIType`, `OCIAnyData`, and `OCIAnyDataSet` APIs and details on how to use them. See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about the interfaces to the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types and about the `DBMS_TYPES` package, which defines constants for built-in and user-defined types, for use with `ANYTYPE`, `ANYDATA`, and `ANYDATASET`.

Part IV

Scenarios and Examples

13

Power Demand Cartridge Example

This chapter explains the power demand sample data cartridge that is discussed throughout this book. The power demand cartridge includes a user-defined object type, extensible indexing, and optimization. This chapter covers the following topics:

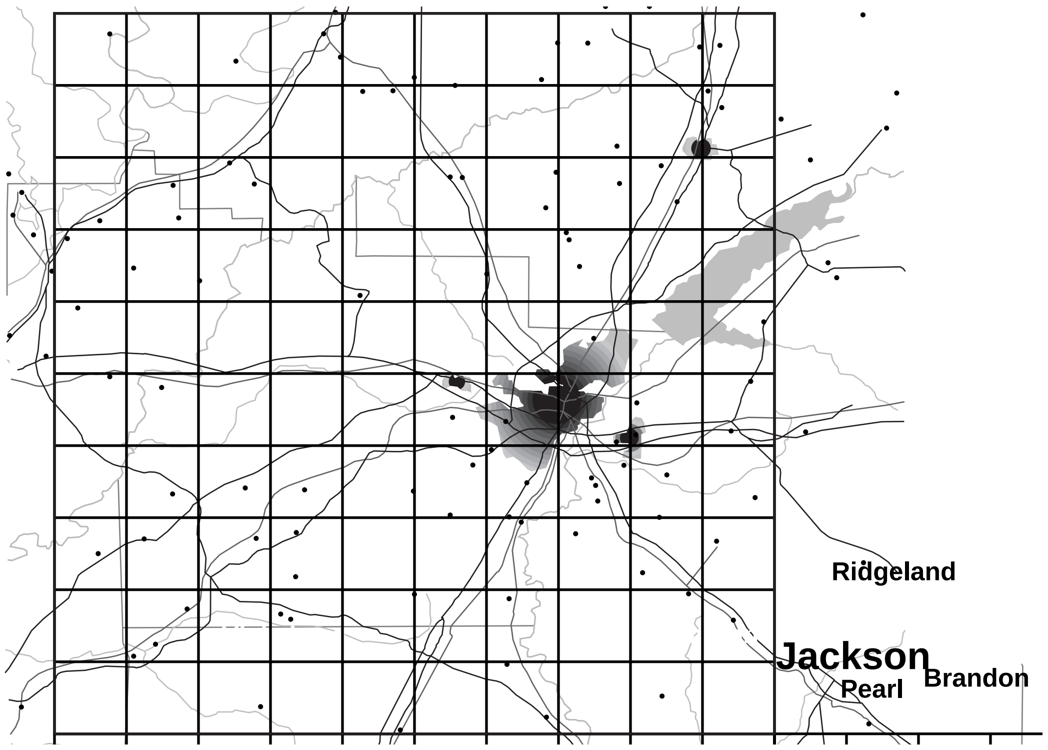
- "Modeling the Application" on page 13-9, including the technical and business scenario
- "Queries and Extensible Indexing" on page 13-13, describing kinds of queries that benefit from domain indexes
- "Creating the Domain Index" on page 13-15, explaining how the index and related structures for the example were created.
- "Defining a Type and Methods for Extensible Optimizing" on page 13-40, explaining how the methods for the extensible optimizer were created.
- "Testing the Domain Index" on page 13-64, explaining how to test the domain index and see if it is causing more efficient execution of queries than would occur without an index

This chapter does not explain in detail the concepts related to the features illustrated. For information about extensible indexing, see Chapter 7, "g Building Domain Indexes". For information about extensible query optimization, see Chapter 8, "Query Optimization". For information about cartridge services, see Chapter 9, "Using Cartridge Services".

This chapter divides the example into segments and provides commentary. The entire cartridge definition is available online in file `extdemo1.sql` in the Oracle demo directory.

Feature Requirements

A power utility, *Power-To-The-People*, develops a sophisticated model to decide how to deploy its resources. The region served by the utility is represented by a grid laid over a geographic area.



This region may be surrounded by other regions some of whose power needs are supplied by other utilities. As pictured, every region is composed of geographic quadrants referred to as "cells" on a 10x10 grid. There are a number of ways of identifying cells — by spatial coordinates (longitude/latitude), by a matrix numbering (1,1; 1,2;...), and by numbering them sequentially:

Figure 13–1 Regional Grid Cells in Numbered Sequence

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Within the area represented by each cell, the power used by consumers in that area is recorded each hour. For example, the power demand readings for a particular hour might be represented by Table 13–1 (cells here represented on a matrix):

Table 13–1 Sample Power Demand Readings for an Hour

-	1	2	3	4	5	6	7	8	9	10
1	23	21	25	23	24	25	27	32	31	30
2	33	32	31	33	34	32	23	22	21	34
3	45	44	43	33	44	43	42	41	45	46
4	44	45	45	43	42	26	19	44	33	43
5	45	44	43	42	41	44	45	46	47	44
6	43	45	98	55	54	43	44	33	34	44
7	33	45	44	43	33	44	34	55	46	34
8	87	34	33	32	31	34	35	38	33	39
9	30	40	43	42	33	43	34	32	34	46
10	43	42	34	12	43	45	48	45	43	32

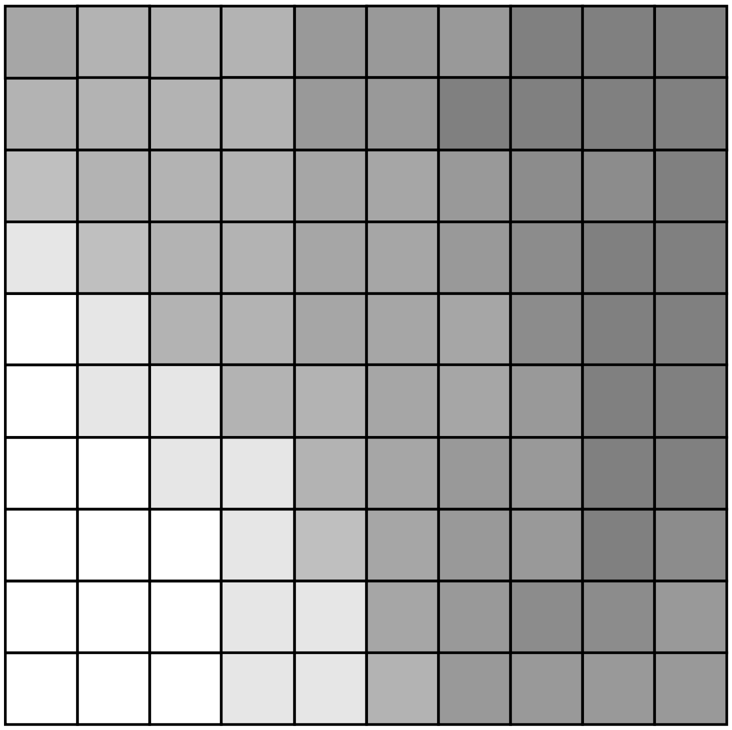
The power stations also receives reports from two other sources:

- *Sensors* on the ground provide temperature readings for every cell

  By analyzing the correlation between historical power demand from cells and the temperature readings for those regions, the utility is able to determine with a close approximation what the demand will be, given specific temperatures.

- *Satellite cameras* provide images regarding current conditions that are converted into grayscale images that match the grid:

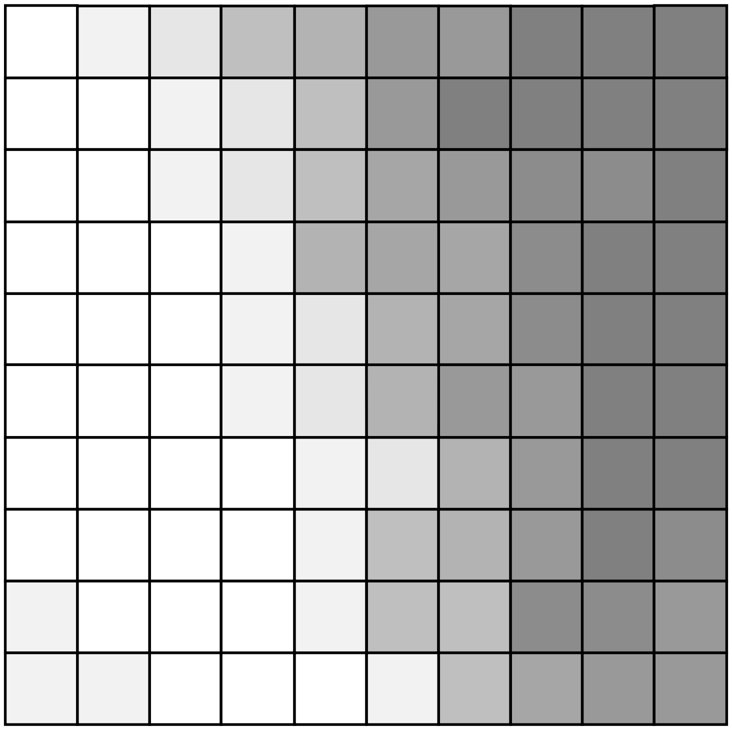
Figure 13–2 Grayscale Representation of Satellite Image



These images are designed so that ’lighter is colder’. Thus, the image shows a cold front moving into the region from the south-west. By correlating the data provided by the grayscale images with temperature readings taken at the same time, the utility has been able to determine what the power demand is given weather conditions viewed from the stratosphere.

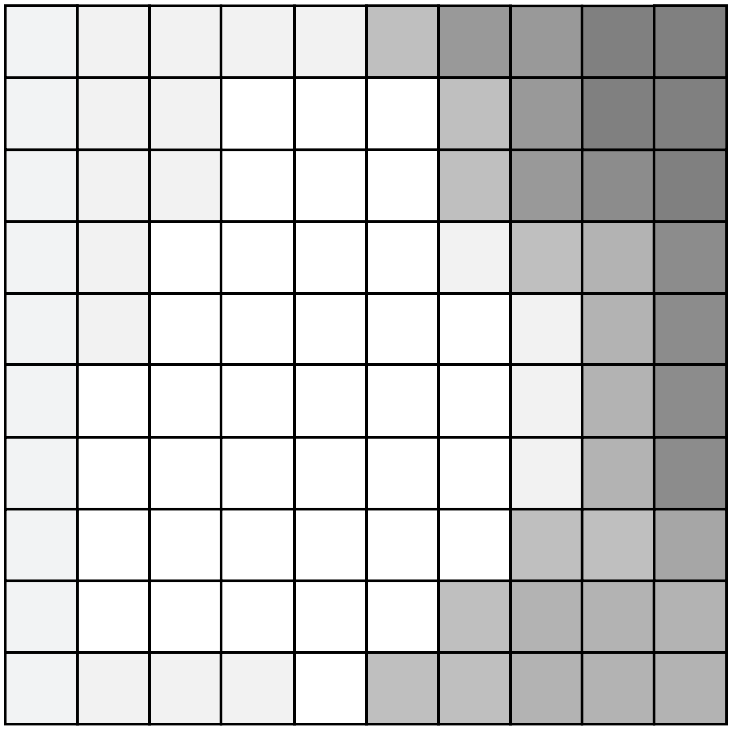
The reason that this is important is that a crucial part of this modeling has to do with noting the rapidity and degree of change in the incoming reports as weather changes and power is deployed. The following diagram shows same cold front at a second recording:

Figure 13–3 Grayscale Representation of Weather Conditions at Second Recording



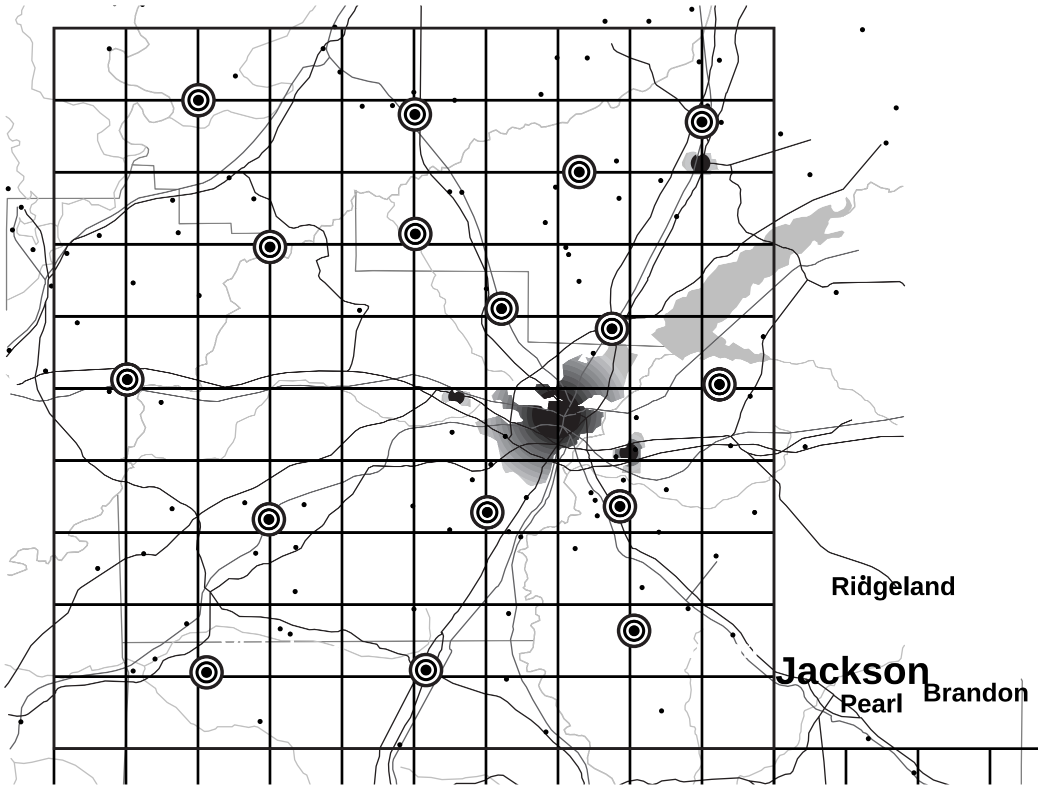
By analyzing the extent and speed of the cold front, the utility is able to project what the conditions are likely to be in the short and medium term:

Figure 13–4 Grayscale Representation of Conditions as Projected



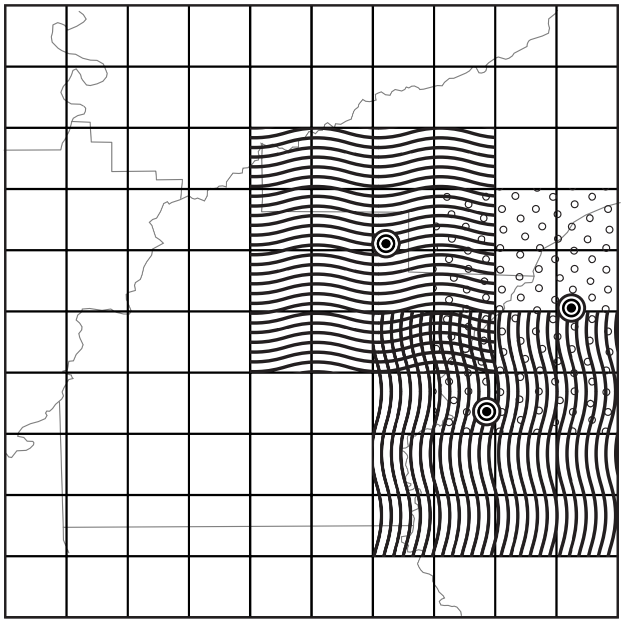
By combing this data about these conditions, and other anomalous situations (such as the failure of a substation) the utility must be able to organize the most optimal deployment of its resources. The following drawing reflects the distribution of substations across the region:

Figure 13–5 Distribution of Power Stations Across the Region



The distribution of power stations means that the utility can redirect its deployment of electricity to the areas of greatest need. The following figure gives a pictorial representation of the overlap between three stations:

Figure 13–6 Areas Served by Three Power Stations

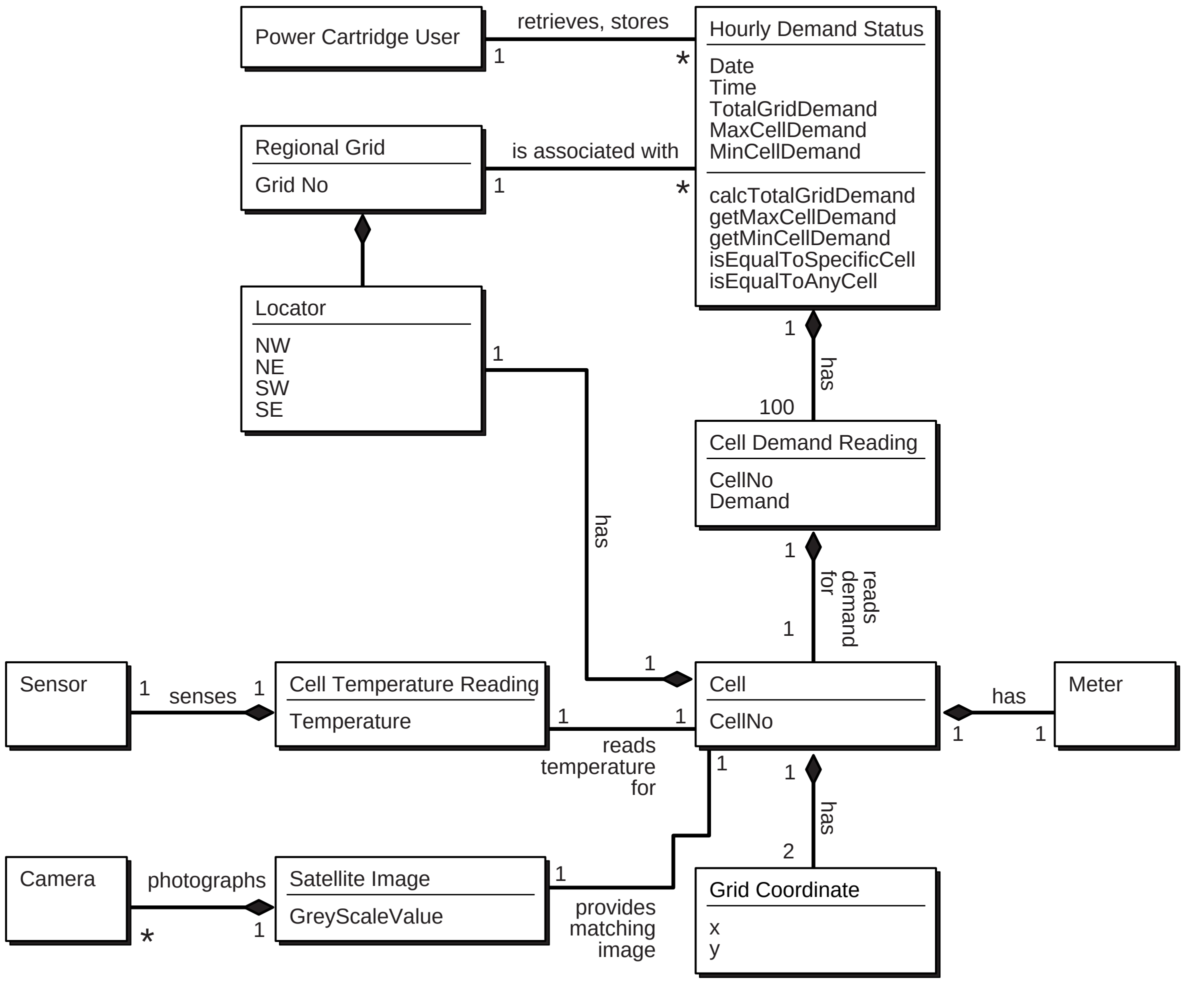


Depending on fluctuating requirements, the utility must be able to decide how to deploy its resources, and even whether to purchase power from a neighboring utility in the event of shortfall.

Modeling the Application

The following Class Diagram describes the application objects using the Unified Modelling Language (UML) notation.

Figure 13–7 Use Case Diagram for Power Demand Cartridge



Sample Queries

Modelling the application in this way, makes possible the following specific queries:

- Find the cell (geographic quadrant) with the highest demand for a specified time-period.

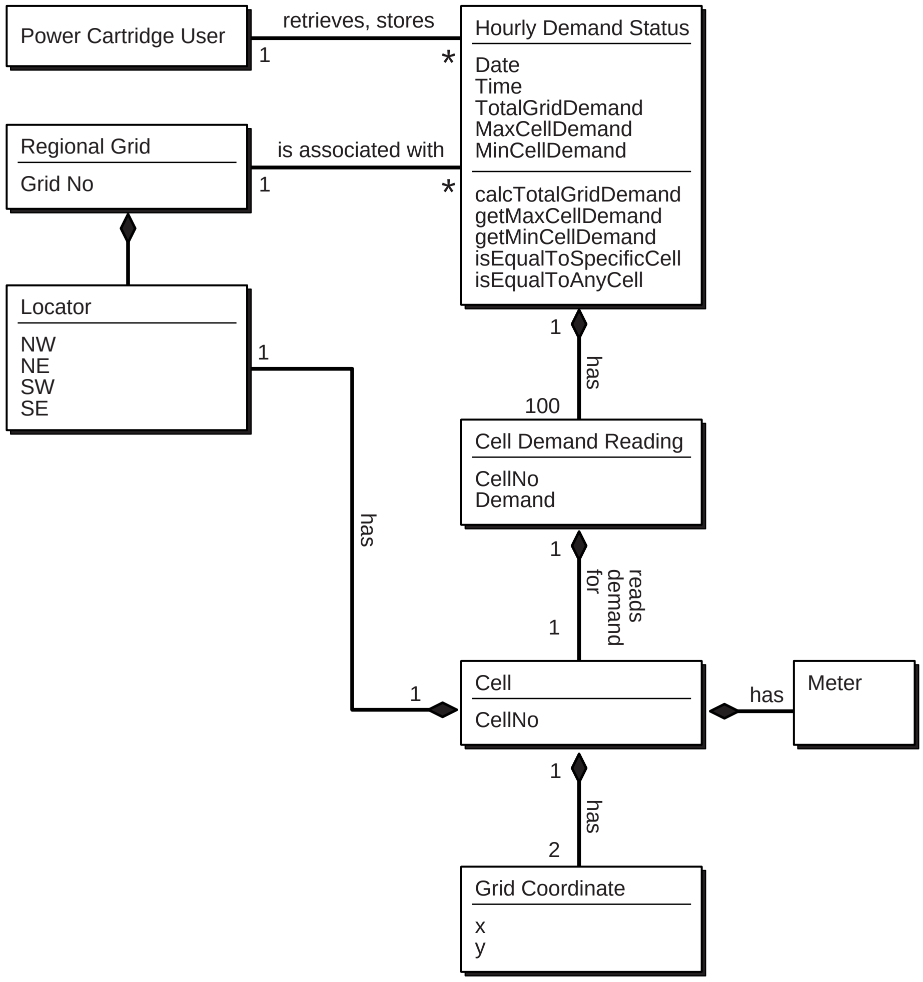
- Find the time-period with the highest total demand.
- Find all cells where demand is greater than some specified value.
- Find any cell at any time where the demand equals some specified value.
- Find any time-period for which 3 or more cells had / have a demand greater than some specified
- Find the time-period for which there was the greatest disparity (difference) between the cell with the minimum demand and the cell with the maximum demand.
- Find the times for which 10 or more cells had demand not less than some specified value.
- Find the times for which the average cell demand was greater than some specified value. (Note: it is assumed that the average is easily computable by TotalPowerDemand / 100.)
- Find the time-periods for which the median cell demand was greater than some specified value. (Note: It is assumed that the median value is not easily computable).
- Find all time-periods for which the total demand rose 10 percent or more over the preceding time's total demand.

These queries are, of course, only a short list of the possible information that could be gleaned from the system. For instance, it is obvious that the developer of such an application would want to build queries that are based on the information derived from prior queries:

- What is the percentage change in demand for a particular cell as compared to a previous time-period?
- Which cells demonstrate rapid increase / decrease in demand measured as percentages greater / lesser than specified values?

The Power Demand cartridge as implemented is described in the following class diagram.

Figure 13–8 Use Case Diagram for Power Demand Cartridge



The utility gets ongoing reports from weather centers about current conditions and from power stations about ongoing power utilization for specific geographical areas (represented by cells on a 10x10 grid). It then compares this information to historical data in order to predict demand for power in the different geographic areas for given time periods.

Each service area for the utility is considered as a 10x10 grid of cells, where each cell's boundaries are associated with spatial coordinates (longitude/latitude). The geographical areas represented by the cells can be uniform or can have different shapes and sizes. Within the area represented by each cell, the power used by consumers in that area is recorded each hour. For example, the power demand readings for a particular hour might be represented by Table 13–2.

Table 13–2 Sample Power Demand Readings for an Hour

-	1	2	3	4	5	6	7	8	9	10
1	23	21	25	23	24	25	27	32	31	30
2	33	32	31	33	34	32	23	22	21	34
3	45	44	43	33	44	43	42	41	45	46
4	44	45	45	43	42	26	19	44	33	43
5	45	44	43	42	41	44	45	46	47	44
6	43	45	98	55	54	43	44	33	34	44
7	33	45	44	43	33	44	34	55	46	34
8	87	34	33	32	31	34	35	38	33	39
9	30	40	43	42	33	43	34	32	34	46
10	43	42	34	12	43	45	48	45	43	32

The numbers in each cell reflect power demand (in some unit of measurement determined by the electric utility) for the hour for that area. For example, the demand for the first cell (1,1) was 23, the demand for the second cell (1,2) was 21, and so on. The demand for the last cell (10, 10) was 32.

The utility uses this data for many monitoring and analytical applications. Readings for individual cells are monitored for unusual surges or decreases in demand. For example, the readings of 98 for (6,3) and 87 for (8,1) might be unusually high, and the readings of 19 for (4,7) and 12 for (10,4) might be unusually low. Trends are also analyzed, such as significant increases or decreases in demand for each neighborhood, for each station, and overall, over time.

Queries and Extensible Indexing

Before you use extensible indexing, you should first ask whether the users of the table will benefit from having the domain index. That is, will they execute queries that could run just as efficiently using a standard Oracle index, or using no index at all.

Queries Not Benefiting from Extensible Indexing

A query does not require a domain index if both of the following are true:

- The desired information can be made an attribute (column) of the table and a standard index can be defined on that column.
- The operations in queries on the data are limited to those operations supported by the standard index, such as `equals`, `lessthan`, `greaterthan`, `max`, and `min` for a b-tree index.

In the `PowerDemand_Typ` object type cartridge example, the values for three columns (`TotGridDemand`, `MaxCellDemand`, and `MinCellDemand`) are set by functions, after which the values do not change. (For example, the total grid power demand for 13:00 on 01-Jan-1998 does not change after it has been computed.) For queries that use these columns, a standard b-tree index on each column is sufficient and recommended for operations like `equals`, `lessthan`, `greaterthan`, `max`, and `min`.

Examples of queries that would not benefit from extensible indexing (using the power demand cartridge) include:

- Find the cell with the highest power demand for a specific time.
- Find the time when the total grid power demand was highest.
- Find all cells where the power demand is greater than a specified value.
- Find the times for which the average cell demand or the median cell demand was greater than a specified value.

  To make this query run efficiently, define two additional columns in the `PowerDemand_Typ` object type (`AverageCellDemand` and `MedianCellDemand`), and create functions to set the values of these columns. (For example, `AverageCellDemand` is `TotGridDemand` divided by 100.) Then, create b-tree indexes on the `AverageCellDemand` and `MedianCellDemand` columns.

Queries Benefiting from Extensible Indexing

A query benefits from a domain index if the data being queried against cannot be made a simple attribute of a table or if the operation to be performed on the data is not one of the standard operations supported by Oracle indexes.

Examples of queries that would benefit from extensible indexing (using the power demand cartridge) include:

- Find the first cell for a specified time where the power demand was equal to a specified value.

  By asking for the *first cell*, the query goes beyond a simple true-false check (such as finding out whether *any* cell for a specified time had a demand equal to a specified value), and thus benefits from a domain index.
- Find the time for which there was the greatest disparity (difference) between the cell with the minimum demand and the cell with the maximum demand.
- Find all times for which 3 or more cells had a demand greater than a specified value.
- Find all times for which 10 or more cells had a demand not less than a specified value.
- Find all times for which the total grid demand rose 10 percent or more over the preceding time's total grid demand.

Creating the Domain Index

This section explains the parts of the power demand cartridge as they relate to extensible indexing. Explanatory text and code segments are mixed.

The entire cartridge definition is available online as extdemo1.sql in the standard Oracle demo directory (location is platform-dependent).

Creating the Schema to Own the Index

Before you create a domain index, create a database user (schema) to own the index. In the power demand example, the user `PowerCartUser` is created and granted the appropriate privileges. All database structures related to the cartridge are created under this user (that is, while the cartridge developer or DBA is connected to the database as `PowerCartUser`).

```
set echo on
connect sys/knl_test7 as sysdba;
drop user PowerCartUser cascade;
create user PowerCartUser identified by PowerCartUser;

-------------------------------------------------------------------
-- INITIAL SET-UP
```

```
-------------------------------------------------------------------
-- grant privileges --
grant connect, resource to PowerCartUser;
-- do we need to grant these privileges --
grant create operator to PowerCartUser;
grant create indextype to PowerCartUser;
grant create table to PowerCartUser;
```

Creating the Object Type (PowerDemand_Typ)

The object type `PowerDemand_Typ` is used to store the hourly power grid readings. This type is used to define a column in the table in which the readings are stored.

First, two types are defined for later use:

- `PowerGrid_Typ`, to define the cells in PowerDemand_Typ
- `NumTab_Typ`, to be used in the table in which the index entries are stored

```
CREATE OR REPLACE TYPE PowerGrid_Typ as VARRAY(100) of NUMBER;
CREATE OR REPLACE TYPE NumTab_Typ as TABLE of NUMBER;
```

The `PowerDemand_Typ` type includes:

- Three attributes (`TotGridDemand`, `MaxCellDemand`, `MinCellDemand`) that are set by three member procedures
- Power demand readings (100 cells in a grid)
- The date/time of the power demand readings. (Every hour, 100 areas transmit their power demand readings.)

```
CREATE OR REPLACE TYPE PowerDemand_Typ AS OBJECT (
  -- Total power demand for the grid
  TotGridDemand NUMBER,
  -- Cell with maximum/minimum power demand for the grid
  MaxCellDemand NUMBER,
  MinCellDemand NUMBER,
  -- Power grid: 10X10 array represented as Varray(100)
  -- using previously defined PowerGrid_Typ
  CellDemandValues PowerGrid_Typ,
  -- Date/time for power-demand samplings: Every hour,
  -- 100 areas transmit their power demand readings.
  SampleTime DATE,
  --
  -- Methods (Set...) for this type:
  -- Total demand for the entire power grid for a
  -- SampleTime: sets the value of TotGridDemand.
  Member Procedure SetTotalDemand,
  -- Maximum demand for the entire power grid for a
  -- SampleTime: sets the value of MaxCellDemand.
  Member Procedure SetMaxDemand,
  -- Minimum demand for the entire power grid for a
  -- SampleTime: sets the value of MinCellDemand.
  Member Procedure SetMinDemand
);
/
```

Defining the Object Type Methods

The `PowerDemand_Typ` object type has methods that set the first three attributes in the type definition:

- `TotGridDemand`, the total demand for the entire power grid for the hour in question (identified by `SampleTime`)
- `MaxCellDemand`, the highest power demand value for all cells for the `SampleTime`
- `MinCellDemand`, the lowest power demand value for all cells for the `SampleTime`

The logic for each procedure is not complicated. `SetTotDemand` loops through the cell values and creates a running total. `SetMaxDemand` compares the first two cell values and saves the higher as the current highest value; it then examines each successive cell, comparing it against the current highest value and saving the higher of the two as the current highest value, until it reaches the end of the cell values. `SetMinDemand` uses the same approach as `SetMaxDemand`, but it continually saves the lower value in comparisons to derive the lowest value overall.

```
CREATE OR REPLACE TYPE BODY PowerDemand_Typ
IS
  --
  -- Methods (Set...) for this type:
  -- Total demand for the entire power grid for a
  -- SampleTime: sets the value of TotGridDemand.
  Member Procedure SetTotalDemand
  IS
  I BINARY_INTEGER;
  Total NUMBER;
  BEGIN
    Total :=0;
    I := CellDemandValues.FIRST;
    WHILE I IS NOT NULL LOOP
   Total := Total + CellDemandValues(I);
        I := CellDemandValues.NEXT(I);
    END LOOP;
    TotGridDemand := Total;
  END;

  -- Maximum demand for the entire power grid for a
  -- SampleTime: sets the value of MaxCellDemand.
  Member Procedure SetMaxDemand
  IS
  I BINARY_INTEGER;
  Temp NUMBER;
  BEGIN
    I := CellDemandValues.FIRST;
    Temp := CellDemandValues(I);
    WHILE I IS NOT NULL LOOP
   IF Temp < CellDemandValues(I) THEN
      Temp := CellDemandValues(I);
   END IF;
        I := CellDemandValues.NEXT(I);
    END LOOP;
    MaxCellDemand := Temp;
  END;

  -- Minimum demand for the entire power grid for a
  -- SampleTime: sets the value of MinCellDemand.
  Member Procedure SetMinDemand
  IS
  I BINARY_INTEGER;
  Temp NUMBER;
  BEGIN
    I := CellDemandValues.FIRST;
    Temp := CellDemandValues(I);
    WHILE I IS NOT NULL LOOP
   IF Temp > CellDemandValues(I) THEN
      Temp := CellDemandValues(I);
   END IF;
        I := CellDemandValues.NEXT(I);
    END LOOP;
    MinCellDemand := Temp;
  END;
END;
/
```

Creating the Functions and Operators

The power demand cartridge is designed so that users can query the power grid for relationships of `equality`, `greaterthan`, or `lessthan`. However, because of the way the cell demand data is stored, the standard operators (=, >, <) cannot be used. Instead, new operators must be created, and a function must be created to define the implementation for each new operator (that is, how the operator is to be interpreted by Oracle).

For this cartridge, each of the three relationships can be checked in two ways:

- Whether a specific cell in the grid satisfies the relationship. (For example, are there grids where cell (3,7) has demand equal to 25?)

  These operators have names in the form `Power_XxxxxSpecific` (such as `Power_EqualsSpecific`), and the implementing functions have names in the form `Power_XxxxxSpecific_Func`.

- Whether any cell in the grid satisfies the relationship. (For example, are there grids where any cell has demand equal to 25?)

  These operators have names in the form `Power_XxxxxAny` (such as `Power_EqualsAny`), and the implementing functions have names in the form `Power_XxxxxAny_Func`.

For each operator-function pair, the function is defined first and then the operator as using the function. The function is the implementation that would be used if there were no index defined. This implementation must be specified so that the Oracle optimizer can determine costs, decide whether the index should be used, and create an execution plan.

Table 13–3 shows the operators and implementing functions:

Table 13–3 Operators and Implementing Functions

Operator	Implementing Function
Power_EqualsSpecific	Power_EqualsSpecific_Func
Power_EqualsAny	Power_EqualsAny_Func
Power_LessThanSpecific	Power_LessThanSpecific_Func
Power_LessThanAny	Power_LessThanAny_Func
Power_GreaterThanSpecific	Power_GreaterThanSpecific_Func
Power_GreaterThanAny	Power_GreaterThanAny_Func

Each function and operator returns a numeric value of 1 if the condition is true (for example, if the specified cell is equal to the specified value), 0 if the condition is not true, or null if the specified cell number is invalid.

The following statements create the implementing functions (`Power_xxx_Func`), first the `specific` and then the `any` implementations.

```
CREATE FUNCTION Power_EqualsSpecific_Func(
  object PowerDemand_Typ, cell NUMBER, value NUMBER)
RETURN NUMBER AS
  BEGIN
  IF cell <= object.CellDemandValues.LAST
  THEN
     IF (object.CellDemandValues(cell) = value) THEN
   RETURN 1;
     ELSE
   RETURN 0;
     END IF;
  ELSE
     RETURN NULL;
  END IF;
  END;
/
CREATE FUNCTION Power_GreaterThanSpecific_Func(
  object PowerDemand_Typ, cell NUMBER, value NUMBER)
RETURN NUMBER AS
  BEGIN
  IF cell <= object.CellDemandValues.LAST
  THEN
     IF (object.CellDemandValues(cell) > value) THEN
   RETURN 1;
     ELSE
   RETURN 0;
     END IF;
  ELSE
     RETURN NULL;
  END IF;
  END;
/
CREATE FUNCTION Power_LessThanSpecific_Func(
  object PowerDemand_Typ, cell NUMBER, value NUMBER)
RETURN NUMBER AS
  BEGIN
  IF cell <= object.CellDemandValues.LAST
  THEN
```

```
      IF (object.CellDemandValues(cell) < value) THEN
    RETURN 1;
      ELSE
    RETURN 0;
      END IF;
  ELSE
      RETURN NULL;
  END IF;
  END;
/
CREATE FUNCTION Power_EqualsAny_Func(
  object PowerDemand_Typ, value NUMBER)
RETURN NUMBER AS
   idx NUMBER;
  BEGIN
    FOR idx IN object.CellDemandValues.FIRST..object.CellDemandValues.LAST LOOP
      IF (object.CellDemandValues(idx) = value) THEN
   RETURN 1;
      END IF;
    END LOOP;
   RETURN 0;
  END;
/
CREATE FUNCTION Power_GreaterThanAny_Func(
  object PowerDemand_Typ, value NUMBER)
RETURN NUMBER AS
   idx NUMBER;
  BEGIN
    FOR idx IN object.CellDemandValues.FIRST..object.CellDemandValues.LAST LOOP
      IF (object.CellDemandValues(idx) > value) THEN
   RETURN 1;
      END IF;
    END LOOP;
   RETURN 0;
  END;
/
CREATE FUNCTION Power_LessThanAny_Func(
  object PowerDemand_Typ, value NUMBER)
RETURN NUMBER AS
   idx NUMBER;
  BEGIN
    FOR idx IN object.CellDemandValues.FIRST..object.CellDemandValues.LAST LOOP
      IF (object.CellDemandValues(idx) < value) THEN
   RETURN 1;
      END IF;
    END LOOP;
   RETURN 0;
  END;
/
```

The following statements create the operators (`Power_xxx`). Each statement specifies an implementing function.

```
CREATE OPERATOR Power_Equals BINDING(PowerDemand_Typ, NUMBER, NUMBER)
  RETURN NUMBER USING Power_EqualsSpecific_Func;
CREATE OPERATOR Power_GreaterThan BINDING(PowerDemand_Typ, NUMBER, NUMBER)
  RETURN NUMBER USING Power_GreaterThanSpecific_Func;
CREATE OPERATOR Power_LessThan BINDING(PowerDemand_Typ, NUMBER, NUMBER)
  RETURN NUMBER USING Power_LessThanSpecific_Func;

CREATE OPERATOR Power_EqualsAny BINDING(PowerDemand_Typ, NUMBER)
  RETURN NUMBER USING Power_EqualsAny_Func;
CREATE OPERATOR Power_GreaterThanAny BINDING(PowerDemand_Typ, NUMBER)
  RETURN NUMBER USING Power_GreaterThanAny_Func;
CREATE OPERATOR Power_LessThanAny BINDING(PowerDemand_Typ, NUMBER)
  RETURN NUMBER USING Power_LessThanAny_Func;
```

Creating the Indextype Implementation Methods

The power demand cartridge creates an object type for the indextype that specifies methods for the domain index. These methods are part of the `ODCIIndex` (Oracle Data Cartridge Interface Index) interface, and they collectively define the behavior of the index in terms of the methods for defining, manipulating, scanning, and exporting the index.

Table 13–4 shows the method functions (all but one starting with `ODCIIndex`) created for the power demand cartridge.

Table 13–4 Indextype Methods

Method	Description
ODCIGetInterfaces	Returns the list of names of the interfaces implemented by the type.

Table 13–4 Indextype Methods (Cont.)

Method	Description
ODCIIndexCreate	Creates a table to store index data. If the base table containing data to be indexed is not empty, this method builds the index for existing data. This method is called when a CREATE INDEX statement is issued that refers to the indextype. Upon invocation, any parameters specified in the PARAMETERS clause are passed in along with a description of the index.
ODCIIndexDrop	Drops the table that stores the index data. This method is called when a DROP INDEX statement specifies the index.
ODCIIndexStart	Initializes the scan of the index for the operator predicate. This method is invoked when a query is submitted involving an operator that can be executed using the domain index.
ODCIIndexFetch	Returns the ROWID of each row that satisfies the operator predicate.
ODCIIndexClose	Ends the current use of the index. This method can perform any necessary clean-up.
ODCIIndexInsert	Maintains the index structure when a record is inserted in a table that contains columns or object attributes indexed by the indextype.
ODCIIndexDelete	Maintains the index structure when a record is deleted from a table that contains columns or object attributes indexed by the indextype.
ODCIIndexUpdate	Maintains the index structure when a record is updated (modified) in a table that contains columns or object attributes indexed by the indextype.
ODCIIndexGet Metadata	Allows the export and import of implementation-specific metadata associated with the index.

Type Definition

The following statement creates the power_idxtype_im object type. The methods of this type are the ODCI methods to define, manipulate, and scan the domain index. The curnum attribute is the cursor number used as context for the scan routines (ODCIIndexStart, ODCIIndexFetch, and ODCIIndexClose).

```
CREATE OR REPLACE TYPE power_idxtype_im AS OBJECT
(
  curnum NUMBER,
  STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexCreate (ia sys.ODCIIndexInfo, parms VARCHAR2,
     env sys.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexDrop(ia sys.ODCIIndexInfo, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexStart(sctx IN OUT power_idxtype_im,
                                 ia sys.ODCIIndexInfo,
                                 op sys.ODCIPredInfo, qi sys.ODCIQueryInfo,
                                 strt NUMBER, stop NUMBER,
                                 cmppos NUMBER, cmpval NUMBER, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexStart(sctx IN OUT power_idxtype_im,
                                 ia sys.ODCIIndexInfo,
                                 op sys.ODCIPredInfo, qi sys.ODCIQueryInfo,
                                 strt NUMBER, stop NUMBER,
                                 cmpval NUMBER, env sys.ODCIEnv)
     RETURN NUMBER,
  MEMBER FUNCTION ODCIIndexFetch(nrows NUMBER, rids OUT sys.ODCIRidList,
     env sys.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIIndexClose (env sys.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexInsert(ia sys.ODCIIndexInfo, rid VARCHAR2,
                                  newval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexDelete(ia sys.ODCIIndexInfo, rid VARCHAR2,
                                  oldval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexUpdate(ia sys.ODCIIndexInfo, rid VARCHAR2,
                                  oldval PowerDemand_Typ,
                                  newval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIIndexGetMetadata(ia sys.ODCIIndexInfo,
                                       expversion VARCHAR2,
                                       newblock OUT PLS_INTEGER,
                                       env sys.ODCIEnv)
     RETURN VARCHAR2
);
/
```

The CREATE TYPE statement is followed by a CREATE TYPE BODY statement that specifies the implementation for each member function:

```
CREATE OR REPLACE TYPE BODY power_idxtype_im
IS
...
```

Each type method is described in a separate section, but the method definitions (except for `ODCIIndexGetMetadata`, which returns a `VARCHAR2` string) have the following general form:

```
STATIC FUNCTION function-name (...)
  RETURN NUMBER
IS
...
END;
```

ODCIGetInterfaces Method

The `ODCIGetInterfaces` function returns the list of names of the interfaces implemented by the type. To specify the Oracle9*i* version of these interfaces, the `ODCIGetInterfaces` routine must return `’SYS.ODCIINDEX2’` in the `OUT` parameter.

```
STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
     RETURN NUMBER IS
 BEGIN
     ifclist := sys.ODCIObjectList(sys.ODCIObject('SYS','ODCIINDEX2'));
     return ODCIConst.Success;
 END ODCIGetInterfaces;
```

Note: In Oracle8*i*, function `ODCIGetInterfaces` specified `SYS.ODCIINDEX1` in the `ODCIObjectList` parameter to specify the Oracle8*i* version of the `ODCIIndex` routines. To continue to use existing Oracle8*i* code that is not updated for any Oracle9*i* changes to the routines, continue to have function `ODCIGetInterfaces` specify `SYS.ODCIINDEX1`.

ODCIIndexCreate Method

The `ODCIIndexCreate` function creates the table to store index data. If the base table containing data to be indexed is not empty, this method inserts the index data entries for existing data.

The function takes the index information as an object parameter whose type is `SYS.ODCIINDEXINFO`. The type attributes include the index name, owner name, and so forth. The `PARAMETERS` string specified in the `CREATE INDEX` statement is also passed in as a parameter to the function.

```
  STATIC FUNCTION ODCIIndexCreate (ia sys.ODCIIndexInfo, parms VARCHAR2,
                                   env sys.ODCIEnv)
     RETURN NUMBER IS
   i INTEGER;
   r ROWID;
   p NUMBER;
   v NUMBER;
   stmt1 VARCHAR2(1000);
   stmt2 VARCHAR2(1000);
   stmt3 VARCHAR2(1000);
   cnum1 INTEGER;
   cnum2 INTEGER;
   cnum3 INTEGER;
   junk NUMBER;
```

The SQL statement to create the table for the index data is constructed and executed. The table includes the `ROWID` of the base table (`r`), the cell position number (`cpos`) in the grid from 1 to 100, and the power demand value in that cell (`cval`).

```
BEGIN
   -- Construct the SQL statement.
   stmt1 := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
           '_pidx' || '( r ROWID, cpos NUMBER, cval NUMBER)';

   -- Dump the SQL statement.
   dbms_output.put_line('ODCIIndexCreate>>>>>');
   sys.ODCIIndexInfoDump(ia);
   dbms_output.put_line('ODCIIndexCreate>>>>>'||stmt1);

   -- Execute the statement.
   cnum1 := dbms_sql.open_cursor;
   dbms_sql.parse(cnum1, stmt1, dbms_sql.native);
   junk := dbms_sql.execute(cnum1);
   dbms_sql.close_cursor(cnum1);
```

The function populates the index by inserting rows into the table. The function "unnests" the VARRAY attribute and inserts a row for each cell into the table. Thus, each 10 X 10 grid (10 rows, 10 values for each row) becomes 100 rows in the table (one row for each cell).

```
   -- Now populate the table.
   stmt2 := ' INSERT INTO '|| ia.IndexSchema || '.' ||
       ia.IndexName || '_pidx' ||
       ' SELECT :rr, ROWNUM, column_value FROM THE' ||
       ' (SELECT CAST (P.'|| ia.IndexCols(1).ColName||'.CellDemandValues
          AS NumTab_Typ)'||
       ' FROM ' || ia.IndexCols(1).TableSchema || '.' ||
       ia.IndexCols(1).TableName || ' P' ||
       ' WHERE P.ROWID = :rr)';

   -- Execute the statement.
   dbms_output.put_line('ODCIIndexCreate>>>>>'||stmt2);

   -- Parse the statement.
   cnum2 := dbms_sql.open_cursor;
   dbms_sql.parse(cnum2, stmt2, dbms_sql.native);

   stmt3 := 'SELECT ROWID FROM '|| ia.IndexCols(1).TableSchema
     || '.' || ia.IndexCols(1).TableName;
   dbms_output.put_line('ODCIIndexCreate>>>>>'||stmt3);
   cnum3 := dbms_sql.open_cursor;
   dbms_sql.parse(cnum3, stmt3, dbms_sql.native);
   dbms_sql.define_column_rowid(cnum3, 1, r);
   junk := dbms_sql.execute(cnum3);

   WHILE dbms_sql.fetch_rows(cnum3) > 0 LOOP
      -- Get column values of the row. --
      dbms_sql.column_value_rowid(cnum3, 1, r);
      -- Bind the row into the cursor for the next insert. --
      dbms_sql.bind_variable_rowid(cnum2, ':rr', r);
      junk := dbms_sql.execute(cnum2);
   END LOOP;
```

The function concludes by closing the cursors and returning a success status.

```
    dbms_sql.close_cursor(cnum2);
    dbms_sql.close_cursor(cnum3);
    RETURN ODCICONST.SUCCESS;
  END;
```

ODCIIndexDrop Method

The `ODCIIndexDrop` function drops the table that stores the index data. This method is called when a `DROP INDEX` statement is issued.

```
  STATIC FUNCTION ODCIIndexDrop(ia sys.ODCIIndexInfo, env sys.ODCIEnv)
     RETURN NUMBER IS
    stmt VARCHAR2(1000);
    cnum INTEGER;
    junk INTEGER;
```

```
BEGIN
  -- Construct the SQL statement.
 stmt := 'drop table ' || ia.IndexSchema || '.' || ia.IndexName
   || '_pidx';

 dbms_output.put_line('ODCIIndexDrop>>>>>');
 sys.ODCIIndexInfoDump(ia);
 dbms_output.put_line('ODCIIndexDrop>>>>>'||stmt);

 -- Execute the statement.
 cnum := dbms_sql.open_cursor;
 dbms_sql.parse(cnum, stmt, dbms_sql.native);
 junk := dbms_sql.execute(cnum);
 dbms_sql.close_cursor(cnum);

 RETURN ODCICONST.SUCCESS;
END;
```

ODCIIndexStart Method (for `Specific` Queries)

The first definition of the `ODCIIndexStart` function initializes the scan of the index to return all rows that satisfy the operator predicate. For example, if a query asks for all instances where cell (3,7) has a value equal to 25, the function initializes the scan to return all rows in the index-organized table for which that cell has that value. (This definition of `ODCIIndexStart` differs from the definition in the next section in that it includes the `cmppos` parameter for the position of the cell.)

The `self` parameter is the context that is shared with the `ODCIIndexFetch` and `ODCIIndexClose` functions. The `ia` parameter contains the index information (an object instance of type `SYS.ODCIINDEXINFO`), and the `op` parameter contains the operator information (an object instance of type `SYS.ODCIOPERINFO`). The `strt` and `stop` parameters are the lower and upper boundary points for the operator return value. The `cmppos` parameter is the cell position and `cmpval` is the value in the cell specified by the operator (`Power_XxxxxSpecific`).

```
STATIC FUNCTION ODCIIndexStart(sctx IN OUT power_idxtype_im,
      ia sys.ODCIIndexInfo,
      op sys.ODCIPredInfo, qi sys.ODCIQueryInfo,
      strt NUMBER, stop NUMBER,
      cmppos NUMBER, cmpval NUMBER, env sys.ODCIEnv ) RETURN NUMBER IS
  cnum INTEGER;
  rid ROWID;
  nrows INTEGER;
  relop VARCHAR2(2);
  stmt VARCHAR2(1000);
```

```
BEGIN
  dbms_output.put_line('ODCIIndexStart>>>>>');
  sys.ODCIIndexInfoDump(ia);
  sys.ODCIPredInfoDump(op);
  dbms_output.put_line('start key : '||strt);
  dbms_output.put_line('stop key : '||stop);
  dbms_output.put_line('compare position : '||cmppos);
  dbms_output.put_line('compare value : '||cmpval);
```

The function checks for errors in the predicate.

```
  -- Take care of some error cases.
  -- The only predicates in which btree operators can appear are
  --    op() = 1     OR    op() = 0
  if (strt != 1) and (strt != 0) then
  raise_application_error(-20101, 'Incorrect predicate for operator');
  END if;

  if (stop != 1) and (stop != 0) then
  raise_application_error(-20101, 'Incorrect predicate for operator');
  END if;
```

The function generates the SQL statement to be executed. It determines the operator name and the lower and upper index value bounds (the start and stop keys). The start and stop keys can both be 1 (= TRUE) or both be 0 (= FALSE).

```
  -- Generate the SQL statement to be executed.
  -- First, figure out the relational operator needed for the statement.
  -- Take into account the operator name and the start and stop keys.
  -- For now, the start and stop keys can both be 1 (= TRUE) or
  -- both be 0 (= FALSE).
  if op.ObjectName = 'POWER_EQUALS' then
    if strt = 1 then
      relop := '=';
    else
      relop := '!=';
    end if;
  elsif op.ObjectName = 'POWER_LESSTHAN' then
    if strt = 1 then
      relop := '<';
    else
      relop := '>=';
    end if;
  elsif op.ObjectName = 'POWER_GREATERTHAN' then
    if strt = 1 then
```

```
        relop := '>';
      else
        relop := '<=';
      end if;
    else
      raise_application_error(-20101, 'Unsupported operator');
    end if;

    stmt := 'select r from '||ia.IndexSchema||'.'||ia.IndexName||'_pidx'||
              ' where cpos '|| '=' ||''''||cmppos||''''||
              ' and cval '||relop||''''||cmpval||'''';

    dbms_output.put_line('ODCIIndexStart>>>>>' || stmt);
    cnum := dbms_sql.open_cursor;
    dbms_sql.parse(cnum, stmt, dbms_sql.native);
    dbms_sql.define_column_rowid(cnum, 1, rid);
    nrows := dbms_sql.execute(cnum);
```

The function stores the cursor number in the context, which is used by the `ODCIIndexFetch` function, and sets a success return status.

```
    -- Set context as the cursor number.
    self := power_idxtype_im(cnum);

    -- Return success.
    RETURN ODCICONST.SUCCESS;
  END;
```

ODCIIndexStart Method (for Any Queries)

This definition of the `ODCIIndexStart` function initializes the scan of the index to return all rows that satisfy the operator predicate. For example, if a query asks for all instances where any cell has a value equal to 25, the function initializes the scan to return all rows in the index-organized table for which that cell has that value. (This definition of `ODCIIndexStart` differs from the definition in the preceding section in that it does not include the `cmppos` parameter.)

The `self` parameter is the context that is shared with the `ODCIIndexFetch` and `ODCIIndexClose` functions. The `ia` parameter contains the index information (an object instance of type `SYS.ODCIINDEXINFO`), and the `op` parameter contains the operator information (an object instance of type `SYS.ODCIOPERINFO`). The `strt` and `stop` parameters are the lower and upper boundary points for the operator return value. The `cmpval` parameter is the value in the cell specified by the operator (`Power_Xxxxx`).

```
  STATIC FUNCTION ODCIIndexStart(sctx IN OUT power_idxtype_im,
        ia sys.ODCIIndexInfo,
        op sys.ODCIPredInfo, qi sys.ODCIQueryInfo,
        strt NUMBER, stop NUMBER,
        cmpval NUMBER, env sys.ODCIEnv ) RETURN NUMBER IS
    cnum INTEGER;
    rid ROWID;
    nrows INTEGER;
    relop VARCHAR2(2);
    stmt VARCHAR2(1000);
  BEGIN
    dbms_output.put_line('ODCIIndexStart>>>>>');
    sys.ODCIIndexInfoDump(ia);
    sys.ODCIPredInfoDump(op);
    dbms_output.put_line('start key : '||strt);
    dbms_output.put_line('stop key : '||stop);
    dbms_output.put_line('compare value : '||cmpval);
```

The function checks for errors in the predicate.

```
    -- Take care of some error cases.
    -- The only predicates in which btree operators can appear are
    --    op() = 1     OR    op() = 0
    if (strt != 1) and (strt != 0) then
    raise_application_error(-20101, 'Incorrect predicate for operator');
    END if;

    if (stop != 1) and (stop != 0) then
    raise_application_error(-20101, 'Incorrect predicate for operator');
    END if;
```

The function generates the SQL statement to be executed. It determines the operator name and the lower and upper index value bounds (the start and stop keys). The start and stop keys can both be 1 (= TRUE) or both be 0 (= FALSE).

```
    -- Generate the SQL statement to be executed.
    -- First, figure out the relational operator needed for the statement.
    -- Take into account the operator name and the start and stop keys.
    -- For now, the start and stop keys can both be 1 (= TRUE) or
    -- both be 0 (= FALSE).
    if op.ObjectName = 'POWER_EQUALSANY' then
      relop := '=';
    elsif op.ObjectName = 'POWER_LESSTHANANY' then
        relop := '<';
    elsif op.ObjectName = 'POWER_GREATERTHANANY' then
```

```
        relop := '>';
    else
      raise_application_error(-20101, 'Unsupported operator');
    end if;

    -- This statement returns the qualifying rows for the TRUE case.
    stmt := 'select distinct r from '||ia.IndexSchema||'.'||ia.IndexName||
            '_pidx'||' where cval '||relop||''''||cmpval||'''';
    -- In the FALSE case, we need to find the  complement of the rows.
    if (strt = 0) then
      stmt := 'select distinct r from '||ia.IndexSchema||'.'||
              ia.IndexName||'_pidx'||' minus '||stmt;
    end if;

    dbms_output.put_line('ODCIIndexStart>>>>>' || stmt);
    cnum := dbms_sql.open_cursor;
    dbms_sql.parse(cnum, stmt, dbms_sql.native);
    dbms_sql.define_column_rowid(cnum, 1, rid);
    nrows := dbms_sql.execute(cnum);
```

The function stores the cursor number in the context, which is used by the `ODCIIndexFetch` function, and sets a success return status.

```
    -- Set context as the cursor number.
    self := power_idxtype_im(cnum);

    -- Return success.
    RETURN ODCICONST.SUCCESS;
  END;
```

ODCIIndexFetch Method

The `ODCIIndexFetch` function returns a batch of `ROWIDs` for the rows that satisfy the operator predicate. Each time `ODCIIndexFetch` is invoked, it returns the next batch of rows (`rids` parameter, a collection of type `SYS.ODCIRIDLIST`) that satisfy the operator predicate. The maximum number of rows that can be returned on each invocation is specified by the `nrows` parameter.

Oracle invokes `ODCIIndexFetch` repeatedly until all rows that satisfy the operator predicate have been returned.

```
  MEMBER FUNCTION ODCIIndexFetch(nrows NUMBER, rids OUT sys.ODCIRidList,
                                 env sys.ODCIEnv)
   RETURN NUMBER IS
    cnum INTEGER;
    idx INTEGER := 1;
    rlist sys.ODCIRidList := sys.ODCIRidList();
    done boolean := FALSE;
```

The function loops through the collection of rows selected by the `ODCIIndexStart` function, using the same cursor number (`cnum`) as in the `ODCIIndexStart` function, and returns the `ROWIDs`.

```
  BEGIN
    dbms_output.put_line('ODCIIndexFetch>>>>>');
    dbms_output.put_line('Nrows : '||round(nrows));

    cnum := self.curnum;

    WHILE not done LOOP
      if idx > nrows then
         done := TRUE;
      else
         rlist.extEND;
         if dbms_sql.fetch_rows(cnum) > 0 then
            dbms_sql.column_value_rowid(cnum, 1, rlist(idx));
            idx := idx + 1;
         else
            rlist(idx) := null;
            done := TRUE;
         END if;
      END if;
    END LOOP;

    rids := rlist;
    RETURN ODCICONST.SUCCESS;
  END;
```

ODCIIndexClose Method

The `ODCIIndexClose` function closes the cursor used by the `ODCIIndexStart` and `ODCIIndexFetch` functions.

```
  MEMBER FUNCTION ODCIIndexClose (env sys.ODCIEnv) RETURN NUMBER IS
    cnum INTEGER;
  BEGIN
    dbms_output.put_line('ODCIIndexClose>>>>>');

    cnum := self.curnum;
    dbms_sql.close_cursor(cnum);
    RETURN ODCICONST.SUCCESS;
```

```
END;
```

ODCIIndexInsert Method

The `ODCIIndexInsert` function is called when a record is inserted in a table that contains columns or `OBJECT` attributes indexed by the indextype. The new values in the indexed columns are passed in as arguments along with the corresponding row identifier.

```
STATIC FUNCTION ODCIIndexInsert(ia sys.ODCIIndexInfo, rid VARCHAR2,
 newval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER AS
     cid INTEGER;
     i BINARY_INTEGER;
     nrows INTEGER;
     stmt VARCHAR2(1000);
 BEGIN
   dbms_output.put_line(' ');
   dbms_output.put_line('ODCIIndexInsert>>>>>'||
    ' TotGridDemand= '||newval.TotGridDemand ||
    ' MaxCellDemand= '||newval.MaxCellDemand ||
    ' MinCellDemand= '||newval.MinCellDemand) ;
   sys.ODCIIndexInfoDump(ia);

   -- Construct the statement.
   stmt := ' INSERT INTO '|| ia.IndexSchema || '.' || ia.IndexName
          || '_pidx' ||' VALUES (:rr, :pos, :val)';

   -- Execute the statement.
   dbms_output.put_line('ODCIIndexInsert>>>>>'||stmt);
   -- Parse the statement.
   cid := dbms_sql.open_cursor;
   dbms_sql.parse(cid, stmt, dbms_sql.native);
   dbms_sql.bind_variable_rowid(cid, ':rr', rid);

   -- Iterate over the rows of the Varray and insert them.
   i := newval.CellDemandValues.FIRST;
   WHILE i IS NOT NULL LOOP
       -- Bind the row into the cursor for insert.
       dbms_sql.bind_variable(cid, ':pos', i);
       dbms_sql.bind_variable(cid, ':val', newval.CellDemandValues(i));
       -- Execute.
       nrows := dbms_sql.execute(cid);
       dbms_output.put_line('ODCIIndexInsert>>>>>('||
                             'RID' ||' , '||
                             i   || ' , '||
                             newval.CellDemandValues(i)|| ')');
       i := newval.CellDemandValues.NEXT(i);
    END LOOP;
   dbms_sql.close_cursor(cid);
   RETURN ODCICONST.SUCCESS;
 END ODCIIndexInsert;
```

ODCIIndexDelete Method

The `ODCIIndexDelete` function is called when a record is deleted from a table that contains columns or object attributes indexed by the indextype. The old values in the indexed columns are passed in as arguments along with the corresponding row identifier.

```
  STATIC FUNCTION ODCIIndexDelete(ia sys.ODCIIndexInfo, rid VARCHAR2,
                                oldval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER AS
       cid INTEGER;
       stmt VARCHAR2(1000);
       nrows INTEGER;
   BEGIN
     dbms_output.put_line(' ');
     dbms_output.put_line('ODCIIndexDelete>>>>>'||
      ' TotGridDemand= '||oldval.TotGridDemand ||
      ' MaxCellDemand= '||oldval.MaxCellDemand ||
      ' MinCellDemand= '||oldval.MinCellDemand) ;
     sys.ODCIIndexInfoDump(ia);

     -- Construct the statement.
     stmt := ' DELETE FROM '|| ia.IndexSchema || '.' || ia.IndexName
            || '_pidx' || ' WHERE r=:rr';
     dbms_output.put_line('ODCIIndexDelete>>>>>'||stmt);

     -- Parse and execute the statement.
     cid := dbms_sql.open_cursor;
     dbms_sql.parse(cid, stmt, dbms_sql.native);
     dbms_sql.bind_variable_rowid(cid, ':rr', rid);
     nrows := dbms_sql.execute(cid);
     dbms_sql.close_cursor(cid);

     RETURN ODCICONST.SUCCESS;
   END ODCIIndexDelete;
```

ODCIIndexUpdate Method

The `ODCIIndexUpdate` function is called when a record is updated in a table that contains columns or object attributes indexed by the indextype. The old and new values in the indexed columns are passed in as arguments along with the row identifier.

```
STATIC FUNCTION ODCIIndexUpdate(ia sys.ODCIIndexInfo, rid VARCHAR2,
 oldval PowerDemand_Typ, newval PowerDemand_Typ, env sys.ODCIEnv)
     RETURN NUMBER AS
     cid INTEGER;
     cid2 INTEGER;
     stmt VARCHAR2(1000);
     stmt2 VARCHAR2(1000);
     nrows INTEGER;
     i NUMBER;
 BEGIN
   dbms_output.put_line(' ');
   dbms_output.put_line('ODCIIndexUpdate>>>>> Old'||
    ' TotGridDemand= '||oldval.TotGridDemand ||
    ' MaxCellDemand= '||oldval.MaxCellDemand ||
    ' MinCellDemand= '||oldval.MinCellDemand) ;
   dbms_output.put_line('ODCIIndexUpdate>>>>> New'||
    ' TotGridDemand= '||newval.TotGridDemand ||
    ' MaxCellDemand= '||newval.MaxCellDemand ||
    ' MinCellDemand= '||newval.MinCellDemand) ;
   sys.ODCIIndexInfoDump(ia);

   -- Delete old entries.
   stmt := ' DELETE FROM '|| ia.IndexSchema || '.' || ia.IndexName
          || '_pidx' || ' WHERE r=:rr';
   dbms_output.put_line('ODCIIndexUpdate>>>>>'||stmt);

   -- Parse and execute the statement.
   cid := dbms_sql.open_cursor;
   dbms_sql.parse(cid, stmt, dbms_sql.native);
   dbms_sql.bind_variable_rowid(cid, ':rr', rid);
   nrows := dbms_sql.execute(cid);
   dbms_sql.close_cursor(cid);

   -- Insert new entries.
   stmt2 := ' INSERT INTO '|| ia.IndexSchema || '.' || ia.IndexName
          || '_pidx' || ' VALUES (:rr, :pos, :val)';
   dbms_output.put_line('ODCIIndexUpdate>>>>>'||stmt2);

   -- Parse and execute the statement.
```

```
    cid2 := dbms_sql.open_cursor;
    dbms_sql.parse(cid2, stmt2, dbms_sql.native);
    dbms_sql.bind_variable_rowid(cid2, ':rr', rid);

    -- Iterate over the rows of the Varray and insert them.
    i := newval.CellDemandValues.FIRST;
    WHILE i IS NOT NULL LOOP
        -- Bind the row into the cursor for insert.
        dbms_sql.bind_variable(cid2, ':pos', i);
        dbms_sql.bind_variable(cid2, ':val', newval.CellDemandValues(i));
        nrows := dbms_sql.execute(cid2);
        dbms_output.put_line('ODCIIndexUpdate>>>>>('||
                             'RID' || ' , '||
                             i   || ' , '||
                             newval.CellDemandValues(i)|| ')');
        i := newval.CellDemandValues.NEXT(i);
     END LOOP;
    dbms_sql.close_cursor(cid2);

    RETURN ODCICONST.SUCCESS;
  END ODCIIndexUpdate;
```

`ODCIIndexUpdate` is the last method defined in the `CREATE TYPE BODY` statement, which ends as follows:

```
END;
/
```

ODCIIndexGetMetadata Method

The optional `ODCIIndexGetMetadata` function, if present, is called by the Export utility in order to write implementation-specific metadata (which is not stored in the system catalogs) into the export dump file. This metadata might be policy information, version information, user settings, and so on. This metadata is written to the dump file as anonymous PL/SQL blocks that are executed at import time, immediately before the associated index is created.

This method returns strings to the Export utility that comprise the code of the PL/SQL blocks. The Export utility repeatedly calls this method until a zero-length string is returned, thus allowing the creation of any number of PL/SQL blocks of arbitrary complexity. Normally, this method calls functions within a PL/SQL package in order to make use of package-level variables, such as cursors and iteration counters, that maintain state across multiple calls by Export.

For information about the Export and Import utilities, see the *Oracle9i Database Utilities* manual.

In the power demand cartridge, the only metadata that is passed is a version string of *V1.0*, identifying the current format of the index-organized table that underlies the domain index. The `power_pkg.getversion` function generates a call to the `power_pkg.checkversion` procedure, to be executed at import time to check that the version string is *V1.0*.

```
STATIC FUNCTION ODCIIndexGetMetadata(ia sys.ODCIIndexInfo, expversion
VARCHAR2, newblock OUT PLS_INTEGER, env sys.ODCIEnv)
  RETURN VARCHAR2 IS

BEGIN
-- Let getversion do all the work since it has to maintain state across calls.

  RETURN power_pkg.getversion (ia.IndexSchema, ia.IndexName, newblock);

EXCEPTION
  WHEN OTHERS THEN
      RAISE;

END ODCIIndexGetMetaData;
```

The `power_pkg` package is defined as follows:

```
CREATE OR REPLACE PACKAGE power_pkg AS
  FUNCTION getversion(idxschema IN VARCHAR2, idxname IN VARCHAR2,
      newblock OUT PLS_INTEGER) RETURN VARCHAR2;
  PROCEDURE checkversion (version IN VARCHAR2);
END power_pkg;
/
SHOW ERRORS;

CREATE OR REPLACE PACKAGE BODY power_pkg AS

-- iterate is a package-level variable used to maintain state across calls
-- by Export in this session.

iterate NUMBER := 0;

FUNCTION getversion(idxschema IN VARCHAR2, idxname IN VARCHAR2,
      newblock OUT PLS_INTEGER) RETURN VARCHAR2 IS

BEGIN
```

```
-- We are generating only one PL/SQL block consisting of one line of code.
  newblock := 1;

  IF iterate = 0
  THEN
-- Increment iterate so we'll know we're done next time we're called.
    iterate := iterate + 1;

-- Return a string that calls checkversion with a version 'V1.0'
-- Note that export adds the surrounding BEGIN/END pair to form the anon.
-- block... we don't have to.

    RETURN 'power_pkg.checkversion(''V1.0'');';
  ELSE
-- reset iterate for next index
    iterate := 0;
-- Return a 0-length string; we won't be called again for this index.
    RETURN '';
  END IF;
END getversion;

PROCEDURE checkversion (version IN VARCHAR2) IS

  wrong_version            EXCEPTION;

BEGIN
  IF version != 'V1.0' THEN
     RAISE wrong_version;
  END IF;
END checkversion;

END power_pkg;
```

Creating the Indextype

The power demand cartridge creates the indextype for the domain index. The specification includes the list of operators supported by the indextype. It also identifies the implementation type containing the OCDI index routines.

```
CREATE OR REPLACE INDEXTYPE power_idxtype
FOR
   Power_Equals(PowerDemand_Typ, NUMBER, NUMBER),
   Power_GreaterThan(PowerDemand_Typ, NUMBER, NUMBER),
   Power_LessThan(PowerDemand_Typ, NUMBER, NUMBER),
   Power_EqualsAny(PowerDemand_Typ, NUMBER),
   Power_GreaterThanAny(PowerDemand_Typ, NUMBER),
   Power_LessThanAny(PowerDemand_Typ, NUMBER)
USING power_idxtype_im;
```

Defining a Type and Methods for Extensible Optimizing

This section explains the parts of the power demand cartridge as they relate to extensible optimization. Explanatory text and code segments are mixed.

Creating the Statistics Table (PowerCartUserStats)

The table `PowerCartUserStats` is used to store statistics about the hourly power grid readings. These statistics will be used by the method `ODCIStatsSelectivity` (described later) to estimate the selectivity of operator predicates. Because of the types of statistics collected, it is more convenient to use a separate table instead of letting Oracle store the statistics.

The `PowerCartUserStats` table contains the following columns:

- The table and column for which statistics are collected
- The cell for which the statistics are collected
- The minimum and maximum power demand for the given cell over all power grid readings
- The number of non-null readings for the given cell over all power grid readings

```
CREATE TABLE PowerCartUserStats (
  -- Table for which statistics are collected
  tab VARCHAR2(30),
  -- Column for which statistics are collected
  col VARCHAR2(30),
  -- Cell position
  cpos NUMBER,
  -- Minimum power demand for the given cell
  lo NUMBER,
  -- Maximum power demand for the given cell
  hi NUMBER,
  -- Number of (non-null) power demands for the given cell
  nrows NUMBER
);
/
```

Creating the Extensible Optimizer Methods

The power demand cartridge creates an object type that specifies methods that will be used by the extensible optimizer. These methods are part of the `ODCIStats` (Oracle Data Cartridge Interface STATisticS) interface and they collectively define the methods that are called when an ANALYZE command is issued or when the optimizer is deciding on the best execution plan for a query.

Table 13–5 shows the method functions created for the power demand cartridge. (Names of all but one of the functions begin with the string *ODCIStats*.)

Table 13–5 Extensible Optimizer Methods

Method	Description
`ODCIGetInterfaces`	Returns the list of names of the interfaces implemented by the type.
`ODCIStatsCollect`	Collects statistics for columns of type PowerDemand_Typ or domain indexes of indextype power_idxtype. This method is called when an ANALYZE statement is issued that refers to a column of the PowerDemand_Typ type or an index of the power_idxtype indextype. Upon invocation, any options specified in the ANALYZE statement are passed in along with a description of the column or index.
`ODCIStatsDelete`	Deletes statistics for columns of type PowerDemand_Typ or domain indexes of indextype power_idxtype. This method is called when an ANALYZE statement is issued to delete statistics for a column of the appropriate type or an index of the appropriate indextype.
`ODCIStatsSelectivity`	Computes the selectivity of a predicate involving an operator or its functional implementation. This method is called by the optimizer when a predicate of the appropriate type appears in the WHERE clause of a query.
`ODCIStatsIndexCost`	Computes the cost of a domain index access path. This method is called by the optimizer to get the cost of a domain index access path assuming the index can be used for the query.
`ODCIStatsFunctionCost`	Computes the cost of a function. This method is called by the optimizer to get the cost of executing a function. The function need not necessarily be an implementation of an operator.

Type Definition

The following statement creates the `power_statistics` object type. This object type's ODCI methods are used to collect and delete statistics about columns and indexes, compute selectivities of predicates with operators or functions, and to compute costs of domain indexes and functions. The curnum attribute is a dummy attribute that is not used.

```
CREATE OR REPLACE TYPE power_statistics AS OBJECT
(
  curnum NUMBER,
  STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
     RETURN NUMBER,
  STATIC FUNCTION ODCIStatsCollect(col sys.ODCIColInfo,
     options sys.ODCIStatsOptions, rawstats OUT RAW, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIStatsDelete(col sys.ODCIColInfo, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIStatsCollect(ia sys.ODCIIndexInfo,
     options sys.ODCIStatsOptions, rawstats OUT RAW, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIStatsDelete(ia sys.ODCIIndexInfo, env sys.ODCIEnv)
     RETURN NUMBER,
  STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
     sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
     object PowerDemand_Typ, cell NUMBER, value NUMBER, env sys.ODCIEnv)
     RETURN NUMBER,
     PRAGMA restrict_references(ODCIStatsSelectivity, WNDS, WNPS),
  STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
     sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
     object PowerDemand_Typ, value NUMBER, env sys.ODCIEnv) RETURN NUMBER,
     PRAGMA restrict_references(ODCIStatsSelectivity, WNDS, WNPS),
  STATIC FUNCTION ODCIStatsIndexCost(ia sys.ODCIIndexInfo,
     sel NUMBER, cost OUT sys.ODCICost, qi sys.ODCIQueryInfo,
     pred sys.ODCIPredInfo, args sys.ODCIArgDescList,
     strt NUMBER, stop NUMBER, cmppos NUMBER, cmpval NUMBER, env sys.ODCIEnv)
     RETURN NUMBER,
     PRAGMA restrict_references(ODCIStatsIndexCost, WNDS, WNPS),
  STATIC FUNCTION ODCIStatsIndexCost(ia sys.ODCIIndexInfo,
     sel NUMBER, cost OUT sys.ODCICost, qi sys.ODCIQueryInfo,
     pred sys.ODCIPredInfo, args sys.ODCIArgDescList,
     strt NUMBER, stop NUMBER, cmpval NUMBER, env sys.ODCIEnv) RETURN NUMBER,
     PRAGMA restrict_references(ODCIStatsIndexCost, WNDS, WNPS),
  STATIC FUNCTION ODCIStatsFunctionCost(func sys.ODCIFuncInfo,
     cost OUT sys.ODCICost, args sys.ODCIArgDescList,
     object PowerDemand_Typ, cell NUMBER, value NUMBER, env sys.ODCIEnv)
```

```
      RETURN NUMBER,
      PRAGMA restrict_references(ODCIStatsFunctionCost, WNDS, WNPS),
   STATIC FUNCTION ODCIStatsFunctionCost(func sys.ODCIFuncInfo,
      cost OUT sys.ODCICost, args sys.ODCIArgDescList,
      object PowerDemand_Typ, value NUMBER, env sys.ODCIEnv) RETURN NUMBER,
      PRAGMA restrict_references(ODCIStatsFunctionCost, WNDS, WNPS)
);
/
```

The `CREATE TYPE` statement is followed by a `CREATE TYPE BODY` statement that specifies the implementation for each member function:

```
CREATE OR REPLACE TYPE BODY power_statistics
IS
...
```

Each member function is described in a separate section, but the function definitions have the following general form:

```
  STATIC FUNCTION function-name (...)
     RETURN NUMBER IS
  END;
```

ODCIGetInterfaces Method

The `ODCIGetInterfaces` function returns the list of names of the interfaces implemented by the type. There is only one set of the extensible optimizer interface routines, called `SYS.ODCISTATS`, but the server supports multiple versions of them for backward compatibility. In Oracle9*i*, most of the routines have a new `ODCIEnv` argument, and several underlying system types used by other arguments have been enhanced. To specify the Oracle9*i* version of the routines, function `ODCIGetInterfaces` must specify `SYS.ODCISTATS2` in the `OUT`, `ODCIObjectList` parameter.

> **Note:** In Oracle8*i*, function `ODCIGetInterfaces` specified `SYS.ODCISTATS1` in the `ODCIObjectList` parameter to specify the Oracle8*i* version of the `ODCIStats` routines. To continue to use existing Oracle8*i* code that is not updated for any Oracle9*i* changes to the routines, continue to have function `ODCIGetInterfaces` specify `SYS.ODCISTATS1`.

```
STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
     RETURN NUMBER IS
```

```
BEGIN
   ifclist := sys.ODCIObjectList(sys.ODCIObject('SYS','ODCISTATS2'));
   RETURN ODCIConst.Success;
END ODCIGetInterfaces;
```

ODCIStatsCollect Method (for `PowerDemand_Typ` columns)

The `ODCIStatsCollect` function collects statistics for columns whose datatype is the `PowerDemand_Typ` object type. The statistics are collected for each cell in the column over all power grid readings. For a given cell, the statistics collected are the minimum and maximum power grid readings, and the number of non-null readings.

The function takes the column information as an object parameter whose type is `SYS.ODCICOLINFO`. The type attributes include the table name, column name, and so on. Options specified in the `ANALYZE` command used to collect the column statistics are also passed in as parameters. For example, if `ANALYZE ESTIMATE` is used, then the percentage or number of rows specified in the `ANALYZE` command is passed in to `ODCIStatsCollect`. Since the power demand cartridge uses a table to store the statistics, the output parameter `rawstats` is not used in this cartridge.

```
STATIC FUNCTION ODCIStatsCollect(col sys.ODCIColInfo,
                                 options sys.ODCIStatsOptions,
                                 rawstats OUT RAW, env sys.ODCIEnv)
   RETURN NUMBER IS
   cnum                INTEGER;
   stmt                VARCHAR2(1000);
   junk                INTEGER;

   cval                NUMBER;
   colname             VARCHAR2(30) := rtrim(ltrim(col.colName, '"'), '"');
   statsexists         BOOLEAN := FALSE;
   pdemands            PowerDemand_Tab%ROWTYPE;
   user_defined_stats  PowerCartUserStats%ROWTYPE;
   CURSOR c1(tname VARCHAR2, cname VARCHAR2) IS
     SELECT * FROM PowerCartUserStats
     WHERE tab = tname
       AND col = cname;
   CURSOR c2 IS
     SELECT * FROM PowerDemand_Tab;

BEGIN
  sys.ODCIColInfoDump(col);
  sys.ODCIStatsOptionsDump(options);
```

```
IF (col.TableSchema IS NULL OR col.TableName IS NULL
    OR col.ColName IS NULL) THEN
  RETURN ODCIConst.Error;
END IF;

dbms_output.put_line('ODCIStatsCollect>>>>>');
dbms_output.put_line('**** Analyzing column '
                     || col.TableSchema
                     || '.' || col.TableName
                     || '.' || col.ColName);

-- Check if statistics exist for this column
FOR user_defined_stats IN c1(col.TableName, colname) LOOP
  statsexists := TRUE;
  EXIT;
END LOOP;
```

The function checks whether statistics for this column already exist. If so, it initializes them to NULL; otherwise, it creates statistics for each of the 100 cells and initializes them to NULL.

```
IF not statsexists THEN
  -- column statistics don't exist; create entries for
  -- each of the 100 cells
  cnum := dbms_sql.open_cursor;
  FOR i in 1..100 LOOP
    stmt := 'INSERT INTO PowerCartUserStats VALUES( '
         || '''' || col.TableName || ''', '
         || '''' || colname || ''', '
         || to_char(i) || ', '
         || 'NULL, NULL, NULL)';
    dbms_sql.parse(cnum, stmt, dbms_sql.native);
    junk := dbms_sql.execute(cnum);
  END LOOP;
  dbms_sql.close_cursor(cnum);
ELSE
  -- column statistics exist; initialize to NULL
  cnum := dbms_sql.open_cursor;
  stmt := 'UPDATE PowerCartUserStats'
       || ' SET lo = NULL, hi = NULL, nrows = NULL'
       || ' WHERE tab = ' || col.TableName
       || ' AND col = ' || colname;
  dbms_sql.parse(cnum, stmt, dbms_sql.native);
  junk := dbms_sql.execute(cnum);
  dbms_sql.close_cursor(cnum);
```

```
    END IF;
```

The function collects statistics for the column by reading rows from the table that is being analyzed. This is done by constructing and executing a SQL statement.

```
    -- For each cell position, the following statistics are collected:
    --   maximum value
    --   minimum value
    --   number of rows (excluding NULLs)
    cnum := dbms_sql.open_cursor;
    FOR i in 1..100 LOOP
      FOR pdemands IN c2 LOOP
        IF i BETWEEN pdemands.sample.CellDemandValues.FIRST AND
                     pdemands.sample.CellDemandValues.LAST THEN
          cval := pdemands.sample.CellDemandValues(i);
          stmt := 'UPDATE PowerCartUserStats SET '
               || 'lo = least(' || 'NVL(' || to_char(cval) || ', lo), '
               || 'NVL(' || 'lo, ' || to_char(cval) || ')), '
               || 'hi = greatest(' || 'NVL(' || to_char(cval) || ', hi), '
               || 'NVL(' || 'hi, ' || to_char(cval) || ')), '
               || 'nrows = decode(nrows, NULL, decode('
               || to_char(cval) || ', NULL, NULL, 1), decode('
               || to_char(cval) || ', NULL, nrows, nrows+1)) '
               || 'WHERE cpos = ' || to_char(i)
               || ' AND tab = ''' || col.TableName || ''''
               || ' AND col = ''' || colname || '''';
          dbms_sql.parse(cnum, stmt, dbms_sql.native);
          junk := dbms_sql.execute(cnum);
        END IF;
      END LOOP;
    END LOOP;
```

The function concludes by closing the cursor and returning a success status.

```
    dbms_sql.close_cursor(cnum);

    rawstats := NULL;

    return ODCIConst.Success;

  END;
```

ODCIStatsDelete Method (for `PowerDemand_Typ` columns)

The `ODCIStatsDelete` function deletes statistics of columns whose datatype is the `PowerDemand_Typ` object type.

The function takes the column information as an object parameter whose type is `SYS.ODCICOLINFO`. The type attributes include the table name, column name, and so on.

```
STATIC FUNCTION ODCIStatsDelete(col sys.ODCIColInfo, env sys.ODCIEnv)
   RETURN NUMBER IS
   cnum               INTEGER;
   stmt               VARCHAR2(1000);
   junk               INTEGER;

   colname            VARCHAR2(30) := rtrim(ltrim(col.colName, '"'), '"');
   statsexists        BOOLEAN := FALSE;
   user_defined_stats PowerCartUserStats%ROWTYPE;
   CURSOR c1(tname VARCHAR2, cname VARCHAR2) IS
     SELECT * FROM PowerCartUserStats
     WHERE tab = tname
       AND col = cname;
BEGIN
  sys.ODCIColInfoDump(col);

  IF (col.TableSchema IS NULL OR col.TableName IS NULL
      OR col.ColName IS NULL) THEN
    RETURN ODCIConst.Error;
  END IF;

  dbms_output.put_line('ODCIStatsDelete>>>>>');
  dbms_output.put_line('**** Analyzing (delete) column '
                       || col.TableSchema
                       || '.' || col.TableName
                       || '.' || col.ColName);
```

The function verifies that statistics for the column exist by checking the statistics table. If statistics were not collected, then there is nothing to be done. If, however, statistics are present, it constructs and executes a SQL statement to delete the relevant rows from the statistics table.

```
  -- Check if statistics exist for this column
  FOR user_defined_stats IN c1(col.TableName, colname) LOOP
    statsexists := TRUE;
    EXIT;
  END LOOP;

  -- If user-defined statistics exist, delete them
  IF statsexists THEN
    stmt := 'DELETE FROM PowerCartUserStats'
```

```
           || ' WHERE tab = ''' || col.TableName || ''''
           || ' AND col = ''' || colname || '''';
      cnum := dbms_sql.open_cursor;
      dbms_output.put_line('ODCIStatsDelete>>>>>');
      dbms_output.put_line('ODCIStatsDelete>>>>>' || stmt);
      dbms_sql.parse(cnum, stmt, dbms_sql.native);
      junk := dbms_sql.execute(cnum);
      dbms_sql.close_cursor(cnum);
    END IF;

    RETURN ODCIConst.Success;
  END;
```

ODCIStatsCollect Method (for `power_idxtype` Domain Indexes)

The `ODCIStatsCollect` function collects statistics for domain indexes whose indextype is `power_idxtype`. In the power demand cartridge, this function simply analyzes the index-organized table that stores the index data.

The function takes the index information as an object parameter whose type is `SYS.ODCIINDEXINFO`. The type attributes include the index name, owner name, and so on. Options specified in the `ANALYZE` command used to collect the index statistics are also passed in as parameters. For example, if `ANALYZE ESTIMATE` is used, then the percentage or number of rows is passed in. The output parameter `rawstats` is not used.

```
  STATIC FUNCTION ODCIStatsCollect (ia sys.ODCIIndexInfo,
     options sys.ODCIStatsOptions, rawstats OUT RAW, env sys.ODCIEnv)
     RETURN NUMBER IS
     cnum             INTEGER;
     stmt             VARCHAR2(1000);
     junk             INTEGER;
  BEGIN
    -- To analyze a domain index, simply analyze the table that
    -- implements the index

    sys.ODCIIndexInfoDump(ia);
    sys.ODCIStatsOptionsDump(options);

    stmt := 'ANALYZE TABLE '
         || ia.IndexSchema || '.' || ia.IndexName || '_pidx'
         || ' COMPUTE STATISTICS';

    dbms_output.put_line('**** Analyzing index '
                         || ia.IndexSchema || '.' || ia.IndexName);
    dbms_output.put_line('SQL Statement: ' || stmt);

    cnum := dbms_sql.open_cursor;
    dbms_sql.parse(cnum, stmt, dbms_sql.native);
    junk := dbms_sql.execute(cnum);
    dbms_sql.close_cursor(cnum);

    rawstats := NULL;

    RETURN ODCIConst.Success;
  END;
```

ODCIStatsDelete Method (for `power_idxtype` Domain Indexes)

The `ODCIStatsDelete` function deletes statistics for domain indexes whose indextype is `power_idxtype`. In the power demand cartridge, this function simply deletes the statistics of the index-organized table that stores the index data.

The function takes the index information as an object parameter whose type is `SYS.ODCIINDEXINFO`. The type attributes include the index name, owner name, and so on.

```
STATIC FUNCTION ODCIStatsDelete(ia sys.ODCIIndexInfo, env sys.ODCIEnv)
   RETURN NUMBER IS
   cnum              INTEGER;
   stmt              VARCHAR2(1000);
   junk              INTEGER;
BEGIN
  -- To delete statistics for a domain index, simply delete the
  -- statistics for the table implementing the index

  sys.ODCIIndexInfoDump(ia);

  stmt := 'ANALYZE TABLE '
       || ia.IndexSchema || '.' || ia.IndexName || '_pidx'
       || ' DELETE STATISTICS';

  dbms_output.put_line('**** Analyzing (delete) index '
                       || ia.IndexSchema || '.' || ia.IndexName);
  dbms_output.put_line('SQL Statement: ' || stmt);

  cnum := dbms_sql.open_cursor;
  dbms_sql.parse(cnum, stmt, dbms_sql.native);
  junk := dbms_sql.execute(cnum);
  dbms_sql.close_cursor(cnum);
```

```
    RETURN ODCIConst.Success;
  END;
```

ODCIStatsSelectivity Method (for `Specific` Queries)

The first definition of the `ODCIStatsSelectivity` function estimates the selectivity of operator or function predicates for `Specific` queries. For example, if a query asks for all instances where cell (3,7) has a value equal to 25, the function estimates the percentage of rows in which the given cell has the specified value. (This definition of `ODCIStatsSelectivity` differs from the definition in the next section in that it includes the `cell` parameter for the position of the cell.)

The `pred` parameter contains the function information (the functional implementation of an operator in an operator predicate); this parameter is an object instance of type `SYS.ODCIPREDINFO`. The selectivity is returned as a percentage in the `sel` output parameter. The `args` parameter (an object instance of type `SYS.ODCIARGDESCLIST`) contains a descriptor for each argument of the function as well as the start and stop values of the function. For example, an argument might be a column in which case the argument descriptor will contain the table name, column name, and so forth. The `strt` and `stop` parameters are the lower and upper boundary points for the function return value. If the function in a predicate contains a literal of type `PowerDemand_Typ`, the `object` parameter will contain the value in the form of an object constructor. The `cell` parameter is the cell position and the `value` parameter is the value in the cell specified by the function (`PowerXxxxxSpecific_Func`).

The selectivity is estimated by using a technique similar to that used for simple range predicates. For example, a simple estimate for the selectivity of a predicate like

```
  c > v
```

is `(M-v)/(M-m)` where `m` and `M` are the minimum and maximum values, respectively, for the column `c` (as determined from the column statistics), provided the value `v` lies between `m` and `M`.

The `get_selectivity` function computes the selectivity of a simple range predicate given the minimum and maximum values of the column in the predicate. It assumes that the column values in the table are uniformly distributed between the minimum and maximum values.

```
CREATE FUNCTION get_selectivity(relop VARCHAR2, value NUMBER,
                                lo NUMBER, hi NUMBER, ndv NUMBER)
  RETURN NUMBER AS
  sel NUMBER := NULL;
  ndv NUMBER;
BEGIN
  -- This function computes the selectivity (as a percentage)
  -- of a predicate
  --            col <relop> <value>
  -- where <relop> is one of: =, !=, <, <=, >, >=
  --       <value> is one of: 0, 1
  -- lo and hi are the minimum and maximum values of the column in
  -- the table.  This function performs a simplistic estimation of the
  -- selectivity by assuming that the range of distinct values of
  -- the column is distributed uniformly in the range lo..hi and that
  -- each distinct value occurs nrows/(hi-lo+1) times (where nrows is
  -- the number of rows).

  IF ndv IS NULL OR ndv <= 0 THEN
    RETURN 0;
  END IF;

  -- col != <value>
  IF relop = '!=' THEN
    IF value between lo and hi THEN
      sel := 1 - 1/ndv;
    ELSE
      sel := 1;
    END IF;

  -- col = <value>
  ELSIF relop = '=' THEN
    IF value between lo and hi THEN
      sel := 1/ndv;
    ELSE
      sel := 0;
    END IF;

  -- col >= <value>
  ELSIF relop = '>=' THEN
    IF lo = hi THEN
      IF value <= lo THEN
        sel := 1;
      ELSE
        sel := 0;
      END IF;
    ELSIF value between lo and hi THEN
      sel := (hi-value)/(hi-lo) + 1/ndv;
    ELSIF value < lo THEN
      sel := 1;
    ELSE
      sel := 0;
    END IF;

  -- col < <value>
  ELSIF relop = '<' THEN
    IF lo = hi THEN
      IF value > lo THEN
        sel := 1;
      ELSE
        sel := 0;
      END IF;
    ELSIF value between lo and hi THEN
      sel := (value-lo)/(hi-lo);
    ELSIF value < lo THEN
      sel := 0;
    ELSE
      sel := 1;
    END IF;

  -- col <= <value>
  ELSIF relop = '<=' THEN
    IF lo = hi THEN
      IF value >= lo THEN
        sel := 1;
      ELSE
        sel := 0;
      END IF;
    ELSIF value between lo and hi THEN
      sel := (value-lo)/(hi-lo) + 1/ndv;
    ELSIF value < lo THEN
      sel := 0;
    ELSE
      sel := 1;
    END IF;

  -- col > <value>
  ELSIF relop = '>' THEN
    IF lo = hi THEN
      IF value < lo THEN
        sel := 1;
      ELSE
        sel := 0;
      END IF;
    ELSIF value between lo and hi THEN
      sel := (hi-value)/(hi-lo);
    ELSIF value < lo THEN
      sel := 1;
    ELSE
      sel := 0;
    END IF;

  END IF;

  RETURN least(100, ceil(100*sel));

END;
/
```

The ODCIStatsSelectivity function estimates the selectivity for function predicates which have constant start and stop values. Further, the first argument of the function in the predicate must be a column of type PowerDemand_Typ and the remaining arguments must be constants.

```
  STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
     sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
     object PowerDemand_Typ, cell NUMBER, value NUMBER, env sys.ODCIEnv)
     RETURN NUMBER IS
     fname              varchar2(30);
     relop              varchar2(2);
     lo                 NUMBER;
     hi                 NUMBER;
     nrows              NUMBER;
     colname            VARCHAR2(30);
     statsexists        BOOLEAN := FALSE;
     stats              PowerCartUserStats%ROWTYPE;
     CURSOR c1(cell NUMBER, tname VARCHAR2, cname VARCHAR2) IS
       SELECT * FROM PowerCartUserStats
       WHERE cpos = cell
         AND tab = tname
         AND col = cname;
  BEGIN
    -- compute selectivity only when predicate is of the form:
    --      fn(col, <cell>, <value>) <relop> <val>
    -- In all other cases, return an error and let the optimizer
    -- make a guess.  We also assume that the function "fn" has
    -- a return value of 0, 1, or NULL.
    -- start value
    IF (args(1).ArgType != ODCIConst.ArgLit AND
        args(1).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
    END IF;

    -- stop value
    IF (args(2).ArgType != ODCIConst.ArgLit AND
        args(2).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
    END IF;

    -- first argument of function
    IF (args(3).ArgType != ODCIConst.ArgCol) THEN
      RETURN ODCIConst.Error;
    END IF;

    -- second argument of function
    IF (args(4).ArgType != ODCIConst.ArgLit AND
        args(4).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
    END IF;

    -- third argument of function
    IF (args(5).ArgType != ODCIConst.ArgLit AND
        args(5).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
    END IF;

    colname := rtrim(ltrim(args(3).colName, '"'), '"');
```

The first (column) argument of the function in the predicate must have statistics collected for it (by issuing the `ANALYZE` command which will call `ODCIStatsCollect` for the column). If statistics have not been collected, `ODCIStatsSelectivity` returns an error status.

```
    -- Check if the statistics table exists (we are using a
    -- user-defined table to store the user-defined statistics).
    -- Get user-defined statistics: MIN, MAX, NROWS
    FOR stats IN c1(cell, args(3).TableName, colname) LOOP
      -- Get user-defined statistics: MIN, MAX, NROWS
      lo := stats.lo;
      hi := stats.hi;
      nrows := stats.nrows;
      statsexists := TRUE;
      EXIT;
    END LOOP;

    -- If no user-defined statistics were collected, return error
    IF not statsexists THEN
      RETURN ODCIConst.Error;
    END IF;
```

Each `Specific` function predicate corresponds to an equivalent range predicate. For example, the predicate:

```
Power_EqualsSpecific_Func(col, 21, 25) = 0
```

which checks that the reading in cell 21 is not equal to 25, corresponds to the equivalent range predicate:

```
col[21] != 25
```

The `ODCIStatsSelectivity` function finds the corresponding range predicates for each `Specific` function predicate. There are several boundary cases where the selectivity can be immediately determined.

```
    -- selectivity is 0 for "fn(col, <cell>, <value>) < 0"
    IF (stop = 0 AND
        bitand(pred.Flags, ODCIConst.PredIncludeStop) = 0) THEN
      sel := 0;
      RETURN ODCIConst.Success;
    END IF;

    -- selectivity is 0 for "fn(col, <cell>, <value>) > 1"
    IF (strt = 1 AND
        bitand(pred.Flags, ODCIConst.PredIncludeStart) = 0) THEN
      sel := 0;
      RETURN ODCIConst.Success;
    END IF;

    -- selectivity is 100% for "fn(col, <cell>, <value>) >= 0"
    IF (strt = 0 AND
        bitand(pred.Flags, ODCIConst.PredExactMatch) = 0 AND
        bitand(pred.Flags, ODCIConst.PredIncludeStart) > 0) THEN
      sel := 100;
      RETURN ODCIConst.Success;
    END IF;

    -- selectivity is 100% for "fn(col, <cell>, <value>) <= 1"
    IF (stop = 1 AND
        bitand(pred.Flags, ODCIConst.PredExactMatch) = 0 AND
        bitand(pred.Flags, ODCIConst.PredIncludeStop) > 0) THEN
      sel := 100;
      RETURN ODCIConst.Success;
    END IF;

    -- get function name
    IF bitand(pred.Flags, ODCIConst.PredObjectFunc) > 0 THEN
      fname := pred.ObjectName;
    ELSE
      fname := pred.MethodName;
    END IF;

    -- convert prefix relational operator to infix:
    -- "Power_EqualsSpecific_Func(col, <cell>, <value>) = 1"
    -- becomes "col[<cell>] = <value>"

    --   Power_EqualsSpecific_Func(col, <cell>, <value>) = 0
    --   Power_EqualsSpecific_Func(col, <cell>, <value>) <= 0
    --   Power_EqualsSpecific_Func(col, <cell>, <value>) < 1
    -- can be transformed to
    --   col[<cell>] != <value>
    IF (fname LIKE upper('Power_Equals%') AND
        (stop = 0 OR
         (stop = 1 AND
          bitand(pred.Flags, ODCIConst.PredIncludeStop) = 0))) THEN
      relop := '!=';

    --   Power_LessThanSpecific_Func(col, <cell>, <value>) = 0
    --   Power_LessThanSpecific_Func(col, <cell>, <value>) <= 0
    --   Power_LessThanSpecific_Func(col, <cell>, <value>) < 1
    -- can be transformed to
    --   col[<cell>] >= <value>
    ELSIF (fname LIKE upper('Power_LessThan%') AND
           (stop = 0 OR
            (stop = 1 AND
             bitand(pred.Flags, ODCIConst.PredIncludeStop) = 0))) THEN
      relop := '>=';

    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) = 0
    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) <= 0
    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) < 1
    -- can be transformed to
    --   col[<cell>] <= <value>
    ELSIF (fname LIKE upper('Power_GreaterThan%') AND
           (stop = 0 OR
            (stop = 1 AND
             bitand(pred.Flags, ODCIConst.PredIncludeStop) = 0))) THEN
      relop := ’<=’;

    --   Power_EqualsSpecific_Func(col, <cell>, <value>) = 1
    --   Power_EqualsSpecific_Func(col, <cell>, <value>) >= 1
    --   Power_EqualsSpecific_Func(col, <cell>, <value>) > 0
    -- can be transformed to
    --   col[<cell>] = <value>
    ELSIF (fname LIKE upper(’Power_Equals%’) AND
           (strt = 1 OR
            (strt = 0 AND
             bitand(pred.Flags, ODCIConst.PredIncludeStart) = 0))) THEN
      relop := ’=’;

    --   Power_LessThanSpecific_Func(col, <cell>, <value>) = 1
    --   Power_LessThanSpecific_Func(col, <cell>, <value>) >= 1
    --   Power_LessThanSpecific_Func(col, <cell>, <value>) > 0
    -- can be transformed to
    --   col[<cell>] < <value>
    ELSIF (fname LIKE upper(’Power_LessThan%’) AND
           (strt = 1 OR
            (strt = 0 AND
             bitand(pred.Flags, ODCIConst.PredIncludeStart) = 0))) THEN
      relop := ’<’;

    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) = 1
    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) >= 1
    --   Power_GreaterThanSpecific_Func(col, <cell>, <value>) > 0
    -- can be transformed to
    --   col[<cell>] > <value>
    ELSIF (fname LIKE upper(’Power_GreaterThan%’) AND
           (strt = 1 OR
            (strt = 0 AND
             bitand(pred.Flags, ODCIConst.PredIncludeStart) = 0))) THEN
      relop := ’>’;

    ELSE
      RETURN ODCIConst.Error;

    END IF;
```

After the `Specific` function predicate is transformed into a simple range predicate, `ODCIStatsSelectivity` calls `get_selectivity` to compute the selectivity for the range predicate (and thus, equivalently, for the Specific function predicate). It returns with a success status.

```
    sel := get_selectivity(relop, value, lo, hi, nrows);
    RETURN ODCIConst.Success;
  END;
```

ODCIStatsSelectivity Method (for Any Queries)

The second definition of the ODCIStatsSelectivity function estimates the selectivity of operator or function predicates for Any queries. For example, if a query asks for all instances where any cell has a value equal to 25, the function estimates the percentage of rows in which any cell has the specified value. (This definition of ODCIStatsSelectivity differs from the definition in the preceding section in that it does not include the cell parameter.)

The pred parameter contains the function information (the functional implementation of an operator in an operator predicate); this parameter is an object instance of type SYS.ODCIPREDINFO. The selectivity is returned as a percentage in the sel output parameter. The args parameter (an object instance of type SYS.ODCIARGDESCLIST) contains a descriptor for each argument of the function as well as the start and stop values of the function. For example, an argument might be a column in which case the argument descriptor will contain the table name, column name, and so forth. The strt and stop parameters are the lower and upper boundary points for the function return value. If the function in a predicate contains a literal of type PowerDemand_Typ, the object parameter will contain the value in the form of an object constructor. The value parameter is the value in the cell specified by the function (Power_XxxxxAny_Func).

The selectivity for Any queries can be calculated as the complement of the probability that none of the cells has the specified value. Thus, if s[i] is the selectivity of the ith cell having the given value, then the selectivity of the Any function predicate can be estimated as:

```
    1 - (1-s[1])(1-s[2])...(1-s[100])
```

assuming that the value of each cell is independent of the values in other cells. This means that this version of the ODCIStatsSelectivity function (for Any queries) can compute its selectivity by calling the first definition of the ODCIStatsSelectivity function (for Specific queries).

```
  STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
     sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
     object PowerDemand_Typ, value NUMBER, env sys.ODCIEnv)
```

```
   RETURN NUMBER IS
   cellsel            NUMBER;
   i                  NUMBER;
   specsel            NUMBER;
   newargs            sys.ODCIArgDescList
                        := sys.ODCIArgDescList(NULL, NULL, NULL,
                                               NULL, NULL);
BEGIN
  -- To compute selectivity for the ANY functions, call the
  -- selectivity function for the SPECIFIC functions.  For example,
  -- the selectivity of the ANY predicate
  --
  --     Power_EqualsAnyFunc(object, value) = 1
  --
  -- is computed as
  --
  --     1 - (1-s[1])(1-s[2])...(1-s[100])
  --
  -- where s[i] is the selectivity of the SPECIFIC predicate
  --
  --     Power_EqualsSpecific_Func(object, i, value) = 1
  --

  sel := 1;
  newargs(1) := args(1);
  newargs(2) := args(2);
  newargs(3) := args(3);
  newargs(4) := sys.ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL);
  newargs(5) := args(4);
  FOR i in 1..100 LOOP
    cellsel := NULL;
    specsel := power_statistics.ODCIStatsSelectivity(pred, cellsel,
                 newargs, strt, stop, object, i, value, env);
    IF specsel = ODCIConst.Success THEN
      sel := sel * (1 - cellsel/100);
    END IF;
  END LOOP;

  sel := (1 - sel) * 100;
  RETURN ODCIConst.Success;
END;
```

ODCIStatsIndexCost Method (for `Specific` Queries)

The first definition of the `ODCIStatsIndexCost` function estimates the cost of the domain index for `Specific` queries. For example, if a query asks for all instances where cell (3,7) has a value equal to 25, the function estimates the cost of the domain index access path to evaluate this query. (This definition of `ODCIStatsIndexCost` differs from the definition in the next section in that it includes the `cmppos` parameter for the position of the cell.)

The `ia` parameter contains the index information (an object instance of type `SYS.ODCIINDEXINFO`). The `sel` parameter is the selectivity of the operator predicate as estimated by the `ODCIStatsSelectivity` function for `Specific` queries. The estimated cost is returned in the `cost` output parameter. The `qi` parameter contains some information about the query and its environment (for example, whether the `ALL_ROWS` or `FIRST_ROWS` optimizer mode is being used). The `pred` parameter contains the operator information (an object instance of type `SYS.ODCIPREDINFO`). The `args` parameter contains descriptors of the value arguments of the operator as well as the start and stop values of the operator. The `strt` and `stop` parameters are the lower and upper boundary points for the operator return value. The `cmppos` parameter is the cell position and `cmpval` is the value in the cell specified by the operator (`Power_XxxxxSpecific`).

In the power demand cartridge, the domain index cost for `Specific` queries is the same as the domain index cost for `Any` queries, so this version of the `ODCIStatsIndexCost` function simply calls the second definition of the function (described in the next section).

```
STATIC FUNCTION ODCIStatsIndexCost(ia sys.ODCIIndexInfo,
   sel NUMBER, cost OUT sys.ODCICost, qi sys.ODCIQueryInfo,
   pred sys.ODCIPredInfo, args sys.ODCIArgDescList,
   strt NUMBER, stop NUMBER, cmppos NUMBER, cmpval NUMBER, env sys.ODCIEnv)
   RETURN NUMBER IS
BEGIN
  -- This is the cost for queries on a specific cell; simply
  -- use the cost for queries on any cell.
  RETURN ODCIStatsIndexCost(ia, sel, cost, qi, pred, args,
                            strt, stop, cmpval, env);
END;
```

ODCIStatsIndexCost Method (for `Any` Queries)

The second definition of the `ODCIStatsIndexCost` function estimates the cost of the domain index for `Any` queries. For example, if a query asks for all instances where any cell has a value equal to 25, the function estimates the cost of the domain index access path to evaluate this query. (This definition of `ODCIStatsIndexCost` differs from the definition in the preceding section in that it does not include the `cmppos` parameter.)

The `ia` parameter contains the index information (an object instance of type `SYS.ODCIINDEXINFO`). The `sel` parameter is the selectivity of the operator predicate as estimated by the `ODCIStatsSelectivity` function for `Any` queries. The estimated cost is returned in the `cost` output parameter. The `qi` parameter contains some information about the query and its environment (for example, whether the `ALL_ROWS` or `FIRST_ROWS` optimizer mode is being used). The `pred` parameter contains the operator information (an object instance of type `SYS.ODCIPREDINFO`). The `args` parameter contains descriptors of the value arguments of the operator as well as the start and stop values of the operator. The `strt` and `stop` parameters are the lower and upper boundary points for the operator return value. The `cmpval` parameter is the value in the cell specified by the operator (`Power_XxxxxAny`).

The index cost is estimated as the number of blocks in the index-organized table implementing the index multiplied by the selectivity of the operator predicate times a constant factor.

```
STATIC FUNCTION ODCIStatsIndexCost(ia sys.ODCIIndexInfo,
   sel NUMBER, cost OUT sys.ODCICost, qi sys.ODCIQueryInfo,
   pred sys.ODCIPredInfo, args sys.ODCIArgDescList,
   strt NUMBER, stop NUMBER, cmpval NUMBER, env sys.ODCIEnv)
   RETURN NUMBER IS
   ixtable             VARCHAR2(40);
   numblocks           NUMBER := NULL;
   get_table           user_tables%ROWTYPE;
   CURSOR c1(tab VARCHAR2) IS
     SELECT * FROM user_tables WHERE table_name = tab;
BEGIN
  -- This is the cost for queries on any cell.

  -- To compute the cost of a domain index, multiply the
  -- number of blocks in the table implementing the index
  -- with the selectivity

  -- Return if we don't have predicate selectivity
  IF sel IS NULL THEN
    RETURN ODCIConst.Error;
  END IF;

  cost := sys.ODCICost(NULL, NULL, NULL, NULL);

  -- Get name of table implementing the domain index
```

```
    ixtable := ia.IndexName || '_pidx';

    -- Get number of blocks in domain index
    FOR get_table IN c1(upper(ixtable)) LOOP
      numblocks := get_table.blocks;
      EXIT;
    END LOOP;

    IF numblocks IS NULL THEN
      -- Exit if there are no user-defined statistics for the index
      RETURN ODCIConst.Error;
    END IF;

    cost.CPUCost := ceil(400*(sel/100)*numblocks);
    cost.IOCost := ceil(1.5*(sel/100)*numblocks);
    RETURN ODCIConst.Success;
  END;
```

ODCIStatsFunctionCost Method

The `ODCIStatsFunctionCost` function estimates the cost of evaluating a function (`Power_XxxxxSpecific_Func` or `Power_XxxxxAny_Func`).

The `func` parameter contains the function information; this parameter is an object instance of type `SYS.ODCIFUNCINFO`. The estimated cost is returned in the output `cost` parameter. The `args` parameter (an object instance of type `SYS.ODCIARGDESCLIST`) contains a descriptor for each argument of the function. If the function contains a literal of type `PowerDemand_Typ` as its first argument, the `object` parameter will contain the value in the form of an object constructor. The `value` parameter is the value in the cell specified by the function (`PowerXxxxxSpecific_Func` or `Power_XxxxxAny_Func`).

The function cost is simply estimated as some default value depending on the function name. Since the functions don't read any data from disk, the I/O cost is set to zero.

```
  STATIC FUNCTION ODCIStatsFunctionCost(func sys.ODCIFuncInfo,
     cost OUT sys.ODCICost, args sys.ODCIArgDescList,
     object PowerDemand_Typ, value NUMBER, env sys.ODCIEnv)
     RETURN NUMBER IS
     fname               VARCHAR2(30);
  BEGIN
    cost := sys.ODCICost(NULL, NULL, NULL, NULL);

    -- Get function name
    IF  bitand(func.Flags, ODCIConst.ObjectFunc) > 0 THEN
      fname := func.ObjectName;
    ELSE
      fname := func.MethodName;
    END IF;

    IF fname LIKE upper('Power_LessThan%') THEN
      cost.CPUCost := 5000;
      cost.IOCost := 0;
      RETURN ODCIConst.Success;
    ELSIF fname LIKE upper('Power_Equals%') THEN
      cost.CPUCost := 7000;
      cost.IOCost := 0;
      RETURN ODCIConst.Success;
    ELSIF fname LIKE upper('Power_GreaterThan%') THEN
      cost.CPUCost := 5000;
      cost.IOCost := 0;
      RETURN ODCIConst.Success;
    ELSE
      RETURN ODCIConst.Error;
    END IF;
  END;
```

Associating the Extensible Optimizer Methods with Database Objects

In order for the optimizer to use the methods defined in the `power_statistics` object type, they have to be associated with the appropriate database objects. The following statements do this.

```
-- Associate statistics type with types, indextypes, and functions
ASSOCIATE STATISTICS WITH TYPES PowerDemand_Typ USING power_statistics;
ASSOCIATE STATISTICS WITH INDEXTYPES power_idxtype USING power_statistics;
ASSOCIATE STATISTICS WITH FUNCTIONS
  Power_EqualsSpecific_Func,
  Power_GreaterThanSpecific_Func,
  Power_LessThanSpecific_Func,
  Power_EqualsAny_Func,
  Power_GreaterThanAny_Func,
  Power_LessThanAny_Func
  USING power_statistics;
```

Analyzing the Database Objects

Analyzing tables, columns, and indexes ensures that the optimizer has the relevant statistics to estimate accurate costs for various access paths and choose a good plan. Further, the selectivity and cost functions defined in the `power_statistics` object type rely on the presence of statistics. The following statements analyze the database objects and verify that statistics were indeed collected.

```
-- Analyze the table
ANALYZE TABLE PowerDemand_Tab COMPUTE STATISTICS;

-- Verify that user-defined statistics were collected
SELECT tab tablename, col colname, cpos, lo, hi, nrows
FROM PowerCartUserStats
WHERE nrows IS NOT NULL
ORDER BY cpos;

-- Delete the statistics
ANALYZE TABLE PowerDemand_Tab DELETE STATISTICS;

-- Verify that user-defined statistics were deleted
SELECT tab tablename, col colname, cpos, lo, hi, nrows
FROM PowerCartUserStats
WHERE nrows IS NOT NULL
ORDER BY cpos;

-- Re-analyze the table
ANALYZE TABLE PowerDemand_Tab COMPUTE STATISTICS;

-- Verify that user-defined statistics were re-collected
SELECT tab tablename, col colname, cpos, lo, hi, nrows
FROM PowerCartUserStats
WHERE nrows IS NOT NULL
ORDER BY cpos;
```

Testing the Domain Index

This section explains the parts of the power demand example that perform some simple tests of the domain index. These tests consist of:

- Creating the power demand table (`PowerDemand_Tab`) and populating it with a small amount of data
- Executing some queries before the index is created (and showing the execution plans without an index being used)

The execution plans show that a full table scan is performed in each case.

- Creating the index on the grid
- Executing the same queries after the index is created (and showing the execution plans with the index being used)

  The execution plans show that Oracle is using the index and not performing full table scans, thus resulting in more efficient execution.

The statements in this section are available online in the example file (tkqxpwr.sql).

Creating and Populating the Power Demand Table

The power demand table is created with two columns:

- `region`, to allow the electric utility to use the grid scheme in multiple areas or states. Each region (for example, New York, New Jersey, Pennsylvania, and so on) is represented by a 10x10 grid.
- `sample`, a collection of samplings (power demand readings from each cell in the grid), defined using the `PowerDemand_Typ` object type.

```
CREATE TABLE PowerDemand_Tab (
  -- Region for which these power demand readings apply
  region NUMBER,
  -- Values for each "sampling" time (for a given hour)
  sample PowerDemand_Typ
);
```

Several rows are inserted, representing power demand data for two regions (1 and 2) for several hourly timestamps. For simplicity, values are inserted only into the first 5 positions of each grid (the remaining 95 values are set to null).

```
-- The next INSERT statements "cheat" by supplying
-- only 5 grid values (instead of 100).

-- First 5 INSERT statements are for region 1 (1 AM to 5 AM on
-- 01-Feb-1998).

INSERT INTO PowerDemand_Tab VALUES(1,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(55,8,13,9,5),
   to_date('02-01-1998 01','MM-DD-YYYY HH'))
);

INSERT INTO PowerDemand_Tab VALUES(1,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(56,8,13,9,3),
   to_date('02-01-1998 02','MM-DD-YYYY HH'))
);

INSERT INTO PowerDemand_Tab VALUES(1,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(55,8,13,9,3),
   to_date('02-01-1998 03','MM-DD-YYYY HH'))
);

INSERT INTO PowerDemand_Tab VALUES(1,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(54,8,13,9,3),
   to_date('02-01-1998 04','MM-DD-YYYY HH'))
);

INSERT INTO PowerDemand_Tab VALUES(1,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(54,8,12,9,3),
   to_date('02-01-1998 05','MM-DD-YYYY HH'))
);

-- Also insert some rows for region 2.

INSERT INTO PowerDemand_Tab VALUES(2,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(9,8,11,16,5),
   to_date('02-01-1998 01','MM-DD-YYYY HH'))
);

INSERT INTO PowerDemand_Tab VALUES(2,
   PowerDemand_Typ(NULL, NULL, NULL, PowerGrid_Typ(9,8,11,20,5),
   to_date('02-01-1998 02','MM-DD-YYYY HH'))
);
```

Finally, the values for `TotGridDemand`, `MaxCellDemand`, and `MinCellDemand` are computed and set for each of the newly inserted rows, and these values are displayed.

```
DECLARE
CURSOR c1 IS SELECT Sample, Region FROM PowerDemand_Tab FOR UPDATE;
s PowerDemand_Typ;
r NUMBER;
BEGIN
  OPEN c1;
  LOOP
     FETCH c1 INTO s,r;
     EXIT WHEN c1%NOTFOUND;
     s.SetTotalDemand;
     s.SetMaxDemand;
```

```
      s.SetMinDemand;
      dbms_output.put_line(s.TotGridDemand);
      dbms_output.put_line(s.MaxCellDemand);
      dbms_output.put_line(s.MinCellDemand);
      UPDATE PowerDemand_Tab SET Sample = s WHERE CURRENT OF c1;
   END LOOP;
   CLOSE c1;
END;
/

-- Examine the values.
SELECT region, P.Sample.TotGridDemand, P.Sample.MaxCellDemand,
   P.Sample.MinCellDemand,
   to_char(P.sample.sampletime, 'MM-DD-YYYY HH')
 FROM PowerDemand_Tab P;
```

Querying Without the Index

The queries is this section are executed by applying the underlying function (`PowerEqualsSpecific_Func`) for every row in the table, because the index has not yet been defined.

The example file includes queries that check, both for a specific cell number and for any cell number, for values equal to, greater than, and less than a specified value. For example, the equality queries are as follows:

```
SET SERVEROUTPUT ON
-------------------------------------------------------------------
-- Query, referencing the operators (without index)
-------------------------------------------------------------------
explain plan for
SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,2,10) = 1;
@tkoqxpll

SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,2,10) = 1;

explain plan for
SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,1,25) = 1;
@tkoqxpll

SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,1,25) = 1;

explain plan for
SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,2,8) = 1;
@tkoqxpll

SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_Equals(P.Sample,2,8) = 1;

explain plan for
SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_EqualsAny(P.Sample,9) = 1;
@tkoqxpll

SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     P.Sample.MinCellDemand
   FROM PowerDemand_Tab P
   WHERE Power_EqualsAny(P.Sample,9) = 1;
```

The execution plans show that a full table scan is performed in each case:

```
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    FULL            POWERDEMAND_TAB
```

Creating the Index

The index is created on the `sample` column in the power demand table.

```
CREATE INDEX PowerIndex ON PowerDemand_Tab(Sample)
   INDEXTYPE IS power_idxtype;
```

Querying with the Index

The queries in this section are the same as those in "Querying Without the Index" on page 13-67, but this time the index is used.

The execution plans show that Oracle is using the domain index and not performing full table scans, thus resulting in more efficient execution. For example:

```
SQLPLUS> -------------------------------------------------------------------
SQLPLUS> -- Query, referencing the operators (with index)
SQLPLUS> -------------------------------------------------------------------
SQLPLUS> explain plan for
     2> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     3>      P.Sample.MinCellDemand
     4>    FROM PowerDemand_Tab P
     5>    WHERE Power_Equals(P.Sample,2,10) = 1;
Statement processed.
SQLPLUS> @tkoqxpll
SQLPLUS> set echo off
Echo                            OFF
Charwidth                       15
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    BY ROWID        POWERDEMAND_TAB
DOMAIN INDEX                    POWERINDEX
3 rows selected.
Statement processed.
Echo                            ON
SQLPLUS>
SQLPLUS> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     2>      P.Sample.MinCellDemand
     3>    FROM PowerDemand_Tab P
     4>    WHERE Power_Equals(P.Sample,2,10) = 1;
REGION     SAMPLE.TOT SAMPLE.MAX SAMPLE.MIN
---------- ---------- ---------- ----------
0 rows selected.
ODCIIndexStart>>>>>
ODCIIndexInfo
Index owner : POWERCARTUSER
Index name : POWERINDEX
Table owner : POWERCARTUSER
Table name : POWERDEMAND_TAB
Indexed column : "SAMPLE"
Indexed column type :POWERDEMAND_TYP
Indexed column type schema:POWERCARTUSER
ODCIPredInfo
Object owner : POWERCARTUSER
Object name : POWER_EQUALS
Method name :
Predicate bounds flag :
     Exact Match
     Include Start Key
     Include Stop Key
start key : 1
stop key : 1
compare position : 2
compare value : 10
ODCIIndexStart>>>>>select r from POWERCARTUSER.POWERINDEX_pidx where cpos ='2'
and cval ='10'
ODCIIndexFetch>>>>>
Nrows : 2000
ODCIIndexClose>>>>>
SQLPLUS>
SQLPLUS> explain plan for
     2> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     3>      P.Sample.MinCellDemand
     4>    FROM PowerDemand_Tab P
     5>    WHERE Power_Equals(P.Sample,2,8) = 1;
Statement processed.
SQLPLUS> @tkoqxpll
SQLPLUS> set echo off
Echo                            OFF
Charwidth                       15
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    BY ROWID        POWERDEMAND_TAB
DOMAIN INDEX                    POWERINDEX
3 rows selected.
Statement processed.
Echo                            ON
SQLPLUS>
SQLPLUS> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     2>      P.Sample.MinCellDemand
     3>    FROM PowerDemand_Tab P
     4>    WHERE Power_Equals(P.Sample,2,8) = 1;
```

```
REGION     SAMPLE.TOT SAMPLE.MAX SAMPLE.MIN
---------- ---------- ---------- ----------
         1         90         55          5
         1         89         56          3
         1         88         55          3
         1         87         54          3
         1         86         54          3
         2         49         16          5
         2         53         20          5
7 rows selected.
ODCIIndexStart>>>>>
ODCIIndexInfo
Index owner : POWERCARTUSER
Index name : POWERINDEX
Table owner : POWERCARTUSER
Table name : POWERDEMAND_TAB
Indexed column : "SAMPLE"
Indexed column type :POWERDEMAND_TYP
Indexed column type schema:POWERCARTUSER
ODCIPredInfo
Object owner : POWERCARTUSER
Object name : POWER_EQUALS
Method name :
Predicate bounds flag :
     Exact Match
     Include Start Key
     Include Stop Key
start key : 1
stop key : 1
compare position : 2
compare value : 8
ODCIIndexStart>>>>>select r from POWERCARTUSER.POWERINDEX_pidx where cpos ='2'
and cval ='8'
ODCIIndexFetch>>>>>
Nrows : 2000
ODCIIndexClose>>>>>
SQLPLUS>
SQLPLUS> explain plan for
     2> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     3>      P.Sample.MinCellDemand
     4>    FROM PowerDemand_Tab P
     5>    WHERE Power_EqualsAny(P.Sample,9) = 1;
Statement processed.
SQLPLUS> @tkoqxpll
SQLPLUS> set echo off
```

```
Echo                            OFF
Charwidth                       15
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    BY ROWID        POWERDEMAND_TAB
DOMAIN INDEX                    POWERINDEX
3 rows selected.
Statement processed.
Echo                            ON
SQLPLUS>
SQLPLUS> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     2>      P.Sample.MinCellDemand
     3>    FROM PowerDemand_Tab P
     4>    WHERE Power_EqualsAny(P.Sample,9) = 1;
REGION     SAMPLE.TOT SAMPLE.MAX SAMPLE.MIN
---------- ---------- ---------- ----------
         1         90         55          5
         1         89         56          3
         1         88         55          3
         1         87         54          3
         1         86         54          3
         2         49         16          5
         2         53         20          5
7 rows selected.
ODCIIndexStart>>>>>
ODCIIndexInfo
Index owner : POWERCARTUSER
Index name : POWERINDEX
Table owner : POWERCARTUSER
Table name : POWERDEMAND_TAB
Indexed column : "SAMPLE"
Indexed column type :POWERDEMAND_TYP
Indexed column type schema:POWERCARTUSER
ODCIPredInfo
Object owner : POWERCARTUSER
Object name : POWER_EQUALSANY
Method name :
Predicate bounds flag :
     Exact Match
     Include Start Key
     Include Stop Key
start key : 1
stop key : 1
compare value : 9
```

```
ODCIIndexStart>>>>>select distinct r from POWERCARTUSER.POWERINDEX_pidx where
cval ='9'
ODCIIndexFetch>>>>>
Nrows : 2000
ODCIIndexClose>>>>>
SQLPLUS>
SQLPLUS> explain plan for
     2> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     3>      P.Sample.MinCellDemand
     4>    FROM PowerDemand_Tab P
     5>    WHERE Power_GreaterThanAny(P.Sample,50) = 1;
Statement processed.
SQLPLUS> @tkoqxpll
SQLPLUS> set echo off
Echo                           OFF
Charwidth                      15
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    BY ROWID        POWERDEMAND_TAB
DOMAIN INDEX                    POWERINDEX
3 rows selected.
Statement processed.
Echo                           ON
SQLPLUS>
SQLPLUS> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     2>      P.Sample.MinCellDemand
     3>    FROM PowerDemand_Tab P
     4>    WHERE Power_GreaterThanAny(P.Sample,50) = 1;
REGION     SAMPLE.TOT SAMPLE.MAX SAMPLE.MIN
---------- ---------- ---------- ----------
         1         90         55          5
         1         89         56          3
         1         88         55          3
         1         87         54          3
         1         86         54          3
5 rows selected.
ODCIIndexStart>>>>>
ODCIIndexInfo
Index owner : POWERCARTUSER
Index name : POWERINDEX
Table owner : POWERCARTUSER
Table name : POWERDEMAND_TAB
Indexed column : "SAMPLE"
Indexed column type :POWERDEMAND_TYP
Indexed column type schema:POWERCARTUSER
ODCIPredInfo
Object owner : POWERCARTUSER
Object name : POWER_GREATERTHANANY
Method name :
Predicate bounds flag :
     Exact Match
     Include Start Key
     Include Stop Key
start key : 1
stop key : 1
compare value : 50
ODCIIndexStart>>>>>select distinct r from POWERCARTUSER.POWERINDEX_pidx where cv
al >'50'
ODCIIndexFetch>>>>>
Nrows : 2000
ODCIIndexClose>>>>>
SQLPLUS>
SQLPLUS> explain plan for
     2> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     3>      P.Sample.MinCellDemand
     4>    FROM PowerDemand_Tab P
     5>    WHERE Power_LessThanAny(P.Sample,50) = 0;
Statement processed.
SQLPLUS> @tkoqxpll
SQLPLUS> set echo off
Echo                            OFF
Charwidth                       15
OPERATIONS      OPTIONS         OBJECT_NAME
--------------- --------------- ---------------
SELECT STATEMEN
TABLE ACCESS    BY ROWID        POWERDEMAND_TAB
DOMAIN INDEX                    POWERINDEX
3 rows selected.
Statement processed.
Echo                            ON
SQLPLUS>
SQLPLUS> SELECT P.Region, P.Sample.TotGridDemand ,P.Sample.MaxCellDemand,
     2>      P.Sample.MinCellDemand
     3>    FROM PowerDemand_Tab P
     4>    WHERE Power_LessThanAny(P.Sample,50) = 0;
REGION     SAMPLE.TOT SAMPLE.MAX SAMPLE.MIN
---------- ---------- ---------- ----------
0 rows selected.
ODCIIndexStart>>>>>
```

```
ODCIIndexInfo
Index owner : POWERCARTUSER
Index name : POWERINDEX
Table owner : POWERCARTUSER
Table name : POWERDEMAND_TAB
Indexed column : "SAMPLE"
Indexed column type :POWERDEMAND_TYP
Indexed column type schema:POWERCARTUSER
ODCIPredInfo
Object owner : POWERCARTUSER
Object name : POWER_LESSTHANANY
Method name :
Predicate bounds flag :
     Exact Match
     Include Start Key
     Include Stop Key
start key : 0
stop key : 0
compare value : 50
ODCIIndexStart>>>>>select distinct r from POWERCARTUSER.POWERINDEX_pidx minus se
lect distinct r from POWERCARTUSER.POWERINDEX_pidx where cval <'50'
ODCIIndexFetch>>>>>
Nrows : 2000
ODCIIndexClose>>>>>
```

14
PSBTREE: An Example of Extensible Indexing

This chapter presents an extensible indexing example in which some of the ODCIIndex interface routines are implemented in C:

Introduction

The example in this chapter gives a general illustration of how to implement the extensible indexing interface routines in C. The example tries to concentrate on topics that are common to all implementations and glosses over domain-specific details.

The code for the example is in the demo directory (see file `extdemo5.sql`). It extends an earlier example (`extdemo2.sql`, also in demo directory) by adding to the indextype support for local domain indexes on range partitioned tables.

Design of the indextype

The indextype implemented here, called `PSBtree,` operates like a btree index. It supports three user-defined operators:

- `gt(Greater Than)`
- `lt(Less Than)`
- `eq(EQuals)`

These operators operate on operands of `VARCHAR2` datatype.

The index data consists of records of the form `<key, rid>` where `key` is the value of the indexed column and `rid` is the row identifier of the corresponding row. To simplify the implementation of the indextype, the index data is stored in an index-organized table.

When an index is a local domain index, one index-organized table is created for each partition to store the index data for that partition. Thus, the index manipulation routines merely translate operations on the `PSBtree` into operations on the table storing the index data.

When a user creates a `PSBtree` index (a local index), `n` tables are created consisting of the indexed column and a `rowid` column, where `n` is the number of partitions in the base table. Inserts into the base table cause appropriate insertions into the affected index table. Deletes and updates are handled similarly. When the `PSBtree` is queried based on a user-defined operator (one of `gt`, `lt` and `eq`), an appropriate query is issued against the index table to retrieve all the satisfying rows. Appropriate partition pruning occurs, and only the index tables that correspond to the relevant, or "interesting," partitions are accessed.

Implementing Operators

The PSBtree indextype supports three operators. Each operator has a corresponding functional implementation. The functional implementations of the eq, gt and lt operators are presented in the following section.

Create Functional Implementations

Functional Implementation of EQ (EQUALS)

The functional implementation for eq is provided by a function (bt_eq) that takes in two VARCHAR2 parameters and returns 1 if they are equal and 0 otherwise.

```
CREATE FUNCTION bt_eq(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
  IF a = b then
    RETURN 1;
  ELSE
    RETURN 0;
  END IF;
END;
```

Functional Implementation of LT (LESS THAN)

The functional implementation for lt is provided by a function (bt_lt) that takes in two VARCHAR2 parameters and returns 1 if the first parameter is less than the second, 0 otherwise.

```
CREATE FUNCTION bt_lt(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
  IF a < b then
    RETURN 1;
  ELSE
   RETURN 0;
  END IF;
END;
```

Functional Implementation of GT (GREATER THAN)

The functional implementation for gt is provided by a function (bt_gt) that takes in two VARCHAR2 parameters and returns 1 if the first parameter is greater than the second, 0 otherwise.

```
CREATE FUNCTION bt_gt(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
  IF a > b then
    RETURN 1;
  ELSE
    RETURN 0;
  END IF;
END;
```

Create Operators

To create the operator, you need to specify the signature of the operator along with its return type and its functional implementation.

Operator EQ

```
CREATE OPERATOR eq
BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER
USING bt_eq;
```

Operator LT

```
CREATE OPERATOR lt
BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER
USING bt_lt;
```

Operator GT

```
CREATE OPERATOR gt
BINDING (VARCHAR2, VARCHAR2) RETURN NUMBER
USING bt_gt;
```

Implementing the Index Routines

1. Define an implementation type that implements the `ODCIIndex` interface routines.

```
CREATE TYPE psbtree_im AS OBJECT
(
  scanctx RAW(4),
  STATIC FUNCTION ODCIGetInterfaces(ifclist OUT SYS.ODCIObjectList)
```

```
    RETURN NUMBER,
  STATIC FUNCTION ODCIIndexCreate (ia SYS.ODCIIndexInfo, parms VARCHAR2,
    env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexDrop(ia SYS.ODCIIndexInfo, env SYS.ODCIEnv)
    RETURN NUMBER,
  STATIC FUNCTION ODCIIndexExchangePartition(ia SYS.ODCIIndexInfo,
    ia1 SYS.ODCIIndexInfo, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexMergePartition(ia SYS.ODCIIndexInfo,
    part_name1 SYS.ODCIPartInfo, part_name2 SYS.ODCIPartInfo, parms VARCHAR2,
    env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexSplitPartition(ia SYS.ODCIIndexInfo,
    part_name1 SYS.ODCIPartInfo, part_name2 SYS.ODCIPartInfo, parms VARCHAR2,
    env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexTruncate(ia SYS.ODCIIndexInfo, env SYS.ODCIEnv)
    RETURN NUMBER,
  STATIC FUNCTION ODCIIndexInsert(ia SYS.ODCIIndexInfo, rid VARCHAR2,
    newval VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexDelete(ia SYS.ODCIIndexInfo, rid VARCHAR2,
    oldval VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexUpdate(ia SYS.ODCIIndexInfo, rid VARCHAR2,
    oldval VARCHAR2, newval VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexStart(sctx IN OUT psbtree_im, ia SYS.ODCIIndexInfo,
    op SYS.ODCIPredInfo, qi sys.ODCIQueryInfo, strt number, stop number,
    cmpval VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIIndexFetch(nrows NUMBER, rids OUT SYS.ODCIridlist,
    env SYS.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIIndexClose(env SYS.ODCIEnv) RETURN NUMBER
);
/
SHOW ERRORS
```

2. Define the implementation type body

You can implement the index routines in any language supported by Oracle. For this example, we will implement the get interfaces routine and the index definition routines in PL/SQL. We will implement the index manipulation and query routines in C.

```
CREATE OR REPLACE TYPE BODY psbtree_im
IS
```

The get interfaces routine returns the expected interface name through its `OUT` parameter.

```
   STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
       RETURN NUMBER IS
   BEGIN
       ifclist := sys.ODCIObjectList(sys.ODCIObject('SYS','ODCIINDEX2'));
       RETURN ODCIConst.Success;
   END ODCIGetInterfaces;
```

The `ODCIIndexCreate` routine creates an index storage table with two columns. The first column stores the `VARCHAR2` indexed column value. The second column in the index table stores the `rowid` of the corresponding row in the base table. When the create routine is invoked during creation of a local domain index, it is invoked `n`+2 times, where `n` is the number of partitions in the base table. The routine creates one index storage table for each partition. The create routine is also invoked during execution of `ALTER TABLE ADD PARTITION` if there are local domain indexes defined on the table. In this case, the routine simply creates a new index storage table to correspond to the newly created partition. The routine makes use of the information passed in to determine the context in which it is invoked. `DBMS_SQL` is used to execute the dynamically constructed SQL statement.

```
STATIC FUNCTION ODCIIndexCreate (ia SYS.ODCIIndexInfo, parms VARCHAR2,
    env SYS.ODCIEnv)
    RETURN NUMBER
  IS
   i INTEGER;
   stmt VARCHAR2(1000);
   cnum INTEGER;
   junk INTEGER;
  BEGIN
   -- construct the sql statement
   stmt := '';

   IF ((env.CallProperty IS NULL) and (ia.IndexPartition IS NULL ))  THEN
     stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
             '_sbtree'  ||
             '( f1 , f2, PRIMARY KEY (f1) ) ORGANIZATION INDEX AS SELECT ' ||
             ia.IndexCols(1).ColName || ', ROWID FROM ' ||
             ia.IndexCols(1).TableSchema || '.' || ia.IndexCols(1).TableName;
   END IF;

   IF ((env.CallProperty IS NOT NULL) AND (ia.IndexPartition IS NOT NULL)) THEN
     stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
             '_' || ia.indexpartition || '_sbtree'  ||
             '( f1 , f2, PRIMARY KEY (f1) ) ORGANIZATION INDEX AS SELECT ' ||
             ia.IndexCols(1).ColName || ', ROWID FROM ' ||
             ia.IndexCols(1).TableSchema || '.' ||
             ia.IndexCols(1).TableName || ' PARTITION (' ||
             ia.IndexCols(1).TablePartition || ')';
   END IF;

   IF ((env.CallProperty IS NULL) AND (ia.IndexPartition IS NOT NULL)) THEN
     stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
             '_' || ia.IndexPartition || '_sbtree'  ||
             '( f1 ' || ia.IndexCols(1).ColTypeName ||'(200) , f2 ROWID, ' ||
             ' PRIMARY KEY (f1)) ORGANIZATION INDEX';
   END IF;

   DBMS_OUTPUT.PUT_LINE('Create');
   DBMS_OUTPUT.PUT_LINE(stmt);

   -- execute the statement
   IF ( (env.CallProperty IS NULL) OR
        (env.CallProperty = SYS.ODCIConst.IntermediateCall) ) THEN
     cnum := DBMS_SQL.OPEN_CURSOR;
     DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
     junk := DBMS_SQL.EXECUTE(cnum);
     DBMS_SQL.CLOSE_CURSOR(cnum);
   END IF;

   RETURN ODCIConst.Success;
  END;
```

The `ODCIIndexDrop` routine drops the index storage table(s). For a local domain index, the routine is invoked `n`+2 times, where `n` is the number of partitions in the base table.

```
STATIC FUNCTION ODCIIndexDrop(ia SYS.ODCIIndexInfo, env SYS.ODCIEnv)
  RETURN NUMBER IS
  stmt VARCHAR2(1000);
  cnum INTEGER;
  junk INTEGER;
 BEGIN
   -- construct the sql statement
  stmt := '';
  IF ((env.CallProperty IS NULL) and (ia.IndexPartition IS NULL) ) THEN
    stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
            '_sbtree';
  ELSE
    IF (ia.IndexPartition IS NOT NULL) THEN
      stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
              '_' || ia.IndexPartition || '_sbtree';
    END IF;
  END IF;

  DBMS_OUTPUT.PUT_LINE('Drop');
  DBMS_OUTPUT.PUT_LINE(stmt);

  -- execute the statement
  IF ( (env.CallProperty IS NULL) OR
       (env.CallProperty = SYS.ODCIConst.IntermediateCall) ) THEN
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
    junk := DBMS_SQL.EXECUTE(cnum);
    DBMS_SQL.CLOSE_CURSOR(cnum);
  END IF;

  RETURN ODCIConst.Success;
 END;
```

To handle partition maintenance operations, the indextype also has addiitonal methods that take appropriate actions on the index storage tables when the base table partitions are merged, split, or exchanged.

The `ODCIIndexMergePartition` routine drops the index storage tables for the two index partitions being merged and creates a new table corresponding to the resulting merged partition. If there is data in the resulting merged partition, the index is marked `UNUSABLE` so that the table is not populated with rows. That is left to be done during a subsequent `ALTER INDEX REBUILD PARTITION`.

```
 STATIC FUNCTION ODCIIndexMergePartition(ia SYS.ODCIIndexInfo,
   part_name1 SYS.ODCIPartInfo, part_name2 SYS.ODCIPartInfo,
   parms VARCHAR2, env SYS.ODCIEnv)
   return number
 IS
  stmt VARCHAR2(2000);
  cnum INTEGER;
  junk INTEGER;
 BEGIN
  DBMS_OUTPUT.PUT_LINE('Merge Partitions');
  stmt := '';

  IF (ia.IndexPartition IS NOT NULL) THEN
     stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
              '_' || ia.IndexPartition || '_sbtree';

     DBMS_OUTPUT.PUT_LINE('drop');
     DBMS_OUTPUT.PUT_LINE(stmt);
```

```
    -- execute the statement
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
    junk := DBMS_SQL.EXECUTE(cnum);
    DBMS_SQL.CLOSE_CURSOR(cnum);
 END IF;

 IF ( part_name1 IS NOT NULL) THEN
    stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
             '_' || part_name1.IndexPartition || '_sbtree';

    DBMS_OUTPUT.PUT_LINE('drop');
    DBMS_OUTPUT.PUT_LINE(stmt);

    -- execute the statement
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
    junk := DBMS_SQL.EXECUTE(cnum);
    DBMS_SQL.CLOSE_CURSOR(cnum);
 END IF;

 IF ( part_name2 IS NOT NULL) THEN
    stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
         '_' || part_name2.IndexPartition ||  '_sbtree'  ||
         '( f1 ' || ia.IndexCols(1).ColTypeName ||
         '(200) , f2 ROWID, PRIMARY KEY (f1) ) ORGANIZATION INDEX';

    DBMS_OUTPUT.PUT_LINE('create');
    DBMS_OUTPUT.PUT_LINE('Parameter string : ' || parms);

    DBMS_OUTPUT.PUT_LINE(stmt);

    -- execute the statement
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
    junk := DBMS_SQL.EXECUTE(cnum);
    DBMS_SQL.CLOSE_CURSOR(cnum);
  END IF;

  RETURN ODCIConst.Success;
END;
```

The `ODCIIndexSplitPartition` routine drops the index storage table corresponding to the partition being split and creates two new index tables that correspond to the two new index partitions created.

```
STATIC FUNCTION ODCIIndexSplitPartition(ia SYS.ODCIIndexInfo,
    part_name1 SYS.ODCIPartInfo, part_name2 SYS.ODCIPartInfo,
    parms VARCHAR2, env sys.odcienv)
    return number
  IS
   stmt VARCHAR2(2000);
   cnum INTEGER;
   junk INTEGER;
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Split Partition');
   stmt := '';

   IF (ia.IndexPartition IS NOT NULL) THEN
      stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
               '_' || ia.IndexPartition || '_sbtree';

      DBMS_OUTPUT.PUT_LINE('drop');
      DBMS_OUTPUT.PUT_LINE(stmt);

      -- execute the statement
      cnum := DBMS_SQL.OPEN_CURSOR;
      DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
      junk := DBMS_SQL.EXECUTE(cnum);
      DBMS_SQL.CLOSE_CURSOR(cnum);
    END IF;

   IF ( part_name1 IS NOT NULL) THEN
      stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
           '_' || part_name1.IndexPartition ||  '_sbtree'  ||
           '( f1 ' || ia.IndexCols(1).ColTypeName ||
           '(200) , f2 ROWID, PRIMARY KEY (f1)) ORGANIZATION INDEX';

      DBMS_OUTPUT.PUT_LINE('create');
      DBMS_OUTPUT.PUT_LINE('Parameter string : ' || parms);
      DBMS_OUTPUT.PUT_LINE(stmt);

      -- execute the statement
      cnum := DBMS_SQL.OPEN_CURSOR;
      DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
      junk := DBMS_SQL.EXECUTE(cnum);
      DBMS_SQL.CLOSE_CURSOR(cnum);
    END IF;

   IF ( part_name2 IS NOT NULL) THEN
      stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName ||
           '_' || part_name2.IndexPartition ||  '_sbtree'  ||
           '( f1 ' || ia.IndexCols(1).ColTypeName ||
           '(200) , f2 ROWID, PRIMARY KEY (f1)) ORGANIZATION INDEX';

      DBMS_OUTPUT.PUT_LINE('create');
      DBMS_OUTPUT.PUT_LINE('Parameter string : ' || parms);
      DBMS_OUTPUT.PUT_LINE(stmt);

      -- execute the statement
      cnum := DBMS_SQL.OPEN_CURSOR;
      DBMS_SQL.PARSE(cnum, stmt, dbms_sql.native);
      junk := DBMS_SQL.EXECUTE(cnum);
      DBMS_SQL.CLOSE_CURSOR(cnum);
    END IF;

    RETURN ODCIConst.Success;
  END;
```

The `ODCIIndexExchangePartition` exchanges the index storage tables for the index partition being exchanged, with the index storage table for the global domain index.

```
STATIC FUNCTION ODCIIndexExchangePartition(ia SYS.ODCIIndexInfo,
    ia1 SYS.ODCIIndexInfo, env SYS.ODCIEnv)
   RETURN NUMBER
  IS
   stmt VARCHAR2(2000);
   cnum INTEGER;
   junk INTEGER;
  BEGIN
   stmt := '';
   DBMS_OUTPUT.PUT_LINE('Exchange Partitions');

    -- construct the sql statement
   stmt := 'ALTER TABLE temp EXCHANGE PARTITION p1 WITH TABLE ' ||
        ia1.IndexSchema || '.' || ia1.IndexName || '_sbtree';
   cnum := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
   junk := DBMS_SQL.EXECUTE(cnum);
   DBMS_SQL.CLOSE_CURSOR(cnum);
   stmt := 'ALTER TABLE temp EXCHANGE PARTITION p1 WITH TABLE ' ||
        ia.IndexSchema || '.' || ia.IndexName ||
        '_' || ia.IndexPartition || '_sbtree';

   cnum := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
   junk := DBMS_SQL.EXECUTE(cnum);
   DBMS_SQL.CLOSE_CURSOR(cnum);

   stmt := 'ALTER TABLE temp EXCHANGE PARTITION p1 WITH TABLE ' ||
        ia1.IndexSchema || '.' || ia1.IndexName || '_sbtree';
   cnum := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(cnum, stmt, DBMS_SQL.NATIVE);
   junk := DBMS_SQL.EXECUTE(cnum);
   DBMS_SQL.CLOSE_CURSOR(cnum);

   RETURN ODCIConst.Success;
  END;
```

The index manipulation and query routines are implemented in C. These require some setup to be done in advance. Specifically, you need to create a library object called `extdemo5l` for your compiled C code.

After the setup, the following statements register the implementation of the index manipulation and query routines in terms of their corresponding C functions.

Register the implementation of the `ODCIIndexInsert` routine.

```
STATIC FUNCTION ODCIIndexInsert(ia SYS.ODCIIndexInfo, rid VARCHAR2,
                              newval VARCHAR2, env SYS.ODCIEnv)
                              RETURN NUMBER AS EXTERNAL
name "qxiqtbi"
library extdemo5l
with context
parameters (
context,
ia,
ia indicator struct,
rid,
rid indicator,
newval,
newval indicator,
env,
env indicator struct,
return OCINumber
           );
```

Register the implementation of the `ODCIIndexDelete` routine.

```
STATIC FUNCTION ODCIIndexDelete(ia SYS.ODCIIndexInfo, rid VARCHAR2,
                                oldval VARCHAR2, env SYS.ODCIEnv)
                                RETURN NUMBER AS EXTERNAL
name "qxiqtbd"
library extdemo5l
with context
parameters (
context,
ia,
ia indicator struct,
rid,
rid indicator,
oldval,
oldval indicator,
env,
env indicator struct,
return OCINumber
           );
```

Register the implementation of the `ODCIIndexUpdate` routine.

```
STATIC FUNCTION ODCIIndexUpdate(ia SYS.ODCIIndexInfo, rid VARCHAR2,
                          oldval VARCHAR2, newval VARCHAR2, env SYS.ODCIEnv)
                                RETURN NUMBER AS EXTERNAL
name "qxiqtbu"
library extdemo5l
with context
parameters (
context,
ia,
ia indicator struct,
rid,
rid indicator,
oldval,
oldval indicator,
newval,
newval indicator,
env,
env indicator struct,
return OCINumber
           );
```

Register the implementation of the `ODCIIndexStart` routine.

```
STATIC FUNCTION ODCIIndexStart(sctx IN OUT psbtree_im, ia SYS.ODCIIndexInfo,
                         op SYS.ODCIPredInfo,
                         qi SYS.ODCIQueryInfo,
                         strt NUMBER,
                         stop NUMBER,
                         cmpval VARCHAR2,
                         env SYS.ODCIEnv)
   RETURN NUMBER AS EXTERNAL
   name "qxiqtbs"
   library extdemo5l
   with context
   parameters (
     context,
     sctx,
     sctx indicator struct,
     ia,
     ia indicator struct,
     op,
     op indicator struct,
     qi,
     qi indicator struct,
     strt,
     strt indicator,
     stop,
     stop indicator,
     cmpval,
     cmpval indicator,
     env,
     env indicator struct,
     return OCINumber
  );
```

Register the implementation of the `ODCIIndexFetch` routine.

```
MEMBER FUNCTION ODCIIndexFetch(nrows NUMBER, rids OUT SYS.ODCIRidList,
     env SYS.ODCIEnv)
 RETURN NUMBER AS EXTERNAL
 name "qxiqtbf"
 library extdemo5l
 with context
 parameters (
   context,
   self,
   self indicator struct,
   nrows,
   nrows indicator,
   rids,
   rids indicator,
   env,
   env indicator struct,
   return OCINumber
 );
```

Register the implementation of the ODCIIndexClose routine.

```
MEMBER FUNCTION ODCIIndexClose (env SYS.ODCIEnv) RETURN NUMBER AS EXTERNAL
name "qxiqtbc"
library extdemo5l
with context
parameters (
  context,
  self,
  self indicator struct,
  env,
  env indicator struct,
  return OCINumber
);
```

The C Code

General Notes

The C structs for mapping the ODCI types are defined in the file `odci.h`. For example, the C struct `ODCIIndexInfo` is the mapping for the corresponding ODCI object type. The C struct `ODCIIndexInfo_ind` is the mapping for the null object.

Common Error Processing Routine

This function is used to check and process the return code from all `OCI` routines. It checks the status code and raises an exception in case of errors.

```
static int qxiqtce(ctx, errhp, status)
OCIExtProcContext *ctx;
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;
  int errnum = 29400;  /* choose some oracle error number */
  int rc = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    rc = 0;
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4)1, (text *)NULL, &errcode,
                       errbuf, (ub4)sizeof(errbuf), OCI_HTYPE_ERROR);
    /* Raise exception */
    OCIExtProcRaiseExcpWithMsg(ctx, errnum, errbuf, strlen((char *)errbuf));
    rc = 1;
    break;
  default:
    (void) sprintf((char *)errbuf, "Warning - some error\n");
    /* Raise exception */
    OCIExtProcRaiseExcpWithMsg(ctx, errnum, errbuf, strlen((char *)errbuf));
    rc = 1;
    break;
  }
  return (rc);
}
```

Implementation Of The ODCIIndexInsert Routine

The insert routine parses and executes a statement that inserts a new row into the index table. The new row consists of the new value of the indexed column and the `rowid` that have been passed in as parameters.

```
OCINumber *qxiqtbi(ctx, ix, ix_ind, rid, rid_ind,
                   newval, newval_ind, env, env_ind)
OCIExtProcContext *ctx;
ODCIIndexInfo     *ix;
ODCIIndexInfo_ind *ix_ind;
char              *rid;
short             rid_ind;
char              *newval;
short             newval_ind;
ODCIEnv           *env;
ODCIEnv_ind       *env_ind;
{
```

```
OCIEnv *envhp = (OCIEnv *) 0;               /* env. handle */
OCISvcCtx *svchp = (OCISvcCtx *) 0;         /* service handle */
OCIError *errhp = (OCIError *) 0;           /* error handle */
OCIStmt *stmthp = (OCIStmt *) 0;            /* statement handle */
OCIBind *bndp = (OCIBind *) 0;              /* bind handle */
OCIBind *bndp1 = (OCIBind *) 0;             /* bind handle */

int retval = (int)ODCI_SUCCESS;             /* return from this function */
OCINumber *rval = (OCINumber *)0;
ub4 key;                                    /* key value set in "self" */

char insstmt[2000];                         /* sql insert statement */

/* allocate memory for OCINumber first */
rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));

/* Get oci handles */
if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
  return(rval);

/* set up return code */
if (qxiqtce(ctx, errhp, OCINumberFromInt(errhp, (dvoid *)&retval,
                                        sizeof(retval),
                                        OCI_NUMBER_SIGNED, rval)))
  return(rval);

/******************************
 * Construct insert Statement *
 ******************************/
 if (ix_ind->IndexPartition == OCI_IND_NULL)
 {
   sprintf(insstmt,
                "INSERT into %s.%s_sbtree values (:newval, :mrid)",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName));
 }
 else
 {
   sprintf(insstmt,
                "INSERT into %s.%s_%s_sbtree values (:newval, :mrid)",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName),
                OCIStringPtr(envhp, ix->IndexPartition));

    }
```

```
/****************************************
 * Parse and Execute Insert Statement   *
 ****************************************/

/* allocate stmt handle */
if (qxiqtce(ctx, errhp, OCIHandleAlloc((dvoid *)envhp,
                                       (dvoid **)&stmthp,
                                       (ub4)OCI_HTYPE_STMT, (size_t)0,
                                       (dvoid **)0)))
  return(rval);

/* prepare the statement */
if (qxiqtce(ctx, errhp, OCIStmtPrepare(stmthp, errhp, (text *)insstmt,
                                       (ub4)strlen(insstmt), OCI_NTV_SYNTAX,
                                       OCI_DEFAULT)))
  return(rval);



/* Set up bind for newval */
if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)1,
                                     (dvoid *)newval,
                                     (sb4)(strlen(newval)+1),
                                     (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                     (ub2 *)0, (ub4)0, (ub4 *)0,
                                     (ub4)OCI_DEFAULT)))
  return(rval);

/* Set up bind for rid */
if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)2,
                                     (dvoid *)rid,
                                     (sb4)(strlen(rid)+1),
                                     (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                     (ub2 *)0, (ub4)0, (ub4 *)0,
                                     (ub4)OCI_DEFAULT)))
  return(rval);


/* Execute statement */
if (qxiqtce(ctx, errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4)1,
                                       (ub4)0, (OCISnapshot *)NULL,
                                       (OCISnapshot *)NULL,
                                       (ub4)OCI_DEFAULT)))
  return(rval);
  return(rval);
}
```

Implementation of the ODCIIndexDelete Routine

The delete routine constructs a SQL statement to delete a row from the index table corresponding to the row being deleted from the base table. The row in the index table is identified by the value of `rowid` that is passed in as a parameter to this routine.

```
OCINumber *qxiqtbd(ctx, ix, ix_ind, rid, rid_ind,
                   oldval, oldval_ind, env, env_ind)
OCIExtProcContext *ctx;
ODCIIndexInfo     *ix;
ODCIIndexInfo_ind *ix_ind;
char              *rid;
short             rid_ind;
char              *oldval;
short             oldval_ind;
ODCIEnv           *env;
ODCIEnv_ind       *env_ind;
{
  OCIEnv *envhp = (OCIEnv *) 0;               /* env. handle */
  OCISvcCtx *svchp = (OCISvcCtx *) 0;         /* service handle */
  OCIError *errhp = (OCIError *) 0;           /* error handle */
  OCIStmt *stmthp = (OCIStmt *) 0;            /* statement handle */
  OCIBind *bndp = (OCIBind *) 0;              /* bind handle */
  OCIBind *bndp1 = (OCIBind *) 0;             /* bind handle */

  int retval = (int)ODCI_SUCCESS;             /* return from this function */
  OCINumber *rval = (OCINumber *)0;
  ub4 key;                                    /* key value set in "self" */

  char delstmt[2000];                         /* sql insert statement */

  /* Get oci handles */
  if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
    return(rval);

  /* set up return code */
  rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
  if (qxiqtce(ctx, errhp, OCINumberFromInt(errhp, (dvoid *)&retval,
                                          sizeof(retval),
                                          OCI_NUMBER_SIGNED, rval)))
    return(rval);

  /*****************************
   * Construct delete Statement *
   ******************************/
   if (ix_ind->IndexPartition == OCI_IND_NULL)
   {
     sprintf(delstmt,
                  "DELETE FROM %s.%s_sbtree WHERE f2 = :rr",
                  OCIStringPtr(envhp, ix->IndexSchema),
                  OCIStringPtr(envhp, ix->IndexName));
   }
   else
   {
     sprintf(delstmt,
                  "DELETE FROM %s.%s_%s_sbtree WHERE f2 = :rr",
                  OCIStringPtr(envhp, ix->IndexSchema),
                  OCIStringPtr(envhp, ix->IndexName),
                  OCIStringPtr(envhp, ix->IndexPartition));

   }

  /****************************************
   * Parse and Execute delete Statement   *
   ****************************************/

  /* allocate stmt handle */
  if (qxiqtce(ctx, errhp, OCIHandleAlloc((dvoid *)envhp,
                                         (dvoid **)&stmthp,
                                         (ub4)OCI_HTYPE_STMT, (size_t)0,
                                         (dvoid **)0)))
    return(rval);

  /* prepare the statement */
  if (qxiqtce(ctx, errhp, OCIStmtPrepare(stmthp, errhp, (text *)delstmt,
                                         (ub4)strlen(delstmt), OCI_NTV_SYNTAX,
                                         OCI_DEFAULT)))
    return(rval);



  /* Set up bind for rid */
  if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)1,
                                       (dvoid *)rid,
                                       (sb4)(strlen(rid)+1),
                                         (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                         (ub2 *)0, (ub4)0, (ub4 *)0,
                                         (ub4)OCI_DEFAULT)))
    return(rval);


  /* Execute statement */
  if (qxiqtce(ctx, errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4)1,
                                          (ub4)0, (OCISnapshot *)NULL,
                                          (OCISnapshot *)NULL,
                                          (ub4)OCI_DEFAULT)))
    return(rval);

  return(rval);
}
```

Implementation of the ODCIIndexUpdate Routine

The update routine constructs a SQL statement to update a row in the index table corresponding to the row being updated in the base table. The row in the index table is identified by the value of `rowid` that is passed in as a parameter to this routine. The old column value (`oldval`) is replaced by the new value (`newval`).

```
OCINumber *qxiqtbu(ctx, ix, ix_ind, rid, rid_ind,
                   oldval, oldval_ind, newval, newval_ind, env, env_ind)
OCIExtProcContext *ctx;
ODCIIndexInfo     *ix;
ODCIIndexInfo_ind *ix_ind;
char              *rid;
short             rid_ind;
char              *oldval;
short             oldval_ind;
char              *newval;
short             newval_ind;
ODCIEnv           *env;
ODCIEnv_ind       *env_ind;
{
  OCIEnv *envhp = (OCIEnv *) 0;              /* env. handle */
  OCISvcCtx *svchp = (OCISvcCtx *) 0;        /* service handle */
  OCIError *errhp = (OCIError *) 0;          /* error handle */
  OCIStmt *stmthp = (OCIStmt *) 0;           /* statement handle */
  OCIBind *bndp = (OCIBind *) 0;             /* bind handle */
  OCIBind *bndp1 = (OCIBind *) 0;            /* bind handle */

  int retval = (int)ODCI_SUCCESS;            /* return from this function */
```

```
OCINumber *rval = (OCINumber *)0;
ub4 key;                                      /* key value set in "self" */

char updstmt[2000];                           /* sql insert statement */

/* Get oci handles */
if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
  return(rval);

/* set up return code */
rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
if (qxiqtce(ctx, errhp, OCINumberFromInt(errhp, (dvoid *)&retval,
                                          sizeof(retval),
                                          OCI_NUMBER_SIGNED, rval)))
  return(rval);

/*****************************
 * Construct update Statement *
 ******************************/
 if (ix_ind->IndexPartition == OCI_IND_NULL)
 {
   sprintf(updstmt,
        "UPDATE %s.%s_sbtree SET f1 = :newval, f2 = :rr WHERE f1 = :oldval",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName));
 }
 else
 {
   sprintf(updstmt,
     "UPDATE %s.%s_%s_sbtree SET f1 = :newval, f2 = :rr WHERE f1 = :oldval",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName),
                OCIStringPtr(envhp, ix->IndexPartition));

 }

/****************************************
 * Parse and Execute Update Statement   *
 ****************************************/

/* allocate stmt handle */
if (qxiqtce(ctx, errhp, OCIHandleAlloc((dvoid *)envhp,
                                        (dvoid **)&stmthp,
                                        (ub4)OCI_HTYPE_STMT, (size_t)0,
                                        (dvoid **)0)))
```

```
      return(rval);

    /* prepare the statement */
    if (qxiqtce(ctx, errhp, OCIStmtPrepare(stmthp, errhp, (text *)updstmt,
                                           (ub4)strlen(updstmt), OCI_NTV_SYNTAX,
                                           OCI_DEFAULT)))
      return(rval);


    /* Set up bind for newval */
    if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)1,
                                         (dvoid *)newval,
                                         (sb4)(strlen(newval)+1),
                                         (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                         (ub2 *)0, (ub4)0, (ub4 *)0,
                                         (ub4)OCI_DEFAULT)))
      return(rval);

    /* Set up bind for rid */
    if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)2,
                                         (dvoid *)rid,
                                         (sb4)(strlen(rid)+1),
                                         (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                         (ub2 *)0, (ub4)0, (ub4 *)0,
                                         (ub4)OCI_DEFAULT)))
      return(rval);

    /* Set up bind for oldval */
    if (qxiqtce(ctx, errhp, OCIBindByPos(stmthp, &bndp, errhp, (ub4)3,
                                         (dvoid *)oldval,
                                         (sb4)(strlen(oldval)+1),
                                         (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                         (ub2 *)0, (ub4)0, (ub4 *)0,
                                         (ub4)OCI_DEFAULT)))
      return(rval);

    /* Execute statement */
    if (qxiqtce(ctx, errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4)1,
                                           (ub4)0, (OCISnapshot *)NULL,
                                           (OCISnapshot *)NULL,
                                           (ub4)OCI_DEFAULT)))
      return(rval);

    return(rval);
  }
```

Implementation of the ODCIIndexStart Routine

The start routine performs the setup for an `sbtree` index scan. The query information in terms of the operator predicate, its arguments, and the bounds on return values are passed in as parameters to this function. The scan context that is shared among the index scan routines is an instance of the type `psbtree_im`. We have defined a C struct (`qxiqtim`) as a mapping for the object type. In addition, there is a C struct (`qxiqtin`) for the corresponding null object. Note that the C structs for the object type and its null object can be generated by using the Object Type Translator (OTT).

```
/* The index implementation type is an object type with a single RAW attribute
 * which will be used to store the context key value.
 * C mapping of the implementation type :
 */
struct qxiqtim
{
   OCIRaw *sctx_qxiqtim;
};
typedef struct qxiqtim qxiqtim;

struct qxiqtin
{
  short atomic_qxiqtin;
  short scind_qxiqtin;
};
typedef struct qxiqtin qxiqtin;
```

This function sets up a cursor that scans the index table. The scan retrieves the stored rowids for the rows in the index table that satisfy the specified predicate. The predicate for the index table is generated based on the operator predicate information that is passed in as parameters. For example, if the operator predicate is of the form:

```
eq(col, 'joe') = 1
```

the predicate on the index table is set up to be

```
f1 = 'joe'
```

There are a set of OCI handles that need to be cached away and retrieved on the next fetch call. A C struct `qxiqtcx` is defined to hold all the necessary scan state. This structure is allocated out of `OCI_DURATION_STATEMENT` memory to ensure that it persists till the end of `fetch`. After populating the structure with the required info, a pointer to the structure is saved in OCI context. The context is identified by a 4-byte key that is generated by calling an OCI routine. The 4-byte key is stashed away in the scan context - `exiting`. This object is returned back to the Oracle server and is passed in as a parameter to the next fetch call.

```
/* The index scan context - should be stored in "statement" duration memory
 * and used by start, fetch and close routines.
 */
struct qxiqtcx
{
  OCIStmt *stmthp;
  OCIDefine *defnp;
  OCIBind *bndp;
  char ridp[19];
};

typedef struct qxiqtcx qxiqtcx;

OCINumber *qxiqtbs(ctx, sctx, sctx_ind, ix, ix_ind, pr, pr_ind, qy, qy_ind,
                   strt, strt_ind, stop, stop_ind, cmpval, cmpval_ind,
                   env, env_ind)
OCIExtProcContext *ctx;
qxiqtim           *sctx;
qxiqtin           *sctx_ind;
ODCIIndexInfo     *ix;
ODCIIndexInfo_ind *ix_ind;
ODCIPredInfo      *pr;
dvoid             *pr_ind;
ODCIQueryInfo     *qy;
dvoid             *qy_ind;
OCINumber         *strt;
short             strt_ind;
OCINumber         *stop;
short             stop_ind;
char              *cmpval;
short             cmpval_ind;
ODCIEnv           *env;
dvoid             *env_ind;
{
  sword status;
  OCIEnv *envhp;                                                /* env. handle */
  OCISvcCtx *svchp;                                         /* service handle */
  OCIError *errhp;                                            /* error handle */
  OCISession *usrhp;                                           /* user handle */
  qxiqtcx *icx;                                /* state to be saved for later calls */
```

```
int strtval;                                                /* start bound */
int stopval;                                                 /* stop bound */

int errnum = 29400;                     /* choose some oracle error number */
char errmsg[512];                                  /* error message buffer */
size_t errmsglen;                               /* Length of error message */

char relop[3];                     /* relational operator used in sql stmt */
char selstmt[2000];                                /* sql select statement */

int retval = (int)ODCI_SUCCESS;                /* return from this function */
OCINumber *rval = (OCINumber *)0;
ub4 key;                                        /* key value set in "sctx" */

/* Get oci handles */
if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
  return(rval);

/* set up return code */
rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
if (qxiqtce(ctx, errhp, OCINumberFromInt(errhp, (dvoid *)&retval,
                                         sizeof(retval),
                                         OCI_NUMBER_SIGNED, rval)))
  return(rval);

/* get the user handle */
if (qxiqtce(ctx, errhp, OCIAttrGet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                                   (dvoid *)&usrhp, (ub4 *)0,
                                   (ub4)OCI_ATTR_SESSION,
                                   errhp)))
  return(rval);


/**********************************************/
/* Allocate memory to hold index scan context */
/**********************************************/
if (qxiqtce(ctx, errhp, OCIMemoryAlloc((dvoid *)usrhp, errhp,
                                       (dvoid **)&icx,
                                       OCI_DURATION_STATEMENT,
                                       (ub4)(sizeof(qxiqtcx)),
                                       OCI_MEMORY_CLEARED)))
  return(rval);

icx->stmthp = (OCIStmt *)0;
icx->defnp = (OCIDefine *)0;
```

```
icx->bndp = (OCIBind *)0;

/**********************************/
/* Check that the bounds are valid */
/**********************************/
/* convert from oci numbers to native numbers */
if (qxiqtce(ctx, errhp, OCINumberToInt(errhp, strt,
                                       sizeof(strtval), OCI_NUMBER_SIGNED,
                                       (dvoid *)&strtval)))
  return(rval);
if (qxiqtce(ctx, errhp, OCINumberToInt(errhp, stop,
                                      sizeof(stopval),
                                      OCI_NUMBER_SIGNED, (dvoid *)&stopval)))
  return(rval);

/* verify that strtval/stopval are both either 0 or 1 */
if (!(((strtval == 0) && (stopval == 0)) ||
      ((strtval == 1) && (stopval == 1))))
  {
    strcpy(errmsg, "Incorrect predicate for sbtree operator");
    errmsglen = (size_t)strlen(errmsg);
    if (OCIExtProcRaiseExcpWithMsg(ctx, errnum, (text *)errmsg, errmsglen)
        != OCIEXTPROC_SUCCESS)
      /* Use cartridge error services here */;
    return(rval);
  }

/********************************************/
/* Generate the SQL statement to be executed */
/********************************************/
if (memcmp((dvoid *)OCIStringPtr(envhp, pr->ObjectName), "EQ", 2)
    == 0)
  if (strtval == 1)
    strcpy(relop, "=");
  else
    strcpy(relop, "!=");
else if (memcmp((dvoid *)OCIStringPtr(envhp, pr->ObjectName), "LT",
                2) == 0)
  if (strtval == 1)
    strcpy(relop, "<");
  else
    strcpy(relop, ">=");
else
  if (strtval == 1)
    strcpy(relop, ">");
  else
    strcpy(relop, "<=");

if (ix_ind->IndexPartition == OCI_IND_NULL)
{
  sprintf(selstmt, "select f2 from %s.%s_sbtree where f1 %s :val",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName), relop);
}
else
{
  sprintf(selstmt, "select f2 from %s.%s_%s_sbtree where f1 %s :val",
                OCIStringPtr(envhp, ix->IndexSchema),
                OCIStringPtr(envhp, ix->IndexName),
                OCIStringPtr(envhp, ix->IndexPartition), relop);
}

/***********************************/
/* Parse, bind, define and execute */
/***********************************/
/* allocate stmt handle */
if (qxiqtce(ctx, errhp,
            OCIHandleAlloc((dvoid *)envhp, (dvoid **)&(icx->stmthp),
                           (ub4)OCI_HTYPE_STMT, (size_t)0,
                           (dvoid **)0)))
  return(rval);
/* prepare the statement */
if (qxiqtce(ctx, errhp, OCIStmtPrepare(icx->stmthp, errhp, (text *)selstmt,
                                       (ub4)strlen(selstmt), OCI_NTV_SYNTAX,
                                       OCI_DEFAULT)))
  return(rval);

/* Set up bind */
if (qxiqtce(ctx, errhp,
            OCIBindByPos(icx->stmthp, &(icx->bndp), errhp, (ub4)1,
                          (dvoid *)cmpval,
                          (sb4)(strlen(cmpval)+1),
                          (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                          (ub2 *)0, (ub4)0, (ub4 *)0,
                          (ub4)OCI_DEFAULT)))
  return(rval);

/* Set up define */
if (qxiqtce(ctx, errhp, OCIDefineByPos(icx->stmthp, &(icx->defnp), errhp,
                                       (ub4)1, (dvoid *)(icx->ridp),
                                         (sb4) sizeof(icx->ridp),
                                         (ub2)SQLT_STR, (dvoid *)0, (ub2 *)0,
                                         (ub2 *)0, (ub4)OCI_DEFAULT)))
    return(rval);

  /* execute */
  if (qxiqtce(ctx, errhp, OCIStmtExecute(svchp, icx->stmthp, errhp, (ub4)0,
                                         (ub4)0, (OCISnapshot *)NULL,
                                         (OCISnapshot *)NULL,
                                         (ub4)OCI_DEFAULT)))
    return(rval);

  /***********************************/
  /* Set index context to be returned */
  /***********************************/
  /* generate a key */
  if (qxiqtce(ctx, errhp, OCIContextGenerateKey((dvoid *)usrhp, errhp, &key)))
    return(rval);
  /* set the memory address of the struct to be saved in the context */
  if (qxiqtce(ctx, errhp, OCIContextSetValue((dvoid *)usrhp, errhp,
                                             OCI_DURATION_STATEMENT,
                                             (ub1 *)&key, (ub1)sizeof(key),
                                             (dvoid *)icx)))
    return(rval);
  /* set the key as the member of "sctx" */
  if (qxiqtce(ctx, errhp, OCIRawAssignBytes(envhp, errhp, (ub1 *)&key,
                                            (ub4)sizeof(key),
                                            &(sctx->sctx_qxiqtim))))
    return(rval);

  sctx_ind->atomic_qxiqtin = OCI_IND_NOTNULL;
  sctx_ind->scind_qxiqtin = OCI_IND_NOTNULL;

  return(rval);
}
```

Implementation of the ODCIIndexFetch Routine

The scan context set up by the start routine is passed in as a parameter to the fetch routine. This function first retrieves the 4-byte key from the scan context. The C mapping for the scan context is `qxiqtim`. Next, the OCI context is looked up based on the key. This gives the memory address of the structure that holds the OCI handles - the `qxiqtcx` structure.

This function returns the next batch of rowids that satisfy the operator predicate. It uses the value of the `nrows` parameter as the size of the batch. It repeatedly fetches rowids from the open cursor and populates the `rowid` list with them. When the batch is full or when there are no more rowids left, the function returns them back to the Oracle server.

```
OCINumber *qxiqtbf(ctx, self, self_ind, nrows, nrows_ind, rids, rids_ind,
                   env, env_ind)
OCIExtProcContext *ctx;
qxiqtim           *self;
qxiqtin           *self_ind;
OCINumber         *nrows;
short             nrows_ind;
OCIArray          **rids;
short             *rids_ind;
ODCIEnv           *env;
dvoid             *env_ind;
{
  sword status;
  OCIEnv *envhp;
  OCISvcCtx *svchp;
  OCIError *errhp;
  OCISession *usrhp;                                        /* user handle */
  qxiqtcx *icx;

  int idx = 1;
  int nrowsval;

  OCIArray *ridarrp = *rids;                           /* rowid collection */
  OCIString *ridstr = (OCIString *)0;

  int done = 0;
  int retval = (int)ODCI_SUCCESS;
  OCINumber *rval = (OCINumber *)0;

  ub1 *key;                                     /* key to retrieve context */
  ub4 keylen;                                             /* length of key */

  /*******************/
  /* Get OCI handles */
  /*******************/
  if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
      return(rval);

  /* set up return code */
```

```
rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
if (qxiqtce(ctx, errhp,
            OCINumberFromInt(errhp, (dvoid *)&retval, sizeof(retval),
                                             OCI_NUMBER_SIGNED, rval)))
  return(rval);

/* get the user handle */
if (qxiqtce(ctx, errhp, OCIAttrGet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                                   (dvoid *)&usrhp, (ub4 *)0,
                                   (ub4)OCI_ATTR_SESSION, errhp)))
  return(rval);

/*********************************/
/* Retrieve context from key    */
/*********************************/
key = OCIRawPtr(envhp, self->sctx_qxiqtim);
keylen = OCIRawSize(envhp, self->sctx_qxiqtim);

if (qxiqtce(ctx, errhp, OCIContextGetValue((dvoid *)usrhp, errhp,
                                           key, (ub1)keylen,
                                           (dvoid **)&(icx))))
  return(rval);

/* get value of nrows */
if (qxiqtce(ctx, errhp, OCINumberToInt(errhp, nrows, sizeof(nrowsval),
                                     OCI_NUMBER_SIGNED, (dvoid *)&nrowsval)))
  return(rval);

/****************/
/* Fetch rowids */
/****************/
while (!done)
{
  if (idx > nrowsval)
    done = 1;
  else
  {
    status = OCIStmtFetch(icx->stmthp, errhp, (ub4)1, (ub2) 0,
                          (ub4)OCI_DEFAULT);
    if (status == OCI_NO_DATA)
    {
      short col_ind = OCI_IND_NULL;
      /* have to create dummy oci string */
      OCIStringAssignText(envhp, errhp, (text *)"dummy",
                          (ub2)5, &ridstr);
```

```
          /* append null element to collection */
          if (qxiqtce(ctx, errhp, OCICollAppend(envhp, errhp,(dvoid *)ridstr,
                                              (dvoid *)&col_ind,
                                              (OCIColl *)ridarrp)))
            return(rval);
          done = 1;
        }
        else if (status == OCI_SUCCESS)
        {
          OCIStringAssignText(envhp, errhp, (text *)icx->ridp,
                              (ub2)18, (OCIString **)&ridstr);
          /* append rowid to collection */
          if (qxiqtce(ctx, errhp, OCICollAppend(envhp, errhp, (dvoid *)ridstr,
                                              (dvoid *)0, (OCIColl *)ridarrp)))
            return(rval);
          idx++;
        }
        else if (qxiqtce(ctx, errhp, status))
          return(rval);
      }
    }

    /* free ridstr finally */
    if (ridstr &&
        (qxiqtce(ctx, errhp, OCIStringResize(envhp, errhp, (ub4)0,
                                           &ridstr))))
      return(rval);

    *rids_ind = OCI_IND_NOTNULL;

    return(rval);
  }
```

Implementation of the ODCIIndexClose Routine

The scan context set up by the start routine is passed in as a parameter to the close routine. This function first retrieves the 4-byte key from the scan context. The C mapping for the scan context is `qxiqtim`. Next, the OCI context is looked up based on the key. This gives the memory address of the structure that holds the OCI handles - the `qxiqtcx` structure.

The function closes and frees all the OCI handles. It also frees the memory that was allocated in the start routine.

```
OCINumber *qxiqtbc(ctx, self, self_ind, env, env_ind)
OCIExtProcContext *ctx;
qxiqtim           *self;
qxiqtin           *self_ind;
ODCIEnv           *env;
dvoid             *env_ind;
{
  sword status;
  OCIEnv *envhp;
  OCISvcCtx *svchp;
  OCIError *errhp;
  OCISession *usrhp;                                         /* user handle */

  qxiqtcx *icx;

  int retval = (int) ODCI_SUCCESS;
  OCINumber *rval = (OCINumber *)0;

  ub1 *key;                                      /* key to retrieve context */
  ub4 keylen;                                              /* length of key */

  if (qxiqtce(ctx, errhp, OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp)))
      return(rval);

  /* set up return code */
  rval = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
  if (qxiqtce(ctx, errhp, OCINumberFromInt(errhp, (dvoid *)&retval,
                                          sizeof(retval),
                                          OCI_NUMBER_SIGNED, rval)))
    return(rval);

  /* get the user handle */
  if (qxiqtce(ctx, errhp, OCIAttrGet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                                    (dvoid *)&usrhp, (ub4 *)0,
                                    (ub4)OCI_ATTR_SESSION, errhp)))
    return(rval);

  /*******************************/
  /* Retrieve context using key   */
  /*******************************/
  key = OCIRawPtr(envhp, self->sctx_qxiqtim);
  keylen = OCIRawSize(envhp, self->sctx_qxiqtim);

  if (qxiqtce(ctx, errhp, OCIContextGetValue((dvoid *)usrhp, errhp,
                                            key, (ub1)keylen,
                                                        (dvoid **)&(icx))))
    return(rval);

  /* Free handles and memory */
  if (qxiqtce(ctx, errhp, OCIHandleFree((dvoid *)icx->stmthp,
                                        (ub4)OCI_HTYPE_STMT)))
    return(rval);

  if (qxiqtce(ctx, errhp, OCIMemoryFree((dvoid *)usrhp, errhp, (dvoid *)icx)))
    return(rval);

  return(rval);
}
```

Implementing the Indextype

Create the indextype object and specify the list of operators that it supports. In addition, specify the name of the implementation type that implements the ODCIIndex interface routines.

```
CREATE INDEXTYPE psbtree
FOR
eq(VARCHAR2, VARCHAR2),
lt(VARCHAR2, VARCHAR2),
gt(VARCHAR2, VARCHAR2)
USING psbtree_im
WITH LOCAL RANGE PARTITION
```

Usage examples

One typical usage scenario is described in the following example. Create a range partitioned table and populate it.

```
CREATE TABLE t1 (f1 NUMBER, f2 VARCHAR2(200))
PARTITION BY RANGE(f1)
(
  PARTITION p1 VALUES LESS THAN (101),
  PARTITION p2 VALUES LESS THAN (201),
  PARTITION p3 VALUES LESS THAN (301),
  PARTITION p4 VALUES LESS THAN (401)
 );
INSERT INTO t1 VALUES (10, 'aaaa');
INSERT INTO t1 VALUES (200, 'bbbb');
INSERT INTO t1 VALUES (100, 'cccc');
INSERT INTO t1 VALUES (300, 'dddd');
INSERT INTO t1 VALUES (400, 'eeee');
COMMIT;
```

Create a `psbtree` index on column f2. The create index statement specifies the indextype to be used.

```
CREATE INDEX it1 ON t1(f2) iINDEXTYPE IS psbtree LOCAL
(PARTITION pe1 PARAMETERS('test1'), PARTITION pe2,
 PARTITION pe3, PARTITION pe4 PARAMETERS('test4'))
PARAMETERS('test');
```

Execute a query that uses one of the `sbtree` operators. •

```
SELECT * FROMM t1 WHERE eq(f2, 'dddd') = 1 AND f1>101 ;
```

Explain Plan Output

```
OPERATION            OPTIONS                    PARTITION_START
PARTITION_STOP
---------------------------------------------------------------

SELECTSTATEMENT

PARTITION RANGE      ITERATOR                             2
4

TABLE ACCESS                   BY LOCAL INDEX ROWID        2
4

DOMAIN INDEX
```

Part V

Reference

This part contains chapters of reference information on cartridge-related APIs:

For information on cartridge services using C, see the chapter on cartridge services in the *Oracle Call Interface Programmer's Guide*.

15

Reference: Cartridge Services Using Java

This reference chapter describes a Java language cartridge service. For more complete details on Java functionality, refer to the *Oracle9i Supplied Java Packages Reference*, and the *Oracle9i Java Stored Procedures Developer's Guide*.

File Installation

The `ODCI.jar` and `CartridgeServices.jar` files must be installed into the `SYS` schema in order to use the Java classes described in this chapter.

If you installed the Java option, then you must install the `ODCI.jar` and `CartridgeServices.jar` files. You do not need to perform this task if you did not install the Java option.

To install `ODCI.jar` and `CartridgeServices.jar` files, run the following commands from the command line:

```
loadjava -user sys/PASSWORD -resolve -synonym -grant public
-verbose ORACLE_HOME/vobs/jlib/CartridgeServices.jar

loadjava -user sys/PASSWORD -resolve -synonym -grant public
-verbose ORACLE_HOME/vobs/jlib/ODCI.jar
```

Substitute the `SYS` password for `PASSWORD`, and substitute the Oracle home directory for `ORACLE_HOME`. These commands install the classes and create the synonyms in the `SYS` schema. See the chapter on what to do after migrating or updating the database, in *Oracle9i Database Migration*, for further details on installing the `jar` files.

Cartridge Services—Maintaining Context

The Java cartridge service is used for maintaining context. It is similar to the OCI context management service. This class should be used when switching context between the server and the cartridge code.

For additional detail on Java functionality, see the *Oracle9i Java Stored Procedures Developer's Guide*.

ContextManager

ContextManager is a Constructor in class Oracle that extends Object.

Class Interface

```
public static Hashtable ctx

extends Object
```

Variable

```
ctx

 public static Hashtable ctx
```

Constructors

```
ContextManager

 public ContextManager()
```

Methods

The following methods are available:

```
setContext (static method in class oracle)
getContext (static method in class oracle)
clearContext (static method in class oracle)
```

CountException()

CountException is a Constructor for class Oracle that extends Exception.

`Class oracle.CartridgeServices.CountException`

CountException(String)

CountException is a Constructor for class Oracle that extends Exception.

```
public CountException(String s)
```

InvalidKeyException()

InvalidKeyException() is a Constructor for class Oracle that extends Exception.

```
public InvalidKeyException(String s)
```

InvalidKeyException(String)

InvalidKeyException(String) is a Constructor for class Oracle that extends Exception.

```
public InvalidKeyException(String s)
```

16

Reference: Extensibility Constants, Types, and Mappings

This chapter first describes System Defined Constants and System Defined Types. Both of these apply generically to all supported languages. Next, in three subsections, this chapter describes mappings that are specific to the PL/SQL, C, and Java languages.

System Defined Constants

All the constants referred to in this chapter are defined in the `ODCIConst` package installed as part of the `catodci.sql` script. There are equivalent definitions for use within C routines in `odci.h`.

We strongly recommend that you use these constants instead of hard coding their underlying values in your routines.

ODCIIndexAlter Options

- `AlterIndexNone`
- `AlterIndexRename`
- `AlterIndexRebuild`
- `AlterIndexUpdBlockRefs`

ODCIArgDesc.ArgType Bits

- `ArgOther`
- `ArgCol`
- `ArgLit`
- `ArgAttr`
- `ArgNull`

ODCIEnv.CallProperty Values

- `None`
- `FirstCall`
- `IntermediateCall`
- `FinalCall`

ODCIIndexInfo.Flags Bits

Bit	Meaning
`Local`	Indicates a local domain index
`RangePartn`	For a local domain index, indicates that the base table is range-partitioned. Is set only in conjunction with the `Local` bit
Parallel	Indicates that a parallel degree was specified for the index creation or alter operation
Unusable	Indicates that `UNUSABLE` was specified during index creation and that the index being created will be marked unusable
`IndexOnIOT`	Indicates that the domain index is defined on an index-organized table
`FunctionIdx`	Indicates that the the index is a function-based domain index

ODCIPredInfo.Flag Bits

- `PredExactMatch`
- `PredPrefixMatch`
- `PredIncludeStart`
- `PredIncludeStop`
- `PredMultiTable`
- `PredObjectFunc`
- `PredObjectPkg`
- `PredObjectType`

ODCIFuncInfo.Flags Bits

- `ObjectFunc`
- `ObjectPkg`
- `ObjectType`

ODCIQueryInfo.Flags Bits

- `QueryFirstRows`
- `QueryAllRows`

ODCIStatsOptions.Flags Bits

- `EstimateStats`
- `ComputeStats`
- `Validate`

ODCIStatsOptions.Options Bits

- `PercentOption`
- `RowOption`

ScnFlg (Function with Index Context) Values

- `RegularCall`
- `CleanupCall`

Status Values

- `Success`
- `Error`
- `Warning`
- `ErrContinue`
- `Fatal`

Status	Description
`Success`	Indicates a successful operation.
`Error`	Indicates an error.
`Warning`	Oracle9*i* introduces a new global temporary table—`ODCI_WARNINGS$(C1 NUMBER, C2 VARCHAR2(2000))`—defined in `catodci.sql` to hold warnings and error messages. The table is specific to a user session. Cartridge warnings and error messages can be put into this table. The column `C1` holds the line number where the warning or error was generated, and `C2` contains the text of the message. If you prefer to produce more structured or formatted messages, you can define your own table to hold them and simply insert a row in the `ODCI_WARNINGS$` table to indicate where your warning or error message can be found. If an `ODCI_WARNING` is returned by an ODCI DDL call, the server reads the `ODCI_WARNINGS$` table and dumps it out.
`ErrContinue`	`ErrContinue` is used by the intermediate calls during DDL for local domain indexes. Having an `ODCIIndex` call for a partition-level operation return `ErrContinue` implies that there was an error in processing the data for the particular partition but that the DDL as a whole can continue. In such cases, the related partition is marked `FAILED`, and processing continues with the next partition. By way of contrast, if any operation returns an `Error` status, processing is immediately halted, and the index as well as the specific partition that returned `Error` is marked `FAILED`.

Status	Description
`Fatal`	When an index creation operation returns this status code, all the dictionary entries corresponding to the index are dropped and the entire `CREATE INDEX` operation is rolled back. Note that this assumes that the data cartridge did not perform a `COMMIT` after creating any schema objects in the routine (or after doing a cleanup of any such objects created) before returning `Fatal`. A `Fatal` status code can be returned only from the `ODCIIndexCreate` call, for a non-local domain index, and from the first call to `ODCIIndexCreate`, for a local domain index. In all other cases, a return of `Fatal` is treated like a return of `Error`.

System Defined Types

A number of *system-defined types* are defined by Oracle and need to be created by running the `catodci.sql` catalog script. The C mappings for these object types are defined in odci.h The `ODCIIndex` and `ODCIStats` routines described in Chapter 17 and Chapter 18 use these types as parameters.

Unless otherwise mentioned, the names parsed as type attributes are unquoted identifiers.

ODCIArgDesc

Name

`ODCIArgDesc`

Datatype

Object type.

Purpose

Stores function / operator arguments.

Table 16–1 Function/Operator Argument Description — Attributes

Name	Datatype	Purpose
`ArgType`	`NUMBER`	Argument type
`TableName`	`VARCHAR2(30)`	Name of table
`TableSchema`	`VARCHAR2(30)`	Schema containing the table
`ColName`	`VARCHAR2(4000)`	Name of column. This could be top level column name such as "A", or a nested column "A"."B" Note that the column name are quoted identifiers.
`TablePartitionLower`	`VARCHAR2(30)`	Contains the name of the lowest table partition that is accessed in the query
`TablePartitionUpper`	`VARCHAR2(30)`	Contains the name of the highest table partition that is accessed in the query

ODCIArgDescList

Name

`ODCIArgDescList`

Datatype

`VARRAY(32767) of ODCIArgDesc`

Purpose

Contains a list of argument descriptors

ODCIRidList

Name

`ODCIRidList`

Datatype

```
VARRAY(32767) OF VARCHAR2("M_URID_SZ")
```

Purpose

Stores list of rowids. The rowids are stored in their character format.

ODCIColInfo

Name

`ODCIColInfo`

Datatype

Object type.

Purpose

Stores information related column.

Table 16–2 Column Related Information — Attributes

Name	Datatype	Purpose
`TableSchema`	`VARCHAR2(30)`	Schema containing table
`TableName`	`VARCHAR2(30)`	Name of table
`ColName`	`VARCHAR2(4000)`	Name of column. This could be top level column name such as "A", or a nested column "A"."B" Note that the column name are quoted identifiers.
`ColTypeName`	`VARCHAR2(30)`	Datatype of column
`ColTypeSchema`	`VARCHAR2(30)`	Schema containing datatype if user-defined datatype
`TablePartition`	`VARCHAR2(30)`	For a local domain index, contains the name of the specific base table partition

ODCIColInfoList

Name

`ODCIColInfoList`

Datatype

`VARRAY(32) OF ODCIColInfo`

Purpose

Stores information related to a list of columns.

ODCIColStats

Name

`ODCIColStats`

Datatype

Object type

Purpose

Stores statistics for a list of columns for a table function.

Table 16–3 ODCIColStats — Attributes

Name	Datatype	Purpose
`Col`	`ODCIColInfo`	Column of table function argument (cursor)
`Num_distinct`	`NUMBER`	Number of distinct values for column
`Low_value`	`RAW(32)`	Minimum value of column
`High_value`	`RAW(32)`	Maximum value of column
`Num_nulls`	`NUMBER`	Number of NULLs in the column
`Avg_col_len`	`NUMBER`	Average length of column in bytes
`User_stats`	`RAW(2000)`	User-defined statistics for column

ODCIColStatsList

Name

`ODCIColStatsList`

Datatype

`VARRAY(32)` of `ODCIColStats`

Purpose

Stores statistics for a list of column for a table function.

ODCICost

Name

`ODCICost`

Datatype

Object type.

Purpose

Stores cost information.

Table 16–4 Cost Information — Attributes

Name	Datatype	Purpose
`CPUCost`	`NUMBER`	CPU cost
`IOCost`	`NUMBER`	I/O cost
`NetworkCost`	`NUMBER`	Communication cost
`IndexCostInfo`	`VARCHAR2(255)`	Optional user-supplied information about the domain index for display in the `PLAN` table (255 characters maximum)

ODCIEnv

Name

ODCIEnv

Datatype

Object type

Purpose

Contains general information about the environment in which the extensibility routines are executing.

Table 16–5 Environment Variable Descriptor Information — Attributes

Name	Datatype	Purpose
EnvFlags	NUMBER	Not currently used
CallProperty	NUMBER	■ 0 = None ■ 1 = First Call ■ 2 = Intermediate Call ■ 3 = Final Call

Usage Notes

CallProperty is used only for local domain indexes. For non-local domain indexes it is always set to 0.

For local domain indexes, CallProperty is set to indicate which is the current call in cases where multiple calls are made to the same routine.

For example, when creating a local domain index, the ODCIIndexCreate routine is called N+2 times, where N is the number of partitions. For the first call, CallProperty is set to FirstCall, for the N intermediate calls, it is set to IntermediateCall, and for the last call it is set to FinalCall.

CallProperty is used only for CREATE INDEX, DROP INDEX, TRUNCATE TABLE, and for some of the extensible optimizer-related calls for local domain indexes. In all other cases, including DML and query routines for local domain indexes, it is set to 0.

ODCIFuncInfo

Name

`ODCIFuncInfo`

Datatype

Object type.

Purpose

Stores function information.

Table 16–6 Function Information — Attributes

Name	Datatype	Purpose
`ObjectSchema`	`VARCHAR2(30)`	`Object schema name`
`ObjectName`	`VARCHAR2(30)`	`Function/package/type name`
`MethodName`	`VARCHAR2(30)`	`Method name for package/type`
`Flags`	`NUMBER`	`Function flags - see ODCIConst`

ODCIIndexInfo

Name

`ODCIIndexInfo`

Datatype

Object type

Purpose

Stores the metadata information related to a domain index. It is passed as a parameter to all `ODCIIndex` routines.

Table 16–7 Index Related Information — Attributes

Name	Datatype	Purpose
`IndexSchema`	`VARCHAR2(30)`	Schema containing domain index
`IndexName`	`VARCHAR2(30)`	Name of domain index
`IndexCols`	`ODCIColInfoList`	List of indexed columns
`IndexPartition`	`VARCHAR2(30)`	For a local domain index, contains the name of the specific index partition
`IndexInfoFlags`	`NUMBER`	Possible flags are: ■ `Local` ■ `RangePartn` ■ `Parallel` ■ `Unusable` ■ `IndexOnIOT` ■ `FunctionIdx`
`IndexParaDegree`	`NUMBER`	The degree of parallelism, if one is specified when creating or rebuilding a domain index or local domain index partition in parallel

ODCIPredInfo

Name

`ODCIPredInfo`

Datatype

Object type

Purpose

Stores the metadata information related to a predicate containing a user-defined operator or function. It is also passed as a parameter to `ODCIIndexStart`() query routine.

Table 16–8 Operator Related Information — Attributes

Name	Datatype	Purpose
`ObjectSchema`	`VARCHAR2(30)`	Schema of operator/function
`ObjectName`	`VARCHAR2(30)`	Name of operator/function
`MethodName`	`VARCHAR2(30)`	Name of method, applies only to package methods type
`Flags`	`NUMBER`	The possible flags that could be set are: PredExactMatch - Exact Match PredPrefixMatch - Prefix Match PredIncludeStart - Bounds include the start key value PredIncludeStop - Bounds include the stop key value PredMultiTable - Predicate involves multiple tables PredObjectFunc - Object is a function PredObjectPlg - Object is a package PredObjectType - Object is a type

ODCIIndexCtx

Name

`ODCIIndexCtx`

Datatype

Object type

Purpose

Stores the index context, including the domain index metadata and the `ROWID`. It is passed as parameter to the functional implementation of an operator that expects index context.

Table 16–9 Index Context Related Information — Attributes

Name	Datatype	Purpose
`IndexInfo`	`ODCIIndexInfo`	Stores the metadata information about the domain index
`rid`	`VARCHAR2("M_URID_SZ")`	Row identifier of the current row

ODCIObject

Name

`ODCIObject`

Datatype

Object type

Purpose

Stores information about a schema object.

Table 16–10 Index Context Related Information — Attributes

Name	Datatype	Purpose
`ObjectSchema`	`VARCHAR2(30)`	Name of schema in which object is located
`ObjectName`	`VARCHAR2(30)`	Name of object

ODCIObjectList

Name

`ODCIObjectList`

Datatype

`VARRAY(32) OF ODCIObject`

Purpose

Stores information about a list of schema objects.

ODCIPartInfo

Name
`ODCIPartInfo`

Datatype
Object type

Purpose
Contains the names of both the table partition and the index partition.

Table 16–11 Index-Related Information — Attributes

Name	Datatype	Purpose
TablePartition	`VARCHAR2(30)`	Contains the table partition name
IndexPartition	`VARCHAR2(30)`	Contains the index partition name

ODCIQueryInfo

Name

`ODCIQueryInfo`

Datatype

Object type

Purpose

Stores information about the context of a query. It is passed as a parameter to the `ODCIIndexStart` routine.

Table 16–12 Index Context Related Information — Attributes

Name	Datatype	Purpose
`Flags`	`NUMBER`	The following flags can be set: ■ `QueryFirstRows` —Set when the optimizer hint `FIRST_ROWS` is specified in the query ■ `QueryAllRows` —Set when the optimizer hint `ALL_ROWS` is specified in the query
`AncOps`	`ODCIObjectList`	Ancillary operators referenced in the query

ODCIStatsOptions

Name

`ODCIStatsOptions`

Datatype

Object type.

Purpose

Stores options information for ANALYZE.

Table 16–13 Cost Information — Attributes

Name	Datatype	Purpose
`Sample`	`NUMBER`	`Sample size`
`Options`	`NUMBER`	`ANALYZE options - see "ODCICost" on page 16-23`
`Flags`	`NUMBER`	`ANALYZE flags - see "ODCICost" on page 16-23`

ODCITabStats

Name

`ODCITabStats`

Datatype

`NUMBER`

Purpose

Stores table statistics for a table function

Table 16–14 ODCITabStats — Attributes

Name	Datatype	Purpose
`Num_rows`	`NUMBER`	Number of rows in table

ODCITableFunctionStats

Name

`ODCITableFunctionStats`

Datatype

Object type

Purpose

Stores table function statistics

Table 16–15 PDCOTab;eFimctopmStats — Attributes

Name	Datatype	Purpose
`ColumnStats`	`ODCIColStatsList`	Column statistics for a table function argument
`TableStats`	`ODCITabStats`	Table statistics for a table function argument

Mappings of Constants and Types

Mappings in PL/SQL

A variety of PL/SQL mappings are common to both Extensible Indexing and the Extensible Optimizer.

- Constants are defined in the `ODCIConst` package found in `catodci.sql`
- Types are defined as object types found in `catodci.sql`

Mappings in C

Mappings of constants and types are defined for C in the public header file `odci.h`. Each C structure to which a type is mapped has a corresponding indicator structure called `structname_ind` and a reference definition called `structname_ref`.

Mappings in Java

The ODCI (Oracle Data Cartridge Interface) interfaces are described in the *Oracle9i Supplied Java Packages Reference.*, To use these classes, they must first be loaded. See Chapter 15 for loading instructions.

Constants Definitions

The following constants create or replace the `ODCIConst IS` package.

To ensure that the database or packet state are not inadvertently corrupted, the following statement is always used with these methods to restrict reads and writes:

```
pragma restrict_references(ODCIConst, WNDS, RNDS, WNPS, RNPS);
```

Constants for Return Status

```
Success          CONSTANT INTEGER  :=  0;
Error            CONSTANT INTEGER  :=  1;
Warning          CONSTANT INTEGER  :=  2;
ErrContinue      CONSTANT INTEGER  :=  3;
Fatal            CONSTANT INTEGER  :=  4;
```

Constants for ODCIPredInfo.Flags

```
PredExactMatch   CONSTANT INTEGER  :=   1;
PredPrefixMatch  CONSTANT INTEGER  :=   2;
PredIncludeStart CONSTANT INTEGER  :=   4;
PredIncludeStop  CONSTANT INTEGER  :=   8;
PredObjectFunc   CONSTANT INTEGER  :=  16;
PredObjectPkg    CONSTANT INTEGER  :=  32;
PredObjectType   CONSTANT INTEGER  :=  64;
PredMultiTable   CONSTANT INTEGER  := 128;
```

Constants for ODCIQueryInfo.Flags

```
QueryFirstRows   CONSTANT INTEGER  :=  1;
QueryAllRows     CONSTANT INTEGER  :=  2;
```

Constants for ScnFlg (Func with Index Context)

```
CleanupCall      CONSTANT INTEGER  :=  1;
RegularCall      CONSTANT INTEGER  :=  2;
```

Constants for ODCIFuncInfo.Flags

```
ObjectFunc       CONSTANT INTEGER  :=  1;
ObjectPkg        CONSTANT INTEGER  :=  2;
ObjectType       CONSTANT INTEGER  :=  4;
```

Constants for ODCIArgDesc.ArgType

```
ArgOther         CONSTANT INTEGER  :=  1;
ArgCol           CONSTANT INTEGER  :=  2;
ArgLit           CONSTANT INTEGER  :=  3;
ArgAttr          CONSTANT INTEGER  :=  4;
ArgNull          CONSTANT INTEGER  :=  5;
```

Constants for ODCIStatsOptions.Options

```
PercentOption    CONSTANT INTEGER  :=  1;
RowOption        CONSTANT INTEGER  :=  2;
```

Constants for ODCIStatsOptions.Flags

```
EstimateStats    CONSTANT INTEGER  :=  1;
ComputeStats     CONSTANT INTEGER  :=  2;
Validate         CONSTANT INTEGER  :=  4;
```

Constants for ODCIIndexAlter parameter alter_option

```
AlterIndexNone           CONSTANT INTEGER  :=  0;
AlterIndexRename         CONSTANT INTEGER  :=  1;
AlterIndexRebuild        CONSTANT INTEGER  :=  2;
AlterIndexUpdBlockRefs   CONSTANT INTEGER  :=  5;
```

Constants for ODCIIndexInfo.IndexInfoFlags

```
Local                    CONSTANT INTEGER  := 1;
RangePartn               CONSTANT INTEGER  := 2;
Parallel                 CONSTANT INTEGER  := 16;
Unusable                 CONSTANT INTEGER  := 32;
IndexOnIOT               CONSTANT INTEGER  := 64;
FunctionIdx              CONSTANT INTEGER  := 256;
```

Constants for ODCIEnv.CallProperty

```
None                     CONSTANT INTEGER  := 0;
FirstCall                CONSTANT INTEGER  := 1;
IntermediateCall         CONSTANT INTEGER  := 2;
FinalCall                CONSTANT INTEGER  := 3;
```

17

Reference: Extensible Indexing Interface

This chapter describes Java language ODCI (Oracle Data Cartridge Interface) Extensible Indexing Interfaces. For more complete details on Java functionality, refer to the *Oracle9i Supplied Java Packages Reference*.

The following interfaces are described:

- ODCIGetInterfaces
- ODCIIndexAlter
- ODCIIndexClose
- ODCIIndexCreate
- ODCIIndexDelete
- ODCIIndexDrop
- ODCIIndexExchangePartition
- ODCIIndexFetch
- ODCIIndexGetMetadata
- ODCIIndexInsert
- ODCIIndexMergePartition
- ODCIIndexSplitPartition
- ODCIIndexStart
- ODCIIndexTruncate
- ODCIIndexUpdate

Extensible Indexing — System Defined Interface Routines

Caution: These routines are invoked by Oracle at the appropriate times based on SQL statements executed by the end user. The user should not try to invoke these routines directly as this may result in corruption of index data.

ODCIGetInterfaces

Syntax

ODCIGetInterfaces(ifclist OUT ODCIObjectList) RETURN NUMBER

Purpose

The `ODCIGetInterfaces` function is invoked when an `INDEXTYPE` is created by a `CREATE INDEXTYPE...` statement or is altered.

Table 17–1 ODCIGetInterfaces Arguments

Argument	Meaning
`ifclist`	Contains information about the interfaces it supports

Returns

- `ODCIConst.Success` on success
- `ODCIConst.Error` on error.

Usage Notes

This function should be implemented as a static type method.

This function must return `'SYS.ODCIINDEX2'` in the `ODCIObjectList` if the indextype uses the second version of the `ODCIIndex` interface, that is, the version described in this book.

In existing code that uses the previous, Oracle8*i* version of the `ODCIIndex` interface, this function was required to return `'SYS.ODCIINDEX1'` to specify the Oracle8*i* version of the interface. That code will still work. To continue to use the Oracle8*i* interface, continue to have this function return `'SYS.ODCIINDEX1'`, and do not implement Oracle9*i* versions of any of the routines.

ODCIIndexAlter

Syntax

ODCIIndexAlter(ia ODCIIndexInfo, parms IN OUT VARCHAR2, alter_option NUMBER, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked when a domain index or a domain index partition is altered using an `ALTER INDEX` or an `ALTER INDEX PARTITION` statement.

Table 17–2 ODCIIndexAlter Arguments

Argument	Meaning
`ia`	Contains information about the index and the indexed column
parms (IN)	Parameter string if ALTER INDEX PARAMETERS or ALTER INDEX REBUILD contains the user specified parameter string if ALTER INDEX RENAME contains the new name of the domain index
parms (OUT)	Parameter string is valid only if ALTER INDEX PARAMETERS or ALTER INDEX REBUILD. Contains the resultant string to be stored in system catalogs
`alter_option`	Specifies one of the following options: ■ `AlterIndexNone` if `ALTER INDEX [PARTITION] PARAMETERS` ■ `AlterIndexRename` if `ALTER INDEX RENAME [PARTITION]` ■ `AlterIndexRebuild` if `ALTER INDEX REBUILD [PARTITION] [PARALLEL (DEGREE deg)] [PARAMETERS]` ■ `AlterIndexUpdBlockRefs` if `ALTER INDEX [schema.]index UPDATE BLOCK REFERENCES`
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

Usage Notes

- This function should be implemented as a static type method.
- An `ALTER INDEX` statement can be invoked for domain indexes in multiple ways.

  ```
  ALTER INDEX index_name
  PARAMETERS (parms);
  ```

  or

  ```
  ALTER INDEX index_name
  REBUILD PARAMETERS (parms);
  ```

  The precise behavior in these two cases is defined by the implementor. One possibility is that the first statement would merely reorganize the index based on the parameters while the second would rebuild it from scratch.
- The maximum length of the input parameters string is 1000 characters. The `OUT` value of the `parms` argument can be set to resultant parameters string to be stored in the system catalogs.
- The `ALTER INDEX` statement can also be used to rename a domain index in the following way:

  ```
  ALTER INDEX index_name
  RENAME TO new_index_name
  ```

  In this case, the new name of the domain index is passed to the `parms` argument.
- If the `PARALLEL` clause is omitted, then the domain index or local domain index partition is rebuilt sequentially.
- If the `PARALLEL` clause is specified, the parallel degree is passed to the `ODCIIndexAlter` invocation in the `IndexParaDegree` attribute of `ODCIIndexInfo`, and the `Parallel` bit of the `IndexInfoFlags` attribute is set. The parallel degree is determined as follows:
  - If `PARALLEL DEGREE` *deg* is specified, *deg* is passed.

  - If only `PARALLEL` is specified, then a constant is passed to indicate that the default degree of parallelism was specified.

- If the `ODCIIndexAlter` routine returns with the `ODCIConst.Success`, the index is valid and usable. If the `ODCIIndexAlter` routine returns with `ODCIConst.Warning`, the index is valid and usable but a warning message is returned to the user. If `ODCIIndexAlter` returns with an error (or exception), the domain index will be marked `FAILED`.
- When the `ODCIIndexAlter` routine is being executed, the domain index is marked `LOADING`.
- Every SQL statement executed by ODCIIndexAlter is treated as an independent operation. The changes made by ODCIIndexCreate are not guaranteed to be atomic.
- The `AlterIndexUpdBlockRefs` alter option applies only to domain indexes on index-organized tables. When the end user executes an `ALTER INDEX` *`<domain_index>`* `UPDATE BLOCK REFERENCES`, `ODCIIndexAlter` is called with the `AlterIndexUpdBlockRefs` bit set to give the cartridge developer the opportunity to update guesses as to the block locations of rows, stored in logical rowids.

ODCIIndexClose

Syntax

ODCIIndexClose(self IN <impltype>, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked to end the processing of an operator.

Table 17–3 ODCIIndexClose Arguments

Argument	Meaning
`self(IN)`	Is the value of the context returned by the previous invocation of `ODCIIndexFetch`
env	The environment handle passed to the routine

Returns

- `ODCIConst.Success` on success
- `ODCIConst.Error` on error.

Usage Notes

- The index implementor can perform any appropriate actions to finish up the processing of an domain index scan, such as freeing memory and other resources.

ODCIIndexCreate

Syntax

ODCIIndexCreate(ia ODCIIndexInfo, parms VARCHAR2, env ODCIEnv) RETURN NUMBER

Purpose

The `ODCIIndexCreate` method is invoked when a domain index is created by a `CREATE INDEX...INDEXTYPE IS...PARAMETERS...` statement issued by the user. The domain index that is created can be a non-partitioned index or a local partitioned domain index.

Table 17–4 ODCIIndexCreate Arguments

Argument	Meaning
`ia`	Contains information about the indexed column
`parms`	Is the PARAMETERS string passed in uninterpreted by Oracle. The maximum size of the parameter string is 1000 characters.
`env`	The environment handle passed to the routine

Returns

- `ODCIConst.Success` on success
- `ODCIConst.Error` on error
- `ODCIConst.Warning`
- `ODCIConst.ErrContinue` if the method is invoked at the partition level for creation of a local partitioned index, to continue to the next partition even in case of an error
- `ODCIConst.Fatal` to signify that all dictionary entries for the index are cleaned up and that the `CREATE INDEX` operation is rolled back. Returning this status code assumes that the cartridge code has not created any objects (or cleaned up any objects created).

Usage Notes

- This function should be implemented as a static type method.

- The `ODCIIndexCreate` routine should create objects (such as tables) to store the index data, generate the index data, and store the data in the index data tables.
- The `ODCIIndexCreate` procedure should handle creation of indexes on both empty and non-empty tables. If the base table is not empty, the `ODCIIndexCreate` procedure can scan the entire table and generate index data.
- Every SQL statement executed by `ODCIIndexCreate` is treated as an independent operation. The changes made by `ODCIIndexCreate` are not guaranteed to be atomic.
- For a non-partitioned domain index, the parallel degree is passed to the `ODCIIndexCreate` invocation in the `IndexParaDegree` attribute of `ODCIIndexInfo`, and the `Parallel` bit of the `IndexInfoFlags` is set. The parallel degree is determined as follows:
  - If `PARALLEL DEGREE` *deg* specified, *deg* is passed.
  - If only `PARALLEL` is specified, then a constant indicating that the default degree of parallelism was specified, is passed.
  - If the `PARALLEL` clause is omitted altogether, the operation is done sequentially
- When the `ODCIIndexCreate` routine is being executed, the domain index is marked `LOADING`.
- If the `ODCIIndexCreate` routine returns with the `ODCIConst.Success`, the index is valid and usable. If the `ODCIIndexCreate` routine returns with `ODCIConst.Warning`, the index is valid and usable but a warning message is returned to the user. If the `ODCIIndexCreate` routine returns with an `ODCIConst.Error` (or exception), the domain index will be marked `FAILED.`
- The only operations permitted on `FAILED` domain indexes is `DROP INDEX`, `TRUNCATE TABLE` or `ALTER INDEX REBUILD`.
- If a domain index is created on an column of object type which contains a `REF` attribute, do not dereference the `REF`s while building your index. Dereferencing a `REF` fetches data from a different table instance. If the data in the other table is modified, you will not be notified and your domain index will become incorrect.
- To create a non-partitioned domain indexe, the `ODCIIndexCreate` method is invoked once, and the only valid return codes are `ODCIConstSuccess`,

ODCIConstWarning or ODCIConstError. The IndexPartition and TablePartition name are NULL and callProperty is also NULL.

To create a local partitioned domain index, the ODCIIndexCreate method is invoked *N*+2 times, where *N* is the number of local index partitions. The first and the final call handle operations on the top-level index object, and the intermediate *N* calls handle partition-level objects. In the first call, a table to hold the index level metadata can be created. In the intermediate calls, independent tables to hold partition level data can be created and populated, and in the last call, indexes can be built on the index metadata tables and so forth.

For local partitioned domain indexes, the first and the last call can return ODCIConstSuccess, ODCIConstWarning or ODCIConstError. The intermediate *N* calls can return ODCIConstSuccess, ODCIConstWarning, ODCIConstError or ODCIConstErrContiue. If a partition level call returns ODCIConstError, the partition is marked FAILED, the index is marked FAILED, and the create operation terminates at that point. If the call returns ODCIConstErrContinue, the partition is marked FAILED, and the method is invoked for the next partition.

This method is invoked during ALTER TABLE ADD PARTITION too. In this case, there is only one call to ODCIIndexCreate, the IndexPartition and TablePartition name are filled in, and the callProperty is set to NULL

Since this routine handles multiple things (namely, creation of a non-partitioned index, creation of a local index and creation of a single index partition), you must take special care to code it appropriately.

ODCIIndexDelete

Syntax

ODCIIndexDelete(ia ODCIIndexInfo, rid VARCHAR2, oldval <icoltype>, env ODCIEnv) RETURN NUMBER

Purpose

This procedure is invoked when a row is deleted from a table that has a domain index defined on one of its columns.

Table 17–5 ODCIIndexDelete Arguments

Argument	Meaning
`ia`	Contains information about the index and the indexed column
`rid`	The row identifier of the deleted row
`oldval`	The value of the indexed column in the deleted row. The datatype is the same as that of the indexed column.
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error

Usage Notes

- This function should be implemented as a static type method.
- This method should delete index data corresponding to the deleted row from the appropriate tables/files storing index data.
- If `ODCIIndexDelete` is invoked at the partition level, then the index partition name is filled in in the `ODCIIndexInfo` argument.

ODCIIndexDrop

Syntax

ODCIIndexDrop(ia ODCIIndexInfo, env ODCIEnv) RETURN NUMBER

Purpose

The `ODCIIndexDrop` procedure is invoked when a domain index is dropped explicitly using a `DROP INDEX` statement, or implicitly through a `DROP TABLE`, or `DROP USER` statement.

Table 17–6 ODCIIndexDrop Arguments

Argument	Meaning
`ia`	Contains information about the indexed column
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

While dropping a local domain index, the first *N*+`1` calls can return `ODCIConst.ErrContinue` too.

Usage Notes

- This method should be implemented as a static type method.
- This method should drop the tables storing the domain index data.
- This method is invoked for dropping a non-partitioned index, dropping a local domain index, and also for dropping a single index partition during `ALTER TABLE DROP PARTITION`.

  For dropping a non-partitioned index, the `ODCIIndexDrop` is invoked once, with the `IndexPartition`, `TablePartition` and `callProperty` set to `NULL`.

  For dropping a local domain index, the routine is invoked *N*+2 times, where *N* is the number of partitions.

For dropping a single index partition during ALTER TABLE DROP PARTITION, this routine is invoked once with the IndexPartition and the TablePartition filled in and the callProperty set to NULL.

The old table and the old index partition's dictionary entries are deleted before the call to ODCIIndexDrop, so the cartridge code for this routine should not rely on the existence of this data in the views.

- Since it is possible that the domain index is marked FAILED (due to abnormal termination of some DDL routine), the ODCIIndexDrop routine should be capable of cleaning up partially created domain indexes. When the ODCIIndexDrop routine is being executed, the domain index is marked LOADING.
- Note that if the ODCIIndexDrop routine returns with an ODCIConst.Error or exception, the DROP INDEX statement fails and the index is marked FAILED. In that case, there is no mechanism to get rid of the domain index except by using the FORCE option. If the ODCIIndexDrop routine returns with ODCIConst.Warning in the case of an explicit DROP INDEX statement, the operation succeeds but a warning message is returned to the user.
- Every SQL statement executed by ODCIIndexDrop is treated as an independent operation. The changes made by ODCIIndexDrop are not guaranteed to be atomic.

ODCIIndexExchangePartition

Syntax

ODCIIndexExchangePartition(ia ODCIIndexInfo, ia1 ODCIIndexInfo, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked when an `ALTER TABLE EXCHANGE PARTITION...INCLUDING INDEXES` is issued on a partitioned table on which a local domain index is defined.

Table 17–7 ODCIIndexExchangePartition Arguments

Argument	Meaning
ia	Contains information about the partition to exchange
ia1	Contains information about the non-local, unpartitioned domain index to exchange
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

Usage Notes

The function should be implemented as a static type method.

This method should handle both converting a partition of a domain index into a non-partitioned domain index table and converting a non-partitioned index to a partition of a partitioned domain index.

ODCIIndexFetch

Syntax

ODCIIndexFetch(self IN [OUT] <impltype>, nrows IN NUMBER, rids OUT ODCIRidList, env ODCIEnv) RETURN NUMBER

Purpose

This procedure is invoked repeatedly to retrieve the rows satisfying the operator predicate.

Table 17–8 ODCIIndexFetch Arguments

Argument	Meaning
`self(IN)`	Is the value of the context returned by the previous call (to `ODCIIndexFetch` or to `ODCIIndexStart` if this is the first time fetch is being called for this operator instance
`self(OUT)`	The context that is passed to the next query-time call.Note that this parameter does not have to be defined as `OUT` if the value is not modified in this routine.
`nrows`	Is the maximum number of result rows that can be returned to Oracle in this call
`rids`	Is the array of row identifiers for the result rows being returned by this call
`env`	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error

Usage Notes

- `ODCIIndexFetch` returns rows satisfying the operator predicate. That is, it returns the row identifiers of all the rows for which the operator return value falls within the specified bounds.
- Each call to `ODCIIndexFetch` can return a maximum of *nrows* number of rows. The value of *nrows* passed in is decided by Oracle based on some internal factors. However, the `ODCIIndexFetch` routine can return lesser than *nrows* number of rows. The row identifiers are returned through the output `rids`

array. A `NULL ROWID` (as an element of the `rids` array) indicates that all satisfying rows have been returned.

Assume that there are 3000 rows which satisfy the operator predicate, and that the value of `nrows` = 2000. The first invocation of `ODCIIndexFetch` can return the first 2000 rows. The second invocation can return a `rid` list consisting of the remaining 1000 rows followed by a `NULL` element. The `NULL` value in `rid` list indicates that all satisfying rows have now been returned.

- If the context value is changed within this call, the new value is passed in to subsequent query-time calls.

ODCIIndexGetMetadata

Syntax

ODCIIndexGetMetadata(ia IN ODCIIndexInfo, version IN VARCHAR2, new_block OUT PLS_INTEGER) RETURN VARCHAR2;

Purpose

This routine is called repeatedly to return a series of strings of PL/SQL code that comprise the non-dictionary metadata associated with the index in `ia`. The routine can pass whatever information is required at import time—for example, policy, version, preferences, and so on. This method is optional unless implementation-specific metadata is required.

Table 17–9 ODCIIndexGetMetadata Arguments

Argument	Description
`ia`	Specifies the index on which export is currently working.
`version`	Version of export making the call in the form `08.01.03.00.00`.
`new_block`	Non-zero (`TRUE`): Returned string starts a new PL/SQL block. Export will terminate the current block (if any) with `END`; and open a new block with `BEGIN` before writing strings to the dump file. The routine is called again. `0` (`FALSE`): Returned string continues current block. Export writes only the returned string to the dump file then calls the routine again.

Developers of domain index implementation types in 8.1.3 **must** implement `ODCIIndexGetMetadata` even if only to indicate that no PL/SQL metadata exists or that the index is not participating in fast rebuild.

Returns

- A null-terminated string containing a piece of an opaque block of PL/SQL code.
- A zero-length string indicates no more data; export stops calling the routine.

Usage Notes

This function should be implemented as a static type method.

The routine will be called repeatedly until the return string length is 0. If an index has no metadata to be exported using PL/SQL, it should return an empty string upon first call.

This routine can be used to build one or more blocks of anonymous PL/SQL code for execution by import.Each block returned will be invoked independently by import. That is, if a block fails for any reason at import time, subsequent blocks will still be invoked. Therefore any dependent code should be incorporated within a single block. The size of an individual block of PL/SQL code is limited only by the size of import's read buffer controlled by its `BUFFER` parameter.

The execution of these PL/SQL blocks at import time will be considered part of the associated domain index's creation. Therefore, their execution will be dependent upon the successful import of the index's underlying base table and user's setting of import's `INDEXES=Y/N` parameter, as is the creation of the index.

The routine should not pass back the `BEGIN`/`END` strings that open and close the individual blocks of PL/SQL code; export will add these to mark the individual units of execution.

The parameter `version` is the version number of the currently executing export client. Since export and import can be used to downgrade a database to the previous functional point release, it also represents the minimum server version you can expect to find at import time; it may be higher, but never lower.

The cartridge developer can use this information to determine what version of information should be written to the dump file. For example, assume the current server version is `08.02.00.00.00`, but the export version handed in is `08.01.04.00.00`. If a cartridge's metadata changed formats between 8.1 and 8.2, it would know to write the data to the dump file in 8.1 format anticipating an import into an 8.1.4 system. Server versions starting at 8.2 and higher will have to know how to convert 8.1 format metadata.

Some points to be aware of:

1. The data contained within the strings handed back to export must be completely platform-independent. That is, they should contain no binary information that may reflect the endian nature of the export platform, which may be different from the import platform. Binary information may be passed as hex strings and converted through `RAWTOHEX` and `HEXTORAW`.
2. The strings are translated from the export server to export client character set and are written to the dump file as such. At import time, they are translated from export client character set to import client character set, then from import

client char set to import server character set when handed over the UPI interface.

3. Specifying a specific target schema in the execution of any of the PL/SQL blocks should be avoided as it will most likely cause an error if you exercise import's `FROMUSER` -> `TOUSER` schema replication feature. For example, a procedure prototype such as:

   ```
   PROCEDURE AQ_CREATE ( schema IN VARCHAR2, que_name IN VARCHAR2) ...
   ```
   Should be avoided since this will fail if you have remapped schema A to schema B on import. You can assume at import time that you are already connected to the target schema.

4. Export dump files from a particular version must be importable into all future versions. This means that all PL/SQL routines invoked within the anonymous PL/SQL blocks written to the dump file must be supported for all time. You may wish to encode some version information to assist with detecting when conversion may be required.

5. Export will be operating in a read-only transaction if its parameter `CONSISTENT=Y`. In this case, no writes are allowed from the export session. Therefore, this method must not write any database state.

6. You can attempt to import the same dump file multiple times, especially when using import's `IGNORE=Y` parameter. Therefore, this method must produce PL/SQL code that is idempotent, or at least deterministic when executed multiple times.

7. Case on database object names must be preserved; that is, objects named '`Foo`' and '`FOO`' are distinct objects. Database object names should be enclosed within double quotes ("") to preserve case.

Error Handling

Any unrecoverable error should raise an exception allowing it to propagate back to `get_domain_index_metadata` and thence back to export. This will cause export to terminate the creation of the current index's DDL in the dump file and to move on to the next index.

At import time, failure of the execution of any metadata PL/SQL block will cause the associated index to not be created under the assumption that the metadata creation is an integral part of the index creation.

ODCIIndexInsert

Syntax

ODCIIndexInsert(ia ODCIIndexInfo, rid VARCHAR2, newval <icoltype>, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked when a new row is inserted into a table that has a domain index defined on one of its columns.

Table 17–10 ODCIIndexInsert Arguments

Argument	Meaning
`ia`	Contains information about the index and the indexed column
`rid`	The row identifier of the new row in the table
`newval`	The value of the indexed column in the inserted row.
`env`	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error

Usage Notes

This function should be implemented as a static type method.

This method should insert index data corresponding to the new row into the appropriate tables/files storing index data.

If `ODCIIndexInsert` is invoked at the partition level, then the index partition name is filled in in the `ODCIIndexInfo` argument.

ODCIIndexMergePartition

Syntax

ODCIIndexMergePartition(ia ODCIIndexInfo, part_name1 ODCIPartInfo, part_name2 ODCIPartInfo, parms VARCHAR2, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked when a `ALTER TABLE MERGE PARTITION` is issued on range partitioned table on which a local domain index is defined.

Table 17–11 ODCIIndexMergePartition Arguments

Argument	Meaning
ia	Contains index and table partition name for one of the partitions to be merged
part_name1	Contains index and table partition name for the second partition to be merged
part_name2	Holds index and table partition name for the merged partition
parms	Contains the parameter string for the resultant merged partition—essentially the default parameter string associated with the index
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

Usage Notes

- The function should be implemented as a static type method.
- You should create a new table representing the resultant merged partition and populate it with data from the merged partitions. Then drop the tables corresponding to the merged index partitions. Also, the newly created partition should pick the default parameter string associated with the index level.

  The old table and the old index partitions' dictionary entries are deleted before the call to `ODCIIndexMergePartition`, so the cartridge code for this routine should not rely on the existence of this data in the views.

ODCIIndexSplitPartition

Syntax

ODCIIndexSplitPartition(ia ODCIIndexInfo, part_name1 ODCIPartInfo, part_name2 ODCIPartInfo, parms VARCHAR2, env ODCIEnv) RETURN NUMBER

Purpose

This method is invoked when an `ALTER TABLE SPLIT PARTITION` is invoked on a partitioned table on which a local domain index is defined.

Table 17–12 ODCIIndexSplitPartition Arguments

Argument	Meaning
ia	Contains the information about the partition to be split
part_name1	Holds the index and table partition names for one of the new partitions
part_name2	Holds the index and table partition names for the other new partition
parms	Contains the parameter string for the new partitions—essentially the parameter string associated with the index partition that is being split
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

Usage Notes

The function should be implemented as a static type method.

Cartridge writers need to drop the metadata corresponding to the partition that is split and create metadata for the two partitions that are created as a result of the split. The index data corresponding to these partitions need not be computed since the indexes are marked `UNUSABLE`. When the user issues `ALTER INDEX REBUILD PARTITION` to make the indexes usable, the indexes can be built.

The old table and the old index partition's dictionary entries are deleted before the call to `ODCIIndexSplitPartition`, so the cartridge code for this routine should not rely on the existence of this data in the views.

ODCIIndexStart

Syntax

ODCIIndexStart(sctx IN OUT <impltype>, ia ODCIIndexInfo, pi ODCIPredInfo, qi ODCIQueryInfo, strt <opbndtype>, stop <opbndtype>, <valargs>, env ODCIEnv) RETURN NUMBER

Purpose

This procedure is invoked to start the evaluation of an operator on an indexed column.

Table 17–13 ODCIIndexStart Arguments

Argument	Meaning
`sctx(IN)`	The value of the scan context returned by some previous related query-time call (such as the corresponding ancillary operator, if invoked before the primary operator); `NULL` otherwise
`sctx(OUT)`	The context that is passed to the next query-time call; the next query-time call will be to `ODCIIndexFetch`
`ia`	Contains information about the index and the indexed column
`pi`	Contains information about the operator predicate
`qi`	Contains query information (hints plus list of ancillary operators referenced)
`strt`	The start value of the bounds on the operator return value. The datatype is the same as that of the operator's return value
`stop`	The stop value of the bounds on the operator return value. The datatype is the same as that of the operator's return value.
`valargs`	The value arguments of the operator invocation. The number and datatypes of these arguments are the same as those of the value arguments to the operator.
`env`	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error

Usage Notes

- The function should be implemented as a static method.

- ODCIIndexStart is invoked to begin the evaluation of an operator on an indexed column. In particular, the following conditions hold:
  - The first argument to the operator is a column which has a domain index defined on it.
  - The indextype of the domain index (specified in ODCIIndexInfo parameter) supports the current operator.
  - All other arguments to the operator are value arguments (literals) which are passed in through the <valargs> parameters.
- The ODCIIndexStart method should initialize the index scan as needed (using the operator-related information in the pi argument) and prepare for the subsequent invocations of ODCIIndexFetch.
- The strt, stop parameters together with the bndflg value in ODCIPredInfo parameter specify the range of values within which the operator return value should lie.
- Bounds for operator return values are specified as follows:
  - If the predicate to be evaluated is of the form op LIKE val, the ODCIIndexPrefixMatch flag is set. In this case, the start key contains the value <val> and the stop key value is irrelevant.
  - If the predicate to be evaluated is of the form op = val, the ODCIIndexExactMatch flag is set. In this case, the start key contains the value <val> and the stop key value is irrelevant.
  - If the predicate to be evaluated is of the form op > val, startkey contains the value <val> and stop key value is set to NULL. If the predicate is of the form op >= <val>, the flag ODCIIndexIncludeStart is also set.
  - If the predicate to be evaluated is of the form op < val, stop key contains the value <val> and the start key value is set to NULL. If the predicate is of the form op <= val, the flag ODCIIndexIncludeStop is also set.
- A context value can be returned to Oracle (through the SELF argument) which will then be passed back to the next query-time call. The next call will be to ODCIIndexFetch if the evaluation continues, or to ODCIIndexStart if the evaluation is restarted. The context value can be used to store the entire evaluation state or just a handle to the memory containing the state.
- Note that if the same indextype supports multiple operators with different signatures, multiple ODCIIndexStart methods need to be implemented, one

for each distinct combination of value argument datatypes. For example, if an indextype supports three operators:

1. `op1(number, number)`
2. `op1(varchar2, varchar2)`
3. `op2(number, number)`

two `ODCIIndexStart` routines would need to be implemented:

- `ODCIIndexStart(...., NUMBER)`— handles cases (1) and (3) which has a `NUMBER` value argument
- `ODCIIndexStart(...., VARCHAR2)`— handles case (2) which has a `VARCHAR2` value argument

- The query information in `qi` parameter can be used to optimize the domain index scan, if possible. The query information includes hints that have been specified for the query and the list of relevant ancillary operators referenced in the query block.

ODCIIndexTruncate

Syntax

ODCIIndexTruncate(ia ODCIIndexInfo, env ODCIEnv) RETURN NUMBER

Purpose

The `ODCIIndexTruncate` procedure is invoked when a `TRUNCATE` statement is issued against a table that has a domain index defined on one of its columns.

Table 17–14   ODCIIndexTruncate Arguments

Argument	Meaning
`ia`	Contains information about the indexed column
env	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error, or `ODCIConst.Warning`.

While truncating a local domain index, the first *N*+1 calls can return `ODCIConst.ErrContinue` too.

Usage Notes

- This function should be implemented as a static type method.
- After this function executes, the domain index should be empty (corresponding to the empty base table).
- While the `ODCIIndexTruncate` routine is being executed, the domain index is marked `LOADING`. If the `ODCIIndexTruncate` routine returns with an `ODCIConst.Error` (or exception), the domain index will be marked `FAILED`. The only operation permitted on `FAILED` domain indexes is `DROP INDEX`, `TRUNCATE TABLE` or `ALTER INDEX REBUILD`. If `ODCIIndexTruncate` returns with `ODCIConst.Warning`, the operation succeeds but a warning message is returned to the user.
- Every SQL statement executed by `ODCIIndexTruncate` is treated as an independent operation. The changes made by `ODCIIndexTruncate` are not guaranteed to be atomic.

- This method is invoked for truncating a non-partitioned index, truncating a local domain index, and also for truncating a single index partition during `ALTER TABLE TRUNCATE PARTITION`.

  For truncating a non-partitioned index, the `ODCIIndexTruncateis` invoked once, with the `IndexPartition`, `TablePartition` and `callProperty` set to `NULL`.

  For truncating a local domain index, the routine is invoked *N*+2 times, where *N* is the number of partitions.

  For truncating a single index partition during `ALTER TABLE TRUNCATE PARTITION`, this routine is invoked once with the `IndexPartition` and the `TablePartition` filled in and the `callProperty` set to `NULL`.

ODCIIndexUpdate

Syntax

ODCIIndexUpdate(ia ODCIIndexInfo, rid VARCHAR2, oldval <icoltype>, newval <icoltype>, env ODCIEnv)RETURN NUMBER

Purpose

This method is invoked when a row is updated in a table that has a domain index defined on one of its columns.

Table 17–15 ODCIIndexUpdate Arguments

Argument	Meaning
`ia`	Contains information about the index and the indexed column
`rid`	The row identifier of the updated row
`oldval`	The value of the indexed column before the update. The datatype is the same as that of the indexed column.
`newval`	The value of the indexed column after the update. The datatype is the same as that of the indexed column.
`env`	The environment handle passed to the routine

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error

Usage Notes

- The function should be implemented as a static type method.
- This method should update the tables/files storing the index data for the updated row.
- In addition to a SQL `UPDATE` statement, a LOB value can be updated through a variety of "`WRITE`" interfaces (see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*). If a domain index is defined on a `LOB` column or an object type containing a `LOB` attribute, the `ODCIIndexUpdate` routine is called when a LOB locator is implicitly or explicitly closed after one or more write operations.
- If `ODCIIndexUpdate` is invoked at the partition level, then the index partition name is filled in in the `ODCIIndexInfo` argument.

18 Reference: Extensible Optimizer Interface

This chapter describes the functions and procedures that comprise the interface to the extensible optimizer.

See Also: The *Oracle9i Supplied Java Packages Reference* for details on Java functionality

Note on the New Interfaces

In Oracle9*i*, the extensible optimizer interfaces have changed to support working with partitioned tables and domain indexes. The changes are of two kinds:

- Additional attributes have been added to some of the system-defined object types that are parameters to the `ODCIStats` interface methods. For example the `ODCIColInfo` type is enhanced to add information about the column's partition.
- Arguments or semantics of the arguments have changed for some `ODCIStats` methods. For example, the `ODCIStatsDelete` interface is changed to add an `OUT` argument to contain updated aggregate statistics.

You do not need to change your code unless you want to use the new functionality. You must, however, recompile your files and reload the shared library on the server machine, and you must not attempt to use the additional information being passed in any newly added system-type attributes.

If, on the other hand, you do want to use the new Oracle9*i* functionality, you must update your code for the new attributes added to the various system-defined types, and you must code for the new arguments added to various `ODCIStats` functions. You must also return `'SYS.ODCISTATS2'` in the `OUT` argument in the `ODCIGetInterfaces` routine. This tells the server to invoke the version of the `ODCIStats` methods that uses the new arguments.

Note that you must update your code for the Oracle9*i*, `ODCIStats2` version of the `ODCIStats` interfaces to use your statistics type with an indextype that implements the `ODCIIndex2` version of the extensible indexing interfaces.

The Extensible Optimizer Interface

EXPLAIN PLAN

`EXPLAIN PLAN` has been enhanced to show the user-defined CPU and I/O costs for domain indexes in the `CPU_COST` and `IO_COST` columns of `PLAN_TABLE`. For example, suppose we have a table `Emp_tab` and a user-defined operator `Contains`. Further, suppose that there is a domain index `EmpResume_indx` on the `Resume_col` column of `Emp_tab`, and that the indextype of `EmpResume_indx` supports the operator `Contains`. Then, the query

```
SELECT * FROM Emp_tab WHERE Contains(Resume_col, ’Oracle’) = 1
```

might have the following plan:

OPERATION	OPTIONS	OBJECT_NAME	CPU_COST	IO_COST
SELECT STATEMENT				
TABLE ACCESS	BY ROWID	EMP_TAB		
DOMAIN INDEX		EMPRESUME_INDX	300	4

INDEX Hint

The index hint will apply to domain indexes. In other words, the index hint will force the optimizer to use the hinted index for a user-defined operator, if possible.

ORDERED_PREDICATES Hint

The hint `ORDERED_PREDICATES` forces the optimizer to preserve the order of predicate evaluation (except predicates used for index keys) as specified in the `WHERE` clause of a SQL DML statement.

Example

Consider an example of how the statistics functions might be used. Suppose, in the schema `SCOTT`, we define the following:

```
CREATE OPERATOR Contains binding (VARCHAR2(4000), VARCHAR2(30))
   RETURN NUMBER USING Contains_fn;

CREATE TYPE stat1 (
   ...,
   STATIC FUNCTION ODCIStatsSelectivity(pred ODCIPredInfo, sel OUT NUMBER,
      args ODCIArgDescList, start NUMBER, stop NUMBER, doc VARCHAR2(4000),
      key VARCHAR2(30)) return NUMBER,
   STACTIC FUNCTION ODCIStatsFunctionCost(func ODCIFuncInfo, cost OUT
      ODCICost, args ODCIArgDescList, doc VARCHAR2(4000), key VARCHAR2(30))
      return NUMBER,
   STATIC FUNCTION ODCIStatsIndexCost(ia ODCIIndexInfo, sel NUMBER,
      cost OUT ODCICost, qi ODCIQueryInfo, pred ODCIPredInfo,
      args ODCIArgDescList, start NUMBER, stop NUMBER,
      key VARCHAR2(30)) return NUMBER,
   ...
);
```

```
CREATE TABLE T (resume VARCHAR2(4000));

CREATE INDEX T_resume on T(resume) INDEXTYPE IS indtype;

ASSOCIATE STATISTICS WITH FUNCTIONS Contains_fn USING stat1;

ASSOCIATE STATISTICS WITH INDEXES T_resume USING stat1;
```

When the optimizer encounters the query

```
SELECT * FROM T WHERE Contains(resume, 'ORACLE') = 1,
```

it will compute the selectivity of the predicate by invoking the user-defined selectivity function for the functional implementation of the `Contains` operator. In this case, the selectivity function is `stat1.ODCIStatsSelectivity`. It will be called as follows:

```
stat1.ODCIStatsSelectivity (
   ODCIPredInfo('SCOTT', 'Contains_fn', NULL, 29),
   sel,
   ODCIArgDescList(
      ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL),
      ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL),
      ODCIArgDesc(ODCIConst.ArgCol, 'T', 'SCOTT', '"resume"'),
      ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL)),
      1,
      1,
      NULL,
      'ORACLE')
```

Suppose the selectivity function returns a selectivity of 3 (percent). When the domain index is being evaluated, then the optimizer will call the user-defined index cost function as follows:

```
stat1.ODCIStatsIndexCost (
   ODCIIndexInfo('SCOTT', 'T_resume',
      ODCIColInfoList(ODCIColInfo('SCOTT', 'T', '"resume"', NULL, NULL))),
      3,
      cost,
      NULL,
      ODCIPredInfo('SCOTT', 'Contains', NULL, 13),
      ODCIArgDescList( ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL),
                       ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL),
                       ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL)),
      1,
      1,
     'ORACLE')
```

Suppose that the optimizer decides not to use the domain index because it is too expensive. Then it will call the user-defined cost function for the functional implementation of the operator as follows:

```
stat1.ODCIStatsFunctionCost (
   ODCIFuncInfo('SCOTT', 'Contains_fn', NULL, 1),
   cost,
   ODCIArgDescList(  ODCIArgDesc(ODCIConst.ArgCol, 'T', 'SCOTT', '"resume"'),
                     ODCIArgDesc(ODCIConst.ArgLit, NULL, NULL, NULL)),
   NULL,
   'ORACLE')
```

The following sections describe each statistics type function in greater detail.

User-Defined ODCIStats Functions

User-defined `ODCIStats` functions are used for table columns, functions, package, type, indextype or domain indexes. These functions are described in the following sections.

ODCIGetInterfaces

Syntax

ODCIGetInterfaces(ifclist OUT ODCIObjectList) RETURN NUMBER

Purpose

`ODCIGetInterfaces` is invoked by the server to discover which version of the `ODCIStats` interface the user has implemented in the methods of the user-defined statistics type.

Table 18–1 ODCIGetInterfaces Parameters

Parameter	Meaning
ifclist (OUT)	The version of the `ODCIStats` interfaces implemented by the statistics type. This value should be `'SYS.ODCISTATS2'`to specify the Oracle9*i* version.

Returns

`ODCIConst.Success` on success, `ODCIConst.Error` otherwise.

Usage Notes

Different versions of `ODCIStats` functions are used by Oracle8*i* and Oracle9*i*: the Oracle9*i* version adds parameters to some functions to support working with statistics on partitions of a table or domain index.

For the Oracle9*i* version of `ODCIStats` functions, `ODCIGetInterfaces` must return the string `'SYS.ODCISTATS2'` in the `ODCIObjectList` parameter. This value indicates that the statistics type uses the second version of the `ODCIStats` interface. Accordingly, the server invokes statistics, cost, or selectivity functions using the Oracle9*i* interface.

ODCIStatsCollect (Column)

Syntax

FUNCTION ODCIStatsCollect(col ODCIColInfo, options ODCIStatsOptions, statistics OUT RAW) return NUMBER

Purpose

`ODCIStatsCollect` is called by the `ANALYZE` command to collect user-defined statistics on a table or a partition of a table.

Table 18–2 ODCIStatsCollect Parameters

Parameter	Meaning
`col`	column for which statistics are being collected
`options`	options passed to `ANALYZE`
`statistics`	user-defined statistics collected

Returns

The function returns `ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning.`

Usage Notes

This function should be implemented as a static type method.

If statistics are being collected for only one partition, the `TablePartition` field in the `ODCIColInfo` type is filled in with the name of the partition. Otherwise (if statistics need to be collected for all the partitions or for the entire table), the `TablePartition` field is null.

If the `ANALYZE` command is executed to collect user-defined statistics on a partitioned table, then `n+1` `ODCIStatsCollect` calls are made, where `n` is the number of partitions in the table. The first `n` calls are made with the `TablePartition` attribute in `ODCIColInfo` filled in with the partition name and the `ODCIStatsOptions.CallProperty` set to `IntermediateCall`. The last call is made with `ODCIEnv.CallPropertyflag` set to `FinalCall` to allow you to collect aggregate statistics for the entire table. The `OUT` statistics in the first call are ignored by the server. The `OUT` statistics in the subsequent `n` calls are inserted into the `USTATS$` table corresponding to the partitions. The `OUT` statistics in the last call are the aggregate statistics for the table. The `ODCIColInfo.Partition` field is `NULL` in the first and last calls.

If user-defined statistics are being collected for only one partition of the table, two `ODCIStatsCollect` calls are made. In the first, you should collect statistics for the partition. For this call, the `TablePartition` attribute of the `ODCIColInfo` structure is filled in and the `ODCIEnv.CallProperty` is set to `FirstCall`. The statistics in the `OUT` arguments in the `ODCIStatsCollect` call are inserted into the `USTATS$` table corresponding to the partition.

In the second call you can update the aggregate statistics of the table based upon the new statistics collected for the partition. In this call, the `ODCIEnv.CallPropertyflag` is set to `FinalCall` to indicate that it is the second call. If you do not want to modify the aggregate statistics, read the aggregate statistics of the table from the catalog and pass that back in the statistics field as the `OUT` argument. Whatever value is present in the statistics argument is written in the `USTATS$` by the server. The `ODCIColInfo.TablePartition` is filled in with the partition name in both the calls.

Return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call to indicate that you are using the Oracle9*i* version of the `ODCISTATS` interface, that supports partitioning.

ODCIStatsCollect (Index)

Syntax

FUNCTION ODCIStatsCollect(ia ODCIIndexInfo, options ODCIStatsOptions, statistics OUT RAW) return NUMBER

Purpose

`ODCIStatsCollect` is called by the `ANALYZE INDEX` command to collect user-defined statistics on an index or a partition of an index.

Table 18–3 ODCIStatsCollect Parameters

Parameter	Meaning
`ia`	domain index for which statistics are being collected
`options`	options passed to `ANALYZE`
`statistics`	user-defined statistics collected

Returns

The function returns `ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning.`

Usage Notes

This function should be implemented as a static type method.

If statistics are being collected for the entire partitioned index, the `IndexPartition` field is null, and `n`+2 calls are made to the `ODCIStatsCollect` function. This scenario is similar to that described for the column version of ODCIStatsCollect on page 18-7.

If the statistics are being collected for a single partition of the index, the `IndexPartition` field contains the name of the partition, and two calls are made to the `ODCIStatsCollect` function. The first call is made to obtain the statistics for the index partition, and the second call is made to obtain the aggregate statistics for the domain index.

To collect statistics on a non-partitioned domain index only a single call is made to the `ODCIStatsCollect` function.

Return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call to indicate that you are using the Oracle9*i* version of the `ODCISTATS` interface, that supports partitioning.

ODCIStatsDelete (Column)

Syntax

FUNCTION ODCIStatsDelete(col ODCIColInfo, statistics OUT RAW, env ODCIEnv) return NUMBER

Purpose

`ODCIStatsDelete` is called by the `ANALYZE` *`<table>`* `DELETE STATISTICS` command to delete user-defined statistics on a table or a partition of a table.

Table 18–4 ODCIStatsDelete Parameters

Parameter	Meaning
`col`	Column for which statistics are being deleted
`statistics OUT`	Contains table-level aggregate statistics for a partitioned table
`env`	Contains information about how many times the function has been called by the server

Returns

`ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning.`

Usage Notes

This function should be implemented as a static method.

When the function is called for a non-partitioned table, the `statistics` argument in the `ODCIStatsDelete` interface is ignored.

If the statistics are being deleted for a partitioned table, the `ODCIStatsDelete` is called `n+1` times. The first `n` calls are with the partition name filled in in the `ODCIColInfo` structure and the `ODCIEnv.CallProperty` set to `IntermediateCall`. The last call is made with the `ODCIEnv.CallProperty` set to `FinalCall`.

The order of operations that you must perform for a delete are the inverse of what you do to collect statistics: In the first call, delete the table-level statistics from your statistics tables; in the intermediate n calls, delete the statistics for the specific partitions; and in the last call drop or clean up any structures created for holding statistics for the deleted table. The `ODCIColInfo.TablePartition` is set to null in the first and last calls. In the intermediate n calls, the `TablePartition` field is filled in.

If statistics are being deleted for only one partition, two `ODCIStatsDelete` calls are made. In each call, `ODCIColInfo.TablePartition` is filled in with the partition name. On the first call, delete any user-defined statistics collected for that partition. On the second call, update the aggregate statistics for the table and return these aggregate statistics as an `OUT` argument.

Return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call to indicate that you are using the Oracle9*i* version of the `ODCISTATS` interface, that supports partitioning.

ODCIStatsDelete (Index)

Syntax

FUNCTION ODCIStatsDelete(ia ODCIIndexInfo, statistics OUT RAW, env ODCIEnv) return NUMBER

Purpose

`ODCIStatsDelete` is called by the `ANALYZE` *<table>* `DELETE STATISTICS` command to delete user-defined statistics on an index or a partition of an index.

Table 18–5 ODCIStatsDelete Parameters

Parameter	Meaning
`ia`	Domain index for which statistics are being deleted
`statistics OUT`	Contains aggregate statistics for a partitioned index
`env`	Contains information about how many times the function has been called by the server

Returns

`ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning.`

Usage Notes

This function should be implemented as a static method.

When the function is called for a non-partitioned index, the `statistics` argument in the `ODCIStatsDelete` interface is ignored.

If statistics are being deleted for a partitioned index, `ODCIStatsDelete` is called `n`+2 times. The first and the last call are made with the `ODCIEnv.CallProperty` set to `FirstCall` and `FinalCall` respectively and do not have the partition name set in the `ODCIIndexInfo` type. The intermediate n calls are made with the partition name filled in in the `ODCIIndexInfo` structure and the `ODCIEnv.CallProperty` set to `IntermediateCall`.

The order of operations that you must perform to delete statistics are the inverse of what you do to collect statistics: In the first call, delete the index-level statistics from your statistics tables; in the intermediate n calls, delete the statistics for the specific partitions; and in the last call drop or clean up any structures created for holding the deleted statistics. The `ODCIIndexInfo.IndexPartition` is set to null in the first and last calls. In the intermediate n calls, the `IndexPartition` field is filled in.

If statistics are being deleted for only one partition, two `ODCIStatsDelete` calls are made. In each call, `ODCIIndexInfo.IndexPartition` is filled in with the partition name. On the first call, delete any user-defined statistics collected for that partition. On the second call, update the aggregate statistics for the index and return these aggregate statistics as an `OUT` argument.

Return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call to indicate that you are using the Oracle9*i* version of the `ODCISTATS` interface, that supports partitioning.

ODCIStatsFunctionCost

Syntax

FUNCTION ODCIStatsFunctionCost(func ODCIFuncInfo, cost OUT ODCICost, args ODCIArgDescList, <list of function arguments>) return NUMBER

Purpose

Computes the cost of a function.

Table 18–6 ODCIStatsFunctionCost Parameters

Parameter	Meaning
`func`	Function or type method for which the cost is being computed
`cost`	Computed cost (must be positive whole numbers)
`args`	Descriptor of actual arguments with which the function or type method was called. If the function has *n* arguments, the `args` array will contain *n* elements, each describing the actual arguments of the function or type method
`<list of function arguments>`	List of actual parameters to the function or type method; the number, position, and type of each argument must be the same as in the function or type method

Returns

`ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning.`

Usage Notes

This function should be implemented as a static type method.

ODCIStatsIndexCost

Syntax

FUNCTION ODCIStatsIndexCost(ia ODCIIndexInfo, sel NUMBER, cost OUT ODCICost, qi ODCIQueryInfo, pred ODCIPredInfo, args ODCIArgDescList, start <operator_return_type>, stop <operator_return_type>, <list of operator arguments>, env ODCIEnv) return NUMBER

Purpose

Calculates the cost of a domain index scan—either a scan of the entire index or a scan of one or more index partitions if a local domain index has been built.

For each table in the query, the optimizer uses partition pruning to determine the range of partitions that may be accessed. These partitions are called **interesting partitions**. The set of interesting partitions for a table is also the set of interesting partitions for all domain indexes on that table. The cost of a domain index can depend on the set of interesting partitions, so the optimizer passes a list of interesting index partitions to `ODCIStatsIndexCost` in the `args` argument (the type of this argument, `ODCIArgDescList`, is a list of `ODCIArgDesc` argument descriptor types) for those arguments that are columns. For non-partitioned domain indexes or for cases where no partition pruning is possible, no partition list is passed to `ODCIStatsIndexCost`, and you should assume that the entire index will be accessed.

The domain index key can contain multiple column arguments (for example, the indexed column and column arguments from other tables appearing earlier in a join order). For each column appearing in the index key, the `args` argument contains the list of interesting partitions for the table. For example, for an index key

```
op(T1.c1, T2.c2) = 1
```

the optimizer passes a list of interesting partitions for tables `T1` and `T2` if they are partitioned and there is partition pruning for them.

Table 18–7   ODCIStatsIndexCost Parameters

Parameter	Meaning
`ia`	domain index for which statistics are being collected
`sel`	the user-computed selectivity of the predicate
`cost`	computed cost (must be positive whole numbers)

Table 18–7 ODCIStatsIndexCost Parameters (Cont.)

Parameter	Meaning
`qi`	Information about the query
`pred`	Information about the predicate
`args`	Descriptor of `start`, `stop`, and actual value arguments with which the operator was called. If the operator has *n* arguments, the `args` array will contain *n+1* elements, the first element describing the start value, the second element describing the stop value, and the remaining *n-1* elements describing the actual value arguments of the operator (that is, the arguments after the first)
`start`	Lower bound of the operator (for example, 2 for a predicate `fn(...) > 2`)
`stop`	Upper bound of the operator (for example, 5 for a predicate `fn(...) < 5`)
`<list of function arguments>`	List of actual parameters to the operator (excluding the first); the number, position, and type of each argument must be the same as in the operator
`env`	Contains general information about the environment in which the routine is executing

Returns

`ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning`

Usage Notes

- This function should be implemented as a static type method.
- Only a single call is made to the `ODCIIndexCost` function for queries on partitioned or non-partitioned tables. For queries on partitioned tables, additional information is passed in the `ODCIIndexCost` function. Note that some partitions in the list passed to `ODCIStatsIndexCost` may not actually be accessed by the query. The list of interesting partitions chiefly serves to exclude partitions that definitely will not be accessed.
- When the `ODCIIndexCost` function is invoked, users can fill in a string in the `IndexCostInfo` field of the cost attribute to supply any additional information that might be helpful. The string (255 characters maximum) is displayed in the `OPTIONS` column in the `EXPLAIN PLAN` output when an execution plan chooses a domain index scan.

- Users implementing this function must return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call.

ODCIStatsSelectivity

Syntax

FUNCTION ODCIStatsSelectivity(pred ODCIPredInfo, sel OUT NUMBER, args ODCIArgDescList, start <function_return_type>, stop <function_return_type>, <list of function arguments>, env ODCIEnv) return NUMBER

Purpose

This function specifies the selectivity of a predicate. The selectivity of a predicate involving columns from a single table is the fraction of rows of that table that satisfy the predicate. For predicates involving columns from multiple tables (for example, join predicates), the selectivity should be computed as a fraction of rows in the Cartesian product of those tables.

As in `ODCIStatsIndexCost`, the args argument contains a list of *interesting* partitions for the tables whose columns are referenced in the predicate for which the selectivity has to be computed. These interesting partitions are partitions that cannot be eliminated by partition pruning as possible candidates to be accessed. The set of interesting partitions is passed to the function only if partition pruning has occurred (in other words, the interesting partitions are a strict subset of all the partitions).

For example, when `ODCIStatsSelectivity` is called to compute the selectivity of the predicate:

```
f(T1.c1, T2.c2) > 4
```

the optimizer passes the list of interesting partitions for the table `T1` (in the argument descriptor for column `T1.c1`) if partition pruning is possible; similarly for the table `T2`.

If a predicate contains columns from more than one table, this information is indicated by the flag bit `PredMultiTable` (new in Oracle9*i*) set in the `Flags` attribute of the `pred` argument.

Table 18–8 ODCIStatsSelectivity Parameters

Parameter	Meaning
`pred`	Predicate for which the selectivity is being computed

Table 18–8 ODCIStatsSelectivity Parameters (Cont.)

Parameter	Meaning
`sel`	The computed selectivity, expressed as a number between (and including) 0 and 100, representing a percentage.
args	Descriptor of `start`, `stop`, and actual arguments with which the function, type method, or operator was called. If the function has *n* arguments, the `args` array will contain *n+2* elements, the first element describing the start value, the second element describing the stop value, and the remaining *n* elements describing the actual arguments of the function, method, or operator
start	Lower bound of the function (for example, 2 for a predicate `fn(...)` > 2)
stop	Upper bound of the function (for example, 5 for a predicate `fn(...)` < 5)
`<list of function arguments>`	List of actual parameters to the function or type method; the number, position, and type of each argument must be the same as in the function, type method, or operator
`env`	Contains general information about the environment in which the routine is executing

Returns

`ODCIConst.Success`, `ODCIConst.Error`, or `ODCIConst.Warning`

Usage Notes

- This function should be implemented as a static type method.
- Users implementing this interface must return `'SYS.ODCISTATS2'` in the `ODCIGetInterfaces` call.
- The selectivity of a predicate involving columns from a single table is the fraction of rows of that table that satisfy the predicate. For predicates involving columns from multiple tables (for example, join predicates), the selectivity should be computed as a fraction of rows in the Cartesian product of those tables. For tables with partition pruning, the selectivity should be expressed relative to the cardinalities of the interesting partitions of the tables involved.

  The selectivity of predicates involving columns on partitioned tables is computed relative to the rows in the interesting partitions. Thus, the selectivity of the predicate

```
g(T1.c1) < 5
```

is the percentage of rows in the set of interesting partitions (or all partitions if no partition pruning is possible) that satisfies this predicate. For predicates with columns from multiple tables, the selectivity must be relative to the number of rows in the cartesian product of the tables.

For example, consider the predicate:

```
f(T1.c1, T2.c2) > 4
```

Suppose that the number of rows in the interesting partitions is 1000 for `T1` and 5000 for `T2`. The selectivity of this predicate must be expressed as the percentage of the 5,000,000 rows in the Cartesian product of `T1` and `T2` that satisfy the predicate.

- If a predicate contains columns from more than one table, this information is indicated by the flag bit `PredMultiTable` (new in Oracle9*i*) set in the `Flags` attribute of the `pred` argument.
- A selectivity expressed relative to the base cardinalities of the tables involved may be only an approximation of the true selectivity if cardinalities (and other statistics) of the tables have been reduced based on single-table predicates or other joins earlier in the join order. However, this approximation to the true selectivity should be acceptable to most applications.
- Only one call is made to the `ODCIStatsSelectivity` function for queries on partitioned or non-partitioned tables. In the case of queries on partitioned tables, additional information is passed while calling the `ODCIStatsSelectivity` function.

19 Reference: User-Defined Aggregates Interface

This chapter describes the routines that need to be implemented to define a user-defined aggregate function. The routines are implemented as methods in an object type. Then the `CREATE FUNCTION` statement is used to actually create the aggregate function.

See Also: Chapter 11, "User-Defined Aggregate Functions"

ODCIAggregateInitialize

Syntax

STATIC FUNCTION ODCIAggregateInitialize(actx IN OUT <impltype>) RETURN NUMBER

Purpose

The `ODCIAggregateInitialize` function is invoked by Oracle as the first step of aggregation. This function typically initializes the aggregation context (an instance of the implementation object type) and returns it (as an OUT parameter) to Oracle.

Table 19–1 ODCIAggregateInitialize Parameters

Parameter	Meaning
actx (IN OUT)	The aggregation context that is initialized by the routine. Its value will be null for regular aggregation cases. In aggregation over windows, `actx` is the context of the previous window. This object instance is passed in as a parameter to the next aggregation routine.

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

Implement this routine as a static method.

ODCIAggregateIterate

Syntax

MEMBER FUNCTION ODCIAggregateIterate(self IN OUT <impltype>, val <inputdatatype>) RETURN NUMBER

Purpose

The `ODCIAggregateIterate` function is invoked by Oracle to process the next input row. The routine is invoked by passing in the aggregation context and the value of the next input to be aggregated. This routine processes the input value, updates the aggregation context accordingly, and returns the context back to Oracle. This routine is invoked by Oracle for every value in the underlying group, including NULL values.

Table 19–2 ODCIAggregateIterate Parameters

Parameter	Meaning
self (IN)	The value of the current aggregation context
self (OUT)	The updated aggregation context returned to Oracle
val (IN)	The input value to be aggregated

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

This is a mandatory routine and is implemented as a member method.

ODCIAggregateMerge

Syntax

MEMBER FUNCTION ODCIAggregateMerge(self IN OUT <impltype>, ctx2 IN <impltype>) RETURN NUMBER

Purpose

The `ODCIAggregateMerge` function is invoked by Oracle to merge two aggregation contexts into a single object instance. Two aggregation contexts may need to be merged during either serial or parallel evaluation of the user-defined aggregate. This function takes the two aggregation contexts as input, merges them, and returns the single, merged instance of the aggregation context.

Table 19–3 ODCIAggregateMerge Parameters

Parameter	Meaning
self (IN)	The value of one aggregation context
ctx2 (IN)	The value of the other aggregation context
self (OUT)	The single, merged aggregation context returned to Oracle

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

This is a mandatory routine and is implemented as a member method.

ODCIAggregateTerminate

Syntax

MEMBER FUNCTION ODCIAggregateTerminate(self IN <impltype>, ReturnValue OUT <return_type>, flags IN number) RETURN NUMBER

Purpose

The `ODCIAggregateTerminate` function is invoked by Oracle as the final step of aggregation. This routine takes the aggregation context as input and returns the resultant aggregate value to Oracle. This routine also typically performs any necessary cleanup operations such as freeing memory, and so on.

Table 19–4 ODCIAggregateTerminate Parameters

Parameter	Meaning
self (IN)	The value of the aggregation context
ReturnValue (OUT)	The resultant aggregate value
flags (IN)	A bit vector that indicates various options. A set bit of `ODCI_AGGREGATE_REUSE_CTX` indicates that the context will be reused and that therefore any external context should not be freed. (See "Reusing the Aggregation Context for Analytic Functions" on page 11-10 for information on using this flag.)

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

This is a mandatory routine and is implemented as a member method.

ODCIAggregateDelete

Syntax

MEMBER FUNCTION ODCIAggregateDelete(self IN OUT <impltype>, val <inputdatatype>) RETURN NUMBER

Purpose

The `ODCIAggregateDelete` function is invoked by Oracle to remove an input value from the current group. The routine is invoked by passing in the aggregation context and the value of the input to be removed. The routine processes the input value, updates the aggregation context accordingly, and returns the context to Oracle. This routine is invoked by Oracle during computation of user-defined aggregates with analytic (windowing) functions.

Table 19–5 ODCIAggregateDelete Parameters

Parameter	Meaning
self (IN)	The value of the current aggregation context
self (OUT)	The updated aggregation context returned to Oracle
val (IN)	The input value to be removed from the current group

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

This is an optional routine and is implemented as a member method.

ODCIAggregateWrapContext

Syntax

MEMBER FUNCTION ODCIAggregateWrapContext(self IN OUT <impltype>) RETURN NUMBER

Purpose

The `ODCIAggregateWrapContext` function is invoked by Oracle if the user-defined aggregate has been declared to have external context and is transmitting partial aggregates from slave processes.

This routine must integrate all external pieces of the current aggregation context to make the context self-contained.

> **See Also:** "Handling Large Aggregation Contexts" on page 11-7 for more information on using this function

Table 19–6 ODCIAggregateWrapContext Parameters

Parameter	Meaning
self (IN)	The value of the current aggregation context
self (OUT)	The updated and self-contained aggregation context returned to Oracle

Returns

`ODCIConst.Success` on success, or `ODCIConst.Error` on error.

Usage Notes

This is an optional routine and is implemented as a member method.

20 Reference: Pipelined and Parallel Table Functions

This chapter describes the routines that need to be implemented to define pipelined and parallel table functions in C.

See Also: Chapter 12 for an overall explanation of pipelined and parallel table functions

ODCITableStart

Syntax

STATIC FUNCTION ODCITableStart(sctx OUT <imptype>, <args>) RETURN NUMBER

Purpose

`ODCITableStart` initializes the scan of a table function.

Table 20–1 ODCITableStart Parameters

Parameter	Meaning
sctx (OUT)	The scan context returned by this routine. This value is passed in as a parameter to the later scan routines. The scan context is an instance of the object type containing the implementation of the `ODCITable` routines.
args (IN)	Set of zero or more arguments specified by the user for the table function

Returns

`ODCIConst.Success` on success, `ODCIConst.Error` otherwise.

Usage Notes

- This is the first routine that is invoked to begin retrieving rows from a table function. This routine typically performs the setup needed for the scan. The scan context is created (as an object instance `sctx`) and returned to Oracle. The arguments to the table function, specified by the user in the `SELECT` statement, are passed in as parameters to this routine.
- Any `REF CURSOR` arguments of the table function must be declared as `SYS_REFCURSOR` type in the declaration of the `ODCITableStart` method.

ODCITableFetch

Syntax

MEMBER FUNCTION ODCITableFetch(self IN OUT <imptype>, nrows IN NUMBER, rws OUT <coll-type>) RETURN NUMBER

Purpose

ODCITableFetch returns the next batch of rows from a table function.

Table 20–2 ODCITableFetch Parameters

Parameter	Meaning
self (IN)	The current scan context. This is the object instance returned to Oracle by the previous invocation of the scan routine.
self (OUT)	The scan context to be passed to later scan routine invocations.
nrows (IN)	The number of rows the system expects in the current fetch cycle. The method can ignore this value and return a different number of rows. If fewer rows are returned, the method is called again; if more rows are returned, they are processed in the next cycle.
rws (OUT)	The next batch of rows from the table function. This is returned as an instance of the same collection type as the return type of the table function.

Returns

`ODCIConst.Success` on success, `ODCIConst.Error` otherwise.

Usage Notes

`ODCITableFetch` is invoked one or more times by Oracle to retrieve all the rows in the collection returned by the table function. The scan context is passed in as a parameter. Typically `ODCITableFetch` uses the input scan context and computes the next set of rows to be returned to Oracle. In addition, it may update the scan context accordingly.

Returning more rows in each invocation of `fetch()` reduces the number of fetch calls that need to be made and thus improves performance.

Oracle calls `ODCITableFetch` repeatedly until all rows in the table function's collection have been returned. When all rows have been returned, `ODCITableFetch` should return a null collection.

ODCITableClose

Syntax

```
MEMBER FUNCTION ODCITableClose(self IN <imptype>) RETURN NUMBER
```

Purpose

ODCITableClose performs cleanup operations after scanning a table function.

Table 20–3 ODCITableClose Parameters

Parameter	Meaning
self (IN)	The scan context set up by previous scan routine invocation

Returns

`ODCIConst.Success` on success, `ODCIConst.Error` otherwise.

Usage Notes

Oracle invokes ODCITableClose after the last fetch call. The scan context is passed in as a parameter. ODCITableClose then performs any necessary cleanup operations, such as freeing memory.

ODCITableDescribe

Syntax

STATIC FUNCTION ODCITableDescribe(rtype OUT ANYTYPE, <args>) RETURN NUMBER

Purpose

`ODCITableDescribe` returns describe information for a table function whose return type is ANYDATASET.

Table 20–4 ODCITableDescribe Parameters

Parameter	Meaning
rtype (OUT)	The AnyType value that describes the returned rows from the table function
args (IN)	The set of zero or more arguments specified by the user for the table function.

Returns

`ODCIConst.Success` on success, `ODCIConst.Error` otherwise.

Usage Notes

Oracle invokes `ODCITableDescribe` at query compilation time to retrieve the specific type information.

Note that this interface is applicable only for table functions whose return type is ANYDATASET. The format of elements within the returned collection is conveyed to Oracle by returning an instance of ANYTYPE. The ANYTYPE instance specifies the actual structure of the returned rows in the context of the specific query.

ANYTYPE provides a datatype to model the metadata of a row—the names and datatypes of all the columns (fields) comprising the row. It also provides a set of PL/SQL and C interfaces for users to construct and access the metadata information. ANYDATASET, like ANYTYPE, contains a description of a given type, but ANYDATASET also contains a set of data instances of that type

See Also: "Transient and Generic Types" in Chapter 12 for a discussion of ANYTYPE, ANYDATA, and ANYDATASET

The following example shows a query on a table function that uses the ANYDATASET type:

```
SELECT * FROM
TABLE(CAST(AnyBooks('http://.../books.xml') AS ANYDATASET));
```

At query compilation time, Oracle invokes the ODCITableDescribe routine. The routine typically uses the user arguments to figure out the nature of the return rows. In this example, ODCITableDescribe consults the DTD of the XML documents at the specified location to determine the appropriate ANYTYPE value to return. Each ANYTYPE instance is constructed by invoking the constructor APIs with this field name and datatype information.

Any arguments of the table function which are not constants are passed to ODCITableDescribe as NULLs because their values are not known at compile time.

A
Example: Pipelined Table Functions: Interface Approach

This appendix supplements the discussion of table functions in Chapter 12. The appendix shows two complete implementations of the `StockPivot` table function using the interface approach. One implementation is done in C and one in Java.

The function `StockPivot` converts a row of the type `(Ticker, OpenPrice, ClosePrice)` into two rows of the form `(Ticker, PriceType, Price)`. For example, from an input row `("ORCL", 41, 42)`, the table function returns the two rows `("ORCL", "O", 41)` and `("ORCL", "C", 42)`.

C Implementation

In this example, the three `ODCITable` interface methods of the implementation type are implemented as external functions in C. The code to implement these methods is shown after the following SQL declarations:

SQL Declarations for C Implementation

```
-- Create the input stock table

CREATE TABLE StockTable (
  ticker VARCHAR(4),
  open_price NUMBER,
  close_price NUMBER
);

-- Create the types for the table function's output collection
-- and collection elements
```

```
CREATE TYPE TickerType AS OBJECT
(
  ticker VARCHAR2(4),
  PriceType VARCHAR2(1),
  price NUMBER
);
/

CREATE TYPE TickerTypeSet AS TABLE OF TickerType;
/

-- Create the external library object

CREATE LIBRARY StockPivotLib IS '/home/bill/libstock.so';
/

-- Create the implementation type

CREATE TYPE StockPivotImpl AS OBJECT
(
  key RAW(4),

  STATIC FUNCTION ODCITableStart(sctx OUT StockPivotImpl, cur SYS_REFCURSOR)
  RETURN PLS_INTEGER
    AS LANGUAGE C
    LIBRARY StockPivotLib
    NAME "ODCITableStart"
    WITH CONTEXT
    PARAMETERS (
      context,
      sctx,
      sctx INDICATOR STRUCT,
      cur,
      RETURN INT
    ),

  MEMBER FUNCTION ODCITableFetch(self IN OUT StockPivotImpl, nrows IN NUMBER,
                               outSet OUT TickerTypeSet) RETURN PLS_INTEGER
    AS LANGUAGE C
    LIBRARY StockPivotLib
    NAME "ODCITableFetch"
    WITH CONTEXT
    PARAMETERS (
      context,
      self,
      self INDICATOR STRUCT,
      nrows,
      outSet,
      outSet INDICATOR,
      RETURN INT
    ),

  MEMBER FUNCTION ODCITableClose(self IN StockPivotImpl) RETURN PLS_INTEGER
    AS LANGUAGE C
    LIBRARY StockPivotLib
    NAME "ODCITableClose"
    WITH CONTEXT
    PARAMETERS (
      context,
      self,
      self INDICATOR STRUCT,
      RETURN INT
    )

);
/

-- Define the ref cursor type

CREATE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
/

-- Create table function

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED USING StockPivotImpl;
/
```

C Implementation of the ODCITable Methods

The following code implements the three `ODCITable` methods as external functions in C:

```
#ifndef OCI_ORACLE
# include <oci.h>
#endif
```

```
#ifndef ODCI_ORACLE
# include <odci.h>
#endif

/*---------------------------------------------------------------------------
                     PRIVATE TYPES AND CONSTANTS
  ---------------------------------------------------------------------------*/

/* The struct holding the user's stored context */

struct StoredCtx
{
  OCIStmt* stmthp;
};
typedef struct StoredCtx StoredCtx;

/* OCI Handles */

struct Handles_t
{
  OCIExtProcContext* extProcCtx;
  OCIEnv* envhp;
  OCISvcCtx* svchp;
  OCIError* errhp;
  OCISession* usrhp;
};
typedef struct Handles_t Handles_t;

/********************** SQL Types C representation **********************/

/* Table function's implementation type */

struct StockPivotImpl
{
  OCIRaw* key;
};
typedef struct StockPivotImpl StockPivotImpl;

struct StockPivotImpl_ind
{
  short _atomic;
  short key;
};
typedef struct StockPivotImpl_ind StockPivotImpl_ind;
```

```
/* Table function's output collection element type */

struct TickerType
{
  OCIString* ticker;
  OCIString* PriceType;
  OCINumber price;
};
typedef struct TickerType TickerType;

struct TickerType_ind
{
  short _atomic;
  short ticker;
  short PriceType;
  short price;
};
typedef struct TickerType_ind TickerType_ind;

/* Table function's output collection type */

typedef OCITable TickerTypeSet;

/*--------------------------------------------------------------------------*/
/* Static Functions */
/*--------------------------------------------------------------------------*/

static int GetHandles(OCIExtProcContext* extProcCtx, Handles_t* handles);

static StoredCtx* GetStoredCtx(Handles_t* handles, StockPivotImpl* self,
                               StockPivotImpl_ind* self_ind);

static int checkerr(Handles_t* handles, sword status);

/*--------------------------------------------------------------------------*/
/* Functions definitions */
/*--------------------------------------------------------------------------*/

/* Callout for ODCITableStart */

int ODCITableStart(OCIExtProcContext* extProcCtx, StockPivotImpl*  self,
                   StockPivotImpl_ind* self_ind, OCIStmt** cur)
{
  Handles_t handles;                   /* OCI hanldes */
  StoredCtx* storedCtx;                /* Stored context pointer */
  ub4 key;                             /* key to retrieve stored context */

  /* Get OCI handles */
  if (GetHandles(extProcCtx, &handles))
    return ODCI_ERROR;

  /* Allocate memory to hold the stored context */
  if (checkerr(&handles, OCIMemoryAlloc((dvoid*) handles.usrhp, handles.errhp,
                                       (dvoid**) &storedCtx,
                                       OCI_DURATION_STATEMENT,
                                       (ub4) sizeof(StoredCtx),
                                       OCI_MEMORY_CLEARED)))
    return ODCI_ERROR;

  /* store the input ref cursor in the stored context */
  storedCtx->stmthp=*cur;

  /* generate a key */
  if (checkerr(&handles, OCIContextGenerateKey((dvoid*) handles.usrhp,
                                              handles.errhp, &key)))
    return ODCI_ERROR;

  /* associate the key value with the stored context address */
  if (checkerr(&handles, OCIContextSetValue((dvoid*)handles.usrhp,
                                           handles.errhp,
                                           OCI_DURATION_STATEMENT,
                                           (ub1*) &key, (ub1) sizeof(key),
                                           (dvoid*) storedCtx)))
    return ODCI_ERROR;

  /* stored the key in the scan context */
  if (checkerr(&handles, OCIRawAssignBytes(handles.envhp, handles.errhp,
                                          (ub1*) &key, (ub4) sizeof(key),
                                          &(self->key))))
    return ODCI_ERROR;

  /* set indicators of the scan context */
  self_ind->_atomic = OCI_IND_NOTNULL;
  self_ind->key = OCI_IND_NOTNULL;

  return ODCI_SUCCESS;
}
```

```
/***********************************************************************/

/* Callout for ODCITableFetch */

int ODCITableFetch(OCIExtProcContext* extProcCtx, StockPivotImpl* self,
                   StockPivotImpl_ind* self_ind, OCINumber* nrows,
                   TickerTypeSet** outSet, short* outSet_ind)
{
  Handles_t handles;                   /* OCI hanldes */
  StoredCtx* storedCtx;                /* Stored context pointer */
  int nrowsval;                        /* number of rows to return */

  /* Get OCI handles */
  if (GetHandles(extProcCtx, &handles))
    return ODCI_ERROR;

  /* Get the stored context */
  storedCtx=GetStoredCtx(&handles,self,self_ind);
  if (!storedCtx) return ODCI_ERROR;

  /* get value of nrows */
  if (checkerr(&handles, OCINumberToInt(handles.errhp, nrows, sizeof(nrowsval),
                                        OCI_NUMBER_SIGNED, (dvoid *)&nrowsval)))
    return ODCI_ERROR;

  /* return up to 10 rows at a time */
  if (nrowsval>10) nrowsval=10;

  /* Initially set the output to null */
  *outSet_ind=OCI_IND_NULL;

  while (nrowsval>0)
  {

    TickerType elem;           /* current collection element */
    TickerType_ind elem_ind;   /* current element indicator */

    OCIDefine* defnp1=(OCIDefine*)0;   /* define handle */
    OCIDefine* defnp2=(OCIDefine*)0;   /* define handle */
    OCIDefine* defnp3=(OCIDefine*)0;   /* define handle */

    sword status;

    char ticker[5];
    float open_price;
    float close_price;
    char PriceType[2];

    /* Define the fetch buffer for ticker symbol */
    if (checkerr(&handles, OCIDefineByPos(storedCtx->stmthp, &defnp1,
                                          handles.errhp, (ub4) 1,
                                          (dvoid*) &ticker,
                                          (sb4) sizeof(ticker),
                                          SQLT_STR, (dvoid*) 0, (ub2*) 0,
                                          (ub2*) 0, (ub4) OCI_DEFAULT)))
      return ODCI_ERROR;

    /* Define the fetch buffer for open price */
    if (checkerr(&handles, OCIDefineByPos(storedCtx->stmthp, &defnp2,
                                          handles.errhp, (ub4) 2,
                                          (dvoid*) &open_price,
                                          (sb4) sizeof(open_price),
                                          SQLT_FLT, (dvoid*) 0, (ub2*) 0,
                                          (ub2*) 0, (ub4) OCI_DEFAULT)))
      return ODCI_ERROR;

    /* Define the fetch buffer for closing price */
    if (checkerr(&handles, OCIDefineByPos(storedCtx->stmthp, &defnp3,
                                          handles.errhp, (ub4) 3,
                                          (dvoid*) &close_price,
                                          (sb4) sizeof(close_price),
                                          SQLT_FLT, (dvoid*) 0, (ub2*) 0,
                                          (ub2*) 0, (ub4) OCI_DEFAULT)))
      return ODCI_ERROR;

    /* fetch a row from the input ref cursor */
    status = OCIStmtFetch(storedCtx->stmthp, handles.errhp, (ub4) 1,
                          (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);

    /* finished if no more data */
    if (status!=OCI_SUCCESS && status!=OCI_SUCCESS_WITH_INFO) break;

    /* Initialize the element indicator struct */

    elem_ind._atomic=OCI_IND_NOTNULL;
    elem_ind.ticker=OCI_IND_NOTNULL;
    elem_ind.PriceType=OCI_IND_NOTNULL;
    elem_ind.price=OCI_IND_NOTNULL;

    /* assign the ticker name */
```

```
elem.ticker=NULL;
if (checkerr(&handles, OCIStringAssignText(handles.envhp, handles.errhp,
                                           (text*) ticker,
                                           (ub2) strlen(ticker),
                                           &elem.ticker)))
  return ODCI_ERROR;

/* assign the price type */
elem.PriceType=NULL;
sprintf(PriceType,"O");
if (checkerr(&handles, OCIStringAssignText(handles.envhp, handles.errhp,
                                           (text*) PriceType,
                                           (ub2) strlen(PriceType),
                                           &elem.PriceType)))
  return ODCI_ERROR;

/* assign the price */
if (checkerr(&handles, OCINumberFromReal(handles.errhp, &open_price,
                                         sizeof(open_price), &elem.price)))
  return ODCI_ERROR;

/* append element to output collection */
if (checkerr(&handles, OCICollAppend(handles.envhp, handles.errhp,
                                     &elem, &elem_ind, *outSet)))
  return ODCI_ERROR;

/* assign the price type */
elem.PriceType=NULL;
sprintf(PriceType,"C");
if (checkerr(&handles, OCIStringAssignText(handles.envhp, handles.errhp,
                                           (text*) PriceType,
                                           (ub2) strlen(PriceType),
                                           &elem.PriceType)))
  return ODCI_ERROR;

/* assign the price */
if (checkerr(&handles, OCINumberFromReal(handles.errhp, &close_price,
                                         sizeof(close_price), &elem.price)))
  return ODCI_ERROR;

/* append row to output collection */
if (checkerr(&handles, OCICollAppend(handles.envhp, handles.errhp,
                 &elem, &elem_ind, *outSet)))
  return ODCI_ERROR;
```

```
    /* set collection indicator to not null */
    *outSet_ind=OCI_IND_NOTNULL;

    nrowsval-=2;
  }

  return ODCI_SUCCESS;
}

/***********************************************************************/

/* Callout for ODCITableClose */

int ODCITableClose(OCIExtProcContext* extProcCtx, StockPivotImpl* self,
                   StockPivotImpl_ind* self_ind)
{
  Handles_t handles;                   /* OCI hanldes */
  StoredCtx* storedCtx;                /* Stored context pointer */

  /* Get OCI handles */
  if (GetHandles(extProcCtx, &handles))
    return ODCI_ERROR;

  /* Get the stored context */
  storedCtx=GetStoredCtx(&handles,self,self_ind);
  if (!storedCtx) return ODCI_ERROR;

  /* Free the memory for the stored context */
  if (checkerr(&handles, OCIMemoryFree((dvoid*) handles.usrhp, handles.errhp,
                                       (dvoid*) storedCtx)))
    return ODCI_ERROR;

  return ODCI_SUCCESS;
}

/***********************************************************************/

/* Get the stored context using the key in the scan context */

static StoredCtx* GetStoredCtx(Handles_t* handles, StockPivotImpl* self,
                               StockPivotImpl_ind* self_ind)
{
  StoredCtx *storedCtx;           /* Stored context pointer */
  ub1 *key;                       /* key to retrieve context */
  ub4 keylen;                     /* length of key */
  /* return NULL if the PL/SQL context is NULL */
  if (self_ind->_atomic == OCI_IND_NULL) return NULL;

  /* Get the key */
  key = OCIRawPtr(handles->envhp, self->key);
  keylen = OCIRawSize(handles->envhp, self->key);

  /* Retrieve stored context using the key */
  if (checkerr(handles, OCIContextGetValue((dvoid*) handles->usrhp,
                                          handles->errhp,
                                          key, (ub1) keylen,
                                          (dvoid**) &storedCtx)))
    return NULL;

  return storedCtx;
}

/*************************************************************************/

/* Get OCI handles using the ext-proc context */

static int GetHandles(OCIExtProcContext* extProcCtx, Handles_t* handles)
{
  /* store the ext-proc context in the handles struct */
  handles->extProcCtx=extProcCtx;

  /* Get OCI handles */
  if (checkerr(handles, OCIExtProcGetEnv(extProcCtx, &handles->envhp,
                        &handles->svchp, &handles->errhp)))
    return -1;

  /* get the user handle */
  if (checkerr(handles, OCIAttrGet((dvoid*)handles->svchp,
                                  (ub4)OCI_HTYPE_SVCCTX,
                                  (dvoid*)&handles->usrhp,
                                  (ub4*) 0, (ub4)OCI_ATTR_SESSION,
                                  handles->errhp)))
    return -1;

  return 0;
}

/*************************************************************************/
```

```
/* Check the error status and throw exception if necessary */

static int checkerr(Handles_t* handles, sword status)
{
  text errbuf[512];     /* error message buffer */
  sb4 errcode;          /* OCI error code */

  switch (status)
  {
  case OCI_SUCCESS:
  case OCI_SUCCESS_WITH_INFO:
    return 0;
  case OCI_ERROR:
    OCIErrorGet ((dvoid*) handles->errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    sprintf((char*)errbuf, "OCI ERROR code %d",errcode);
    break;
  default:
    sprintf((char*)errbuf, "Warning - error status %d",status);
    break;
  }

  OCIExtProcRaiseExcpWithMsg(handles->extProcCtx, 29400, errbuf,
    strlen((char*)errbuf));

  return -1;
}
```

Java Implementation

In this example, the declaration of the implementation type references Java methods instead of C functions. This is the only change from the preceding, C example: all the other objects (`TickerType`, `TickerTypeSet`, `refcur_pkg`, `StockTable`, and `StockPivot`) are the same. The code to implement the Java methods is shown after the SQL declarations in the following section.

SQL Declarations for Java Implementation

```
// create the directory object

CREATE OR REPLACE DIRECTORY JavaDir AS '/home/bill/Java';

// compile the java source
```

```
CREATE AND COMPILE JAVA SOURCE NAMED source01
USING BFILE (JavaDir,'StockPivotImpl.java');
/
show errors

-- Create the implementation type

CREATE TYPE StockPivotImpl AS OBJECT
(
  key INTEGER,

  STATIC FUNCTION ODCITableStart(sctx OUT StockPivotImpl, cur SYS_REFCURSOR)
    RETURN NUMBER
    AS LANGUAGE JAVA
   NAME 'StockPivotImpl.ODCITableStart(oracle.sql.STRUCT[], java.sql.ResultSet)
return java.math.BigDecimal',

  MEMBER FUNCTION ODCITableFetch(self IN OUT StockPivotImpl, nrows IN NUMBER,
                               outSet OUT TickerTypeSet) RETURN NUMBER
    AS LANGUAGE JAVA
    NAME 'StockPivotImpl.ODCITableFetch(java.math.BigDecimal,
oracle.sql.ARRAY[]) return java.math.BigDecimal',

  MEMBER FUNCTION ODCITableClose(self IN StockPivotImpl) RETURN NUMBER
    AS LANGUAGE JAVA
    NAME 'StockPivotImpl.ODCITableClose() return java.math.BigDecimal'

);
/
show errors
```

Java Implementation of the ODCITable Methods

The following code implements the three `ODCITable` methods as external functions in Java:

```
import java.io.*;
import java.util.*;
import oracle.sql.*;
import java.sql.*;
import java.math.BigDecimal;
import oracle.CartridgeServices.*;
```

```
// stored context type

public class StoredCtx
{
  ResultSet rset;
  public StoredCtx(ResultSet rs) { rset=rs; }
}

// implementation type

public class StockPivotImpl implements SQLData
{
  private BigDecimal key;

  final static BigDecimal SUCCESS = new BigDecimal(0);
  final static BigDecimal ERROR = new BigDecimal(1);

  // Implement SQLData interface.

  String sql_type;
  public String getSQLTypeName() throws SQLException
  {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    key = stream.readBigDecimal();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeBigDecimal(key);
  }

  // type methods implementing ODCITable interface

  static public BigDecimal ODCITableStart(STRUCT[] sctx,ResultSet rset)
    throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    // create a stored context and store the result set in it
    StoredCtx ctx=new StoredCtx(rset);

    // register stored context with cartridge services
    int key;
    try {
      key = ContextManager.setContext(ctx);
    } catch (CountException ce) {
      return ERROR;
    }

    // create a StockPivotImpl instance and store the key in it
    Object[] impAttr = new Object[1];
    impAttr[0] = new BigDecimal(key);
    StructDescriptor sd = new StructDescriptor("STOCKPIVOTIMPL",conn);
    sctx[0] = new STRUCT(sd,conn,impAttr);

    return SUCCESS;
  }

  public BigDecimal ODCITableFetch(BigDecimal nrows, ARRAY[] outSet)
    throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    // retrieve stored context using the key
    StoredCtx ctx;
    try {
      ctx=(StoredCtx)ContextManager.getContext(key.intValue());
    } catch (InvalidKeyException ik ) {
      return ERROR;
    }

    // get the nrows parameter, but return up to 10 rows
    int nrowsval = nrows.intValue();
    if (nrowsval>10) nrowsval=10;

    // create a vector for the fetched rows
    Vector v = new Vector(nrowsval);
    int i=0;

    StructDescriptor outDesc =
      StructDescriptor.createDescriptor("TICKERTYPE", conn);
    Object[] out_attr = new Object[3];

    while(nrowsval>0 && ctx.rset.next()){
      out_attr[0] = (Object)ctx.rset.getString(1);
      out_attr[1] = (Object)new String("O");
      out_attr[2] = (Object)new BigDecimal(ctx.rset.getFloat(2));
      v.add((Object)new STRUCT(outDesc, conn, out_attr));

      out_attr[1] = (Object)new String("C");
      out_attr[2] = (Object)new BigDecimal(ctx.rset.getFloat(3));
      v.add((Object)new STRUCT(outDesc, conn, out_attr));

      i+=2;
      nrowsval-=2;
    }

    // return if no rows found
    if(i==0) return SUCCESS;

    // create the output ARRAY using the vector
    Object out_arr[] = v.toArray();
    ArrayDescriptor ad = new ArrayDescriptor("TICKERTYPESET",conn);
    outSet[0] = new ARRAY(ad,conn,out_arr);

    return SUCCESS;
  }

  public BigDecimal ODCITableClose() throws SQLException {

    // retrieve stored context using the key, and remove from ContextManager
    StoredCtx ctx;
    try {
      ctx=(StoredCtx)ContextManager.clearContext(key.intValue());
    } catch (InvalidKeyException ik ) {
      return ERROR;
    }

    // close the result set
    Statement stmt = ctx.rset.getStatement();
    ctx.rset.close();
    if(stmt!=null) stmt.close();

    return SUCCESS;
  }

}
```

Index