Oracle9*i*

Heterogeneous Connectivity Administrator's Guide

Release 2 (9.2)

March 2002

Part No. A96544-01

ORACLE®

Oracle9*i* Heterogeneous Connectivity Administrator's Guide, Release 2 (9.2)

Part No. A96544-01

Primary Author: Ruth Baylis

Contributing Authors: Ted Burroughs, Vira Goorah, and Raghu Mani

Graphics Designer: Valerie Moore

Contributors: Jacco Draaijer, Kishan Peyetti, Sridhar Rajagopal, Paul Raveling, and Eric Voss

Contents

3 Features of Oracle Transparent Gateways and Generic Connectivity

4 Using Heterogeneous Services Agents

A Heterogeneous Services Initialization Parameters

B Data Type Mapping

C DBMS_HS_PASSTHROUGH for Pass-Through SQL

D Data Dictionary Translation Support

Index

Send Us Your Comments

Oracle9*i* Heterogeneous Connectivity Administrator's Guide, Release 2 (9.2)

Part No. A96544-01

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

Oracle9i Heterogeneous Connectivity Administrator's Guide describes Oracle's approach for information integration in a heterogeneous environment. Specifically, it describes Oracle Transparent Gateways and Generic Connectivity and is meant to be an administrators guide for these Oracle products.

This preface contains these topics:

Audience

The *Oracle9i* Heterogeneous Connectivity Administrator's Guide is intended for the following users:

- Database administrators who want to administer distributed database systems that involve Oracle to non-Oracle database links
- Application developers who want to make use of the heterogeneous connectivity functionality in the Oracle database server
- Readers who want a high-level understanding of Oracle's architecture for heterogeneous connectivity and how it works.

To use this document, you should be familiar with the following information:

- Relational database concepts and basic database or applications administration as described in the following books:
  - *Oracle9i Database Concepts*
  - *Oracle9i Database Administrator's Guide*
  - *Oracle9i Application Developer's Guide - Fundamentals*
- The operating system environment under which database administrators are running Oracle.

Organization

This document contains the following chapters:

Chapter 1, "Introduction"

This chapter describes the challenges of operating in a heterogeneous environment. Oracle recognizes these challenges and offers both synchronous and asynchronous solutions that enable companies to easily operate in such an environment. The two synchronous solutions, Oracle Transparent Gateways and Generic Connectivity, are discussed this book.

Chapter 2, "The Role of the Heterogeneous Services Component"

Oracle's synchronous solutions for operating in a heterogeneous environment are Oracle Transparent Gateways and Generic Connectivity. The common component of the Oracle database server for supporting these solutions is Heterogeneous Services. This chapter describes the architecture and functionality of the Heterogeneous

Services component and its interaction with Oracle Transparent Gateways and Generic Connectivity.

Chapter 3, "Features of Oracle Transparent Gateways and Generic Connectivity"

This chapter describes the major features provided by Oracle Transparent Gateways and Generic Connectivity.

Chapter 4, "Using Heterogeneous Services Agents"

This chapter explains how to use Oracle Transparent Gateways.

Chapter 5, "Multithreaded Agents"

This chapter explains what multithreaded agents are, how they contribute to the overall efficiency of a distributed database system, and how to administer multithreaded agents.

Chapter 6, "Performance Tips"

This chapter explains how to optimize distributed SQL statements, how to use partition views with Oracle Transparent Gateways, and how to optimize the performance of distributed queries.

Chapter 7, "Generic Connectivity"

This chapter describes the configuration and usage of generic connectivity agents.

Appendix A, "Heterogeneous Services Initialization Parameters"

This appendix lists heterogeneous services initialization parameters and provides instructions on how to set them.

Appendix B, "Data Type Mapping"

The tables in this appendix show how Oracle maps ANSI datatypes through ODBC and OLE DB interfaces to supported Oracle datatypes when it is retrieving data from a non-Oracle system.

Appendix C, "DBMS_HS_PASSTHROUGH for Pass-Through SQL"

The package, `DBMS_HS_PASSTHROUGH`, contains the procedures and functions for pass-through SQL of heterogeneous services. This appendix documents each of them.

Appendix D, "Data Dictionary Translation Support"

This appendix documents data dictionary translation support. It explains how to access non-Oracle data dictionaries, lists heterogeneous services data dictionary views, describes how to use supported views and tables, and explains data dictionary mapping.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Database Concepts*
- *Oracle9i Database Administrator's Guide*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Database New Features*

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.

Convention	Meaning	Example
`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]`

Convention	Meaning	Example
. . .	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;`
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`
`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.

1

Introduction

This chapter describes the challenges of operating in a heterogeneous environment. Oracle recognizes these challenges and offers both synchronous and asynchronous solutions that enable companies to easily operate in such an environment. The two synchronous solutions, Oracle Transparent Gateways and Generic Connectivity, are discussed this book.

This chapter contains these topics:

- The Information Integration Challenge
- How Oracle Addresses Synchronous Information Integration
- Benefits of Oracle's Solution for Synchronous Information Integration

> **See Also:** For information pertaining to a specific Oracle Transparent Gateway information, please consult the Oracle documentation for that specific gateway

The Information Integration Challenge

Information integration is a challenge that affects many organizations. Many run several different database systems. Each of these systems stores data and has a set of applications that runs against it. This data is just bits and bytes on a file system - and only a database can turn the bits and bytes of data into business information. Integration and consolidation of all business information would allow an organization to easily and quickly take advantage of the synergies inherent in business information.

Consolidation of all data into one database system is often difficult. This is in large part because many of the applications that run against one database may not have an equivalent that runs against another. Until such time as migration to one consolidated database system is made feasible, it is necessary for the various heterogeneous database systems to work together.

There are several problems to overcome before such interoperability becomes possible. The database systems can have different access interfaces, different data types, different capabilities, and different ways of handling error conditions. Even when one relational database is trying to access another relational database the differences are significant. In such a situation, the common features of the databases include data access through SQL, two phase commit, and similar data types.

However, there are significant differences as well. SQL dialects can be different, as can transaction semantics. There can be some data types in one database that do not exist in the other. The most significant area of difference is in the data dictionaries of the two databases. Most data dictionaries contain similar information but the information is structured for each in a completely different way. There are several possible ways of overcoming this problem. In this book, we describe the approach that Oracle has taken for synchronously accessing information from multiple sources.

How Oracle Addresses Synchronous Information Integration

If a client program needs to access or modify data at several Oracle databases, it can open connections to each of them. This approach, however, has several drawbacks. Among them, are the following. If you want to join data from the databases, then the client must contain logic that does that. If data integrity must be guaranteed, then the client will need to contain transaction coordination logic.

Oracle provides another approach called distributed processing, where the client connects to one Oracle database and shifts the burden of joining data and transaction coordination to that database. The database that the client program

connects to is called the local database. Any database other than this one is a remote database. The client program can access objects at any of the remote databases using database links. The Oracle query processor takes care of the joins and its transaction engine takes care of the transaction coordination.

The approach that Oracle has taken to solving the heterogeneous connectivity problem is to allow a non-Oracle system to be one of the remote nodes in the previously described scenario. From the client's point of view the remote non-Oracle system functions like a remote Oracle system. It will appear to understand the same SQL dialect and to have the same data dictionary structure as an Oracle system. Access to a non-Oracle system in this manner is done through a component in the Oracle server called Heterogeneous Services.

The work done by the Heterogeneous Services component is, for the most part, completely transparent to the end user. With only a few exceptions (these are noted in later chapters), you are not required to do anything different to access a non-Oracle system than is required for accessing an Oracle system. The Heterogeneous Services component is used as the foundation for implementing Oracle's access to non-Oracle databases.

The following are two methods that Oracle uses for solving the challenges of information sharing and integration in a heterogeneous environment. Because they are both based on a foundation that is integrated into the database, they can exploit all of the features of the database.

- Oracle Transparent Gateways

  An Oracle Transparent Gateway works in conjunction with the Oracle database server Heterogeneous Services component, to access a particular, commercially available, non-Oracle system for which that Oracle Transparent Gateway has been designed. For example, you use the Oracle Transparent Gateway for Sybase on Solaris to access a Sybase database operating on a Sun Solaris platform.

  Using an Oracle Transparent Gateway, you can access data anywhere in a distributed database system without being required to know either the location of the data, or how it is stored.

- Generic Connectivity

  Oracle provides a set of agents, containing only generic code, that interface with the Heterogeneous Services component and comprise Generic Connectivity. These agents must be linked with customer provided drivers. Oracle provides Generic Connectivity agents for ODBC and OLE DB that enable you to use

ODBC and OLE DB drivers to access non-Oracle databases that have an ODBC or OLE DB interface.

The functionality of Generic Connectivity is more limited that of Oracle Transparent Gateways.

Oracle also offers asynchronous information integration solutions that are mentioned here, but that are not discussed in this book. Briefly, these solutions include:

- Oracle Streams

  Oracle Streams enables the propagation of data, transactions and events in a single data stream or queue, either within a database, or between multiple databases. Not only can Oracle Streams capture, propagate, and apply changes to data, it can also handle data structure changes (DDL) and user-defined events. Changes can be captured and applied as is, or transformed at any point in the capture, propagation, and apply processing.

- Messaging Gateway

  The messaging gateway enables communication between Oracle and other non-Oracle message queuing.

- Open System Interfaces

  Oracle offers a number of open interfaces, such as OCI, JDBC, and ODBC, that enable customers to use third party applications or to write their own client applications to access the Oracle database.

Benefits of Oracle's Solution for Synchronous Information Integration

Much of the processing power of Generic Connectivity and Transparent Gateways is integrated into the database. This provides an efficient solution for information integration that enables full exploitation of the power and features of the Oracle database. This includes such features as powerful SQL parsing and distributed optimization capabilities.

The following sections explore the benefits of Oracle's approach to resolving the challenges of a heterogeneous environment:

- Remote Data Can Be Accessed Transparently
- There is No Unnecessary Data Duplication
- SQL Statements Can Query Several Different Databases

Remote Data Can Be Accessed Transparently

Both Generic Connectivity and Oracle Transparent Gateways provide the ability to transparently access data in non-Oracle databases from an Oracle environment. You can create synonyms for the objects in a non-Oracle database and refer to them without having to specify a physical location. This transparency eliminates the need for application developers to customize their applications to access data from different non-Oracle systems, thus decreasing development efforts and increasing the mobility of the application.

Instead of requiring applications to interoperate with non-Oracle systems using their native interfaces (which can result in intensive application-side processing), applications can be built upon a consistent Oracle interface for both Oracle and non-Oracle systems.

There is No Unnecessary Data Duplication

Generic Connectivity and Oracle Transparent Gateways provide applications direct access to data in non-oracle databases. This eliminates the need to upload and download large amounts of data to different locations, thus reducing data duplication and saving disk storage space. Also, by eliminating this need to upload and download large amounts of data there is a further benefit of a reduced risk for unsynchronized or inconsistent data.

SQL Statements Can Query Several Different Databases

The Oracle database server accepts SQL statements that query data stored in several different databases. The Oracle database server with the Heterogeneous Services component processes the SQL statement and passes the appropriate SQL directly to other Oracle databases and through gateways to non-Oracle databases. The Oracle database server then combines the results and returns them to the client. This enables a query to be processed so that it spans the non-Oracle database system, other databases, and local and remote Oracle data.

Oracle's Application Development and End User Tools Can Be Used

Generic Connectivity and Oracle Transparent Gateways extend the range of Oracle's database and application development tools. Oracle has tools that increase

application development and user productivity by reducing prototype, development, and maintenance time.

You are not required to develop new tools or learn how to use other tools to access data stored in non-Oracle databases. Instead, you can access Oracle and non-Oracle data with a single set of Oracle tools. These tools can run on remote machines connected through Oracle Net to the Oracle database server.

Users Can Talk to a Remote Database in its Own Language

Oracle enables you to transparently access non-Oracle systems using Oracle SQL. In some cases, however, it becomes necessary to use non-Oracle system SQL to access the non-Oracle system. For such cases, Oracle has a pass-through feature that allows you to bypass Oracle's query processor and to talk to the remote database in its own language

2

The Role of the Heterogeneous Services Component

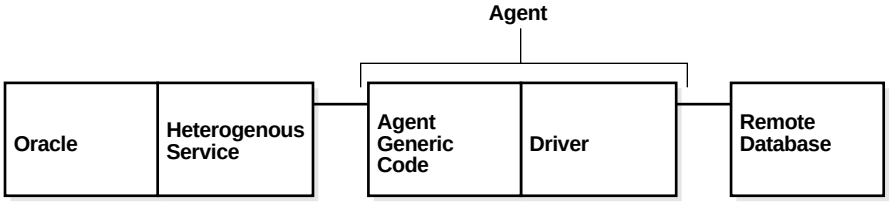
Oracle's synchronous solutions for operating in a heterogeneous environment are Oracle Transparent Gateways and Generic Connectivity. The common component of the Oracle database server for supporting these solutions is Heterogeneous Services. This chapter describes the architecture and functionality of the Heterogeneous Services component and its interaction with Oracle Transparent Gateways and Generic Connectivity.

This chapter contains these topics:

Heterogeneous Connectivity Process Architecture

At a high level, Oracle heterogeneous connectivity process architecture is structured as shown in Figure 2–1.

Figure 2–1 Oracle Heterogeneous Connectivity Process Architecture



The Heterogeneous Services component in the Oracle database server talks to a Heterogeneous Services agent process which, in turn, talks to the non-Oracle system. We can conceptually divide the code into three parts:

- The Heterogeneous Services component in the Oracle database server. Most of the heterogeneous connectivity related processing is done in this module.
- Agent generic code. This is code in the agent that is generic to all Heterogeneous Services based products. This consists, for the most part, of code to communicate with the database and multithreading support.
- The driver. This is the module that communicates with the non-Oracle system. It is used to map calls from the Heterogeneous Services external application programming interface (API) onto the native API of the non-Oracle system and it is non-Oracle system specific.

Heterogeneous Services Agents

A Heterogeneous Service agent is the process through which an Oracle server connects to a non-Oracle system. This agent process that accesses a non-Oracle system is called a gateway. Access to all gateways goes through the Heterogeneous Services component in the Oracle server and all gateways contain the same agent-generic code. Each gateway has a different driver linked in that maps the Heterogeneous Services application programming interface (API) to the client API of the non-Oracle system.

The agent process consists of two components. These are agent generic code and a non-Oracle system-specific driver. An agent exists primarily to isolate the Oracle

database server from third-party code. In order for a process to access the non-Oracle system, the non-Oracle system client libraries have to be linked into it. In the absence of the agent process, these libraries would have to be directly linked into the Oracle database and problems in this code could cause the Oracle server to go down. Having an agent process isolates the Oracle server from any problems in third-party code so that even if a fatal error takes place, only the agent process will end.

An agent can reside in the following places:

- On the same machine as the non-Oracle system
- On the same machine as the Oracle server
- On a machine different from either of these two

Agent processes are usually started when a user session makes its first non-Oracle system access through a database link. These connections are made using Oracle's remote data access software, Oracle Net Services, which enables both client-server and server-server communication. The agent process continues to run until the user session is disconnected or the database link is explicitly closed.

Multithreaded agents behave slightly differently. They have to be explicitly started and shut down by a database administrator instead of automatically being spawned by Oracle Net Services.

> **See Also:** Chapter 5, "Multithreaded Agents" for information on multithreaded agents and how to use them

Types of Heterogeneous Services Agents

There are two types of Heterogeneous Services agents:

- Oracle Transparent Gateways
- Generic Connectivity

Oracle Transparent Gateways

An Oracle Transparent Gateway is a gateway that is designed for accessing a specific non-Oracle system. Oracle Corporation provides gateways to access several commercially produced non-Oracle systems; many of these gateways have been ported to several platforms. For example, an Oracle Transparent Gateway for Sybase on Solaris is the Solaris port of a gateway designed to access Sybase database systems.

With Oracle Transparent Gateways, you can use an Oracle database server to access data anywhere in a distributed database system without being required to know either the location of the data or how it is stored. When the results of your queries are returned to you by the Oracle database server, they are presented to you as if the data stores from which they were taken all resided within a remote instance of an Oracle distributed database.

Generic Connectivity

In addition to Oracle Transparent Gateways for various non-Oracle database systems, there is a set of agents that comprise the Oracle Generic Connectivity feature. These agents contain only generic code and the customer is responsible for providing the necessary drivers. Oracle has Generic Connectivity agents for ODBC and OLE DB that enable you to use ODBE and OLEDB drivers to access non-Oracle systems that have an ODBC or an OLE DB interface.

To access a specific non-Oracle system using Generic Connectivity, you must connect an ODBC or OLE DB driver to the gateway for that non-Oracle system. These drivers are not provided by Oracle corporation. However, as long as Oracle Corporation supports the ODBC and OLE DB protocols, you can use the Generic Connectivity feature to access any non-Oracle system that can be accessed using an ODBC or OLE DB driver.

Generic Connectivity has some limitations. Connecting to one of these gateways from another Oracle database server is not supported. Functionality of these gateways, especially when compared to Oracle Transparent Gateways, is limited.

> **See Also:** For more information, see Chapter 7, "Generic Connectivity"

Heterogeneous Services Components

This section discusses the components of Heterogeneous Services in the Oracle database server. These components are:

- Transaction Service
- SQL Service

Transaction Service

The transaction service component of the Heterogeneous Services component makes it possible for non-Oracle systems to be integrated into Oracle database

server transactions and sessions. When you access a non-Oracle system for the first time over a database link within your Oracle user session, you transparently set up an authenticated session in the non-Oracle system. At the end of your Oracle user session, the authenticated session in the non-Oracle database system transparently closes at the non-Oracle system.

Additionally, one or more non-Oracle systems can participate in an Oracle distributed transaction. When an application commits a transaction, Oracle's two-phase commit protocol accesses the non-Oracle database system to coordinate transparently the distributed transaction. Even in those cases where the non-Oracle system does not support all aspects of Oracle two-phase commit protocol, Oracle can (with some limitations) support distributed transactions with the non-Oracle system.

SQL Service

The Structured Query Language (SQL) service handles the processing of all SQL-related operations. The work done by the SQL service includes:

1. Mapping Oracle internal SQL-related calls to the Heterogeneous Services driver application programing interface (API); this is in turn mapped by the driver to the client API of the non-Oracle system.
2. Translating SQL statements from Oracle's SQL dialect to the SQL dialect of the non-Oracle system.
3. Translating queries that reference Oracle data dictionary tables to queries that extract the necessary information from the non-Oracle system data dictionary.
4. Translating data from non-Oracle system data types to Oracle data types and back.
5. Making up for missing functionality at the non-Oracle system by issuing multiple queries to get the necessary data and doing post processing to get the desired results

Configuring Heterogeneous Services

In the previous section, we described what the different heterogeneous components do. These components consist entirely of generic code and, in order to work with so many different non-Oracle systems, their behavior has to be configured. Each gateway has configuration information stored in the driver module and this information is uploaded to the server immediately after the connection to the

gateway has been established. We can divide this configuration information into three parts:

- Data Dictionary Translations
- Heterogeneous Services Initialization Parameters
- Capabilities

Data Dictionary Translations

Data dictionary translations are views on non-Oracle system data dictionary tables that help Heterogeneous Services translate references to Oracle data dictionary tables into queries needed to retrieve the equivalent information from the non-Oracle system data dictionary.

> **See Also:** For a more detailed explanation of data dictionary translations, please see Appendix D, "Data Dictionary Translation Support"

Heterogeneous Services Initialization Parameters

Heterogeneous Services initialization parameters serve two functions.

- They give the user a means of fine-tuning the gateway to optimize performance and memory utilization for the gateway and the Heterogeneous Services component.
- They enable the user to tell the gateway (and, thereby, Heterogeneous Services) how the non-Oracle system has been configured (for example what language the non-Oracle system is running in). To put it another way, they give Heterogeneous Services information about the configurable properties of the non-Oracle system.

You can examine the Heterogeneous Services initialization parameters for a session by querying the view V$HS_PARAMETER. Users can set initialization parameters in gateway initialization files.

Capabilities

Capabilities tell Heterogeneous Services about the limitations of the non-Oracle system (such as what types of SQL statements are and are not supported) and how to map Oracle data types and SQL expressions to their non-Oracle system equivalents. In other words, they tell Heterogeneous Services about the

non-configurable properties of the non-Oracle system. Capabilities cannot be changed by the user.

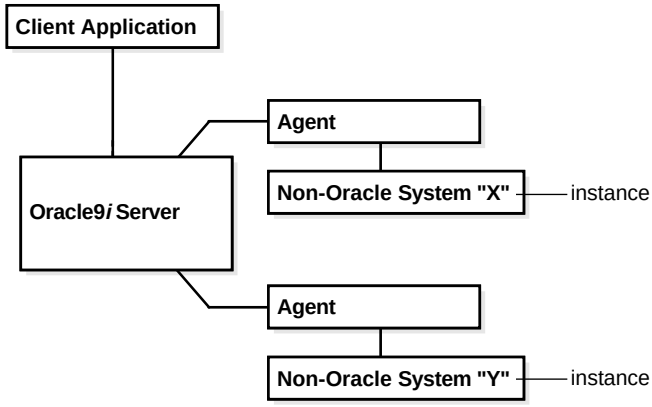
The Heterogeneous Services Data Dictionary

As mentioned in the previous section, configuration information is uploaded from an agent to the Heterogeneous Services component immediately after the connection to the agent has been established. Since this information can be very large in size, it is inefficient to do uploads on each connection. Therefore, the first time an Oracle database talks to an agent, the configuration information is uploaded and stored in Heterogeneous Services data dictionary tables and thereafter no upload takes place until something at the agent changes (for example, if a patch is applied or if the agent is upgraded to a new version).

Classes and Instances

Using Heterogeneous Services, a user can access several non-Oracle systems from a single Oracle database. This is illustrated in Figure 2–2, which shows two non-Oracle systems being accessed.

Figure 2–2 Accessing Multiple Non-Oracle Instances



Both agents upload configuration information, which is stored as part of the Heterogeneous Services data dictionary information on the Oracle database server.

Although it is possible to store data dictionary information at one level of granularity by having completely separate definitions in the Heterogeneous

Services data dictionary for each individual instance, this might lead to an unnecessarily large amount of redundant data dictionary information. To avoid this, Oracle organizes the Heterogeneous Services data dictionary by two levels of granularity, called class and instance.

A class pertains to a specific type of non-Oracle system. For example, you might want to access the class of Sybase database systems with your Oracle database server. An instance defines specializations within a class. For example, you might want to access several separate instances within a Sybase database system. Each class definition (one level of granularity) is shared by all the particular instances (a second level of granularity) under that class. Further, instance information takes precedence over class information, and class information takes precedence over server-supplied defaults.

For example, suppose that the Oracle database server accesses three instances of Sybase and two instances of Ingres II. Sybase and Ingres II each have their own code, requiring separate class definitions for the Oracle database server to access them. The Heterogeneous Services data dictionary therefore would contain two class definitions, one for Sybase and one for Ingres II, with five instance definitions, one for each instance being accessed by the Oracle database server.

Note that instance level capability and data dictionary information are session specific and hence are not stored in the Heterogeneous Services data dictionary of the Oracle database server. However, instance level initialization parameters can be stored in the database.

Data Dictionary Views

The Heterogeneous Services data dictionary views contain the following kinds of information:

- Names of instances and classes uploaded into the Oracle data dictionary
- Capabilities, including SQL translations, defined for each class or instance
- Data Dictionary translations defined for each class or instance
- Initialization parameters defined for each class or instance

You can access information from the Oracle data dictionary by using fixed views. The views are categorized into three main types:

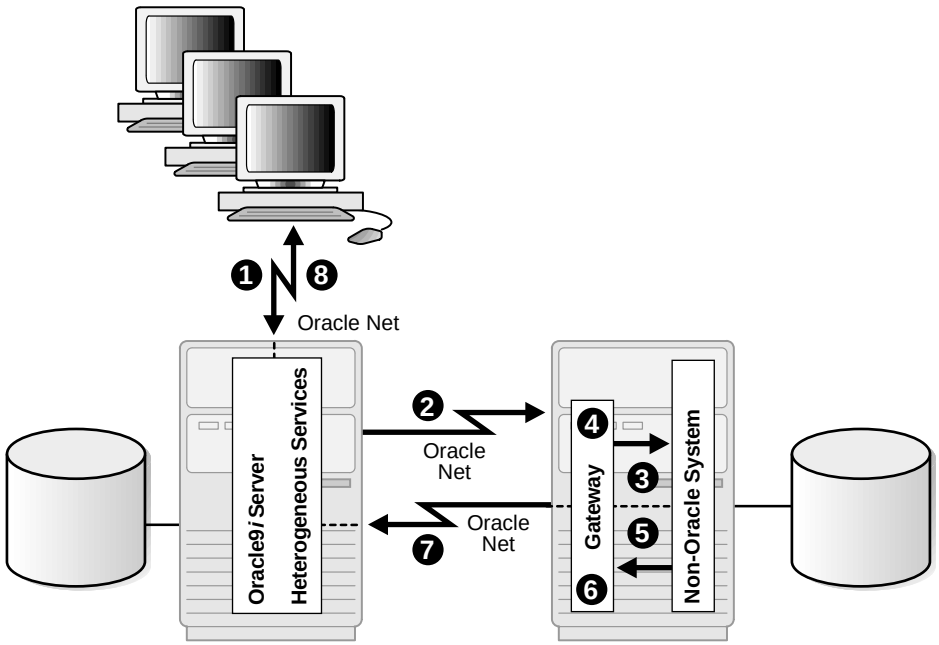
- General views
- Views used for the transaction service
- Views used for the SQL service

See Also: For more information on data dictionary views, see Appendix D, "Data Dictionary Translation Support"

Gateway Process Flow

Figure 2–3 shows a typical gateway process flow. The steps explain the sequence of events that occurs when a client application queries the non-Oracle database system database through the gateway.

Figure 2–3 Gateway Process Flow



1. The client application sends a query over Oracle Net to the Oracle database server.
2. The Oracle database server sends the query over to the gateway using Oracle Net.
3. For the first transaction in a session, the gateway logs into non-Oracle database system using a username and password that is valid in the non-Oracle system.

4. The gateway converts the Oracle SQL statement into a SQL statement understood by non-Oracle database system.
5. The gateway retrieves data using non-Oracle database system SQL statements.
6. The gateway converts retrieved data into a format compatible with the Oracle database server.
7. The gateway returns query results to the Oracle database server, again using Oracle Net Services.
8. The Oracle database server passes the query results to the client application by using Oracle Net. The database link remains open until the gateway session is finished or the database link is explicitly closed.

3

Features of Oracle Transparent Gateways and Generic Connectivity

This chapter describes the major features provided by Oracle Transparent Gateways and Generic Connectivity. Descriptions of these features are contained in the following topics:

- SQL and PL/SQL Support
- Heterogeneous Replication
- Pass-through SQL
- Result Set Support
- Data Dictionary Translations
- Datetime Data Types
- Two Phase Commit Protocol
- Piecewise Long
- SQL*Plus Describe Command
- Constraints on SQL in a Distributed Environment
- Optimization

> **Note:** These features are not necessarily available in all Heterogeneous Services based gateways. Not only must there be generic support for these features, which Heterogeneous Services provides, but there must also be support added to the driver for them. Please consult your gateways documentation to determine if a particular feature is supported for your gateway.

SQL and PL/SQL Support

SQL statements are translated and data types are mapped according to capabilities. PL/SQL calls are mapped to non-Oracle system stored procedures. In the case of SQL statements, if functionality is missing at the remote system, then either a simpler query is issued or the statement is broken up into multiple queries and the desired results are obtained by post processing in the Oracle database.

Even though Heterogeneous Services can, for the most part, incorporate non-Oracle systems into Oracle distributed sessions, there are several limitations to this. Some of the generic limitations are:

1. Data manipulation language statements that update objects on the remote non-Oracle system should not reference any objects on the local Oracle database. For example, a statement such as the following will cause an error to be raised:

   ```
   INSERT INTO remote_table@link as SELECT * FROM local_table;
   ```

2. There is no support for `CONNECT BY` clauses in SQL statements.
3. ROWID support is limited; consult individual gateway documentation for more details. The Oracle Universal Rowid data type is not supported in any Oracle9*i* gateway.
4. LOBs, ADTs, and REFs are not supported.
5. PL/SQL in SQL is not supported. For example, a statement such as the following will cause an error to be raised:

   ```
   SELECT remote_func@link(a,b) FROM remote_table@link
   ```

6. Remote packages are not supported.
7. Remote stored procedures can have `out` arguments of type `REF CURSOR` but not `in` or `in-out` objects.
8. None of the Oracle9*i* gateways supports shared database links.

Note: In addition to these generic limitation, each gateway can have additional limitations. Please consult the gateway documentation for individual gateways for a complete list of limitations of the product.

Heterogeneous Replication

Data can be replicated between a non-Oracle system and an Oracle server using materialized views.

> **Note:** Starting with Oracle9i, Release 2, there is another means of sharing information between databases. Called Streams, this functionality includes the replication of information between Oracle and non-Oracle databases.
>
> For information about using Streams, see *Oracle9i Streams*.

Materialized views instantiate data captured from tables at the non-Oracle master site at a particular point in time. This instant is defined by a refresh operation, which copies this data to the Oracle server and synchronizes the copy on Oracle with the master copy on the non-Oracle system. The "materialized" data is then available as a view on the Oracle server.

Replication facilities provide mechanisms to schedule refreshes and to collect materialized views into replication groups to facilitate their administration. Refresh groups permit refreshing multiple materialized views just as if they were a single object.

Heterogeneous replication support is necessarily limited to a subset of the full Oracle-to-Oracle replication functionality:

- Only the non-Oracle system can be the master site. This is because materialized views can be created only on an Oracle server.
- Materialized views must use complete refresh. This is because fast refresh would require Oracle-specific functionality in the non-Oracle system.
- Not all types of materialized views can be created to reference tables on a non-Oracle system. Primary key and subquery materialized views are supported, but rowid and object id materialized views are not supported. This is because there is no SQL standard for the format and contents of rowids, and non-Oracle systems do not implement Oracle objects.

Other restrictions apply to any access to non-Oracle data through Oracle's Heterogeneous Services facilities. The most important of these are:

- Non-Oracle data types in table columns mapped to a fixed view must be compatible with (that is, have a mapping to or from) Oracle data types. This is usually true for data types defined by ANSI SQL standards.

- A subquery materialized view may not be able to use language features restricted by individual non-Oracle systems. In many cases Heterogeneous Services supports such language features by processing queries within the Oracle server, but occasionally the non-Oracle systems impose limitations that cannot be diagnosed until Heterogeneous Services attempts to execute the query.

The following examples illustrate basic setup and use of three materialized views to replicate data from a non-Oracle system to an Oracle data store.

Note: For the following examples, *remote_db* refers to the non-Oracle system which you are accessing from your Oracle database server.

Example 1: Create materialized views for heterogeneous replication

This example creates three materialized views that are then used in succeeding examples.

1. Create a primary key materialized view of table customer@*remote_db*.

```
CREATE MATERIALIZED VIEW pk_mv REFRESH COMPLETE AS
  SELECT * FROM customer@remote_db WHERE "zip" = 94555;
```

2. Create a subquery materialized view of tables orders@*remote_db* and customer@*remote_db*.

```
CREATE MATERIALIZED VIEW sq_mv REFRESH COMPLETE AS
  SELECT * FROM orders@remote_db o WHERE EXISTS
    (SELECT c."c_id" FROM customer@remote_db c
       WHERE c."zip" = 94555 and c."c_id"  = o."c_id" );
```

3. Create a complex materialized view of data from multiple tables on *remote_db.*

```
CREATE MATERIALIZED VIEW cx_mv
  REFRESH COMPLETE AS
  SELECT  c."c_id", o."o_id"
    FROM customer@remote_db c,
         orders@remote_db o,
         order_line@remote_db ol
    WHERE c."c_id" = o."c_id"
    AND o."o_id" = ol."o_id";
```

Example 2: Set up a refresh group for heterogeneous replication

```
BEGIN
  dbms_refresh.make('refgroup1',
   'pk_mv, sq_mv, cx_mv',
   NULL, NULL);
 END;
 /
```

Example 3: Force refresh of all 3 materialized views

```
BEGIN
   dbms_refresh.refresh('refgroup1');
END;
 /
```

See Also: *Oracle9i Replication* for a full description of materialized views and replication facilities

Pass-through SQL

The pass-through SQL feature enables you to send a statement directly to a non-Oracle system without being interpreted by the Oracle9*i* server. This feature can be useful if the non-Oracle system allows for operations in statements for which there is no equivalent in Oracle.

This section contains the following topics:

- Pass-through SQL
- Considering the Implications of Using Pass-Through SQL
- Executing Pass-Through SQL Statements

Using the DBMS_HS_PASSTHROUGH Package

You can execute pass-through SQL statements directly at the non-Oracle system using the PL/SQL package DBMS_HS_PASSTHROUGH. Any statement executed with this package is executed in the same transaction as standard SQL statements.

The DBMS_HS_PASSTHROUGH package is a virtual package. It conceptually resides at the non-Oracle system. In reality, however, calls to this package are intercepted by Heterogeneous Services and mapped onto one or more Heterogeneous Services application programming interface (API) calls. The driver, in turn, maps these Heterogeneous Services API calls onto the API of the non-Oracle system. The client application should invoke the procedures in the

package through a database link in exactly the same way as it would invoke a non-Oracle system stored procedure. The special processing done by Heterogeneous Services is transparent to the user.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about this package

Considering the Implications of Using Pass-Through SQL

When you execute a pass-through SQL statement that implicitly commits or rolls back a transaction in the non-Oracle system, the transaction is affected. For example, some systems implicitly commit the transaction containing a data definition language (DDL) statement. Because the Oracle database server is bypassed, the Oracle database server is unaware of the commit in the non-Oracle system. Consequently, the data at the non-Oracle system can be committed while the transaction in the Oracle database server is not.

If the transaction in the Oracle database server is rolled back, data inconsistencies between the Oracle database server and the non-Oracle server can occur. This situation results in **global data inconsistency.**

Note that if the application executes a regular `COMMIT` statement, the Oracle database server can coordinate the distributed transaction with the non-Oracle system. The statement executed with the pass-through facility is part of the distributed transaction.

Executing Pass-Through SQL Statements

The table below shows the functions and procedures provided by the `DBMS_HS_PASSTHROUGH` package that allow you to execute pass-through SQL statements.

Procedure/Function	Description
`OPEN_CURSOR`	Opens a cursor
`CLOSE_CURSOR`	Closes a cursor
`PARSE`	Parses the statement
`BIND_VARIABLE`	Binds `IN` variables
`BIND_OUT_VARIABLE`	Binds `OUT` variables
`BIND_INOUT_VARIABLE`	Binds `IN OUT` variables
`EXECUTE_NON_QUERY`	Executes non-query

Procedure/Function	Description
EXECUTE_IMMEDIATE	Executes non-query without bind variables
FETCH_ROW	Fetches rows from query
GET_VALUE	Retrieves column value from SELECT statement or retrieves OUT bind parameters

Executing Non-Queries

Non-queries include the following statements and types of statements:

- INSERT
- UPDATE
- DELETE
- DDL

To execute non-query statements, use the EXECUTE_IMMEDIATE function. For example, to execute a DDL statement at a non-Oracle system that you can access using the database link SalesDB, execute:

```
DECLARE
  num_rows INTEGER;

BEGIN
  num_rows := DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE@SalesDB
            ('CREATE TABLE DEPT (n SMALLINT, loc CHARACTER(10))');
END;
```

The variable *num_rows* is assigned the number of rows affected by the execution. For DDL statements, zero is returned. Note that you cannot execute a query with EXECUTE_IMMEDIATE and you cannot use bind variables.

Using Bind Variables: Overview Bind variables allow you to use the same SQL statement multiple times with different values, reducing the number of times a SQL statement needs to be parsed. For example, when you need to insert four rows in a particular table, you can parse the SQL statement once and bind and execute the SQL statement for each row. One SQL statement can have zero or more bind variables.

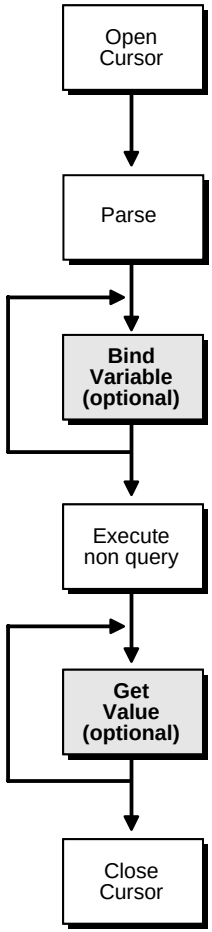
To execute pass-through SQL statements with bind variables, you must:

1. Open a cursor.

2. Parse the SQL statement at the non-Oracle system.
3. Bind the variables.
4. Execute the SQL statement at the non-Oracle system.
5. Close the cursor.

Figure 3–1 shows the flow diagram for executing non-queries with bind variables.

Figure 3–1 Flow Diagram for Non-Query Pass-Through SQL



Using IN Bind Variables The syntax of the non-Oracle system determines how a statement specifies a bind variable. For example, in Oracle you define bind variables with a preceding colon, as in:

```
UPDATE EMP
SET SAL=SAL*1.1
WHERE ENAME=:ename
```

In this statement, ename is the bind variable. In other non-Oracle systems you may need to specify bind variables with a question mark, as in:

```
UPDATE EMP
SET SAL=SAL*1.1
WHERE ENAME= ?
```

In the bind variable step, you must positionally associate host program variables (in this case, PL/SQL) with each of these bind variables.

For example, to execute the above statement, you can use the following PL/SQL program:

```
DECLARE
  c INTEGER;
  nr INTEGER;
BEGIN
  c := DBMS_HS_PASSTHROUGH.OPEN_CURSOR@SalesDB;
  DBMS_HS_PASSTHROUGH.PARSE@SalesDB(c,
        'UPDATE EMP SET SAL=SAL*1.1 WHERE ENAME=?');
  DBMS_HS_PASSTHROUGH.BIND_VARIABLE(c,1,'JONES');
  nr:=DBMS_HS_PASSTHROUGH.EXECUTE_NON_QUERY@SalesDB(c);
  DBMS_OUTPUT.PUT_LINE(nr||' rows updated');
  DBMS_HS_PASSTHROUGH.CLOSE_CURSOR@salesDB(c);
END;
```

Using OUT Bind Variables In some cases, the non-Oracle system can also support OUT bind variables. With OUT bind variables, the value of the bind variable is not known until *after* the execution of the SQL statement.

Although OUT bind variables are populated after the SQL statement is executed, the non-Oracle system must know that the particular bind variable is an OUT bind variable *before* the SQL statement is executed. You must use the BIND_OUT_VARIABLE procedure to specify that the bind variable is an OUT bind variable.

After the SQL statement is executed, you can retrieve the value of the OUT bind variable using the GET_VALUE procedure.

Using IN OUT Bind Variables A bind variable can be both an IN and an OUT variable. This means that the value of the bind variable must be known before the SQL statement is executed but can be changed after the SQL statement is executed.

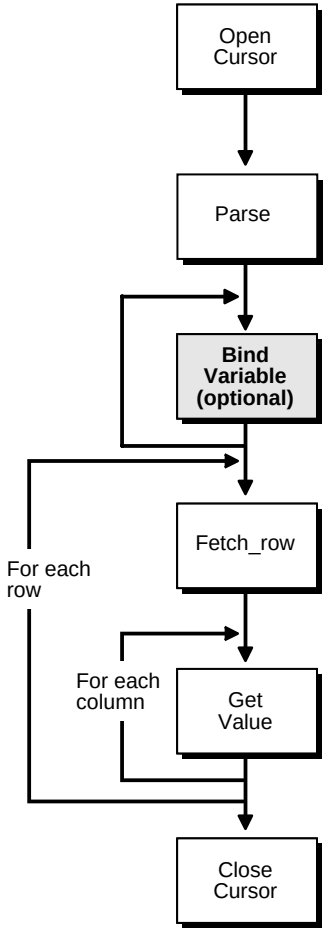
For IN OUT bind variables, you must use the BIND_INOUT_VARIABLE procedure to provide a value *before* the SQL statement is executed. *After* the SQL statement is executed, you must use the GET_VALUE procedure to retrieve the new value of the bind variable.

Executing Queries

The difference between queries and non-queries is that queries retrieve a result set from a SELECT statement. The result set is retrieved by iterating over a cursor.

Figure 3–2 illustrates the steps in a pass-through SQL query. After the system parses the SELECT statement, each row of the result set can be fetched with the FETCH_ROW procedure. After the row is fetched, use the GET_VALUE procedure to retrieve the select list items into program variables. After all rows are fetched you can close the cursor.

Figure 3–2 Pass-Through SQL for Queries



You do not have to fetch all the rows. You can close the cursor at any time after opening the cursor, for example, after fetching a few rows.

Note: Although you are fetching one row at a time, Heterogeneous Services optimizes the round trips between the Oracle9*i* server and the non-Oracle system by buffering multiple rows and fetching from the non-Oracle data system in one round trip.

The next example executes a query:

```
DECLARE
   val  VARCHAR2(100);
   c    INTEGER;
   nr   INTEGER;
BEGIN
  c := DBMS_HS_PASSTHROUGH.OPEN_CURSOR@SalesDB;
  DBMS_HS_PASSTHROUGH.PARSE@SalesDB(c,
    'select ename
     from   emp
     where  deptno=10');
  LOOP
    nr := DBMS_HS_PASSTHROUGH.FETCH_ROW@SalesDB(c);
    EXIT WHEN nr = 0;
    DBMS_HS_PASSTHROUGH.GET_VALUE@SalesDB(c, 1, val);
    DBMS_OUTPUT.PUT_LINE(val);
  END LOOP;
  DBMS_HS_PASSTHROUGH.CLOSE_CURSOR@SalesDB(c);
END;
```

After parsing the SELECT statement, the rows are fetched and printed in a loop until the function FETCH_ROW returns the value 0.

> **See Also:** Appendix C, "DBMS_HS_PASSTHROUGH for Pass-Through SQL"

Result Set Support

Various relational databases allow stored procedures to return result sets. In other words, stored procedures will be able to return one or more sets of rows. This is a relatively new feature for any database.

Traditionally, database stored procedures worked exactly like procedures in any high-level programming language. They had a fixed number of arguments which could be of types in, out, or in-out. If a procedure had n arguments, it could return at most n values as results. However, suppose that somebody wanted a stored procedure to execute a query such as SELECT * FROM emp and return the results. The emp table might have a fixed number of columns but there is no way of telling, at procedure creation time, the number of rows it has. Because of this, no traditional stored procedure can be created that can return the results of a such a query. As a result, several relational database vendors added the capability of returning results sets from stored procedures, but each kind of relational database returns result sets from stored procedures in a different way.

Oracle has a data type called a REF CURSOR. Like every other Oracle data type, a stored procedure can take this data type as an in or out argument. In Oracle, a stored procedure can return a result set in the following way. To return a result set, a stored procedure must have an output argument of type REF CURSOR. It then opens a cursor for a SQL statement and places a handle to that cursor in that output parameter. The caller can then fetch from the REF CURSOR the same way as from any other cursor.

Oracle can do a lot more than simply return result sets. REF CURSORs can be passed as input arguments to PL/SQL routines to be passed back and forth between client programs and PL/SQL routines or between several PL/SQL routines. Until recently, REF CURSORs in Oracle did not work in a distributed environment. This meant that you could pass REF CURSOR values between PL/SQL routines in the same database or between a client program and a PL/SQL routine, but they could not be passed from one database to another. As of Oracle9*i*, that restriction has been removed in the case of Heterogeneous Services.

Result Set Support In Non-Oracle Systems

Several non-Oracle systems allow stored procedures to return result sets but do so in completely different ways. No other relational database management system (RDBMS) has anything like the Oracle REF CURSOR data type. Result sets are supported to some extend in DB2, Sybase, Microsoft SQL Server, and Informix. Result set support in these databases is based on one of the following two models.

Model 1

When creating a stored procedure, the user can explicitly specify the maximum number of result sets that can be returned by that stored procedure. While executing, the stored procedure can open anywhere from zero to its pre-specified maximum number of result sets. After the execution of the stored procedure, a client program can obtain handles to these result sets by using either an embedded SQL directive or calling a client library function. After that the client program can fetch from the result in the same way as from a regular cursor.

Model 2

In this model, there is no pre-specified limit to the number of result sets that can be returned by a stored procedure. Both Model 1 and Oracle have a limit. For Oracle the number of result sets returned by a stored procedure can be at most the number of REF CURSOR `out` arguments; for Model 1, the upper limit is specified using a directive in the stored procedure language. Another way that Model 2 differs from Oracle and Model 1 is that they do not return a handle to the result sets but instead

place the entire result set on the wire when returning from a stored procedure. For Oracle, the handle is the REF CURSOR out argument; for Model 1, it is obtained separately after the execution of the stored procedure. For both Oracle and Model 1, once the handle is obtained, data from the result set is obtained by doing a fetch on the handle; we have a bunch of cursors open and can fetch in any order. In the case of Model 2, however, all the data is already on the wire, with the result sets coming in the order determined by the stored procedure and the output arguments of the procedures coming at the end. So the whole of the first result set must be fetched, then the whole of the second one, until all of the results have been fetched. Finally, the stored procedure out arguments must be fetched.

Heterogeneous Services Support for Result Sets

As can be seen in the preceding sections, result set support exists among non-Oracle databases in a variety of forms. All of these have to be mapped onto the Oracle REF CURSOR model. Due to the considerable differences in behavior among the various non-Oracle systems, Heterogeneous Services result set support will have to behave in one of two different ways depending on the non-Oracle system it is connected to.

Please note the following about Heterogeneous Services result set support:

- Result set support is present in Heterogeneous Services generic code but in order for the feature to work in a gateway, the driver has to implement it as well. Not all drivers have implemented result set support and the customer must check in his gateway-specific documentation to determine whether it is supported in that gateway.
- Heterogeneous Services will support REF CURSOR out arguments from stored procedures. In and in-out arguments will not be supported.
- The REF CURSOR out arguments will all be anonymous REF CURSORs. No typed REF CURSORs are returned by Heterogeneous Services.

Cursor mode

Oracle generally behaves such that each result set returned by the non-Oracle system stored procedure is mapped by the driver to an out argument of type REF CURSOR. The client program sees a stored procedure with several out arguments of type REF CURSOR. After executing the stored procedure, the client program can fetch from the REF CURSOR in exactly the same way as it would from a REF CURSOR returned by an Oracle stored procedure. When connecting to the gateway as described in Model 1, Heterogeneous Services will be in cursor mode.

Sequential Mode

In Oracle, there is a pre-specified maximum number of result sets that a particular stored procedure can return. The number of result sets returned is at most the number of `REF CURSOR out` arguments for the stored procedure. It can, of course, return fewer result sets, but it can never return more.

For the system described in Model 2, there is no pre-specified maximum of result sets that can be returned. In the case of Model 1, we know the maximum number of result sets that a procedure can return, and the driver can return to Heterogeneous Services a description of a stored procedure with that many `REF CURSOR out` arguments. If, on execution of the stored procedure, fewer result sets than the maximum are returned, then the other `REF CURSOR out` arguments will be set to `NULL`.

Another problem for Model 2 database servers is that result sets have to be retrieved in the order in which they were placed on the wire by the database. This prevents Heterogeneous Services from running in cursor mode when connecting to these databases. To access result sets returned by these stored procedures, you must operate Heterogeneous Services in sequential mode.

In sequential mode, the procedure description returned by the driver contains the following:

- All the input arguments of the remote stored procedure
- None of the output arguments
- One `out` argument of type `REF CURSOR` (corresponding to the first result set returned by the stored procedure)

The client fetches from this `REF CURSOR` and then calls the virtual package function `dbms_hs_result_set.get_next_result_set` to get the `REF CURSOR` corresponding to the next result set. This function call is repeated until all result sets have been fetched. The last result set returned will actually be the `out` arguments of the remote stored procedure.

The major limitations of sequential mode are as follows:

- Result sets returned by a remote stored procedure have to be retrieved in the order in which they were placed on the wire
- On execution of a stored procedure, all result sets returned by a previously executed stored procedure will be closed (regardless of whether the data has been completely fetched or not).

Code Examples Using Result Sets

All examples in this section use the following non-Oracle system stored procedure.

> **Note:** For purposes of illustration, the following examples are presented as if they were Oracle PL/SQL stored procedures. However, you can create equivalent stored procedures for the DB2, Microsoft SQL Server, and Sybase.

```
create or replace package rcpackage is
  type rctype is ref cursor;
end rcpackage;
/

create or replace procedure refcurproc
  (arg1 in varchar2, arg2 out varchar2,
   rc1 out rcpackage.rctype,
   rc2 out rcpackage.rctype) is
begin
  arg2 := arg1;
  open rc1 for select * from emp;
  open rc2 for select * from dept;
end;
/
```

This stored procedure assigns the input parameter `arg1` to the output parameter `arg2`, opens the query `SELECT * FROM emp` in `REF CURSOR rc1`, and opens the query `SELECT * FROM dept` in `REF CURSOR rc2`.

Example 1: OCI program fetching from result sets in cursor mode

The following example shows OCI program fetching from result sets in cursor mode.

```
  OCIEnv  *ENVH;
  OCISvcCtx *SVCH;
  OCIStmt *STMH;
  OCIError *ERRH;
  OCIBind *BNDH[4];
  OraText  arg1[20];
  OraText  arg2[20];
  OCIResult *arg3, *arg4;
  OCIStmt   *rstmt1, *rstmt2;
  ub2      rcode[4];
```

```
ub2      rlens[4];
sb2      inds[4];
OraText *stmt = (OraText *) "begin refcurproc@link(:1,:2,:3,:4); end;";

/* Handle Initialization code skipped */

/* Prepare procedure call statement */

OCIStmtPrepare(STMH, ERRH, stmt, strlen(stmt), OCI_NTV_SYNTAX,
               OCI_DEFAULT);

/* Bind procedure arguments */

inds[0] = 0;
strcpy((char *) arg1, "Hello World");
rlens[0] = strlen(arg1);
OCIBindByPos(STMH, &BNDH[0], ERRH, 1, (dvoid *) arg1, 20, SQLT_CHR,
             (dvoid *) &(inds[0]), &(rlens[0]), &(rcode[0]),
             0, (ub4 *) 0, OCI_DEFAULT);

inds[1] = 0;
rlens[1] = 0;
OCIBindByPos(STMH, &BNDH[1], ERRH, 2, (dvoid *) arg2, 20, SQLT_CHR,
             (dvoid *) &(inds[1]), &(rlens[1]), &(rcode[1]),
             0, (ub4 *) 0, OCI_DEFAULT);

inds[2] = 0;
rlens[2] = 0;
OCIDescriptorAlloc(ENVH, (dvoid **) &arg3, OCI_DTYPE_RSET, 0,
                   (dvoid **) 0);
OCIBindByPos(STMH, &BNDH[2], ERRH, 3, (dvoid *) arg3, 0, SQLT_RSET,
             (dvoid *) &(inds[2]), &(rlens[2]), &(rcode[2]),
             0, (ub4 *) 0, OCI_DEFAULT);

inds[3] = 0;
rlens[3] = 0;
OCIDescriptorAlloc(ENVH, (dvoid **) &arg4, OCI_DTYPE_RSET, 0,
                   (dvoid **) 0);
OCIBindByPos(STMH, &BNDH[3], ERRH, 4, (dvoid *) arg4, 0, SQLT_RSET,
             (dvoid *) &(inds[3]), &(rlens[3]), &(rcode[3]),
             0, (ub4 *) 0, OCI_DEFAULT);

/* Execute procedure */

OCIStmtExecute(SVCH, STMH, ERRH, 1, 0, (CONST OCISnapshot *) 0,
```

```
                (OCISnapshot *) 0, OCI_DEFAULT);

  /* Convert result set descriptors to statement handles */

  OCIResultSetToStmt(arg3, ERRH);
  OCIResultSetToStmt(arg4, ERRH);
  rstmt1 = (OCIStmt *) arg3;
  rstmt2 = (OCIStmt *) arg4;

  /* After this the user can fetch from rstmt1 and rstmt2 */
```

Example 2: OCI program fetching from result sets in sequential mode

The following example shows OCI program fetching from result sets in sequential mode.

```
  OCIEnv  *ENVH;
  OCISvcCtx *SVCH;
  OCIStmt *STMH;
  OCIError *ERRH;
  OCIBind *BNDH[2];
  OraText  arg1[20];
  OCIResult *rset;
  OCIStmt   *rstmt;
  ub2      rcode[2];
  ub2      rlens[2];
  sb2      inds[2];
  OraText *stmt = (OraText *) "begin refcurproc@link(:1,:2); end;";
  OraText *n_rs_stm = (OraText *)
    "begin :ret := DBMS_HS_RESULT_SET.GET_NEXT_RESULT_SET@link; end;";

  /* Prepare procedure call statement */

  /* Handle Initialization code skipped */

  OCIStmtPrepare(STMH, ERRH, stmt, strlen(stmt), OCI_NTV_SYNTAX,
                 OCI_DEFAULT);

  /* Bind procedure arguments */

  inds[0] = 0;
  strcpy((char *) arg1, "Hello World");
  rlens[0] = strlen(arg1);
  OCIBindByPos(STMH, &BNDH[0], ERRH, 1, (dvoid *) arg1, 20, SQLT_CHR,
               (dvoid *) &(inds[0]), &(rlens[0]), &(rcode[0]),
```

```
            0, (ub4 *) 0, OCI_DEFAULT);

inds[1] = 0;
rlens[1] = 0;
OCIDescriptorAlloc(ENVH, (dvoid **) &rset, OCI_DTYPE_RSET, 0,
                   (dvoid **) 0);
OCIBindByPos(STMH, &BNDH[1], ERRH, 2, (dvoid *) rset, 0, SQLT_RSET,
            (dvoid *) &(inds[1]), &(rlens[1]), &(rcode[1]),
            0, (ub4 *) 0, OCI_DEFAULT);

/* Execute procedure */

OCIStmtExecute(SVCH, STMH, ERRH, 1, 0, (CONST OCISnapshot *) 0,
               (OCISnapshot *) 0, OCI_DEFAULT);

/* Convert result set to statement handle */

OCIResultSetToStmt(rset, ERRH);
rstmt = (OCIStmt *) rset;

/* After this the user can fetch from rstmt */

/* Issue get_next_result_set call to get handle to next_result set */

/* Prepare Get next result set procedure call */

OCIStmtPrepare(STMH, ERRH, n_rs_stm, strlen(n_rs_stm), OCI_NTV_SYNTAX,
               OCI_DEFAULT);

/* Bind return value */

OCIBindByPos(STMH, &BNDH[1], ERRH, 1, (dvoid *) rset, 0, SQLT_RSET,
            (dvoid *) &(inds[1]), &(rlens[1]), &(rcode[1]),
            0, (ub4 *) 0, OCI_DEFAULT);

/* Execute statement to get next result set*/

OCIStmtExecute(SVCH, STMH, ERRH, 1, 0, (CONST OCISnapshot *) 0,
               (OCISnapshot *) 0, OCI_DEFAULT);

/* Convert next result set to statement handle */

OCIResultSetToStmt(rset, ERRH);
rstmt = (OCIStmt *) rset;
```

```
/* Now rstmt will point to the second result set returned by the
   remote stored procedure */

/* Repeat execution of get_next_result_set to get the output
   arguments */
```

Example 3: PL/SQL program fetching from result sets in cursor mode

Assume that the table `loc_emp` is a local table exactly like the remote `emp` table. The same assumption applies for `loc_dept`.

```
declare
  rc1   rcpackage.rctype;
  rec1  loc_emp%rowtype;
  rc2   rcpackage.rctype;
  rec2  loc_dept%rowtype;
  arg2  varchar2(20);

begin

  -- Execute procedure

  refcurproc@link('Hello World', arg2, rc1, rc2);

  -- Fetch 20 rows from the remote emp table and insert them
  -- into loc_emp

  for i in 1 .. 20 loop
    fetch rc1 into rec1;
    insert into loc_emp (rec1.empno, rec1.ename, rec1.job,
                         rec1.mgr, rec1.hiredate, rec1.sal,
                         rec1.comm, rec1.deptno);
  end loop;

  -- Close the ref cursor

  close rc1;

  -- Fetch 5 rows from the remote dept table and insert them
  -- into loc_dept

  for i in 1 .. 5 loop
    fetch rc2 into rec2;
    insert into loc_dept values (rec2.deptno, rec2.dname, rec2.loc);
  end loop;
```

```
  -- Close the ref cursor

  close rc2;

end;
```

PL/SQL program fetching from result sets in sequential mode

The tables `loc_emp` and `loc_dept` are same as above. The table `outarguments` contains columns corresponding to the out arguments of the remote stored procedure

```
declare
  rc1   rcpackage.rctype;
  rec1  loc_emp%rowtype;
  rc2   rcpackage.rctype;
  rec2  loc_dept%rowtype;
  rc3   rcpackage.rctype;
  rec3  outargs%rowtype;

begin

  -- Execute procedure

  refcurproc@link('Hello World', rc1);

  -- Fetch 20 rows from the remote emp table and insert them
  -- into loc_emp

  for i in 1 .. 20 loop
    fetch rc1 into rec1;
    insert into loc_emp (rec1.empno, rec1.ename, rec1.job,
                         rec1.mgr, rec1.hiredate, rec1.sal,
                         rec1.comm, rec1.deptno);
  end loop;

  -- Close ref cursor

  close rc1;

  -- Get the next result set returned by the stored procedure

  rc2 := dbms_hs_result_set.get_next_result_set@link;
```

```
    -- Fetch 5 rows from the remote dept table and insert them
    -- into loc_dept

    for i in 1 .. 5 loop
      fetch rc2 into rec2;
      insert into loc_dept values (rec2.deptno, rec2.dname, rec2.loc);
    end loop;

    --Close ref cursor

    close rc2;

    -- Get the output arguments from the remote stored procedure
    -- Since we are in sequential mode, they will be returned in the
    -- form of a result set

    rc3 := dbms_hs_result_set.get_next_result_set@link;

    --Fetch them and insert them into the outarguments table

    fetch rc3 into rec3;
    insert into outarguments (rec3.col);

--Close ref cursor

    close rc3;

  end;
```

Data Dictionary Translations

Most database systems have some form of data dictionary. A data dictionary is a collection of information about the database objects that have been created by various users of the system. For a relational database, a data dictionary is a set of tables and views which contain information about the data in the database. This information includes information on the users who are using the system and on the objects that they have created (such as tables, views, triggers and so forth). For the most part, all data dictionaries (regardless of the database system) contain the same information but each database system organizes the information in a different way.

For example, the ALL_CATLOG Oracle data dictionary view gives a list of tables, views, and sequences in the database. It has three columns: the first is called OWNER and is the name of the owner of the object, the second is called TABLE_NAME and is

the name of the object, and the third is called TABLE_TYPE and is the type. This field has value TABLE, VIEW, SEQUENCE and so forth depending on the object type. However, in Sybase, the same information is stored in two tables called sysusers and sysobjects whose column names are quite different than those of Oracle ALL_CATALOG table. Additionally, in Oracle, the table type is a string with value TABLE, VIEW and so forth but in Sybase it is a letter. For example, in Sybase, U means user table, S means system table, V means view, and so forth.

If the client program wanted information from the table ALL_CATALOG at Sybase then all it would have to do is to send a query referencing ALL_CATALOG@*database link* to a gateway and Heterogeneous Services will translate this query to the appropriate one on systables and send the translated query to Sybase.

```
SELECT SU."name" OWNER, SO."name" TABLE_NAME,
   DECODE(SO."type", 'U ','TABLE', 'S ', 'TABLE', 'V ', 'VIEW')
TABLE_TYPE
FROM "dbo"."sysusers"@link  SU, "dbo"."sysobjects"@link SO
WHERE SU."uid" = SO."uid" AND
   (SO."type" = 'V' OR SO."type" = 'S' OR SO."type" = 'U')>
```

To relay such a translation of a query on an Oracle data dictionary table to the equivalent one on the non-Oracle system data dictionary table, Heterogeneous Services needs data dictionary translations for that non-Oracle system. A data dictionary translation is a view definition (essentially a select statement) over one or more non-Oracle system data dictionary tables such that the view looks exactly like the Oracle data dictionary table, with the same column names and the same information formatting. A data dictionary translation need not be as simple as the one above. Often the information needed is not found in one or two tables but is scattered over many tables and the data dictionary translation is a complex join over those tables.

In some cases, an Oracle data dictionary table does not have a translation because the information needed does not exist at the non-Oracle system. In such cases, the gateway can decide not to upload a translation at all or can resort to an alternative approach called **mimicking.** If the gateway wants to mimic a data dictionary table then it will let Heterogeneous Services know and Heterogeneous Services will obtain the description of the data dictionary table by querying the local database but when asked to fetch data, it will report that no rows were selected.

Examples of Data Dictionary Queries

The examples given below show the output of some data dictionary queries sent to Informix, and they compare the results with those produced when querying the same view on Oracle.

Note: The following examples use Informix as the non-Oracle system.

Example 1: Check current session's user name on Oracle and on Informix.

To check the current session's user name on Oracle and on Informix, enter the following:

```
SQL> SELECT a.USERNAME, b.USERNAME FROM USER_USERS a, USER_USERS@remote_db b;

USERNAME                       USERNAME
------------------------------ ------------------------------
THSU                           thsu
```

Note: Oracle maintains usernames in uppercase, Informix maintains them in lowercase.

Example 2: Check current session's user ID on Oracle and on Informix.

To check the current session's user ID on Oracle and on Informix, enter the following:

```
SQL> SELECT a.USER_ID, b.USER_ID FROM USER_USERS a, USER_USERS@remote_db b;

   USER_ID    USER_ID
---------- ----------
        25          0
```

Note: The Informix user ID is defaulted to zero because Informix does not maintain numeric USER_ID values.This also illustrates the need to use caution when accessing other information on the Oracle server. Even if the connected non-Oracle system returns a value for its equivalent of USER_ID, this is not a USER_ID that is meaningful to the Oracle server since it applies only to the non-Oracle system. It would not be meaningful to do other Oracle data dictionary queries using the non-Oracle USER_ID as a key.

Example 3: Check constraints defined on a non-Oracle system for tables owned by an arbitrary user.

To check constraints defined on a non-Oracle system for tables owned by an arbitrary user, enter the following:

```
SQL SELECT CONSTRAINT_NAME, TABLE_NAME FROM ALL_CONSTRAINTS@remote_db
  2  WHERE OWNER = 'thsu';

CONSTRAINT_NAME                TABLE_NAME
------------------------------ ------------------------------
u19942_5270                    thsmv_order_line
u24612_7116                    thsmv_customer
u24613_7117                    thsmv_orders
```

Note: Informix uses a different form of constraint names than Oracle, and its data dictionary maintains the table names in lowercase instead of uppercase.

See Also: Appendix D, "Data Dictionary Translation Support" for more information on data dictionary translations

Datetime Data Types

Oracle has five datetime data types:

- Timestamp
- Timestamp with time zone
- Timestamp with local time zone

- Interval year to month
- Interval day to second

Heterogeneous Services generic code supports Oracle datetime data types in SQL and stored procedures. Oracle does not support these data types in data dictionary translations or queries involving data dictionary translations.

Even though Heterogeneous Services generic code supports this, support for a particular gateway depends on whether or not the driver for that non-Oracle system has implemented datetime support. Support even when the driver implements it may be partial because of the limitations of the non-Oracle system. Users should consult the documentation for their particular gateway on this issue.

The user must set the timestamp formats of the non-Oracle system in the gateway initialization file. The parameters to set are `HS_NLS_TIMESTAMP_FORMAT` and `HS_NLS_TIMESTAMP_TZ_FORMAT`. The user should also set the local time zone for the non-Oracle system in the initialization file. Parameter to set is `HS_TIME_ZONE`.

> **See Also:** Oracle9i SQL Reference for information on datetime data types

Two Phase Commit Protocol

Heterogeneous Services provides the infrastructure for the implementation of the two-phase commit mechanism. The extent to which this is supported depends on the gateway, and the remote system. Please refer to individual gateway manuals for more information.

> **See Also:** Oracle9*i Administrator's Guide* for more information about the two-phase commit protocol

Piecewise Long

Earlier versions of gateways had limited support for the `LONG` data type. `LONG` is an Oracle data type that can be used to store up to 2 gigabytes (GB) of character / raw data (`LONG RAW`). These earlier versions restricted the amount of `LONG` data to 4 MB. This was because they would treat `LONG` data as a single piece. This led to restrictions of memory and network bandwidth on the size of the data that could be handled. Current gateways have extended the functionality to support the full 2 GB of heterogeneous `LONG` data. They handle the data piecewise between the agent and the Oracle server, thereby doing away with the large memory and network bandwidth requirements.

There is a new Heterogeneous Services initialization parameter, HS_LONG_PIECE_TRANSFER_SIZE, that can be used to set the size of the transferred pieces. For example, let us consider fetching 2 GB of LONG data from a heterogeneous source. A smaller piece size means less memory requirement, but more round trips to fetch all the data. A larger piece size means fewer round trips, but more of a memory requirement to store the intermediate pieces internally. Thus, the initialization parameter can be used to tune a system for the best performance, that is, for the best trade-off between round-trips and memory requirements. If the initialization parameter is not set, the system defaults to a piece size of 64 KB.

Note: This feature is not to be confused with piecewise operations on LONG data on the client side. Piecewise fetch and insert operations on the client side did work with the earlier versions of the gateways, and continue to do so. The only difference on the client side is that, where earlier versions of the gateways were able to fetch only up to 4 megabytes (MB) of LONG data, now they can fetch the entire 2 GB of LONG data. This is a significant improvement, considering that 4 MB is only 0.2% of the data type's full capacity.

SQL*Plus Describe Command

Until Oracle9*i*, you could not describe non-Oracle system objects using the SQL*Plus DESCRIBE command. As of Oracle9*i*, functionality to do this has been added to Heterogeneous Services. There are still some limitations. For instance, using Heterogeneous links, you still cannot describe packages, sequences, synonyms, or types.

The SQL*Plus DESCRIBE command is implemented using the OCIDescribeAny call, which was likewise unavailable before Oracle9*i*. The OCIDescribeAny call can also describe databases and schemas, which you cannot do through the SQL*Plus DESCRIBE command. With Heterogeneous Services, you can do both.

In order to implement this functionality some additional driver logic is needed; not all drivers may have implemented it. Please consult individual gateway documentation to see if this feature is supported in that gateway.

Constraints on SQL in a Distributed Environment

This section explains some of the constraints that exist on SQL in a distributed environment. These constraints apply to distributed environments that involve access to non-Oracle systems or remote Oracle databases.

This section contains the following topics:

Resolving Remote and Heterogeneous References

> **Note:** Many of the rules for Heterogeneous access also apply to remote references. For more information, please see the distributed database section of the *Oracle9i Database Administrator's Guide*.

A statement can, with restrictions, be executed on any database node referenced in the statement or the local node. If all objects referenced are resolved to a single, referenced node, then Oracle will attempt to execute a query at that node. You can force execution at a referenced node by using the `/*+ REMOTE_MAPPED */` or `/*+ DRIVING_SITE */` hints. If a statement is forwarded to a different node than the node it was issued at, then the statement is said to be **remote mapped**.

The ways in which statements can, must, and cannot be remote mapped are subject to specific rules or restrictions. If these rules are not all followed, then an error will occur. As long as the statements issued are consistent with all these rules, the order in which the rules are applied does not matter.

Different constraints exist when you are using SQL for remote mapping in a distributed environment. This distributed environment can include remote Oracle databases as well as databases that involve Oracle Transparent Gateways or Generic Connectivity connections between Oracle and non-Oracle systems.

Resolving Important Restrictions

The following section lists some of the different constraints that exist when you are using SQL for remote mapping in a distributed environment.

Note: In the examples that follow, *remote_db* refers to a remote non-Oracle system while *remote_oracle_db* refers to a remote Oracle server.

Rule A: A data definition language statement cannot be remote mapped.

In Oracle data definition language, the target object syntactically has no place for a remote reference. Data definition language statements that contain remote references are always executed locally. For Heterogeneous Services, this means it cannot directly create database objects in a non-Oracle database using SQL.

However, there is an indirect way using pass-through SQL.

Consider the following example:

```
BEGIN
  DBMS_HS.PASSTHROUGHSQL.EXECUTE_IMMEDIATE@remote_db
  (
     'create table x1 (c1 char, c2 number)'
  );
END;
```

Rule B: INSERT, UPDATE and DELETE statements with a remote target table must be remote mapped.

This rule is more restrictive for non-Oracle remote databases than for a remote Oracle database. This is because the remote system cannot fetch data from the originating Oracle database while executing DML statements targeting tables in a non-Oracle system.

For example, to insert all local employees from the local `emp1` table to a remote Oracle `emp2` table, use the following statement:

```
INSERT INTO emp2@remote_oracle_db SELECT * FROM emp1;
```

This statement is remote mapped to the remote database. The remote mapped statement sent to the remote database contains a remote reference back to the originating database for `emp1`. Such a remote link received by the remote database is called a callback link.

In general however, gateways callback links are not supported. When you try to insert into a non-Oracle system using a select statement referencing a local table, an error occurs.

For example, consider the following statement:

```
INSERT INTO emp2@remote_db SELECT * FROM emp1;
```

The statement returns the following error message:

```
ORA-02025: all tables in the SQL statement must be at the remote database
```

The work around is to write a PL/SQL block:

```
DECLARE
CURSOR remote_insert IS SELECT * FROM emp2;
BEGIN

   FOR rec IN remote_insert LOOP
    INSERT INTO emp1@remote_db (empno, ename, deptno) VALUES (
      rec.empno,
      rec.ename,
      rec.deptno
    );
   END loop;
END;
/
```

Another special case involves session specific SQL functions such as USER, USERENV and SYSDATE. These functions may need to be executed at the originating site. A remote mapped statement containing these functions will contain a callback link. For a non-Oracle database where callbacks are not supported this could (by default) result in a restriction error.

For example, consider the following statement:

```
DELETE FROM emp1@remote_db WHERE hiredate > sysdate;
```

The statement returns the following error message:

```
ORA-02070: database REMOTE_DB does not support special functions in this context
```

This often must be resolved by replacing special functions with a bind variable:

```
DELETE FROM emp1@remote_db WHERE hiredate > :1;
```

Rule C: Object features like tables with nested table columns, ADT columns, Opaque columns or Ref Columns cannot be remote mapped.

Currently, the above column types are not supported for heterogeneous access. Hence, this limitation is not directly encountered.

Rule D: SQL statements containing operators and constructs that are not supported at the remote site cannot be remote mapped.

Note that in our description of Rule B we already encountered special constructs such as callback links and special functions as examples of this.

If the statement is a `select` (or `dml` with the target table local) and none of the remaining rules would require the statement to be remote mapped the statement can still be executed by processing the query locally using the local SQL engine and the remote `select` operation.

The remote `select` operation is the operation to retrieve rows for remote table data as opposed to other operations like full table scan and index access which retrieve rows of local table data. The remote table scan has a SQL statement associated with the operation. A full table scan of table `emp1` is issued as `SELECT * FROM emp1` (with the * expanded to the full column list). Access for indexes is converted back to where clause predicates and also filters that can be supported are passed down to the `WHERE` clause of the remote row source.

You can check the SQL statement generated by the Oracle server by explaining the statement and querying the `OTHER` column of the explain plan table for each `REMOTE` operation.

> **See Also:** Example of Using Index and Table Statistics for more information on how to interpret explain plans with remote references

For example consider the following statement:

```
SELECT COUNT(*) FROM emp1@remote_db WHERE hiredate < sysdate;
```

The statement returns the following output:

```
COUNT(*)
----------
        14
1 row selected.
```

The remote table scan is:

```
SELECT hiredate FROM emp1
```

Since the predicate converted to a filter cannot be generated back and passed down to the remote operation because sysdate is not supported by the *remote_db* or evaluation rules, `sysdate` must be executed locally.

> **Note:** Because the remote table scan operation is only partially related to the original query, the number of row retrieved can be significantly larger than you would expect and can have a significant impact on performance.

Rule E: SQL statement containing a table expression cannot be remote mapped.

This limitation is not directly encountered since table expressions are not supported in the heterogeneous access module.

Rule F: If a SQL statement selects a long, the statement must be mapped to the node where the table containing the long resides.

For example, consider the following statement:

```
SELECT long1 FROM table_with_long@remote_db, dual;
```

The statement returns the following error message:

```
ORA-02025: all tables in the SQL statement must be at the remote database
```

This can be resolved by the following statement:

```
SELECT long1 FROM table_with_long@remote_db WHERE long_idx = 1;
```

Rule G: The statement must be mapped to the node on which the table or tables with columns referenced in the FOR UPDATE OF clause resides when the SQL statement is of form "SELECT...FOR UPDATE OF..."

When the SQL statement is of the form `SELECT...FOR UPDATE OF...`, the statement must be mapped to the node on which the table or tables with columns referenced in the `FOR UPDATE OF` clause resides.

For example, consider the following statement:

```
SELECT ename FROM emp1@remote_db WHERE hiredate < sysdate FOR UPDATE OF empno
```

The statement returns the following error message:

```
ORA-02070: database REMOTE_DB does not support special functions in this context
```

Rule H: If the SQL statement contains a SEQUENCE or sequences, the statement must be mapped to the site where each sequence resides.

This rule is not encountered for the heterogeneous access since remote non-Oracle sequences are not supported. The restriction for remote non-Oracle access is already present because of the callback link restriction.

Rule I: If the statement contains a user defined operator or operators, the statement must be mapped to the node where each operator is defined.

This rule is also already covered under the callback link restriction discussed in Rule B.

Rule J: A statement containing duplicate bind variables cannot be remote mapped.

The work around for this restriction is to use unique bind variables and bind by number.

Updates, Inserts and Deletes

As discussed in the previous section, updates to remote non-Oracle objects through an Oracle server are restricted by the missing callback feature support present in the Oracle database. This restricts data manipulation language (DML) upon remote non-Oracle database objects to statements that reference all objects in that remote non-Oracle database or are literals or bind variables.

Because of this, no objects can be referenced from the originating Oracle server or other remote objects.

Also, as with any remote update, whether non-Oracle or a previous remote update, if a SQL update in an Oracle format is not supported, then an error is returned in the following format:

```
ORA-2070: database ... does not support ... in this context.
```

Note: These restrictions do not apply to DML with a local target object referencing non-Oracle or remote Oracle database objects.

You can perform DML to remote Oracle or non-Oracle target tables in an Oracle format that is not supported by using PL/SQL. Declare a cursor that selects the appropriate row and executes the update for each row selected. The row may need to be unique, identified by selecting a primary key, or, if not available, a rowid.

Consider the following example:

```
DECLARE
  CURSOR c1 IS SELECT empno FROM emp e, dept d
               WHERE  e.deptno = d.deptno
               AND    d.dname = 'SALES';
BEGIN
  FOR REC IN c1 LOOP
    UPDATE emp1@remote_db SET comm = .1 * sal;
    WHERE empno = rec.empno;
  END loop;
END;
/
```

Optimization

Oracle's optimizer can be used with Heterogeneous Services. Heterogeneous Services collects certain table and index statistics information on the respective non-Oracle system tables and passes this information back to the Oracle server. The Oracle cost based optimizer uses this information when building the query plan.

There are several other optimizations that the cost based optimizer performs. The most important ones are remote sort elimination and remote joins.

Example of Using Index and Table Statistics

Consider the following statement where you create a table in the Oracle database with 10 rows:

```
CREATE table_T1 (C1 number);
```

Analyze the table by issuing the following SQL statement:

```
ANALYZE table_T1 COMPUTE STATISTICS;
```

Now create a table in the non-Oracle system with 1000 rows:

```
CREATE TABLE remote_t1 (C1 number)
```

Issue the following SQL statement:

```
SELECT a.* FROM remote_t1@remote_db a, T1 b
    WHERE a.C1 = b.C1
```

The Oracle optimizer issues the following SQL statement to the agent:

```
SELECT C1 FROM remote_t1
```

This fetches all the 1000 rows from the non-Oracle system and performs the join in the Oracle database.

Now, if we add a unique index on the column `C1` in the table `remote_t1`, and issue the same SQL statement again, the agent receives the following SQL statement:

```
SELECT C1 FROM remote_t1 WHERE C1 = ?
```

for each value of `C1` in the local `t1.`

Note: ('?') is the bind parameter marker. Also, join predicates containing bind variables generated by Oracle are only generated for nested loop join methods.

To verify the SQL execution plan, generate an explain plan for the SQL statement. Load `utlxplan` in the `admin` directory first.

At the command prompt, type:

```
EXPLAIN PLAN FOR SELECT a.* FROM   remote_t1@remote_db a, T1 b
    WHERE a.C1 = b.C1;
```

Then, run the `utlxpls` utility script by entering the following statement.

```
@utlxpls
```

The operation remote indicates that remote SQL is being referenced.

To find out what statement is sent, type the following statement at the command prompt:

```
SELECT ID, OTHER FROM EXPLAIN_PLAN WHERE OPERATION = 'REMOTE';
```

Example of Remote Join Optimization

The following is an example of the remote join optimization capability of the Oracle database.

Note: The explain plan that uses tables from a non-Oracle system can differ from similar statements with local or remote Oracle table scans. This is because of the limitation on the statistics available to Oracle for non-Oracle tables. Most importantly, column selectivity is not available for non-unique indexes of non-Oracle tables. Because of the limitation of the statistics available, the following example is not necessarily what you encounter when doing remote joins for yourself and is intended for illustration only.

Consider the following example:

```
EXPLAIN PLAN FOR
SELECT e.ename, d.dname, f.ename, f.deptno FROM
 dept d,
 emp@remote_db e,
 emp@remote_db f
  WHERE e.mgr = f.empno
  AND e.deptno = d.deptno
  AND e.empno = f.empno;

@utlxpls
```

Table 3–1 Explain Plan

Operation	Name	Rows	Bytes	Cost	Pstar
SELECT STATEMENT	-	1	101	128	-
HASH JOIN	-	2K	132K	19	-
TABLE ACCESS FULL	DEPT	21	462	1	-
REMOTE	-	2K	89K	16	-

Issue the following statement:

```
SET longwidth 300
SELECT other FROM plan_table WHERE operation = 'REMOTE';
```

You get the following output:

```
SELECT
A1."ENAME",A1."MGR",A1."DEPTNO",A1."EMPNO",A2."ENAME",A2."DEPTNO",A2."EMPNO",A2.
```

```
"EMPNO" FROM "EMP" A1,"EMP" A2 WHERE A1."EMPNO"=A2."EMPNO" AND
A1."MGR"=A2."EMPNO"
```

Optimizer Restrictions for Non-Oracle Access

1. There are no column statistics for remote objects. This can result in poor execution plans. Verify the execution plan and use hints to improve the plan.
2. There is no optimizer hint to force a remote join. However, there is a remote query block optimization that can be used to rewrite the query slightly in order to get a remote join.

   For instance, the earlier example can be rewritten to the form:

   ```
   SELECT v.ename, d.dname, d.deptno FROM
    dept d,
       (SELECT /*+ NO_MERGE */
        e.deptno deptno, e.ename ename emp@remote_db e, emp@remote_db f
            WHERE e.mgr = f.empno
            AND e.empno = f.empno;
       )
     WHERE v.deptno = d.deptno;
   ```

   This guarantees a remote join because it has been isolated in a nested query with the NO_MERGE hint.

4

Using Heterogeneous Services Agents

This chapter explains how to use Heterogeneous Services (HS) agents. It contains the following sections:

Setting Up Access to Non-Oracle Systems

This section explains the generic steps to configure access to a non-Oracle system.

> **Note:** The instructions for configuring your agent may differ slightly from the following instructions. Please see the *Installation and User's Guide* for your agent for more complete installation information.

The steps for setting up access to a non-Oracle system are:

Step 1: Install the Heterogeneous Services Data Dictionary

Step 2: Set Up the Environment to Access Heterogeneous Services Agents

Step 3: Create the Database Link to the Non-Oracle System

Step 4: Test the Connection

Step 1: Install the Heterogeneous Services Data Dictionary

To install the data dictionary tables and views for Heterogeneous Services, you must run a script that creates all the Heterogeneous Services data dictionary tables, views, and packages. On most systems the script is called `caths.sql` and resides in `$ORACLE_HOME/rdbms/admin`.

> **Note:** The data dictionary tables, views, and packages may already be installed on your Oracle9*i* server. Check for the existence of Heterogeneous Services data dictionary views, for example, `SYS.HS_FDS_CLASS`.

Step 2: Set Up the Environment to Access Heterogeneous Services Agents

To initiate a connection to the non-Oracle system, the Oracle9*i* server starts an agent process through the Oracle Net listener. For the Oracle9*i* server to be able to connect to the agent, you must:

1. Set up a Oracle Net service name for the agent that can be used by the Oracle9*i* server. The Oracle Net service name descriptor includes protocol-specific information needed to access the Oracle Net listener. The service name descriptor must include the `(HS=OK)` clause to ensure the connection uses Oracle9*i* Heterogeneous Services. The description of this service name is

defined in `tnsnames.ora`, the Oracle Names server, or in third-party name servers using the Oracle naming adapter.

The following is a sample entry for service name in the `tnsnames.ora` file:

```
Sybase_sales= (DESCRIPTION=
                    (ADDRESS=(PROTOCOL=tcp)
                             (HOST=dlsun206)
                             (PORT=1521)
                    )
                    (CONNECT_DATA = (SERVICE_NAME=SalesDB)
                    )
                    (HS = OK)
              )
```

2. Set up the listener on the gateway to listen for incoming request from the Oracle9*i* server and spawn Heterogeneous Services agents. Then, start the listener on the gateway machine.

   The following is a sample entry for the listener in `listener.ora`:

```
LISTENER =
   (ADDRESS_LIST =
      (ADDRESS= (PROTOCOL=tcp)
                (HOST = dlsun206)
                (PORT = 1521)
      )
  )
...
SID_LIST_LISTENER =
  (SID_LIST =
      (SID_DESC = (SID_NAME=SalesDB)
                  (ORACLE_HOME=/home/oracle/megabase/9.0.1)
                  (PROGRAM=tg4mb80)
                  (ENVS=LD_LIBRARY_PATH=non_oracle_system_lib_directory)
      )
  )
```

   The value associated with `PROGRAM` keyword defines the name of the agent executable. The full path of the directory which contains the DLLs that are loaded by the Heterogeneous Services agent is specified by `LD_LIBRARY_PATH`. Typically, you use `SID_NAME` to define the initialization parameter file for the agent.

See Also:

- *Oracle9i Net Services Administrator's Guide* for additional information about the Oracle Net service name descriptor and listener entry for heterogeneous connectivity
- "Administering Multithreaded Agents" on page 5-7 for information about starting multithreaded agents

Step 3: Create the Database Link to the Non-Oracle System

To create a database link to the non-Oracle system, use the `CREATE DATABASE LINK` statement. The service name that is used in the `USING` clause of the `CREATE DATABASE LINK` statement is the Oracle Net service name.

For example, to create a database link to the `sales` database on Sybase, enter:

```
CREATE DATABASE LINK sales
USING 'Sybase_sales';
```

Step 4: Test the Connection

To test the connection to the non-Oracle system, use the database link in a SQL or PL/SQL statement. If the non-Oracle system is a SQL-based database, you can execute a `SELECT` statement from an existing table or view using the database link. For example, issue:

```
SELECT * FROM product@sales
WHERE product_name like '%pencil%';
```

When you try to access the non-Oracle system for the first time, the Heterogeneous Services agent uploads information into the Heterogeneous Services data dictionary. The uploaded information includes:

Type of Data	Explanation
Capabilities of the non-Oracle system	For example, the agent specifies whether it can perform a join, or a `GROUP BY`.
SQL translation information	The agent specifies how to translate Oracle functions and operators into functions and operators of the non-Oracle system.
Data dictionary translations	To make the data dictionary information of the non-Oracle system available just as if it were an Oracle data dictionary, the agent specifies how to translate Oracle data dictionary tables into tables and views of the non-Oracle system.

Note: Most agents upload information into the Oracle9*i* data dictionary automatically the first time they are accessed. Some agent vendors may provide scripts, however, that you must run on the Oracle9*i* server.

See Also: Heterogeneous Services Data Dictionary Views on page 4-18 and Appendix D, "Data Dictionary Translation Support"

Setting Initialization Parameters

As mentioned in "Configuring Heterogeneous Services" on page 2-5, you can configure the gateway using initialization parameters. This is done by creating an initialization file and setting the desired parameters in this file

Heterogeneous Services initialization parameters are distinct from Oracle database server initialization parameters. Heterogeneous Services initialization parameters are set in the Heterogeneous Services initialization file and not in the Oracle database server initialization parameter file (`init.ora` file). There is a Heterogeneous Services initialization file for each gateway instance.

Name and Location of Heterogeneous Services Initialization Parameter File

The name of the file is `init`*`sid`*`.ora`, where *`sid`* is the Oracle system identifier used for the gateway.

In the case of Generic Connectivity, the Heterogeneous Services initialization file is located in the directory `$ORACLE_HOME/hs/admin`. In the case of Transparent Gateways it is located in the directory `$ORACLE_HOME/product_name/admin` where *`product_name`* is the name of the product. So, the Sybase gateway initialization file is located in the directory `$ORACLE_HOME/tg4sybs/admin`.

Syntax for Initialization Parameter Settings

The initialization file contains a list of initialization parameter settings each of which should be on a separate line. The syntax to set an initialization parameter is:

`[set] [private]` *`parameter`* `= parameter_`*`value`*

The set and private keywords are optional. If the set keyword is present then the variable will also be set in the environment. If the private keyword is present, the parameter will not be uploaded to the server. In general, it recommended that this

keyword not be used - unless the initialization parameter value contains sensitive information (like a password) that should not be sent over the network from gateway to Oracle server.

In the initialization parameter syntax, all keywords (SET, PRIVATE and IFILE) are case insensitive. Initialization parameter names and values are case sensitive. Most initialization parameters names are uppercase. String values for Heterogeneous Services parameters must be lowercase. Exceptions to this rule are explicitly noted.

Another initialization file can be included in an Heterogeneous Services initialization file by using the IFILE directive. The syntax for this is

IFILE = *pathname for file to be included*

Initialization Parameters

Gateway initialization parameters can be divided into two groups. One is a set of generic initialization parameters that are common to all gateways and the other is a set of initialization parameters that are specific to individual gateways. The list of generic initialization parameters is given below and are the only ones discussed in the document.

- HS_COMMIT_POINT_STRENGTH
- HS_DB_DOMAIN
- HS_DB_INTERNAL_NAME
- HS_DB_NAME
- HS_DESCRIBE_CACHE_HWM
- HS_FDS_CONNECT_INFO
- HS_FDS_DEFAULT_SCHEMA_NAME
- HS_FDS_SHAREABLE_NAME
- HS_FDS_TRACE_LEVEL
- HS_FDS_TRACE_FILE_NAME
- HS_LANGUAGE
- HS_LONG_PIECE_TRANSFER_SIZE
- HS_NLS_DATE_FORMAT
- HS_NLS_DATE_LANGUAGE

- HS_NLS_NCHAR
- HS_NLS_TIMESTAMP_FORMAT
- HS_NLS_TIMESTAMP_TZ_FORMAT
- HS_OPEN_CURSORS
- HS_ROWID_CACHE_SIZE
- HS_RPC_FETCH_REBLOCKING
- HS_RPC_FETCH_SIZE
- HS_TIME_ZONE

Do not use the private keyword when setting any of these parameters. Doing that would prevent the parameter from being uploaded to the server and could cause errors in SQL processing. None of these parameters are required to be set in the environment, so the set keyword need not be used either.

See Also:

- Appendix A, "Heterogeneous Services Initialization Parameters" for descriptions of the generic Heterogeneous Services initialization parameters
- Individual gateway documentation for the list of initialization parameters specific to a gateway

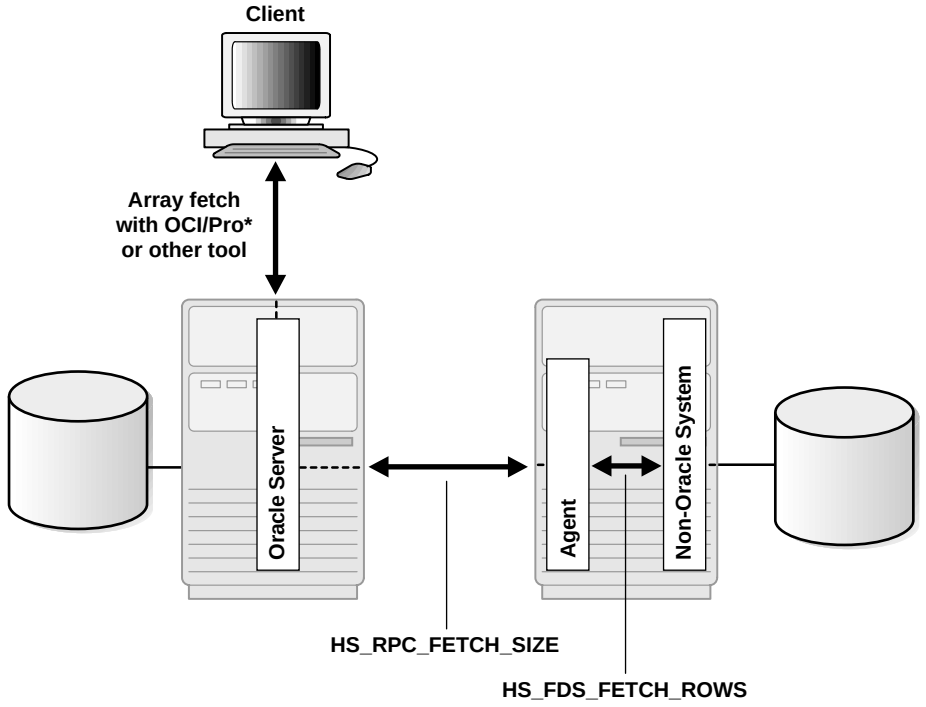
Optimizing Data Transfers Using Bulk Fetch

When an application fetches data from a non-Oracle system using Heterogeneous Services, data is transferred:

1. From the non-Oracle system to the agent process
2. From the agent process to the Oracle database server
3. From the Oracle database server to the application

Oracle allows you to optimize all three data transfers, as illustrated in Figure 4–1.

Figure 4–1 Optimizing data transfers



This section contains the following topics:

- Using OCI, an Oracle Precompiler, or Another Tool for Array Fetches
- Controlling the Array Fetch Between Oracle Database Server and Agent
- Controlling the Array Fetch Between Agent and Non-Oracle Server
- Controlling the Reblocking of Array Fetches

Using OCI, an Oracle Precompiler, or Another Tool for Array Fetches

You can optimize data transfers between your application and the Oracle9*i* server by using array fetches. See your application development tool documentation for information about array fetching and how to specify the amount of data to be sent or each network round trip.

Controlling the Array Fetch Between Oracle Database Server and Agent

When Oracle retrieves data from a non-Oracle system, the Heterogeneous Services initialization parameter HS_RPC_FETCH_SIZE defines the number of bytes sent for each fetch between the agent and the Oracle9*i* server. The agent fetches data from the non-Oracle system until one of the following occurs:

- It has accumulated the specified number of bytes to send back to the Oracle database server.
- The last row of the result set is fetched from the non-Oracle system.

Controlling the Array Fetch Between Agent and Non-Oracle Server

The initialization parameter HS_FDS_FETCH_ROWS determines the number of rows to be retrieved from a non-Oracle system. Note that the array fetch must be supported by the agent. See your agent-specific documentation to ensure that your agent supports array fetching.

Controlling the Reblocking of Array Fetches

By default, an agent fetches data from the non-Oracle system until it has enough data retrieved to send back to the server. That is, it keeps going until the number of bytes fetched from the non-Oracle system is equal to or higher than the value of HS_RPC_FETCH_SIZE. In other words, the agent *reblocks* the data between the agent and the Oracle database server in sizes defined by the value of HS_RPC_FETCH_SIZE.

When the non-Oracle system supports array fetches, you can immediately send the data fetched from the non-Oracle system by the array fetch to the Oracle database server without waiting until the exact value of HS_RPC_FETCH_SIZE is reached. That is, you can stream the data from the non-Oracle system to the Oracle database server and disable reblocking by setting the value of initialization parameter HS_RPC_FETCH_REBLOCKING to OFF.

For example, assume that you set HS_RPC_FETCH_SIZE to 64 kilobytes (KB) and HS_FDS_FETCH_ROWS to 100 rows. Assume that each row is approximately 600 bytes in size, so that the 100 rows are approximately 60 KB. When HS_RPC_FETCH_REBLOCKING is set to ON, the agent starts fetching 100 rows from the non-Oracle system.

Because there is only 60 KB of data in the agent, the agent does not send the data back to the Oracle database server. Instead, the agent fetches the next 100 rows from

the non-Oracle system. Because there is now 120 KB of data in the agent, the first 64 KB can be sent back to the Oracle database server.

Now there is 56 KB of data left in the agent. The agent fetches another 100 rows from the non-Oracle system before sending the next 64 KB of data to the Oracle database server. By setting the initialization parameter HS_RPC_FETCH_REBLOCKING to OFF, the first 100 rows are immediately sent back to the Oracle9*i* server.

Registering Agents

Registration is an operation through which Oracle stores information about an agent in the data dictionary. Agents do not have to be registered. If an agent is not registered, Oracle stores information about the agent in memory instead of in the data dictionary; when a session involving an agent terminates, this information ceases to be available.

Self-registration is an operation in which a database administrator sets an initialization parameter that lets the agent automatically upload information into the data dictionary. In release 8.0 of the Oracle database server, an agent could determine whether to self-register. In Oracle9*i*, self-registration occurs only when the HS_AUTOREGISTER initialization parameter is set to TRUE (default).

Note: HS_AUTOREGISTER is an Oracle initialization parameter that you set in the init.ora file; it is not a Heterogeneous Services initialization parameter that is set in the gateway initialization file.

This section contains the following topics:

- Enabling Agent Self-Registration
- Disabling Agent Self-Registration

Enabling Agent Self-Registration

To ensure correct operation over heterogeneous database links, agent self-registration automates updates to Heterogeneous Services configuration data that describe agents on remote hosts. Agent self-registration is the default behavior. If you do not want to use the agent self-registration feature, then set the initialization parameter HS_AUTOREGISTER to FALSE.

Both the server and the agent rely on three types of information to configure and control operation of the Heterogeneous Services connection. These three sets of information are collectively called **HS configuration data**:

Heterogeneous Services Configuration Data	Description
Heterogeneous Services initialization parameters	Provide control over various connection-specific details of operation.
Capability definitions	Identify details like SQL language features supported by the non-Oracle datasource.
Data dictionary translations	Map references to Oracle data dictionary tables and views into equivalents specific to the non-Oracle data source.

See Also: "Specifying HS_AUTOREGISTER" on page 4-14

Using Agent Self-Registration to Avoid Configuration Mismatches

HS configuration data is stored in the Oracle database server's data dictionary. Because the agent is possibly remote, and may therefore be administered separately, several circumstances can lead to configuration mismatches between servers and agents:

- An agent can be newly installed on a separate machine so that the server has no Heterogeneous Services data dictionary content to represent the agent's HS configuration data.
- A server can be newly installed and lack the necessary HS configuration data for existing agents and non-Oracle data stores.
- A non-Oracle instance can be upgraded from an older version to a newer version, requiring modification of the HS configuration data.
- An Heterogeneous Services agent at a remote site can be upgraded to a new version or patched, requiring modification of the HS configuration data.
- A database administrator (DBA) at the non-Oracle site can change the agent setup, possibly for tuning or testing purposes, in a manner which affects HS configuration data.

Agent self-registration permits successful operation of Heterogeneous Services in all these scenarios. Specifically, agent self-registration enhances interoperability between any Oracle database server and any Heterogeneous Services agent,

provided that each is at least as recent as Version 8.0.3. The basic mechanism for this functionality is the ability to upload HS configuration data from agents to servers.

Self-registration provides automatic updating of HS configuration data residing in the Oracle database server data dictionary. This update ensures that the agent self-registration uploads need to be done only once, on the initial use of a previously unregistered agent. Instance information is uploaded on each connection, not stored in the server data dictionary.

Understanding Agent Self-Registration

The Heterogeneous Services agent self-registration feature can:

- Identify the agent and the non-Oracle data store to the Oracle database server.
- Permit agents to define Heterogeneous Services initialization parameters for use both by the agent and connected Oracle9*i* servers.
- Upload capability definitions and data dictionary translations, if available, from an Heterogeneous Services agent during connection initialization.

> **Note:** When both the server and the agent are release 8.1 or higher, the upload of class information occurs only when the class is undefined in the server data dictionary. Similarly, instance information is uploaded only if the instance is undefined in the server data dictionary.

The information required to accomplish the above is accessed in the server data dictionary by using these agent-supplied names:

- `FDS_CLASS`
- `FDS_CLASS_VERSION`

> **See Also:** "Heterogeneous Services Data Dictionary Views" on page 4-18 to learn how to use the Heterogeneous Services data dictionary views

FDS_CLASS and FDS_CLASS_VERSION `FDS_CLASS` and `FDS_CLASS_VERSION` are defined by Oracle or by third-party vendors for each individual Heterogeneous Services agent and version. Oracle Heterogeneous Services concatenates these

names to form FDS_CLASS_NAME, which is used as a primary key to access class information in the server data dictionary.

FDS_CLASS should specify the type of non-Oracle data store to be accessed and FDS_CLASS_VERSION should specify a version number for both the non-Oracle data store and the agent that connects to the it. Note that when any component of an agent changes, FDS_CLASS_VERSION must also change to uniquely identify the new release.

Note: This information is uploaded when you initialize each connection.

FDS_INST_NAME Instance-specific information can be stored in the server data dictionary. The instance name, FDS_INST_NAME, is configured by the DBA who administers the agent; how the DBA performs this configuration depends on the specific agent in use.

The Oracle database server uses FDS_INST_NAME to look up instance-specific configuration information in its data dictionary. Oracle uses the value as a primary key for columns of the same name in these views:

- FDS_INST_INIT
- FDS_INST_CAPS
- FDS_INST_DD

Server data dictionary accesses that use FDS_INST_NAME also use FDS_CLASS_NAME to uniquely identify configuration information rows. For example, if you port a database from class Sybase8.1.6 to class Sybase8.1.7, both databases can simultaneously operate with instance name SCOTT and use separate sets of configuration information.

Unlike class information, instance information is not automatically self-registered in the server data dictionary.

- If the server data dictionary contains instance information, it represents DBA-defined setup details which fully define the instance configuration. No instance information is uploaded from the agent to the server.
- If the server data dictionary contains no instance information, any instance information made available by a connected agent is uploaded to the server for use in that connection. The uploaded instance data is not stored in the server data dictionary.

Specifying HS_AUTOREGISTER

The Oracle database server initialization parameter HS_AUTOREGISTER enables or disables automatic self-registration of Heterogeneous Services agents. Note that this parameter is specified in the Oracle initialization parameter file, not the agent initialization file. For example, you can set the parameter as follows:

```
HS_AUTOREGISTER = TRUE
```

When set to TRUE, the agent uploads information describing a previously unknown agent class or a new agent version into the server's data dictionary.

Oracle recommends that you use the default value for this parameter (TRUE), which ensures that the server's data dictionary content always correctly represents definitions of class capabilities and data dictionary translations as used in Heterogeneous Services connections.

> **See Also:** *Oracle9i Database Reference* for a description of this parameter

Disabling Agent Self-Registration

To disable agent self-registration, set the HS_AUTOREGISTER initialization parameter as follows:

```
HS_AUTOREGISTER = FALSE
```

Disabling agent self-registration entails that agent information is not stored in the data dictionary. Consequently, the Heterogeneous Services data dictionary views are not useful sources of information. Nevertheless, the Oracle server still requires information about the class and instance of each agent. If agent self-registration is disabled, the server stores this information in local memory.

Oracle Database Server SQL Construct Processing

The gateway rewrites SQL statements when the statements need to be translated or post-processed.

For example, consider a program that requests the following from the non-Oracle database system database:

```
SELECT "COL_A" FROM "test"@SYBS
    WHERE "COL_A" = INITCAP('jones');
```

The non-Oracle database system database does not recognize INITCAP, so the Oracle database server does a table scan of test and filters the results locally. The gateway rewrites the SELECT statement as follows:

```
SELECT "COL_A" FROM "test"@SYBS
```

The results of the query are sent to the gateway and are filtered by the Oracle database server.

Consider the following UPDATE request:

```
UPDATE "test"@SYBS WHERE "COL_A" = INITCAP('jones');
```

In this case, the Oracle database server and the gateway cannot compensate for the lack of support at the non-Oracle database system side, so an error is issued.

If you are performing operations on large amounts of data stored in the non-Oracle database system database, keep in mind that some functions require data to be moved to the integrating Oracle database server before processing can occur.

Using Synonyms

You can provide complete data location transparency and network transparency by using the synonym feature of the Oracle database server. When a synonym is defined, you do not have to know the underlying table or network protocol. A synonym can be public, which means that all Oracle users can refer to the synonym. A synonym can also be defined as private, which means every Oracle user must have a synonym defined to access the non-Oracle database system table.

The following statement creates a system wide synonym for the emp table in the schema of user ORACLE in the non-Oracle database system database:

```
CREATE PUBLIC SYNONYM EMP FOR "ORACLE"."EMP"@SYBS
```

> **See Also:** *Oracle9i Database Administrator's Guide* for information about synonyms

Example of a Distributed Query

The following statement joins data between the Oracle database server, an IBM DB2 database, and the non-Oracle database system database:

```
SELECT O.CUSTNAME, P.PROJNO, E.ENAME, SUM(E.RATE*P."HOURS")
       FROM ORDERS@DB2 O, EMP@ORACLE9 E, "PROJECTS"@SYBS P
       WHERE O.PROJNO = P."PROJNO"
          AND P."EMPNO" = E.EMPNO
```

```
      GROUP BY O.CUSTNAME, P."PROJNO", E.ENAME;
```

Through a combination of views and synonyms, using the following SQL statements, the process of distributed queries is transparent to the user:

```
SQL> CREATE SYNONYM ORDERS FOR ORDERS@DB2
SQL> CREATE SYNONYM PROJECTS FOR "PROJECTS"@SYBS
SQL> CREATE VIEW DETAILS (CUSTNAME,PROJNO,ENAME,SPEND)
      AS
      SELECT O.CUSTNAME, P."PROJNO", E.ENAME, SUM(E.RATE*P."HOURS")
      SPEND
      FROM ORDERS O, EMP E, PROJECTS P
      WHERE O.PROJNO = P."PROJNO"
      AND P."EMPNO" = E.EMPNO
      GROUP BY O.CUSTNAME, P."PROJNO", E.ENAME
```

Use the following SQL statement to retrieve information from the data stores in one statement:

```
SQL> SELECT * FROM DETAILS;
```

The statement retrieves the following table:

```
CUSTNAME         PROJNO          ENAME          SPEND
--------         ------          -----          -----
ABC Co.             1            Jones            400
ABC Co.             1            Smith            180
XYZ Inc.            2            Jones            400
XYZ Inc.            2            Smith            180
```

Copying Data from the Oracle Database Server to the Non-Oracle Database System

Use the SQL*Plus COPY command to copy data from the local database to the non-Oracle database system database. The syntax is as follows:

```
COPY FROM username/password@db_name
INSERT destination_table USING query
```

The following example selects all rows from the local Oracle emp table, inserts them into the emp table on the non-Oracle database system database, and commits the transaction:

```
SQL> COPY FROM SCOTT/TIGER@ORACLE9-
```

```
  2> INSERT SCOTT.EMP@SYBS -
  3> USING SELECT * FROM EMP
```

The COPY command supports APPEND, CREATE, INSERT, and REPLACE options. However, INSERT is the only option supported when copying to non-Oracle database system. The SQL*Plus COPY command does not support copying to tables with lowercase table names. Use the following PL/SQL syntax with lowercase table names:

```
DECLARE
    v1 oracle_table.column1%TYPE;
    v2 oracle_table.column2%TYPE;
    v3 oracle_table.column3%TYPE;
    .
    .
    .
    CURSOR cursor_name IS SELECT * FROM oracle_table;
BEGIN
    OPEN cursor_name;
    LOOP
       FETCH cursor_name INTO v1, v2, v3, ... ;
       EXIT WHEN cursor_name%NOTFOUND;
       INSERT INTO destination_table VALUES (v1, v2, v3, ...);
    END LOOP;

    CLOSE cursor_name;
END;
/
```

> **See Also:** *SQL*Plus User's Guide and Reference* for more information about the COPY command

The following Oracle SQL INSERT statement is not supported for copying data from the Oracle database server to non-Oracle database system:

```
INSERT INTO table_name  SELECT column_list FROM table_name
```

For example, consider the following statement:

```
SQL> INSERT INTO SYBS_TABLE SELECT * FROM MY_LOCAL_TABLE
```

The statement returns the following error message:

```
ORA-2025:  All tables in the SQL statement must be at the remote database
```

Copying Data from the Non-Oracle Database System to the Oracle Database Server

The CREATE TABLE statement lets you copy data from a non-Oracle database system database to the Oracle database server. To create a table on the local database and insert rows from the non-Oracle database system table, use the following syntax:

```
CREATE TABLE table_name AS query
```

The following example creates the table emp in the local Oracle database and inserts the rows from the emp table of the non-Oracle database system database:

```
SQL> CREATE TABLE EMP AS SELECT * FROM SCOTT."EMP"@SYBS
```

Alternatively, you can use the SQL*Plus COPY command to copy data from the non-Oracle database system database to the Oracle database server.

> **See Also:** *SQL*Plus User's Guide and Reference* for more information about the COPY command

Heterogeneous Services Data Dictionary Views

You can use the Heterogeneous Services data dictionary views to access information about Heterogeneous Services. This section addresses the following topics:

- Understanding the Types of Views
- Understanding the Sources of Data Dictionary Information
- Using the General Views
- Using the Transaction Service Views
- Using the SQL Service Views

Understanding the Types of Views

The Heterogeneous Services data dictionary views, which all begin with the prefix HS_, can be divided into four main types:

- General views
- Views used for the transaction service

- Views used for the SQL service

Most of the data dictionary views are defined for both classes and instances. Consequently, for most types of data there is a *_CLASS and an *_INST view.

Table 4–1 Data Dictionary Views for Heterogeneous Services

View	Type	Identifies
HS_BASE_CAPS	SQL service	All capabilities supported by Heterogeneous Services
HS_BASE_DD	SQL service	All data dictionary translation table names supported by Heterogeneous Services
HS_CLASS_CAPS	Transaction service, SQL service	Capabilities for each class
HS_CLASS_DD	SQL service	Data dictionary translations for each class
HS_CLASS_INIT	General	Initialization parameters for each class
HS_FDS_CLASS	General	Classes accessible from this Oracle9*i* server
HS_FDS_INST	General	Instances accessible from this Oracle9*i* server

Like all Oracle data dictionary tables, these views are read-only. Do not change the content of any of the underlying tables.

Understanding the Sources of Data Dictionary Information

The values used for data dictionary content in any particular connection on a Heterogeneous Services database link can come from any of the following sources, in order of precedence:

- Instance information uploaded by the connected Heterogeneous Services agent at the start of the session. This information overrides corresponding content in the Oracle data dictionary, but is never stored into the Oracle data dictionary.
- Instance information stored in the Oracle data dictionary. This data overrides any corresponding content for the connected class.
- Class information stored in the Oracle data dictionary.

If the Oracle database server runs with the HS_AUTOREGISTER server initialization parameter set to FALSE, then no information is stored automatically in the Oracle data dictionary. The equivalent data is uploaded by the Heterogeneous Services agent on a connection-specific basis each time a connection is made, with any instance-specific information taking precedence over class information.

Note: It is not possible to determine positively what capabilities and what data dictionary translations are in use for a given session due to the possibility that an agent can upload instance information.

You can determine the values of Heterogeneous Services initialization parameters by querying the VALUE column of the V$HS_PARAMETER view. Note that the VALUE column of V$HS_PARAMETER truncates the actual initialization parameter value from a maximum of 255 characters to a maximum of 64 characters, and it truncates the parameter name from a maximum of 64 characters to a maximum of 30 characters.

Using the General Views

The views that are common for all services are as follows:

View	Contains
HS_FDS_CLASS HS_FDS_INST	Names of the instances and classes that are uploaded into the Oracle8*i* data dictionary
HS_CLASS_INIT	Information about the Heterogeneous Services initialization parameters

For example, you can access multiple Sybase gateways from an Oracle database server. After accessing the gateways for the first time, the information uploaded into the Oracle database server could appear as follows:

```
SQL> SELECT * FROM hs_fds_class;

FDS_CLASS_NAME        FDS_CLASS_COMMENTS             FDS_CLASS_ID
--------------------- ------------------------------ ------------
Sybase816             Uses Sybase driver, R1.1                1
Sybase817             Uses Sybase driver, R1.2               21
```

Two classes are uploaded: a class that accesses Sybase816 and a class that accesses Sybase817. The data dictionary in the Oracle database server now contains capability information, SQL translations, and data dictionary translations for both Sybase816 and Sybase817.

In addition to this information, the Oracle database server data dictionary also contains instance information in the HS_FDS_INST view for each non-Oracle system instance that is accessed.

Using the Transaction Service Views

When a non-Oracle system is involved in a distributed transaction, the transaction capabilities of the non-Oracle system and the agent control whether it can participate in distributed transactions. Transaction capabilities are stored in the HS_CLASS_CAPS tables.

The ability of the non-Oracle system and agent to support two-phase commit protocols is specified by the 2PC type capability, which can specify one of the following five types:

Type	Capability
Read-only (RO)	The non-Oracle system can only be queried with SQL SELECT statements. Procedure calls are not allowed because procedure calls are assumed to write data.
Single-Site (SS)	The non-Oracle system can handle remote transactions but not distributed transactions. That is, it cannot participate in the two-phase commit protocol.
Commit Confirm (CC)	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol but only as the Commit Point Site. That is, it cannot prepare data, but it can remember the outcome of a particular transaction if asked by the global coordinator.
Two-Phase Commit	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol, as a regular two-phase commit node, but not as a Commit Point Site. That is, it can prepare data, but it cannot remember the outcome of a particular transaction if asked to by the global coordinator.

Type	Capability
Two-Phase Commit Confirm	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol as a regular two-phase commit node or as the Commit Point Site. That is, it can prepare data and it can remember the outcome of a particular transaction if asked by the global coordinator.

The transaction model supported by the driver and non-Oracle system can be queried from Heterogeneous Services' data dictionary view HS_CLASS_CAPS.

One of the capabilities is of the 2PC type:

```
SELECT cap_description, translation
FROM   hs_class_caps
WHERE  cap_description LIKE '2PC%'
AND    fds_class_name='MegaBase6';

CAP_DESCRIPTION                          TRANSLATION
---------------------------------------- -----------
2PC type (RO-SS-CC-PREP/2P-2PCC)                  CC
```

When the non-Oracle system and agent support distributed transactions, the non-Oracle system is treated like any other Oracle9*i* server. When a failure occurs during the two-phase commit protocol, the transaction is recovered automatically. If the failure persists, the in-doubt transaction may need to be manually overridden by the database administrator.

Using the SQL Service Views

Data dictionary views that are specific for the SQL service contain information about:

- SQL capabilities and SQL translations of the non-Oracle data source
- Data Dictionary translations to map Oracle data dictionary views to the data dictionary of the non-Oracle system

> **Note:** This section describes only a portion of the SQL Service-related capabilities. Because you should never need to alter these settings for administrative purposes, these capabilities are not discussed here.

Using Views for Capabilities and Translations

The HS_*_CAPS data dictionary tables contain information about the SQL capabilities of the non-Oracle data source and required SQL translations. These views specify whether the non-Oracle data store or the Oracle database server implements certain SQL language features. If a capability is turned off, then Oracle9*i* does not send any SQL statements to the non-Oracle data source that require this particular capability, but it still performs post-processing.

Using Views for Data Dictionary Translations

In order to make the non-Oracle system appear similar to an Oracle database server, Heterogeneous Services connections map a limited set of Oracle data dictionary views onto the non-Oracle system's data dictionary. This mapping permits applications to issue queries as if these views belonged to an Oracle data dictionary. Data dictionary translations make this access possible. These translations are stored in Heterogeneous Services views whose names are suffixed with _DD.

For example, the following SELECT statement transforms into a Sybase query that retrieves information about emp tables from the Sybase data dictionary table:

```
SELECT * FROM USER_TABLES@salesdb
WHERE UPPER(TABLE_NAME)='EMP';
```

Data dictionary tables can be mimicked instead of translated. If a data dictionary translation is not possible because the non-Oracle data source does not have the required information in its data dictionary, Heterogeneous Services causes it to appear as if the data dictionary table is available, but the table contains no information.

To retrieve information for which Oracle data dictionary views or tables are translated or mimicked for the non-Oracle system, you can issue the following query on the HS_CLASS_DD view:

```
SELECT DD_TABLE_NAME, TRANSLATION_TYPE
FROM   HS_CLASS_DD
WHERE  FDS_CLASS_NAME='Sybase';

DD_TABLE_NAME                  T
------------------------------ -
ALL_ARGUMENTS                  M
ALL_CATALOG                    T
ALL_CLUSTERS                   T
ALL_CLUSTER_HASH_EXPRESSIONS   M
ALL_COLL_TYPES                 M
```

```
ALL_COL_COMMENTS              T
ALL_COL_PRIVS                 M
ALL_COL_PRIVS_MADE            M
ALL_COL_PRIVS_RECD            M
...
```

The translation type T specifies that a translation exists. When the translation type is M, the data dictionary table is mimicked.

> **See Also:** Appendix D, "Data Dictionary Translation Support" for a list of data dictionary views that are supported through Heterogeneous Services mapping

Using the Heterogeneous Services Dynamic Performance Views

The Oracle database server stores information about agents, sessions, and parameter. You can use the V$ dynamic performance views to access this information. This section contains the following topics:

- Determining Which Agents Are Running on a Host
- Determining the Open Heterogeneous Services Sessions

Determining Which Agents Are Running on a Host

The following view shows generation information about agents:

View	Purpose
V$HS_AGENT	Identifies the set of Heterogeneous Services agents currently running on a given host, using one row for each agent process.

Use this view to determine general information about the agents running on a specified host. The following table shows the most relevant columns (for a description of all the columns in the view, see *Oracle9i Database Reference*):

Table 4–2 V$HS_AGENT

Column	Description
AGENT_ID	Oracle Net session identifier used for connections to agent (`listener.ora SID`)
MACHINE	Operating system machine name

Table 4–2 V$HS_AGENT

Column	Description
PROGRAM	Program name of agent
AGENT_TYPE	Type of agent
FDS_CLASS_ID	The ID of the foreign data store class
FDS_INST_ID	The instance name of the foreign data store

Determining the Open Heterogeneous Services Sessions

The following view shows which Heterogeneous Services sessions are open for the Oracle database server:

View	Purpose
V$HS_SESSION	Lists the sessions for each agent, specifying the database link used.

The following table shows the most relevant columns (for an account of all the columns in the view, see *Oracle9i Database Reference*):

Table 4–3 V$HS_SESSION

Column	Description
HS_SESSION_ID	Unique Heterogeneous Services session identifier
AGENT_ID	Oracle Net session identifier used for connections to agent (`listener.ora SID`)
DB_LINK	Server database link name used to access the agent `NULL` means that no database link is used (for example, when using external procedures)
DB_LINK_OWNER	Owner of the database link in `DB_LINK`

Determining the Heterogeneous Services Parameters

The following view shows which Heterogeneous Services parameters are set in the Oracle database server:

View	Purpose
V$HS_PARAMETER	Lists Heterogeneous Services parameters and values registered in the Oracle database server.

The following table shows the most relevant columns (for an account of all the columns in the view, see *Oracle9i Database Reference*):

Table 4–4 V$HS_SESSION

Column	Description
HS_SESSION_ID	Unique Heterogeneous Services session identifier
PARAMETER	The name of the Heterogeneous Services parameter
VALUE	The value of the Heterogeneous Services parameter

Information about the database link that was used for establishing the distributed connection, the startup time, and the set of initialization parameters used for the session is also available.

All of the runtime information is derived from dynamically updated V$ tables. The Distributed Access Manager has a refresh capability available through the menu and toolbar that allows users to rerun queries if necessary and update the data. When the data is refreshed, the tool verifies that the set of registered agents remains the same. If it is not, the global view is updated.

> **See Also:** *Oracle Enterprise Manager Administrator's Guide* and online help for more information about the Distributed Access Manager

5

Multithreaded Agents

This chapter explains what multithreaded agents are, how they contribute to the overall efficiency of a distributed database system, and how to administer multithreaded agents.

This chapter contains the following sections:

> **Note:** Heterogeneous Services supports multithreaded agents, however this functionality may not be available in all Heterogeneous Services based gateways. In addition to the generic support for multithreaded agents that Heterogeneous Services provides, multithreaded agents support must be added to the driver.

Why Use Multithreaded Agents?

This section explains how multithreaded agents contribute to the overall efficiency of Heterogeneous Services and Oracle Transparent Gateways.

This section contains the following topics:

The Challenge of Dedicated Agent Architecture

In the architecture of past releases of Heterogeneous Service, agents are started up on a one for each user session basis and one for each database link basis. When a user session attempts to access a non-Oracle system by means of a particular database link, an agent process is started up that is exclusively dedicated to that user session and that database link. The agent process terminates only when the user session ends or when the database link is closed. Separate agent processes are started under the following conditions:

- The same user session uses two different database links to connect to the same non-Oracle system
- Two different user sessions use the same database link to access the same non-Oracle system.

This architecture is simple and straightforward. However, it has the disadvantage of potentially consuming an unnecessarily large amount of system resources.

For example, suppose that there are several thousand user sessions simultaneously accessing the same non-Oracle system. Because an agent process is started up for each one of them, there are several thousand agent processes running concurrently as well as several thousand connections open to these agent processes. The agent processes are all running regardless of whether each individual agent process is actually active at the moment or not. Because of this, agent processes and open connections can consume a disproportionate amount of system resources without any discernible benefit.

In the case of connections to the Oracle database server, this problem is addressed by starting the server in shared server mode. Shared server mode allows database connections to be shared by a small number of server processes.

The Advantage of Multithreading

The Oracle shared server architecture assumes that even when there are several thousand user sessions currently open, only a small percentage of these connections will be active at any given time. In shared server mode, there is a pool of shared server processes. User sessions connect to dispatcher processes that place the tasks requested by the user sessions on a queue. The tasks are picked up by the first available shared server processes. The number of shared server processes is usually considerably less that the number of user sessions.

Multithreaded Heterogeneous Services agents provide similar functionality for connections to non-Oracle systems. The multithreaded agent architecture uses a pool of shared agent threads. The tasks requested by the user sessions are put on a queue and are picked up by the first available multithreaded agent thread. Because only a small percentage of user connections are actually active at a given moment, using a multithreaded architecture allows for more efficient use of system resources.

Multithreaded Agent Architecture

Multithreaded agents must be prestarted on a one for each system identifier (SID) basis. This is done using the agent control utility `agtctl`. This utility is also used to configure the agent and to shut down the agent.

Each TNS listener that is running on a system listens for incoming connection requests for a set of SIDs. If the SID in an incoming Oracle Net connect string is one of the SIDs that the listener is listening for, then that listener will process the connection. Further, if a multithreaded agent has been started for the SID, then the listener will pass the request to that agent.

In the architecture for multithreaded agents, each incoming connection request is processed by means of the three different kinds of threads:

- A single **monitor** thread

  The monitor thread is responsible for the following:

  - Maintaining communication with the listener
  - Monitoring the load on the process
  - Starting and stopping threads when required

- Several **dispatcher** threads

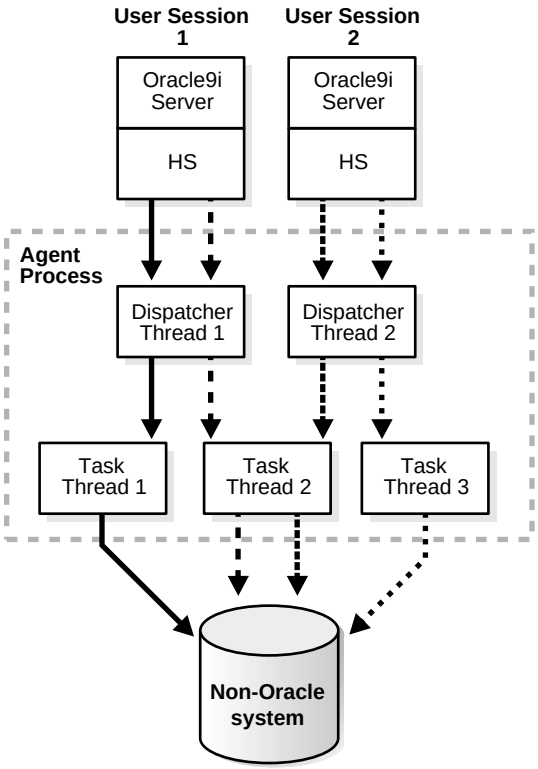
  The dispatcher threads are responsible for the following:

  - Handling communication with the Oracle server
  - Passing task requests on to the task threads
- Several **task** threads

  The task threads handle requests from the Oracle processes.

The multithreaded agent architecture is illustrated in Figure 5–1 where each request issued by a user session is represented in by a separate type of arrow. There is no representation of the monitor thread in this illustration, because that thread is created once when the multithreaded agent is started and it creates and monitors the other threads. Typically there are many more task threads than dispatcher threads.

Figure 5–1 Multithreaded Agent Architecture



These three thread types roughly correspond to the Oracle multithreaded server's PMON, dispatcher and shared server processes respectively.

Note: All requests from a user session go through the same dispatcher thread, but can be serviced by different task threads. It is also possible for several task threads to use the same connection to the non-Oracle system.

Each type of thread is discussed in more detail in the following sections:

- The Monitor Thread
- Dispatcher Threads
- Task Threads

See Also: Administering Multithreaded Agents on page 5-7 for more information on how to start and stop multithreaded agents using the agent control utility

The Monitor Thread

When a multithreaded agent is started for a SID by the agent control utility, the monitor thread is created. The monitor thread performs the following functions:

- It creates the dispatcher and task threads
- It registers the dispatcher threads it has created with all the listeners that are handling connections to this agent.

  While the dispatcher for this SID is running, the listener does not start a new process when it gets an incoming connection. Instead, the listener hands over the connection to this same dispatcher.
- It monitors the other threads and sends load information about the dispatcher threads to all the listener processes handling connections to this agent.

  This enables the listeners to hand over incoming connections to the least loaded dispatcher.
- It continues to monitors each of the threads it has created.

Dispatcher Threads

Dispatcher threads perform the following functions:

- They accept incoming connections and task requests from Oracle servers.
- They place incoming requests on a queue for a task thread to pick up.
- They send results of a request back to the server that issued the request.

Note: Once a user session establishes a connection with a dispatcher, all requests from that user session will go to the same dispatcher until the end of the user session.

Task Threads

Task threads perform the following functions:

- They pick up requests from a queue.
- They perform the necessary operations.
- They place the results on a queue for a dispatcher to pick up

Administering Multithreaded Agents

As discussed earlier, multithreaded Heterogeneous Services agents must be prestarted on a one for each system identifier (SID) basis before any attempt is made to connect to the non-Oracle system. Any agent not spawned in this fashion will not function in multithreaded mode, and must be set up as described in "Setting Up Access to Non-Oracle Systems" on page 4-2.

A multithreaded agent is started, stopped, and configured by an agent control utility called `agtctl`, which works much like `lsnrctl`. However, unlike `lsnrctl`, which reads a configuration file (`listener.ora`), `agtctl` takes configuration information from the command line and writes it to a control file.

The following topics are discussed in this section:

- Agent Control Utility (agtctl) Commands
- Using Single-Line Command Mode
- Using Shell Mode Commands
- Configuration Parameters for Multithreaded Agent Control

Agent Control Utility (agtctl) Commands

You start and stop `agtctl`, and create and maintain its control file, using the commands shown in Table 5–1.

Table 5–1 Agent Control Utility Commands

Command	Description
`startup`	Starts a multithreaded agent
`shutdown`	Stops a multithreaded agent
`set`	Sets a configuration parameter for a multithreaded agent
`unset`	Causes a parameter to revert to its default value
`show`	Displays the value of a configuration parameter
`delete`	Deletes the entry for a particular SID from the control file

Table 5–1 Agent Control Utility Commands

Command	Description
`exit`	Exits shell mode
`help`	Lists available commands

These commands can be issued in one of two ways:

- You can issue commands from the UNIX (or DOS) shell.

  This mode is called single-line command mode.

- You can type `agtctl` and a "`AGTCTL>`" prompt appears. You then can type commands from within the `agtctl` shell.

  This mode is called shell mode

The syntax and parameters for `agtctl` commands vary depending upon the mode in which they are issued.

Note:

- All commands are case sensitive.
- The agent control utility puts its control file in either the directory pointed to by the environment variable `AGTCTL_ADMIN` or in the directory pointed to by the environment variable `TNS_ADMIN`. Ensure that at least one of these environment variables is set and that it points to a directory that the agent has access to
- If the Heterogeneous Services agent requires an environment variable to be set, or if the `ENVS` parameter was used when configuring the `listener.ora` entry for the agent working in dedicated mode, then all required environment variables must be set in the UNIX (or DOS) shell which runs the `agtctl` utility.

Using Single-Line Command Mode

This section describes the use of `agtctl` commands. They are presented in single-line command mode.

Setting Configuration Parameters for a Multithreaded Agent

You should set the configuration parameters for a multithreaded agent before you start the agent. They determine how the agent will be configured. If a configuration parameter is not specifically set, a default value is used. Configuration parameters and their default values are shown in Table 5–2.

Use the `set` command to set multithreaded agent configuration parameters.

Syntax

```
agtctl set  parameter  parameter_value  agent_sid
```

Where:

- *`parameter`* is the parameter that you are setting.
- *`parameter_value`* is the value being assigned to the parameter.
- *`agent_sid`* is the SID which this agent will service. Must be specified for single-line command mode.

Example

```
agtctl  set  max_dispatchers 5 salesDB
```

Starting a Multithreaded Agent

Use the `startup` command to start an agent in multithreaded mode.

Syntax

```
agtctl startup  agent_name  agent_sid
```

Where:

- *`agent_name`* is the name of the agent. E.g., `tg4sybs`, `tg4msql`, `hsodbc`, or other agent.
- *`agent_sid`* is the SID which this agent will service. Must be specified for single-line command mode.

Example

```
agtctl startup  tg4sybs  salesDB
```

Shutting Down a Multithreaded Agent

Use the `shutdown` command to stop a multithreaded agent. There are three forms of shutdown.

- Normal

  This form of shutdown is the default. It causes `agtctl` will talk to the agent and ask it to terminate itself gracefully. All sessions complete the operations they are currently doing and then shutdown.

- Immediate

  In this form of shutdown, `agtctl` talks to the agent and tells it to terminate immediately. The agent process exits immediately regardless of the state of current sessions.

- Abort

  In this form of shutdown, `agtctl` does not talk to the agent at all. It just issues a system call to kill the agent process.

Syntax

```
agtctl shutdown [immediate|abort] agent_sid
```

Where:

- *`agent_sid`* is the SID which this agent will service. Must be specified for single-line command mode.

Example

```
agtctl shutdown immediate salesDB
```

Examining the Value of Configuration Parameters

To examine the value of a configuration parameter use the `show` command.

Syntax

```
agtctl  show  parameter agent_sid
```

Where:

- *`parameter`* is the parameter that you are examining.

Example

```
agtctl  show  max_dispatchers salesDB
```

Resetting a Configuration Parameter to Its Default Value

You can reset a configuration parameter to its default value using the `unset` command.

Syntax

```
agtctl  unset  parameter agent_sid
```

Where:

- *parameter* is the parameter that you are examining.
- *agent_sid* is the SID which this agent will service. Must be specified for single-line command mode.

Example

```
agtctl  unset  max_dispatchers salesDB
```

Deleting an Entry for a Specific SID from the Control File

The `delete` command deletes the entry for the specified SID from the control file.

Syntax

```
agtctl  delete  agent_sid
```

Where:

- *agent_sid* is the SID entry to delete.

Example

```
agtctl  delete  salesDB
```

Requesting Help

Use the `help` command to view a list of available commands for `agtctl`, or to see the syntax for a particular command.

Syntax

```
agtctl help [command]
```

Where:

- *command* is the command whose syntax you want to view.

Example

```
agtctl help set
```

Using Shell Mode Commands

In shell mode, you start `agtctl` by typing `agtctl` whereupon you will see an "`AGTCTL>`" prompt. Thereafter, since you are issuing commands from within the `agtctl` shell, you do not prefix the command string with the word `agtctl`.

Next, set the name of the agent SID that you are working with by typing

```
set  agent_sid  agent_sid
```

All commands issued after this are assumed to be for this particular SID until the `agent_sid` value is changed. Unlike single-line command mode, you do not specify *agent_sid* in the command string.

You can optionally set the language for error messages, to other than English, as follows:

```
set  language  language
```

The commands themselves are the same as those for the single-line command mode. To exit shell mode, type `exit`.

The following are some examples of shell mode commands.

Example: Setting a Configuration Parameter

This example sets a new value for the `shutdown_address` configuration parameter.

```
set shutdown_address (address=(protocol=ipc)(key=oraDBsalesDB))
```

Example: Starting a Multithreaded Agent

This example starts a multithreaded agent.

```
startup tg4sybs
```

Configuration Parameters for Multithreaded Agent Control

The following table lists the configuration parameters for the agent control utility.

Table 5–2 Initialization Parameters for agtctl

Parameter	Description	Default Value
`max_dispatchers`	Maximum number of dispatchers	1
`tcp_dispatchers`	Number of dispatchers listening on tcp (the rest are using ipc)	0
`max_task_threads`	Number of task threads	2
`max_sessions`	Maximum number of sessions	5

`listener_address`	Address on which the listener is listening (needed for registration)	`(ADDRESS_LIST=`
		`(ADDRESS=`
		`(PROTOCOL=IPC)`
		`(KEY=PNPKEY))`
		`(ADDRESS=`
		`(PROTOCOL=IPC)`
		`(KEY=`*`oracle_sid`*`))`
		`(ADDRESS=`
		`(PROTOCOL=TCP)`
		`(HOST=127.0.0.1)`
		`(PORT=1521)))`
		Note: *oracle_sid* is the SID of the Oracle database.

`shutdown_address`	Address on which the agent should listen for shutdown messages from `agtctl`	`(ADDRESS=`
		`(PROTOCOL=IPC)`
		`(KEY=`*`oracle_sid`*`||`*`agent_sid`*`))`
		Notes:
		■ *agent_sid* is the SID of the multithreaded agent.
		■ || indicates that *oracle_sid* and *agent_sid* are concatenated into one string.

6

Performance Tips

This chapter explains how to optimize distributed SQL statements, how to use partition views with Oracle Transparent Gateways, and how to optimize the performance of distributed queries.

This chapter includes the following sections:

Optimizing Heterogeneous Distributed SQL Statements

When a SQL statement accesses data from non-Oracle systems, it is said to be a heterogeneous distributed SQL statement. To optimize heterogeneous distributed SQL statements, follow the same guidelines as for optimizing distributed SQL statements that access Oracle databases only. However, you must consider that the non-Oracle system usually does not support all the functions and operators that Oracle9*i* supports.

The Oracle Transparent Gateways tell Oracle (at connect time) which functions and operators they do support. If the other data source does not support a function or operator, then Oracle performs that function or operator. In this case, Oracle obtains the data from the other data source and applies the function or operator locally. This affects the way in which the SQL statements are decomposed and can affect performance, especially if Oracle is not on the same machine as the other data source.

Using Gateways and Partition Views

You can use partition views with Oracle Transparent Gateways release 8 or higher. Ensure you adhere to the following rules:

- The cost-based optimizer must be used.

  Use hints or set the parameter `OPTIMIZER_MODE` to `ALL_ROWS` or `FIRST_ROWS_K`, or `FIRST_ROWS`.

- Indexes used for each partition must be the same.

  See the gateway-specific documentation to find out whether the gateway sends index information of the non-Oracle system to the Oracle Server. If the gateway sends index information to the optimizer, then make sure that each partition uses the same number of indexes and that you have indexed the same columns.

  If the gateway does not send index information, then the Oracle optimizer is not aware of the indexes on partitions. Indexes are, therefore, considered to be the same for each partition in the non-Oracle system. If one partition resides on an Oracle server, then you cannot have an index defined on that partition.

- The column names and column data types for all branches in the `UNION ALL` view must be the same.

  Non-Oracle system data types are mapped onto Oracle data types. Make sure that the data types of each partition that reside in the different non-Oracle

systems all map to the same Oracle data types. To see how data types are mapped onto Oracle data types, execute a `DESCRIBE` statement in SQL*Plus.

Optimizing Performance of Distributed Queries

You can improve performance of distributed queries in several ways. These are:

- Choose the best SQL statement.

  In many cases, there are several SQL statements that can achieve the same result. If all tables are on the same database, then the difference in performance between these SQL statements might be minimal. But, if the tables are located on different databases, then the difference in performance might be more significant.

- Use the cost-based optimizer.

  The cost-based optimizer uses indexes on remote tables, considers more execution plans than the rule-based optimizer, and generally gives better results. With the cost-based optimizer, performance of distributed queries is generally satisfactory. Only on rare occasions is it necessary to change SQL statements, create views, or use procedural code.

- Use views.

  In some situations, views can be used to improve performance of distributed queries. For example:

  – Joining several remote tables on the remote database

  – Sending a different table through the network

  – Using procedural code

- Use procedural code.

  On some rare occasions, it can be more efficient to replace a distributed query by procedural code, such as a PL/SQL procedure or a precompiler program. This option is mentioned here only for completeness, not because it is often needed.

7

Generic Connectivity

This chapter describes the configuration and usage of Generic Connectivity agents.

This chapter contains these topics:

What Is Generic Connectivity?

Generic Connectivity is intended for low-end data integration solutions requiring the ad hoc query capability to connect from an Oracle database server to non-Oracle database systems. Generic Connectivity is enabled by Oracle's Heterogeneous Services component, allowing you to connect to non-Oracle systems with improved performance and throughput.

Generic Connectivity is implemented as either a Heterogeneous Services ODBC agent or a Heterogeneous Services OLE DB agent. An ODBC agent and OLE DB agent are included as part of your Oracle system. Be sure to use the agents shipped with your particular Oracle system, installed in the same `$ORACLE_HOME`.

Any data source compatible with the ODBC or OLE DB standards described in this chapter can be accessed using a Generic Connectivity agent.

This section contains the following topics:

Types of Agents

Generic Connectivity is implemented as one of the following types of Heterogeneous Services agents:

- ODBC agent for accessing ODBC data providers
- OLE DB agent for accessing OLE DB data providers that support SQL processing—sometimes referred to as **OLE DB (SQL)**
- OLE DB agent for accessing OLE DB data providers without SQL processing support—sometimes referred to as **OLE DB (FS***)*

Each user session receives its own dedicated agent process spawned by the first use in that user session of the database link to the non-Oracle system. The agent process ends when the user session ends.

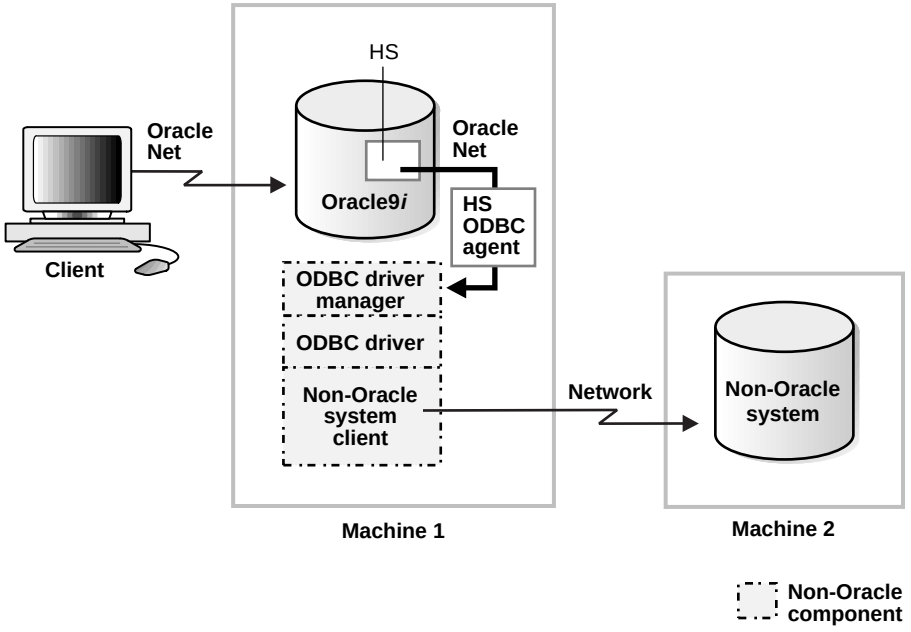
Generic Connectivity Architecture

To access the non-Oracle data store using Generic Connectivity, the agents work with an ODBC or OLE DB driver. The Oracle database server provides support for the ODBC or OLE DB driver interface. The driver that you use must be on the same platform as the agent. The non-Oracle data stores can reside on the same machine as the Oracle database server or on a different machine.

Oracle and Non-Oracle Systems on Separate Machines

Figure 7–1 shows an example of a configuration in which an Oracle and non-Oracle database are on separate machines, communicating through a Heterogeneous Services ODBC agent.

Figure 7–1 Oracle and Non-Oracle Systems on a Separate Machines



In this configuration:

1. A client connects to the Oracle database server through Oracle Net.

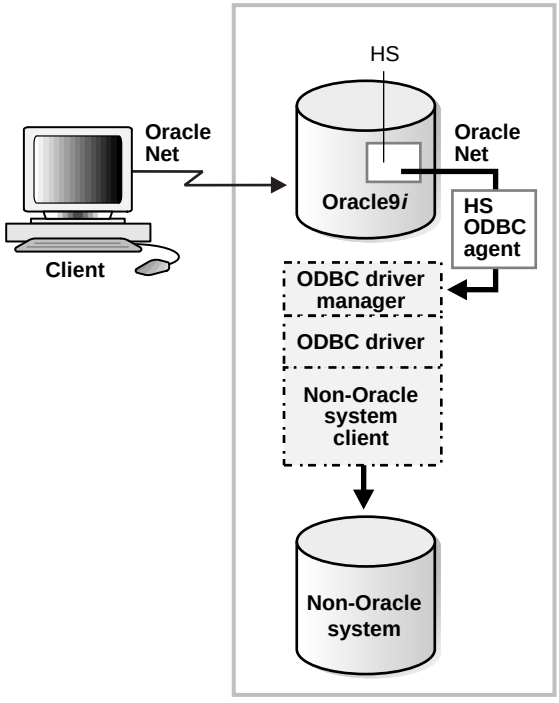
2. The Heterogeneous Services component of the Oracle database server connects through Oracle Net to the Heterogeneous Services ODBC agent.
3. The agent communicates with the following non-Oracle components:
   - An ODBC driver manager
   - An ODBC driver
   - A non-Oracle client application

This client connects to the non-Oracle data store through a network.

Oracle and Non-Oracle Systems on the Same Machine

Figure 7–2 shows an example of a different configuration in which an Oracle and non-Oracle database are on the same machine, again communicating through an Heterogeneous Services ODBC agent.

Figure 7–2 Oracle and non-Oracle Systems on the Same Machine



In this configuration:

1. A client connects to the Oracle database server through Oracle Net.
2. The Heterogeneous Services component of the Oracle database server connects through Oracle Net to the Heterogeneous Services ODBC agent.
3. The agent communicates with the following non-Oracle components:
   - An ODBC driver manager
   - An ODBC driver

The driver then allows access to the non-Oracle data store.

> **Note:** The ODBC driver may require non-Oracle client libraries even if the non-Oracle database is located on the same machine.

SQL Execution

SQL statements sent using a Generic Connectivity agent are executed differently depending on the type of agent you are using: ODBC, OLE DB (SQL), or OLE DB (FS). For example, if a SQL statement involving tables is sent using an ODBC agent for a file-based storage system, the file can be manipulated as if it were a table in a relational database. The naming conventions used at the non-Oracle system can also depend on whether you are using an ODBC or OLE DB agent.

Data Type Mapping

The Oracle database server maps the data types used in ODBC and OLE DB compliant data sources to supported Oracle data types. When the results of a query are returned, the Oracle database server converts the ODBC or OLE DB data types to Oracle data types. For example, the ODBC data type `SQL_TIMESTAMP` and the OLE DB data type `DBTYPE_DBTIMESTAMP` are converted to Oracle's `DATE` data type.

Generic Connectivity Restrictions

Generic Connectivity restrictions include:

- A table including a BLOB column must have a separate column that serves as a primary key
- BLOB / CLOB data cannot be read through passthrough queries
- Updates or deletes that include unsupported functions within a `WHERE` clause are not allowed
- Stored procedures are not supported
- Generic Connectivity agents cannot participate in distributed transactions; they support single-site transactions only

Supported Oracle SQL Statements and Functions

Generic Connectivity supports the following statements, but only if the ODBC or OLE DB driver and non-Oracle system can execute them *and* the statements contain supported Oracle SQL functions:

- DELETE
- INSERT
- SELECT
- UPDATE

Only a limited set of functions are assumed to be supported by the non-Oracle system. Most Oracle functions have no equivalent function in this limited set. Consequently, although post-processing is performed by the Oracle database server, many Oracle functions are not supported by Generic Connectivity, possibly impacting performance.

If an Oracle SQL function is not supported by Generic Connectivity, then this function is not supported in DELETE, INSERT, or UPDATE statements. In SELECT statements, these functions are evaluated by the Oracle database server and post-processed after they are returned from the non-Oracle system.

If an unsupported function is used in a DELETE, INSERT, or UPDATE statement, it generates this Oracle error:

```
ORA-02070: database db_link_name does not support function in this context
```

Generic Connectivity assumes that the following minimum set of SQL functions is supported:

- AVG(*exp*)
- LIKE(*exp*)
- COUNT(*)
- MAX(*exp*)
- MIN(*exp*)
- NOT

Configuring Generic Connectivity Agents

To implement Generic Connectivity on a non-Oracle data source, you must set the agent parameters.

This section contains the following topics:

Creating the Initialization File

You must create and customize an initialization file for your Generic Connectivity agent. Oracle Corporation supplies sample initialization files named `iniths`*`agent`*`.ora`, where *`agent`* is `odbc` or `oledb`, indicating which agent the sample file can be used for, as in the following:

```
inithsodbc.ora
inithsoledb.ora
```

The sample files are stored in the `$ORACLE_HOME/hs/admin directory`.

To create an initialization file for an ODBC or OLE DB agent, copy the applicable sample initialization file and rename the file to `init`*`HS_SID`*`.ora`, where *HS_SID* is the system identifier you want to use for the instance of the non-Oracle system to which the agent connects.

The *HS_SID* is also used to identify how to connect to the agent when you configure the listener by modifying the `listener.ora` file. The *HS_SID* you add to the `listener.ora` file must match the *HS_SID* in an `init`*`HS_SID`*`.ora` file, because the agent spawned by the listener searches for a matching `init`*`HS_SID`*`.ora` file. That is how each agent process gets its initialization information. When you copy and rename your `init`*`HS_SID`*`.ora` file, ensure it remains in the `$ORACLE_HOME/hs/admin` directory.

Editing the Initialization File

Customize the `init`*`HS_SID`*`.ora` file by setting the parameter values used for Generic Connectivity agents to values appropriate for your system, agent, and drivers. You must edit the `init`*`HS_SID`*`.ora` file to change the `HS_FDS_`

CONNECT_INFO initialization parameter. HS_FDS_CONNECT_INFO specifies the information required for connecting to the non-Oracle system.

> **See Also:** "Setting Initialization Parameters" on page 4-5 for more information on parameters

Set the parameter values as follows:

```
[SET][PRIVATE] parameter=value
```

where:

- [SET]and [PRIVATE]are optional keywords. If you do not specify either SET or PRIVATE, the parameter and value are simply used as an initialization parameter for the agent.

  SET specifies that in addition to being used as an initialization parameter, the parameter value is set as an environment variable for the agent process.

  PRIVATE specifies that the parameter value is private and not transferred to the Oracle database server and does not appear in V$ tables or in an graphical user interfaces.

  SET PRIVATE specifies that the parameter value is set as an environment variable for the agent process and is also private (not transferred to the Oracle database server, not appearing in V$ tables or graphical user interfaces).

- *parameter* is the Heterogeneous Services initialization parameter that you are specifying. See "Setting Initialization Parameters" on page 4-5 for a description of all Heterogeneous Services parameters and their possible values. The parameter is case-sensitive.

- *value* is the value you want to specify for the Heterogeneous Services parameter. The value is case-sensitive.

For example, to enable tracing for an agent, set the HS_FDS_TRACE_LEVEL parameter as follows:

```
HS_FDS_TRACE_LEVEL=ON
```

Typically, most parameters are only needed as initialization parameters, so you do not need to use SET or PRIVATE. Use SET for parameter values that the drivers or non-Oracle system need as environment variables.

PRIVATE is only supported for the follow Heterogeneous Services parameters:

- HS_FDS_CONNECT_INFO

- HS_FDS_SHAREABLE_NAME
- HS_FDS_TRACE_LEVEL

You should only use PRIVATE for these parameters if the parameter value includes sensitive information such as a username or password.

Setting Initialization Parameters for an ODBC-based Data Source

The settings for the initialization parameters vary depending on the type of operating system.

Setting Agent Parameters on Windows NT

Specify a file data source name (DSN) or a system DSN which has previously been defined using the ODBC Driver Manager.

When connecting using a file DSN, specify the value as follows:

```
HS_FDS_CONNECT_INFO=FILEDSN=file_dsn
```

When connecting using a system DSN, specify the value as follows:

```
HS_FDS_CONNECT_INFO=system_dsn
```

If you are connecting to the data source through the driver for that data source, precede the DSN by the name of the driver, followed by a semi-colon (;).

Setting Parameters on NT: Example Assume a system DSN has been defined in the Windows ODBC Data Source Administrator. In order to connect to this SQL Server database through the gateway, the following line is required in init*HS_SID*.ora:

```
HS_FDS_CONNECT_INFO=sqlserver7
```

where *sqlserver7* is the name of the system DSN defined in the Windows ODBC Data Source Administrator.

The following procedure enables you to define a system DSN in the Windows ODBC Data Source Administrator:

1. From the **Start** menu, choose **Settings** > **Control Panel** and select the **ODBC** icon.
2. Select the **system DSN** tab to display the system data sources.
3. Click **Add**.

4. From the list of installed ODBC drivers, select the name of the driver that the data source will use. For example, select **SQL Server**.
5. Click **Finish**.
6. Enter a name for the DSN and an optional description. Enter other information depending on the ODBC driver. For example, for SQL Server enter the SQL Server machine.

> **Note:** The name entered for the DSN must match the value of the parameter HS_FDS_CONNECT_INFO that is specified in init*HS_SID*.ora.

7. Continue clicking **Next** and answering the prompts until you click **Finish**.
8. Click **OK** until you exit the ODBC Data Source Administrator.

Setting Agent Parameters on UNIX platforms

Specify a DSN and the path of the ODBC shareable library, as follows:

```
HS_FDS_CONNECT_INFO=dsn_value
HS_FDS_SHAREABLE_NAME=full_odbc_library_path_of_odbc_driver
```

HS_FDS_CONNECT_INFO is required for all platforms for an ODBC agent. HS_FDS_SHAREABLE_NAME is required on UNIX platforms for an ODBC agent. Other initialization parameters have defaults or are optional. You can use the default values and omit the optional parameters, or you can specify the parameters with values tailored for your installation.

> **Note:** Before deciding to accept the default values or change them, see "Setting Initialization Parameters" on page 4-5 for detailed information on all the initialization parameters.

Setting Parameters on UNIX: Example Assume that the odbc.ini file for connecting to Informix using the Intersolve ODBC driver is located in /opt/odbc and includes the following information:

```
[ODBC Data Sources]
Informix=INTERSOLV 3.11 Informix Driver

[Informix]
```

```
Driver=/opt/odbc/lib/ivinf13.so
Description=Informix
Database=personnel@osf_inf72
HostName=osf
LogonID=uid
Password=pwd
```

In order to connect to this Informix database through the gateway, the following lines are required in `init`*`HS_SID`*`.ora`:

```
HS_FDS_CONNECT_INFO=Informix
HS_FDS_SHAREABLE_NAME=/opt/odbc/lib/libodbc.so
set INFORMIXDIR=/users/inf72
set INFORMIXSERVER=osf_inf72
set ODBCINI=/opt/odbc/odbc.ini
```

Note that the set statements are optional as long as they are specified in the working account. Each database has its own set statements.

The `HS_FDS_CONNECT_INFO` parameter value must match the ODBC data source name in the `odbc.ini` file.

Setting Initialization Parameters for an OLE DB-based Data Source

You can only set these parameters on the Windows NT platform.

Specify a data link (UDL) that has previously been defined:

`SET|PRIVATE|SET PRIVATE` `HS_FDS_CONNECT_INFO="UDLFILE=`*`data_link`*`"`

> **Note:** If the parameter value includes an equal sign (=), then it must be surrounded by quotation marks.

`HS_FDS_CONNECT_INFO` is required for an OLE DB agent. Other initialization parameters have defaults or are optional. You can use the default values and omit the optional parameters, or you can specify the parameters with values tailored for your installation.

> **Note:** Before deciding to accept the default values or change them, see "Setting Initialization Parameters" on page 4-5 for detailed information on all the initialization parameters.

ODBC Connectivity Requirements

To use an ODBC agent, you must have an ODBC driver installed on the same machine as the Oracle database server. On Windows NT, you must have an ODBC driver manager also located on the same machine. The ODBC driver manager and driver must meet the following requirements:

- On Windows NT machines, a thread-safe, 32-bit ODBC driver Version 2.x or 3.x is required. You can use the native driver manager supplied with your Windows NT system.
- On UNIX machines, ODBC driver Version 2.5 is required. A driver manager is not required.

The ODBC driver and driver manager on Windows NT must conform to ODBC application program interface (API) conformance Level 1 or higher. If the ODBC driver or driver manager does not support multiple active ODBC cursors, then it restricts the complexity of SQL statements that you can execute using Generic Connectivity.

The ODBC driver you use must support all of the core `SQL ODBC` data types and should support SQL grammar level `SQL_92`. The ODBC driver should also expose the following ODBC APIs:

Table 7–1 ODBC Functions

ODBC Function	Comment
`SQLAllocConnect`	-
`SQLAllocEnv`	-
`SQLAllocStmt`	-
`SQLBindCol`	-
`SQLBindParameter`	-
`SQLColumns`	-
`SQLConnect`	-
`SQLDescribeCol`	-
`SQLDisconnect`	-
`SQLDriverConnect`	-
`SQLError`	-
`SQLExecDirect`	-

Table 7–1 ODBC Functions(Cont.)

ODBC Function	Comment
SQLExecute	-
SQLExtendedFetch	Recommended if used by the non-Oracle system.
SQLFetch	-
SQLForeignKeys	Recommended if used by the non-Oracle system.
SQLFreeConnect	-
SQLFreeEnv	-
SQLFreeStmt	-
SQLGetConnectOption	-
SQLGetData	-
SQLGetFunctions	-
SQLGetInfo	-
SQLGetTypeInfo	-
SQLNumParams	Recommended if used by the non-Oracle system.
SQLNumResultCols	-
SQLParamData	-
SQLPrepare	-
SQLPrimaryKeys	Recommended if used by the non-Oracle system.
SQLProcedureColumns	Recommended if used by the non-Oracle system.
SQLProcedures	Recommended if used by the non-Oracle system.
SQLPutData	-
SQLRowCount	-
SQLSetConnectOption	-
SQLSetStmtOption	-
SQLStatistics	-
SQLTables	-
SQLTransact	Recommended if used by the non-Oracle system.

OLE DB (SQL) Connectivity Requirements

These requirements apply to OLE DB data providers that have an SQL processing capability and expose the OLE DB interfaces.

Generic Connectivity passes the username and password to the provider when calling `IDBInitialize::Initialize()`.

OLE DB (SQL) connectivity requires that the data provider expose the following OLE DB interfaces:

Table 7–2 OLE DB (SQL) Interfaces

Interface	Methods
`IAccessor`	`CreateAccessor`, `ReleaseAccessor`
`IColumnsInfo`	`GetColumnsInfo` (Command and Rowset objects)
`ICommand`	`Execute`
`ICommandPrepare`	`Prepare`
`ICommandProperties`	`SetProperties`
`ICommandText`	`SetCommandText`
`ICommandWithParameters`	`GetParameterInfo`
`IDBCreateCommand`	`CreateCommand`
`IDBCreateSession`	`CreateSession`
`IDBInitialize`	`Initialize`
`IDBSchemaRowset`	`GetRowset` (tables, columns, indexes; optionally also procedures, procedure parameters)
`IErrorInfo`[1]	`GetDescription`, `GetSource`
`IErrorRecords`	`GetErrorInfo`
`ILockBytes (OLE)`[2]	`Flush, ReadAt, SetSize, Stat, WriteAt`
`IRowset`	`GetData, GetNextRows, ReleaseRows, RestartPosition`
`IStream (OLE)`	`Read, Seek, SetSize, Stat, Write`

Table 7–2 OLE DB (SQL) Interfaces

Interface	Methods
ISupportErrorInfo	InterfaceSupportsErrorInfo
ITransactionLocal (optional)	StartTransaction, Commit, Abort

[1] You can also use IErrorLookup with the GetErrorDescription method.

[2] Required only if BLOBs are used in the OLE DB provider.

OLE DB (FS) Connectivity Requirements

These requirements apply to OLE DB data providers that do not have SQL processing capabilities. If the provider exposes them, then OLE DB (FS) connectivity uses OLE DB Index interfaces.

OLE DB Interfaces for Data Providers to Expose

OLE DB (FS) connectivity requires that the data provider expose the following OLE DB interfaces:

Table 7–3 OLE DB (FS) Interfaces

Interface	Methods
IAccessor	CreateAccessor, ReleaseAccessor
IColumnsInfo	GetColumnsInfo (Command and Rowset objects)
IOpenRowset	OpenRowset
IDBCreateSession	CreateSession
IRowsetChange	DeleteRows, SetData, InsertRow
IRowsetLocate	GetRowsByBookmark
IRowsetUpdate	Update (optional)
IDBInitialize	Initialize, Uninitialize
IDBSchemaRowset	GetRowset (tables, columns, indexes; optionally also procedures, procedure parameters)
ILockBytes (OLE)[1]	Flush, ReadAt, SetSize, Stat, WriteAt

Table 7–3 OLE DB (FS) Interfaces (Cont.)

Interface	Methods
IRowsetIndex[2]	SetRange
IErrorInfo[3]	GetDescription, GetSource
IErrorRecords	GetErrorInfo
IRowset	GetData, GetNextRows, ReleaseRows, RestartPosition
IStream (OLE)	Read, Seek, SetSize, Stat, Write
ITransactionLocal (optional)	StartTransaction, Commit, Abort
ISupportErrorInfo	InterfaceSupportsErrorInfo
ITableDefinition	CreateTable, DropTable
IDBProperties	SetProperties

[1] Required only if BLOBs are used in the OLE DB provider.

[2] Required only if indexes are used in the OLE DB provider.

[3] You can use IErrorLookup with the GetErrorDescription method as well.

Because OLE DB (FS) connectivity is generic, it can connect to a number of different data providers that expose OLE DB interfaces. Every such data provider must meet the certain requirements.

Note: The data provider must expose bookmarks. This enables tables to be updated. Without bookmarks being exposed, the tables are read-only.

Data Source Properties

The OLE DB data source must support the following initialization properties:

- DBPROP_INIT_DATASOURCE
- DBPROP_AUTH_USERID

Note: Required if the userid has been supplied in the security file

- DBPROP_AUTH_PASSWORD

> **Note:** Required if the userid and password have been supplied in the security file

The OLE DB data source must also support the following rowset properties:

- DBPROP_IRowsetChange = TRUE
- DBPROP_UPDATABILITY = CHANGE+DELETE+INSERT
- DBPROP_OWNUPDATEDELETE = TRUE
- DBPROP_OWNINSERT = TRUE
- DBPROP_OTHERUPDATEDELETE = TRUE
- DBPROP_CANSCROLLBACKWARDS = TRUE
- DBPROP_IRowsetLocate = TRUE
- DBPROP_OTHERINSERT = FALSE

A

Heterogeneous Services Initialization Parameters

The Heterogeneous Services initialization parameter file contains configuration settings stored as a text file.

This section contains the following topics:

- HS_OPEN_CURSORS
- HS_ROWID_CACHE_SIZE
- HS_RPC_FETCH_REBLOCKING
- HS_RPC_FETCH_SIZE
- HS_TIME_ZONE
- IFILE

See Also:

- "Setting Initialization Parameters" on page 4-5 contains instructions for setting the Heterogeneous Services initialization parameters
- "Configuring Generic Connectivity Agents" on page 7-8 contains instructions for setting Heterogeneous Services initialization parameters specific to Generic Connectivity

HS_COMMIT_POINT_STRENGTH

Default value: 0

Range of values: 0 to 255

Specifies a value that determines the commit point site in a heterogeneous distributed transaction. HS_COMMIT_POINT_STRENGTH is similar to COMMIT_POINT_STRENGTH, described in the *Oracle9i Database Reference*.

Set HS_COMMIT_POINT_STRENGTH to a value relative to the importance of the site that is the commit point site in a distributed transaction. The Oracle database server or non-Oracle system with the highest commit point strength becomes the commit point site. To ensure that a non-Oracle system *never* becomes the commit point site, set the value of HS_COMMIT_POINT_STRENGTH to zero.

HS_COMMIT_POINT_STRENGTH is important only if the non-Oracle system can participate in the two-phase protocol as a regular two-phase commit partner and as the commit point site. This is only the case if the transaction model is two-phase commit confirm (2PCC).

HS_DB_DOMAIN

Default value: WORLD

Range of values: 1 to 119 characters

Specifies a unique network sub-address for a non-Oracle system. HS_DB_DOMAIN is similar to DB_DOMAIN, described in the *Oracle9i Database Reference*. HS_DB_DOMAIN is required if you use the Oracle Names server. HS_DB_NAME and HS_DB_DOMAIN define the global name of the non-Oracle system.

Note: HS_DB_NAME and HS_DB_DOMAIN must combine to form a unique address.

HS_DB_INTERNAL_NAME

Default value: 01010101

Range of values: 1 to 16 hexadecimal characters

Specifies a unique hexadecimal number identifying the instance to which the Heterogeneous Services agent is connected. This parameter's value is used as part of a transaction ID when global name services are activated. Specifying a non-unique number can cause problems when two-phase commit recovery actions are necessary for a transaction.

HS_DB_NAME

Default value: HO

Range of values: 1 to 8 lowercase characters

Specifies a unique alphanumeric name for the data store given to the non-Oracle system. This name identifies the non-Oracle system within the cooperative server environment. `HS_DB_NAME` and `HS_DB_DOMAIN` define the global name of the non-Oracle system.

HS_DESCRIBE_CACHE_HWM

Default value: 100

Range of values: 1 to 4000

Specifies the maximum number of entries in the describe cache used by Heterogeneous Services. This limit is known as the describe cache high water mark. The cache contains descriptions of the mapped tables that Heterogeneous Services reuses so that it does not have to re-access the non-Oracle data store.

If you are accessing many mapped tables, then increase the high water mark to improve performance. Note that increasing the high water mark improves performance at the cost of memory usage.

HS_FDS_CONNECT_INFO

Default value: None

Range of values: Not applicable

Specifies the information needed to bind to the data provider, that is, the non-Oracle system. For Generic Connectivity, you can bind to an ODBC-based data

source or to an OLE DB-based data source. The information that you provide depends on the platform and whether the data source is ODBC or OLE DB-based.

This parameter is required if you are using Generic Connectivity.

ODBC-based Data Source on Windows

You can use either a file DSN (data source name) or a system DSN as follows:

- When connecting using a file DSN the parameter format is:

  HS_FDS_CONNECT_INFO=FILEDSN=*file_dsn*

- When connecting using a system DSN the parameter format is:

  HS_FDS_CONNECT_INFO=*system_dsn*

If you are connecting to the data source through the driver for that data source, then precede the DSN by the name of the driver, followed by a semi-colon (;).

ODBC-based Data Source on UNIX

Use a DSN with the following format:

HS_FDS_CONNECT_INFO=dsn

OLE DB-based Data Source (Windows NT Only)

Use a universal data link (UDL) with the following formats:

- HS_FDS_CONNECT_INFO="UDLFILE=*data_link*"

> **Note:** Whenever the parameter value includes an equal sign (=), it must be enclosed in quotation marks.

HS_FDS_DEFAULT_SCHEMA_NAME

Default value: None

Range of values: Not applicable

Specifies a default value for the owner column that will be returned in the ddtrans, when the value of the owner is null. For example:

```
HS_FDS_DEFAULT_SCHEMA_NAME = PUBLIC
```

HS_FDS_SHAREABLE_NAME

Default value: None

Range of values: Not applicable

Specifies the full path name to the ODBC library. This parameter is required when you are using Generic Connectivity to access data from an ODBC provider on a UNIX machine.

HS_FDS_TRACE_LEVEL

Default value: OFF

Range of values: ON or OFF

Specifies whether error tracing is enabled or disabled for Generic Connectivity. Enable the tracing to see which error messages occur when you encounter problems. The results are written to a Generic Connectivity log file, in the /log directory under the $ORACLE_HOME directory.

HS_LANGUAGE

Default value: System specific

Range of values: Any valid language name (up to 255 characters)

Provides Heterogeneous Services with character set, language, and territory information of the non-Oracle data source. The value must use the following format:

language[*_territory.character_set*]

Note: The national language support initialization parameters affect error messages, the data for the SQL Service, and parameters in distributed external procedures.

Character Sets

Ideally, the character sets of the Oracle database server and the non-Oracle data source are the same. If they are not the same, Heterogeneous Services attempts to translate the character set of the non-Oracle data source to the Oracle database character set, and back again. The translation can degrade performance. In some cases, Heterogeneous Services cannot translate a character from one character set to another.

> **Note:** The specified character set must be a superset of the operating system character set on the platform where the agent is installed.

Language

The language component of the HS_LANGUAGE initialization parameter determines:

- Day and month names of dates
- AD, BC, PM, and AM symbols for date and time
- Default sorting mechanism

Note that Oracle9*i* does not determine the language for error messages for the generic Heterogeneous Services messages (ORA-25000 through ORA-28000). These are controlled by the session settings in the Oracle database server.

> **Note:** Use the HS_NLS_DATE_LANGUAGE initialization parameter to set the day and month names, and the AD, BC, PM, and AM symbols for dates and time independently from the language.

Territory

The territory clause specifies the conventions for day and week numbering, default date format, decimal character and group separator, and ISO and local currency symbols. Note that:

- You can override the date format using the initialization parameter HS_NLS_DATE_FORMAT.
- The level of National Language Support between the Oracle database server and the non-Oracle data source depends on how the driver is implemented. See

the installation documentation for your platform for more information about the level of National Language Support.

HS_LONG_PIECE_TRANSFER_SIZE

Default value: 64 KB

Range of values: Any value up to 2 GB

Sets the size of the piece of LONG data being transferred. A smaller piece size means less memory requirement, but more round trips to fetch all the data. A larger piece size means fewer round trips, but more of a memory requirement to store the intermediate pieces internally. Thus, the initialization parameter can be used to tune a system for the best performance, with the best trade-off between round trips and memory requirements.

HS_NLS_DATE_FORMAT

Default value: Value determined by HS_LANGUAGE parameter

Range of values: Any valid date format mask (up to 255 characters)

Defines the date format for dates used by the target system. This parameter has the same function as the NLS_DATE_FORMAT parameter for an Oracle database server. The value of can be any valid date mask listed in the *Oracle9i SQL Reference*, but must match the date format of the target system. For example, if the target system stores the date February 14, 2001 as 2001/02/14, set the parameter to yyyy/mm/dd. Note that characters must be lowercase.

HS_NLS_DATE_LANGUAGE

Default value: Value determined by HS_LANGUAGE parameter

Range of values: Any valid NLS_LANGUAGE value (up to 255 characters)

Specifies the language used in character date values coming from the non-Oracle system. Date formats can be language independent. For example, if the format is dd/mm/yyyy, all three components of the character date are numbers. In the format dd-mon-yyyy, however, the month component is the name abbreviated to three characters. The abbreviation is very much language dependent. For example, the abbreviation for the month April is "apr", which in French is "avr" (Avril).

Heterogeneous Services assumes that character date values fetched from the non-Oracle system are in this format. Also, Heterogeneous Services sends character date bind values in this format to the non-Oracle system.

HS_NLS_NCHAR

Default value: Value determined by HS_LANGUAGE parameter

Range of values: Any valid national character set (up to 255 characters)

Informs Heterogeneous Services of the value of the national character set of the non-Oracle data source. This value is the non-Oracle equivalent to the NATIONAL CHARACTER SET parameter setting in the Oracle CREATE DATABASE statement. The HS_NLS_NCHAR value should be the character set ID of a character set supported by the Oracle NLSRTL library.

See Also: HS_LANGUAGE on page A-6

HS_NLS_TIMESTAMP_FORMAT

Default value: Derived from NLS_TERRITORY

Range of values: Any valid datetime format mask

Defines the timestamp format for dates used by the target system. This parameter has the same function as the NLS_TIMESTAMP_FORMAT parameter for an Oracle database server. The value of can be any valid timestamp mask listed in the *Oracle9i SQL Reference*, but it must match the date format of the target system. Note that characters must be lowercase. For example:

```
HS_NLS_TIMESTAMP_FORMAT = yyyy-mm-dd hh:mi:ss.ff
```

HS_NLS_TIMESTAMP_TZ_FORMAT

Default value: Derived from NLS_TERRITORY

Range of values: Any valid datetime with time zone format mask

Defines the default timestamp with time zone format for the timestamp with time zone format used by the target system. This parameter has the same function as the NLS_TIMESTAMP_TZ_FORMAT parameter for an Oracle database server. The value

of can be any valid timestamp with time zone mask listed in the *Oracle9i SQL Reference*, but must match the date format of the target system. Note that characters must be lowercase. For example:

```
HS_NLS_TIMESTAMP_TZ_FORMAT = yyyy-mm-dd hh:mi:ss.ff tzh:tzm
```

HS_OPEN_CURSORS

Default value: 50

Range of values: 1 - value of Oracle's OPEN_CURSORS initialization parameter

Defines the maximum number of cursors that can be open on one connection to a non-Oracle system instance.

The value never exceeds the number of open cursors in the Oracle database server. Therefore, setting the same value as the OPEN_CURSORS initialization parameter in the Oracle database server is recommended.

HS_ROWID_CACHE_SIZE

Default value: 3

Range of values: 1 to 32767

Specifies the size of the Heterogeneous Services cache containing the non-Oracle system equivalent of ROWIDs. The cache contains non-Oracle system ROWIDs needed to support the WHERE CURRENT OF clause in a SQL statement or a SELECT FOR UPDATE statement.

When the cache is full, the first slot in the cache is reused, then the second, and so on. Only the last HS_ROWID_CACHE_SIZE non-Oracle system ROWIDs are cached.

HS_RPC_FETCH_REBLOCKING

Default value: ON

Range of values: OFF, ON

Controls whether Heterogeneous Services attempts to optimize performance of data transfer between the Oracle database server and the Heterogeneous Services agent connected to the non-Oracle data store.

The following values are possible:

- OFF disables reblocking of fetched data so that data is immediately sent from agent to server
- ON enables reblocking, which means that data fetched from the non-Oracle system is buffered in the agent and is not sent to the Oracle database server until the amount of fetched data is equal or higher than HS_RPC_FETCH_SIZE. However, any buffered data is returned immediately when a fetch indicates that no more data exists or when the non-Oracle system reports an error.

HS_RPC_FETCH_SIZE

Default value: 4000

Range of values: Decimal integer (byte count)

Tunes internal data buffering to optimize the data transfer rate between the server and the agent process.

Increasing the value can reduce the number of network round trips needed to transfer a given amount of data, but also tends to increase data bandwidth and to reduce response time or **latency** as measured between issuing a query and completion of all fetches for the query. Nevertheless, increasing the fetch size can increase latency for the initial fetch results of a query, because the first fetch results are not transmitted until additional data is available.

After the gateway is installed and configured, you can use the gateway to access non-Oracle database system data, pass non-Oracle database system commands from applications to the non-Oracle database system database, perform distributed queries, and copy data.

HS_TIME_ZONE

Default value for '[+|-] hh:mm': Derived from NLS_TERRITORY

Range of values for '[+|-] hh:mm': Any valid datetime format mask

Specifies the default local time zone displacement for the current SQL session. The format mask, [+|-]hh:mm, is specified to indicate the hours and minutes before or

after UTC (Coordinated Universal Time—formerly Greenwich Mean Time). For example:

```
HS_TIME_ZONE = [+ | -] hh:mm
```

IFILE

Default value: None

Range of values: Valid parameter filenames

Use `IFILE` to embed another initialization file within the current initialization file; the value should be an absolute path and should not contain environment variables; the three levels of nesting limit does not apply.

See Also: `IFILE` in *Oracle9i Database Reference*

B

Data Type Mapping

Oracle9*i* maps the ANSI data types through ODBC and OLE DB interfaces to supported Oracle data types. When the results of a query are returned, Oracle9i converts the ODBC or OLE DB data types to Oracle data types.

The tables in this appendix show how Oracle maps ANSI data types through ODBC and OLE DB interfaces to supported Oracle data types when it is retrieving data from a non-Oracle system.

This appendix contains the following tables

Mapping ANSI Data Types to Oracle Data Types Through an ODBC Interface

Table B–1 Mapping ANSI Data Types to Oracle Data Types Through an ODBC Interface

ANSI	ODBC	Oracle
NUMERIC(19,0)	SQL_BIGINT	NUMBER(19,0)
N/A	SQL_BINARY	RAW
CHAR	SQL_CHAR	CHAR
DATE	SQL_DATE	DATE
DECIMAL(p,s)	SQL_DECIMAL(p,s)	NUMBER(p,s)
DOUBLE PRECISION	SQL_DOUBLE	FLOAT(49)
FLOAT	SQL_FLOAT	FLOAT(49)
INTEGER	SQL_INTEGER	NUMBER(10)[1]
N/A	SQL_LONGVARBINARY	LONG RAW
N/A	SQL_LONGVARCHAR	LONG[2]
REAL	SQL_REAL	FLOAT(23)
SMALLINT	SQL_SMALLINT	NUMBER(5)
TIME	SQL_TIME	DATE
TIMESTAMP	SQL_TIMESTAMP	DATE
NUMERIC(3,0)	SQL_TINYINT	NUMBER(3)
VARCHAR	SQL_VARCHAR	VARCHAR

[1] It's possible under some circumstance for the INTEGER ANSI data type to map to Precision 38, but it usually maps to Precision 10.

[2] If an ANSI SQL implementation defines a large value for the maximum length of VARCHAR data, then it is possible that ANSI VARCHAR will map to SQL_LONGVARCHAR and Oracle LONG. The same is true for OLE DB DBTYPE_STRING (long attribute).

Note: This table maps ODBC data types into equivalent ANSI and Oracle data types. In some cases equivalence to ANSI data types is not guaranteed to be exact because the ANSI SQL standard delegates definition of numeric precision and maximum length of character data to individual implementations. This table reflects a probable mapping between ANSI and ODBC data types for a typical implementation of ANSI SQL.

Mapping ANSI Data Types to Oracle Data Types Through an OLE DB Interface

Table B–2 Mapping ANSI Data Types to Oracle Data Types Through an OLE DB Interface

ANSI	OLE DB	Oracle
NUMERIC(3,0)	DBTYPE_UI1	NUMBER(3)
NUMERIC(3,0)	DBTYPE_I1	NUMBER(3)
SMALLINT	DBTYPE_UI2	NUMBER(5)
SMALLINT	DBTYPE_I2	NUMBER(5)
NUMERIC(3,0)	DBTYPE_BOOL	NUMBER(5)
INTEGER	DBTYPE_UI4	NUMBER(10)
INTEGER	DBTYPE_I4	NUMBER(10)
NUMERIC(19,0)	DBTYPE_UI8	NUMBER(19,0)
NUMERIC(19,0)	DBTYPE_I8	NUMBER(19,0)
NUMERIC(p,s)	DBTYPE_NUMERIC(p,s)	NUMBER(p,s)
FLOAT	DBTYPE_R4	FLOAT(23)
DOUBLE PRECISION	DBTYPE_R8	FLOAT(49)
N/A	DBTYPE_DECIMAL	FLOAT(49)
VARCHAR	DBTYPE_STR	VARCHAR2
VARCHAR	DBTYPE_WSTR	VARCHAR2
NUMERIC(19,0)	DBTYPE_CY	NUMBER(19,0)
DATE	DBTYPE_DBDATE	DATE

Table B–2 Mapping ANSI Data Types to Oracle Data Types Through an OLE DB Interface

ANSI	OLE DB	Oracle
TIME	DBTYPE_DBTIME	DATE
TIMESTAMP	DBTYPE_TIMESTAMP	DATE
N/A	DBTYPE_BYTES	RAW
N/A	DBTYPE_BYTES (long attribute)	LONG RAW
N/A	DBTYPE_STRING (long attribute)	LONG

C

DBMS_HS_PASSTHROUGH for Pass-Through SQL

The package, DBMS_HS_PASSTHROUGH, contains the procedures and functions for pass-through SQL of Heterogeneous Services. This appendix documents each of them.

This appendix contains these topics:

Summary of Subprograms

Table C–1 DBMS_HS_PASSTHROUGH Package Subprograms

Subprogram	Description
BIND_VARIABLE Procedure	Binds an IN variable positionally with a PL/SQL program variable.
BIND_VARIABLE_NCHAR Procedure	Binds IN variables of type NVARCHAR2.
BIND_VARIABLE_RAW Procedure	Binds IN variables of type RAW.
BIND_OUT_VARIABLE Procedure	Binds an OUT variable with a PL/SQL program variable.
BIND_OUT_VARIABLE_NCHAR Procedure	Binds an OUT variable of data type NVARCHAR2 with a PL/SQL program variable.
BIND_OUT_VARIABLE_RAW Procedure	Binds an OUT variable of data type RAW with a PL/SQL program variable.
BIND_INOUT_VARIABLE Procedure	Binds IN OUT bind variables.
BIND_INOUT_VARIABLE_NCHAR Procedure	Binds IN OUT bind variables of data type NVARCHAR2
BIND_INOUT_VARIABLE_RAW Procedure	Binds IN OUT bind variables of data type RAW
CLOSE_CURSOR Function	Closes the cursor and releases associated memory after the SQL statement has been executed at the non-Oracle system
EXECUTE_IMMEDIATE Function	Executes a SQL statement immediately
EXECUTE_NON_QUERY Function	Executes any SQL statement other than a SELECT statement
FETCH_ROW Function	Fetches rows from a result set
GET_VALUE Procedure	Retrieves the select list items of SELECT statements after a row has been fetched, and retrieves the OUT bind values after the SQL statement has been executed
GET_VALUE_NCHAR Procedure	Retrieves the select list items of SELECT statements after a row has been fetched, and retrieves the OUT bind values after the SQL statement has been executed. This procedure operates on the NVARCHAR2 data type

Table C–1 DBMS_HS_PASSTHROUGH Package Subprograms (Cont.)

Subprogram	Description
GET_VALUE_RAW Procedure	Retrieves the select list items of SELECT statements after a row has been fetched, and retrieves the OUT bind values after the SQL statement has been executed. This procedure operates on the RAW data type
OPEN_CURSOR Function	Opens a cursor for executing a pass-through SQL statement at the non-Oracle system
PARSE Procedure	Parses a SQL statement at non-Oracle system

Subprogram Descriptions

This section contains descriptions for the subprograms of the DBMS_HS_PASSTHROUGH package.

BIND_VARIABLE Procedure

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_VARIABLE_NCHAR Procedure
- BIND_VARIABLE_RAW Procedure

This procedure binds an IN variable positionally with a PL/SQL program variable.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_VARIABLE (
     c      IN    BINARY_INTEGER NOT NULL,
     pos    IN    BINARY_INTEGER NOT NULL,
     val    IN    dty);
```

Where *dty* is one of the following data types:

- DATE

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- NUMBER
- TIMESTAMP
- TIMESTAMP WITH TIMEZONE
- TIMESTAMP WITH LOCAL TIMEZONE
- VARCHAR2

See Also:

- BIND_VARIABLE_NCHAR Procedure
- BIND_VARIABLE_RAW Procedure

Parameters

Table C–2 BIND_VARIABLE Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. The cursor must be opened and parsed using the routines OPEN_CURSOR and PARSE.
pos	Position of the bind variable in the SQL statement. Starts from 1
val	Value that must be passed to the bind variable

Exceptions

Table C–3 BIND_VARIABLE Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid
ORA-28552	The procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range
ORA-28555	A NULL value was passed for a NOT NULL parameter

Pragmas

Purity levels defined: WNDS, RNDS

BIND_VARIABLE_NCHAR Procedure

This procedure binds IN variables of type NVARCHAR2.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_VARIABLE_NCHAR (
     c      IN    BINARY_INTEGER NOT NULL,
     pos    IN    BINARY_INTEGER NOT NULL,
     val    IN    NVARCHAR2);
```

Parameters

Table C–4 BIND_VARIABLE_NCHAR Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed using the routines OPEN_CURSOR and PARSE.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Value that must be passed to the bind variable

Exceptions

Table C–5 BIND_VARIABLE_NCHAR Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range
ORA-28555	A NULL value was passed for a NOT NULL parameter

Pragmas

Purity level defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure

BIND_VARIABLE_RAW Procedure

This procedure binds IN variables of type RAW.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW (
     c      IN    BINARY_INTEGER NOT NULL,
     pos    IN    BINARY_INTEGER NOT NULL,
     val    IN    RAW);
```

Parameters

Table C–6 BIND_VARIABLE_RAW Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed using the routines OPEN_CURSOR and PARSE.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Value that must be passed to the bind variable

Exceptions

Table C–7 BIND_VARIABLE_RAW Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid

Table C–7 BIND_VARIABLE_RAW Procedure Exceptions

Exception	Description
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range
ORA-28555	A NULL value was passed for a NOT NULL parameter

Pragmas

Purity level defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- Subprogram Descriptions
- BIND_OUT_VARIABLE Procedure

BIND_OUT_VARIABLE Procedure

This procedure binds an OUT variable with a PL/SQL program variable.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE (
     c         IN    BINARY_INTEGER NOT NULL,
     pos       IN    BINARY_INTEGER NOT NULL,
     val       OUT   dty);
```

Where *dty* is one of

- DATE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- NUMBER
- TIMESTAMP
- TIMESTAMP WITH TIMEZONE

- TIMESTAMP WITH LOCAL TIMEZONE
- VARCHAR2

> **See Also:** BIND_INOUT_VARIABLE_NCHAR Procedure for more information about OUT variables of data type RAW

Parameters

Table C–8 BIND_OUT_VARIABLE Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable will store its value. The package will remember only the size of the variable. After the SQL statement is executed, you can use GET_VALUE to retrieve the value of the OUT parameter. The size of the retrieved value should not exceed the size of the parameter that was passed using BIND_OUT_VARIABLE.

Exceptions

Table C–9 BIND_OUT_VARIABLE Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- Subprogram Descriptions
- BIND_VARIABLE_NCHAR Procedure
- GET_VALUE Procedure

BIND_OUT_VARIABLE_NCHAR Procedure

This procedure binds an OUT variable of data type NVARCHAR2 with a PL/SQL program variable.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE (
     c       IN    BINARY_INTEGER NOT NULL,
     pos     IN    BINARY_INTEGER NOT NULL,
     val     OUT   NVARCHAR2);
```

Parameters

Table C–10 BIND_OUT_VARIABLE_NCHAR Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable will store its value. The package will remember only the size of the variable. After the SQL statement is executed, you can use GET_VALUE to retrieve the value of the OUT parameter. The size of the retrieved value should not exceed the size of the parameter that was passed using BIND_OUT_VARIABLE_RAW.

Exceptions

Table C–11 BIND_OUT_VARIABLE_NCHAR Parameter Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Pragmas defined: WNDS, RNDS

BIND_OUT_VARIABLE_RAW Procedure

This procedure binds an OUT variable of data type RAW with a PL/SQL program variable.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE (
     c       IN    BINARY_INTEGER NOT NULL,
     pos     IN    BINARY_INTEGER NOT NULL,
     val     OUT   RAW);
```

Parameters

Table C–12 BIND_OUT_VARIABLE_RAW Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.

Table C–12 BIND_OUT_VARIABLE_RAW Procedure Parameters

Parameter	Description
val	Variable in which the OUT bind variable will store its value. The package will remember only the size of the variable. After the SQL statement is executed, you can use GET_VALUE to retrieve the value of the OUT parameter. The size of the retrieved value should not exceed the size of the parameter that was passed using BIND_OUT_VARIABLE_RAW.

Exceptions

Table C–13 BIND_OUT_VARIABLE_RAW Parameter Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Pragmas defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_OUT_VARIABLE Procedure
- Subprogram Descriptions
- BIND_VARIABLE_NCHAR Procedure
- GET_VALUE Procedure

BIND_INOUT_VARIABLE Procedure

This procedure binds IN OUT bind variables.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE (
     c       IN     BINARY_INTEGER NOT NULL,
     pos     IN     BINARY_INTEGER NOT NULL,
     val     IN OUT <dty>);
```

Where *dty* is one of

- DATE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- NUMBER
- TIMESTAMP
- TIMESTAMP WITH TIMEZONE
- TIMESTAMP WITH LOCAL TIMEZONE
- VARCHAR2

Parameters

Table C–14 BIND_INOUT_VARIABLE Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	This value will be used for two purposes: ■ To provide the IN value before the SQL statement is executed ■ To determine the size of the OUT value

Exceptions

Table C–15 BIND_INOUT_VARIABLE Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- BIND_OUT_VARIABLE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- Subprogram Descriptions
- BIND_VARIABLE_NCHAR Procedure
- GET_VALUE Procedure

BIND_INOUT_VARIABLE_NCHAR Procedure

This procedure binds IN OUT bind variables of data type NVARCHAR2.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_NCHAR (
     c        IN     BINARY_INTEGER NOT NULL,
     pos      IN     BINARY_INTEGER NOT NULL,
     val      IN OUT NVARCHAR2);
```

Parameters

Table C–16 BIND_INOUT_VARIABLE_NCHAR Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed' using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	This value will be used for two purposes: ■ To provide the IN value before the SQL statement is executed ■ To determine the size of the out value

Exceptions

Table C–17 BIND_INOUT_VARIABLE_NCHAR Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Pragmas defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_INOUT_VARIABLE Procedure
- BIND_OUT_VARIABLE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- Subprogram Descriptions
- BIND_VARIABLE_NCHAR Procedure
- GET_VALUE Procedure

BIND_INOUT_VARIABLE_RAW Procedure

This procedure binds IN OUT bind variables of data type RAW.

Syntax

```
DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW (
     c        IN     BINARY_INTEGER NOT NULL,
     pos      IN     BINARY_INTEGER NOT NULL,
     val      IN OUT RAW);
```

Parameters

Table C–18 BIND_INOUT_VARIABLE_RAW Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	This value will be used for two purposes: ■ To provide the IN value before the SQL statement is executed ■ To determine the size of the OUT value

Exceptions

Table C–19 BIND_INOUT_VARIABLE_RAW Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Pragmas defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- BIND_INOUT_VARIABLE Procedure
- BIND_OUT_VARIABLE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- Subprogram Descriptions
- BIND_VARIABLE_NCHAR Procedure
- GET_VALUE Procedure

CLOSE_CURSOR Function

This function closes the cursor and releases associated memory after the SQL statement has been executed at the non-Oracle system. If the cursor was not open, the operation is a no operation.

Syntax

```
DBMS_HS_PASSTHROUGH.CLOSE_CURSOR (
    c   IN   BINARY_INTEGER NOT NULL);
```

Parameter

Table C–20 CLOSE_CURSOR Procedure Parameters

Parameter	Description
c	Cursor to be released.

Exceptions

Table C–21 CLOSE_CURSOR Procedure Exceptions

Exception	Description
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS, RNDS

See Also: OPEN_CURSOR Function

EXECUTE_IMMEDIATE Function

This function executes a SQL statement immediately. Any valid SQL statement except SELECT can be executed immediately, but the statement must not contain any bind variables. The statement is passed in as a VARCHAR2 in the argument. Internally, the SQL statement is executed using the PASSTHROUGH_SQL protocol sequence of OPEN_CURSOR, PARSE, EXECUTE_NON_QUERY, CLOSE_CURSOR.

Syntax

```
EXECUTE_IMMEDIATE ( s IN VARCHAR2 NOT NULL )
RETURN BINARY_INTEGER);
```

Parameter Description

Table C–22 EXECUTE_IMMEDIATE Procedure Parameters

Parameter	Description
s	VARCHAR2 variable with the statement to be executed immediately.

Returns

The number of rows affected by the execution of the SQL statement.

Exceptions

Table C–23 EXECUTE_IMMEDIATE Procedure Exceptions

Exception	Description
ORA-28544	Max open cursors.
ORA-28551	SQL statement is invalid.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: NONE

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- EXECUTE_NON_QUERY Function
- BIND_INOUT_VARIABLE Procedure

EXECUTE_NON_QUERY Function

This function executes any SQL statement other than a SELECT statement. A cursor has to be open and the SQL statement has to be parsed before the SQL statement can be executed.

Syntax

```
DBMS_HS_PASSTHROUGH.EXECUTE_NON_QUERY (
     c    IN    BINARY_INTEGER NOT NULL)
     RETURN    BINARY_INTEGER);
```

Parameter

Table C–24 EXECUTE_NON_QUERY Function Parameters

Parameter	Description
C	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.

Returns

The number of rows affected by the SQL statement in the non-Oracle system.

Exceptions

Table C–25 EXECUTE_NON_QUERY Function Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	BIND_VARIABLE procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: NONE

See Also:

- OPEN_CURSOR Function
- PARSE Procedure

FETCH_ROW Function

This function fetches rows from a result set. The result set is defined with a SQL SELECT statement.

Before the rows can be fetched, a cursor has to be opened, and the SQL statement has to be parsed. When there are no more rows to be fetched, the function returns 0. After a 0 return, the NO_DATA_FOUND exception occurs when:

- A subsequent FETCH_ROW is attempted

- A GET_VALUE is attempted

Syntax

```
DBMS_HS_PASSTHROUGH.FETCH_ROW (
     c        IN    BINARY_INTEGER NOT NULL
     [,first  IN    BOOLEAN])
     RETURN  BINARY_INTEGER);
```

Parameters and Descriptions

Table C–26 FETCH_ROW Function Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
first	Optional parameter to reexecute a SELECT statement. Possible values: ■ TRUE: reexecute SELECT statement. ■ FALSE: fetch the next row, or if executed for the first time execute and fetch rows (default).

Returns

The returns the number of rows fetched. The function will return 0 if the last row was already fetched.

Exceptions

Table C–27 FETCH_ROW Function Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure

GET_VALUE Procedure

This procedure has two purposes:

- To retrieve the select list items of SELECT statements after a row has been fetched.
- To retrieve the OUT bind values after the SQL statement has been executed.

Syntax

```
DBMS_HS_PASSTHROUGH.GET_VALUE (
     c     IN    BINARY_INTEGER NOT NULL,
     pos   IN    BINARY_INTEGER NOT NULL,
     val   OUT   <dty>);
```

Where *dty* is one of:

- DATE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND
- NUMBER
- TIMESTAMP
- TIMESTAMP WITH TIMEZONE
- TIMESTAMP WITH LOCAL TIMEZONE
- VARCHAR2

For retrieving values of data type RAW, see GET_VALUE_RAW.

Parameters

Table C–28 GET_VALUE Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable or select list item in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable or select list item will store its value.

Exceptions

Table C–29 GET_VALUE Procedure Exceptions

Exception	Description
ORA-1403	Returns NO_DATA_FOUND exception when executing the GET_VALUE after the last row was fetched (FETCH_ROW returned 0).
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor, parse and execute (or fetch) the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- FETCH_ROW Function
- GET_VALUE_NCHAR Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- BIND_INOUT_VARIABLE_RAW Procedure

GET_VALUE_NCHAR Procedure

This procedure, which operates on NVARCHAR2 data types, has two purposes:

- To retrieve the select list items of SELECT statements after a row has been fetched.
- To retrieve the OUT bind values after the SQL statement has been executed.

Syntax

```
DBMS_HS_PASSTHROUGH.GET_VALUE_NCHAR (
     c      IN    BINARY_INTEGER NOT NULL,
     pos    IN    BINARY_INTEGER NOT NULL,
     val    OUT   NVARCHAR2);
```

Parameters

Table C–30 GET_VALUE_NCHAR Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable or select list item in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable or select list item will store its value.

Exceptions

Table C–31 GET_VALUE_NCHAR Procedure Exceptions

Exception	Description
ORA-1403	Returns NO_DATA_FOUND exception when executing the GET_VALUE after the last row was fetched (FETCH_ROW returned 0).
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor, parse and execute (or fetch) the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- FETCH_ROW Function
- GET_VALUE Procedure
- GET_VALUE_RAW Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- BIND_INOUT_VARIABLE_RAW Procedure

GET_VALUE_RAW Procedure

This procedure, which operates on RAW data types, has two purposes:

- To retrieve the select list items of SELECT statements after a row has been fetched.
- To retrieve the OUT bind values after the SQL statement has been executed.

Syntax

```
DBMS_HS_PASSTHROUGH.GET_VALUE_RAW (
     c      IN    BINARY_INTEGER NOT NULL,
     pos    IN    BINARY_INTEGER NOT NULL,
     val    OUT   RAW);
```

Parameters

Table C–32 GET_VALUE_RAW Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable or select list item in the SQL statement. Starts from 1.

Table C–32 GET_VALUE_RAW Procedure Parameters

Parameter	Description
val	Variable in which the OUT bind variable or select list item will store its value.

Exceptions

Table C–33 GET_VALUE_RAW Procedure Exceptions

Exception	Description
ORA-1403	Returns NO_DATA_FOUND exception when executing the GET_VALUE after the last row was fetched (FETCH_ROW returned 0).
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor, parse and execute (or fetch) the SQL statement?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- FETCH_ROW Function
- GET_VALUE Procedure
- GET_VALUE_NCHAR Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- BIND_INOUT_VARIABLE_RAW Procedure

OPEN_CURSOR Function

This function opens a cursor for executing a pass-through SQL statement at the non-Oracle system. This function must be called for any type of SQL statement The function returns a cursor, which must be used in subsequent calls. This call allocates

memory. To deallocate the associated memory, you call the procedure DBMS_HS_PASSTHROUGH.CLOSE_CURSOR.

Syntax

```
DBMS_HS_PASSTHROUGH.OPEN_CURSOR ()
     RETURN    BINARY_INTEGER;
```

Returns

The cursor to be used on subsequent procedure and function calls.

Exceptions

Table C–34 OPEN_CURSOR Function Exceptions

Exception	Description
ORA-28554	Maximum number of open cursor has been exceeded. Increase Heterogeneous Services OPEN_CURSORS initialization parameter.

Pragmas

Purity level defined: WNDS, RNDS

See Also: BIND_INOUT_VARIABLE Procedure

PARSE Procedure

This procedure parses a SQL statement at non-Oracle system.

Syntax

```
DBMS_HS_PASSTHROUGH.GET_VALUE_RAW (
     c     IN    BINARY_INTEGER NOT NULL,
     stmt  IN    VARCHAR2       NOT NULL);
```

Parameters

Table C–35 PARSE Procedure Parameters

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened using function OPEN_CURSOR.

Table C–35 PARSE Procedure Parameters

Parameter	Description
stmt	Statement to be parsed.

Exceptions

Table C–36 PARSE Procedure Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28551	SQL statement is illegal.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Pragmas

Purity level defined: WNDS, RNDS

See Also:

- OPEN_CURSOR Function
- PARSE Procedure
- FETCH_ROW Function
- GET_VALUE Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure
- BIND_INOUT_VARIABLE_NCHAR Procedure

D

Data Dictionary Translation Support

Data dictionary information is stored in the non-Oracle system as system tables and is accessed through ODBC or OLE DB application programming interfaces (APIs). This appendix documents data dictionary translation support. It explains how to access non-Oracle data dictionaries, lists Heterogeneous Services data dictionary views, describes how to use supported views and tables, and explains data dictionary mapping.

This appendix contains the following topics:

Accessing the Non-Oracle Data Dictionary

Accessing a non-Oracle data dictionary table or view is identical to accessing a data dictionary in an Oracle database. You issue a SELECT statement specifying a database link. The Oracle9*i* data dictionary view and column names are used to access the non-Oracle data dictionary. Synonyms of supported views are also acceptable.

For example, the following statement queries the data dictionary table ALL_USERS to retrieve all users in the non-Oracle system:

```
SQL SELECT * FROM all_users@sid1;
```

When you issue a data dictionary access query, the ODBC or OLE DB agent:

1. Maps the requested table, view, or synonym to one or more ODBC or OLE DB APIs (see "Data Dictionary Mapping"). The agent translates all data dictionary column names to their corresponding non-Oracle column names within the query.
2. Sends the sequence of APIs to the non-Oracle system.
3. Possibly converts the retrieved non-Oracle data to give it the appearance of the Oracle8*i* data dictionary table.
4. Passes the data dictionary information from the non-Oracle system table to the Oracle8*i*.

Note: The values returned when querying the Generic Connectivity data dictionary may not be the same as the ones returned by the Oracle Enterprise Manager DESCRIBE command.

Heterogeneous Services Data Dictionary Views

Heterogeneous Services mapping supports the following list of data dictionary views:

- ALL_CATALOG
- ALL_COL_COMMENTS
- ALL_COL_PRIVS
- ALL_COL_PRIVS_MADE
- ALL_COL_PRIVS_RECD

- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- ALL_DB_LINKS
- ALL_DEF_AUDIT_OPTS
- ALL_DEPENDENCIES
- ALL_ERRORS
- ALL_INDEXES
- ALL_IND_COLUMNS
- ALL_OBJECTS
- ALL_SEQUENCES
- ALL_MVIEWS
- ALL_SOURCE
- ALL_SYNONYMS
- ALL_TABLES
- ALL_TAB_COLUMNS
- ALL_TAB_COMMENTS
- ALL_TAB_PRIVS
- ALL_TAB_PRIVS_MADE
- ALL_TAB_PRIVS_RECD
- ALL_TRIGGERS
- ALL_USERS
- ALL_VIEWS
- AUDIT_ACTIONS
- COLUMN_PRIVILEGES
- DBA_CATALOG
- DBA_COL_COMMENTS
- DBA_COL_PRIVS
- DBA_OBJECTS

- DBA_ROLES
- DBA_ROLE_PRIVS
- DBA_SYS_PRIVS
- DBA_TABLES
- DBA_TAB_COLUMNS
- DBA_TAB_COMMENTS
- DBA_TAB_PRIVS
- DBA_USERS
- DICTIONARY
- DICT_COLUMNS
- DUAL
- INDEX_STATS
- PRODUCT_USER_PROFILE
- RESOURCE_COST
- ROLE_ROLE_PRIVS
- ROLE_SYS_PRIVS
- ROLE_TAB_PRIVS
- SESSION_PRIVS
- SESSION_ROLES
- TABLE_PRIVILEGES
- USER_AUDIT_OBJECT
- USER_AUDIT_SESSION
- USER_AUDIT_STATEMENT
- USER_AUDIT_TRAIL
- USER_CATALOG
- USER_CLUSTERS
- USER_CLU_COLUMNS
- USER_COL_COMMENTS

- USER_COL_PRIVS
- USER_COL_PRIVS_MADE
- USER_COL_PRIVS_RECD
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_DB_LINKS
- USER_DEPENDENCIES
- USER_ERRORS
- USER_EXTENTS
- USER_FREE_SPACE
- USER_INDEXES
- USER_IND_COLUMNS
- USER_OBJECTS
- USER_OBJ_AUDIT_OPTS
- USER_RESOURCE_LIMITS
- USER_ROLE_PRIVS
- USER_SEGMENTS
- USER_SEQUENCES
- USER_MVIEW_LOGS
- USER_SOURCE
- USER_SYNONYMS
- USER_SYS_PRIVS
- USER_TABLES
- USER_TABLESPACES
- USER_TAB_COLUMNS
- USER_TAB_COMMENTS
- USER_TAB_PRIVS
- USER_TAB_PRIVS_MADE

- USER_TAB_PRIVS_RECD
- USER_TRIGGERS
- USER_TS_QUOTAS
- USER_USERS
- USER_VIEWS

Views and Tables Supported by Generic Connectivity

Generic Connectivity supports only the views and tables shown in Table D–1.

If you use an unsupported view, then you receive the Oracle8*i* message for no rows selected.

If you want to query data dictionary views using SELECT... FROM DBA_*, first connect as Oracle user SYSTEM or SYS. Otherwise, you receive the following error message:

```
ORA-28506: Parse error in data dictionary translation for %s stored in %s
```

Using Generic Connectivity, queries of the supported data dictionary tables and views beginning with the characters ALL_ may return rows from the non-Oracle system when you do not have access privileges for those non-Oracle objects. When querying an Oracle database with the Oracle data dictionary, rows are returned only for those objects you are permitted to access.

Data Dictionary Mapping

The tables in this section list Oracle data dictionary view names and the equivalent ODBC or OLE DB APIs used.

Table D–1 Generic Connectivity Data Dictionary Mapping

View	ODBC API	OLE DB API
ALL_CATALOG	SQLTables	DBSCHEMA_CATALOGS
ALL_COL_COMMENTS	SQLColumns	DBSCHEMA_COLUMNS
ALL_CONS_COLUMNS	SQLPrimaryKeys, SQLForeignKeys	DBSCHEMA_PRIMARY_KEYS, DBSCHEMA_FOREIGN_KEYS
ALL_CONSTRAINTS	SQLPrimaryKeys, SQLForeignKeys	DBSCHEMA_PRIMARY_KEYS, DBSCHEMA_FOREIGN_KEYS

Table D–1 Generic Connectivity Data Dictionary Mapping (Cont.)

View	ODBC API	OLE DB API
ALL_IND_COLUMNS	SQLStatistics	DBSCHEMA_STATISTICS
ALL_INDEXES	SQLStatistics	DBSCHEMA_STATISTICS
ALL_OBJECTS	SQLTables, SQLProcedures, SQLStatistics	DBSCHEMA_TABLES, DBSCHEMA_PROCEDURES, DBSCHEMA_STATISTICS
ALL_TAB_COLUMNS	SQLColumns	DBSCHEMA_COLUMNS
ALL_TAB_COMMENTS	SQLTables	DBSCHEMA_TABLES
ALL_TABLES	SQLStatistics	DBSCHEMA_STATISTICS
ALL_USERS	SQLTables	DBSCHEMA_TABLES
ALL_VIEWS	SQLTables	DBSCHEMA_TABLES
DICTIONARY	SQLTables	DBSCHEMA_TABLES
USER_CATALOG	SQLTables	DBSCHEMA_TABLES
USER_COL_COMMENTS	SQLColumns	DBSCHEMA_COLUMNS
USER_CONS_COLUMNS	SQLPrimaryKeys, SQLForeignKeys	DBSCHEMA_PRIMARY_KEYS, DBSCHEMA_FOREIGN_KEYS
USER_CONSTRAINTS	SQLPrimaryKeys, SQLForeignKeys	DBSCHEMA_PRIMARY_KEYS, DBSCHEMA_FOREIGN_KEYS
USER_IND_COLUMNS	SQLStatistics	DBSCHEMA_STATISTICS
USER_INDEXES	SQLStatistics	DBSCHEMA_STATISTICS
USER_OBJECTS	SQLTables, SQLProcedures, SQLStatistics	DBSCHEMA_TABLES, DBSCHEMA_PROCEDURES, DBSCHEMA_STATISTICS
USER_TAB_COLUMNS	SQLColumns	DBSCHEMA_COLUMNS
USER_TAB_COMMENTS	SQLTables	DBSCHEMA_TABLES
USER_TABLES	SQLStatistics	DBSCHEMA_STATISTICS
USER_USERS	SQLTables	DBSCHEMA_TABLES
USER_VIEWS	SQLTables	DBSCHEMA_TABLES

Generic Connectivity Data Dictionary Descriptions

The Generic Connectivity data dictionary tables and views provide this information:

- Name, data type, and width of each column
- The contents of columns with fixed values

In the descriptions that follow, the values in the Null? column may differ from the Oracle9*i* data dictionary tables and views. Any default value is shown to the right of an item.

ALL_CATALOG

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_TYPE	-	VARCHAR2(11)	"TABLE" or "VIEW" or SYNONYM

ALL_COL_COMMENTS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	NOT NULL	VARCHAR2(30)	-
COMMENTS	-	VARCHAR2(4000)	NULL

ALL_CONS_COLUMNS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-

Name	Null?	Type	Value
COLUMN_NAME	-	VARCHAR2(4000)	-
POSITION	-	NUMBER	-

ALL_CONSTRAINTS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_TYPE	-	VARCHAR2(1)	"R" or "P"
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
SEARCH_CONDITION	-	LONG	NULL
R_OWNER	-	VARCHAR2(30)	-
R_CONSTRAINT_NAME	-	VARCHAR2(30)	-
DELETE_RULE	-	VARCHAR2(9)	"CASCADE" or "NO ACTION" or "SET NULL"
STATUS	-	VARCHAR2(8)	NULL
DEFERRABLE	-	VARCHAR2(14)	NULL
DEFERRED	-	VARCHAR2(9)	NULL
VALIDATED	-	VARCHAR2(13)	NULL
GENERATED	-	VARCHAR2(14)	NULL
BAD	-	VARCHAR2(3)	NULL
RELY	-	VARCHAR2(4)	NULL
LAST_CHANGE	-	DATE	NULL

ALL_IND_COLUMNS

Name	Null?	Type	Value
INDEX_OWNER	NOT NULL	VARCHAR2(30)	-
INDEX_NAME	NOT NULL	VARCHAR2(30)	-

Name	Null?	Type	Value
TABLE_OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	-	VARCHAR2(4000)	-
COLUMN_POSITION	NOT NULL	NUMBER	-
COLUMN_LENGTH	NOT NULL	NUMBER	-
DESCEND	-	VARCHAR2(4)	"DESC" or "ASC"

ALL_INDEXES

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
INDEX_NAME	NOT NULL	VARCHAR2(30)	-
INDEX_TYPE	-	VARCHAR2(27)	NULL
TABLE_OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_TYPE	-	CHAR(5)	"TABLE"
UNIQUENESS	-	VARCHAR2(9)	"UNIQUE" or "NONUNIQUE"
COMPRESSION	-	VARCHAR2(8)	NULL
PREFIX_LENGTH	-	NUMBER	0
TABLESPACE_NAME	-	VARCHAR2(30)	NULL
INI_TRANS	-	NUMBER	0
MAX_TRANS	-	NUMBER	0
INITIAL_EXTENT	-	NUMBER	0
NEXT_EXTENT	-	NUMBER	0
MIN_EXTENTS	-	NUMBER	0
MAX_EXTENTS	-	NUMBER	0
PCT_INCREASE	-	NUMBER	0

Name	Null?	Type	Value
PCT_THRESHOLD	-	NUMBER	0
INCLUDE_COLUMNS	-	NUMBER	0
FREELISTS	-	NUMBER	0
FREELIST_GROUPS	-	NUMBER	0
PCT_FREE	-	NUMBER	0
LOGGING	-	VARCHAR2(3)	NULL
BLEVEL	-	NUMBER	0
LEAF_BLOCKS	-	NUMBER	0
DISTINCT_KEYS	-	NUMBER	
AVG_LEAF_BLOCKS_PER_KEY	-	NUMBER	0
AVG_DATA_BLOCKS_PER_KEY	-	NUMBER	0
CLUSTERING_FACTOR	-	NUMBER	0
STATUS	-	VARCHAR2(8)	NULL
NUM_ROWS	-	NUMBER	0
SAMPLE_SIZE	-	NUMBER	0
LAST_ANALYZED	-	DATE	NULL
DEGREE	-	VARCHAR2(40)	NULL
INSTANCES	-	VARCHAR2(40)	NULL
PARTITIONED	-	VARCHAR2(3)	NULL
TEMPORARY	-	VARCHAR2(1)	NULL
GENERATED	-	VARCHAR2(1)	NULL
SECONDARY	-	VARCHAR2(1)	NULL
BUFFER_POOL	-	VARCHAR2(7)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
DURATION	-	VARCHAR2(15)	NULL
PCT_DIRECT_ACCESS	-	NUMBER	0
ITYP_OWNER	-	VARCHAR2(30)	NULL
ITYP_NAME	-	VARCHAR2(30)	NULL

Name	Null?	Type	Value
PARAMETERS	-	VARCHAR2(1000)	NULL
GLOBAL_STATS	-	VARCHAR2(3)	NULL
DOMIDX_STATUS	-	VARCHAR2(12)	NULL
DOMIDX_OPSTATUS	-	VARCHAR2(6)	NULL
FUNCIDX_STATUS	-	VARCHAR2(8)	NULL

ALL_OBJECTS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
OBJECT_NAME	NOT NULL	VARCHAR2(30)	-
SUBOBJECT_NAME	-	VARCHAR2(30)	NULL
OBJECT_ID	NOT NULL	NUMBER	0
DATA_OBJECT_ID	-	NUMBER	0
OBJECT_TYPE	-	VARCHAR2(18)	"TABLE" or "VIEW" or "SYNONYM" or "INDEX" or "PROCEDURE"
CREATED	NOT NULL	DATE	NULL
LAST_DDL_TIME	NOT NULL	DATE	NULL
TIMESTAMP	-	VARCHAR2(19)	NULL
STATUS	-	VARCHAR2(7)	NULL
TEMPORARY	-	VARCHAR2(1)	NULL
GENERATED	-	VARCHAR2(1)	NULL
SECONDARY	-	VARCHAR2(1)	NULL

ALL_TAB_COLUMNS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	NOT NULL	VARCHAR2(30)	-
DATA_TYPE	-	VARCHAR2(106)	-
DATA_TYPE_MOD	-	VARCHAR2(3)	NULL
DATA_TYPE_OWNER	-	VARCHAR2(30)	NULL
DATA_LENGTH	NOT NULL	NUMBER	-
DATA_PRECISION	-	NUMBER	-
DATA_SCALE	-	NUMBER	-
NULLABLE	-	VARCHAR2(1)	"Y" or "N"
COLUMN_ID	NOT NULL	NUMBER	-
DEFAULT_LENGTH	-	NUMBER	0
DATA_DEFAULT	-	LONG	NULL
NUM_DISTINCT	-	NUMBER	0
LOW_VALUE	-	RAW(32)	NULL
HIGH_VALUE	-	RAW(32)	NULL
DENSITY	-	NUMBER	0
NUM_NULLS	-	NUMBER	0
NUM_BUCKETS	-	NUMBER	0
LAST_ANALYZED	-	DATE	NULL
SAMPLE_SIZE	-	NUMBER	0
CHARACTER_SET_NAME	-	VARCHAR2(44)	NULL
CHAR_COL_DEC_LENGTH	-	NUMBER	0
GLOBAL_STATS	-	VARCHAR2(3)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
AVG_COL_LEN	-	NUMBER	0

ALL_TAB_COMMENTS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_TYPE	-	VARCHAR2(11)	"TABLE" or "VIEW"
COMMENTS	-	VARCHAR2(4000)	NULL

ALL_TABLES

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLESPACE_NAME	-	VARCHAR2(30)	NULL
CLUSTER_NAME	-	VARCHAR2(30)	NULL
IOT_NAME	-	VARCHAR2(30)	NULL
PCT_FREE	-	NUMBER	0
PCT_USED	-	NUMBER	0
INI_TRANS	-	NUMBER	0
MAX_TRANS	-	NUMBER	0
INITIAL_EXTENT	-	NUMBER	0
NEXT_EXTENT	-	NUMBER	0
MIN_EXTENTS	-	NUMBER	0
MAX_EXTENTS	-	NUMBER	0
PCT_INCREASE	-	NUMBER	0
FREELISTS	-	NUMBER	0
FREELIST_GROUPS	-	NUMBER	0
LOGGING	-	VARCHAR2(3)	NULL
BACKED_UP	-	VARCHAR2(1)	NULL

Name	Null?	Type	Value
NUM_ROWS	-	NUMBER	-
BLOCKS	-	NUMBER	-
EMPTY_BLOCKS	-	NUMBER	0
AVG_SPACE	-	NUMBER	0
CHAIN_CNT	-	NUMBER	0
AVG_ROW_LEN	-	NUMBER	0
AVG_SPACE_FREELIST_BLOCKS	-	NUMBER	0
NUM_FREELIST_BLOCKS	-	NUMBER	0
DEGREE	-	VARCHAR2(10)	NULL
INSTANCES	-	VARCHAR2(10)	NULL
CACHE	-	VARCHAR2(5)	NULL
TABLE_LOCK	-	VARCHAR2(8)	NULL
SAMPLE_SIZE	-	NUMBER	0
LAST_ANALYZED	-	DATE	NULL
PARTITIONED	-	VARCHAR2(3)	NULL
IOT_TYPE	-	VARCHAR2(12)	NULL
TEMPORARY	-	VARHCAR2(1)	NULL
SECONDARY	-	VARCHAR2(1)	NULL
NESTED	-	VARCHAR2(3)	NULL
BUFFER_POOL	-	VARCHAR2(7)	NULL
ROW_MOVEMENT	-	VARCHAR2(8)	NULL
GLOBAL_STATS	-	VARCHAR2(3)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
DURATION	-	VARHCAR2(15)	NULL
SKIP_CORRUPT	-	VARCHAR2(8)	NULL
MONITORING	-	VARCHAR2(3)	NULL

ALL_USERS

Name	Null?	Type	Value
USERNAME	NOT NULL	VARCHAR2(30)	-
USER_ID	NOT NULL	NUMBER	0
CREATED	NOT NULL	DATE	NULL

ALL_VIEWS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
VIEW_NAME	NOT NULL	VARCHAR2(30)	-
TEXT_LENGTH	-	NUMBER	0
TEXT	NOT NULL	LONG	NULL
TYPE_TEXT_LENGTH	-	NUMBER	0
TYPE_TEXT	-	VARCHAR2(4000)	NULL
OID_TEXT_LENGTH	-	NUMBER	0
OID_TEXT	-	VARCHAR2(4000)	NULL
VIEW_TYPE_OWNER	-	VARCHAR2(30)	NULL
VIEW_TYPE	-	VARCHAR2(30)	NULL

DICTIONARY

Name	Null?	Type	Value
TABLE_NAME	-	VARCHAR2(30)	-
COMMENTS	-	VARCHAR2(4000)	NULL

USER_CATALOG

Name	Null?	Type	Value
TABLE_NAME	NOT NULL	VARCHAR2(30)	-

Name	Null?	Type	Value
TABLE_TYPE	-	VARCHAR2(11)	"TABLE" or, "VIEW" or "SYNONYM"

USER_COL_COMMENTS

Name	Null?	Type	Value
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	NOT NULL	VARCHAR2(30)	-
COMMENTS	-	VARCHAR2(4000)	NULL

USER_CONS_COLUMNS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	-	VARCHAR2(4000)	-
POSITION	-	NUMBER	-

USER_CONSTRAINTS

Name	Null?	Type	Value
OWNER	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)	-
CONSTRAINT_TYPE	-	VARCHAR2(1)	R or P
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
SEARCH_CONDITION	-	LONG	NULL
R_OWNER	-	VARCHAR2(30)	-
R_CONSTRAINT_NAME	-	VARCHAR2(30)	-

Name	Null?	Type	Value
DELETE_RULE	-	VARCHAR2(9)	"CASCADE" or "NOACTION" or "SET NULL"
STATUS	-	VARCHAR2(8)	NULL
DEFERRABLE	-	VARCHAR2(14)	NULL
DEFERRED	-	VARCHAR2(9)	NULL
VALIDATED	-	VARCHAR2(13)	NULL
GENERATED	-	VARCHAR2(14)	NULL
BAD	-	VARCHAR2(3)	NULL
RELY	-	VARCHAR2(4)	NULL
LAST_CHANGE	-	DATE	NULL

USER_IND_COLUMNS

Name	Null?	Type	Value
INDEX_NAME	-	VARCHAR2(30)	-
TABLE_NAME	-	VARCHAR2(30)	-
COLUMN_NAME	-	VARCHAR2(4000)	-
COLUMN_POSITION	-	NUMBER	-
COLUMN_LENGTH	-	NUMBER	-
DESCEND	-	VARCHAR2(4)	"DESC" or "ASC"

USER_INDEXES

Name	Null?	Type	Value
INDEX_NAME	NOT NULL	VARCHAR2(30)	-
INDEX_TYPE	-	VARCHAR2(27)	NULL
TABLE_OWNER	NOT NULL	VARCHAR2(30)	-
TABLE_NAME	NOT NULL	VARCHAR2(30)	-

Name	Null?	Type	Value
TABLE_TYPE	-	VARCHAR2(11)	"TABLE"
UNIQUENESS	-	VARCHAR2(9)	"UNIQUE" or "NONUNIQUE"
COMPRESSION	-	VARCHAR2(8)	NULL
PREFIX_LENGTH	-	NUMBER	0
TABLESPACE_NAME	-	VARCHAR2(30)	NULL
INI_TRANS	-	NUMBER	0
MAX_TRANS	-	NUMBER	0
INITIAL_EXTENT	-	NUMBER	0
NEXT_EXTENT	-	NUMBER	0
MIN_EXTENTS	-	NUMBER	0
MAX_EXTENTS	-	NUMBER	0
PCT_INCREASE	-	NUMBER	0
PCT_THRESHOLD	-	NUMBER	0
INCLUDE_COLUMNS	-	NUMBER	0
FREELISTS	-	NUMBER	0
FREELIST_GROUPS	-	NUMBER	0
PCT_FREE	-	NUMBER	0
LOGGING	-	VARCHAR2(3)	NULL
BLEVEL	-	NUMBER	0
LEAF_BLOCKS	-	NUMBER	0
DISTINCT_KEYS	-	NUMBER	-
AVG_LEAF_BLOCKS_PER_KEY	-	NUMBER	0
AVG_DATA_BLOCKS_PER_KEY	-	NUMBER	0
CLUSTERING_FACTOR	-	NUMBER	0
STATUS	-	VARCHAR2(8)	NULL
NUM_ROWS	-	NUMBER	0
SAMPLE_SIZE	-	NUMBER	0

Name	Null?	Type	Value
LAST_ANALYZED	-	DATE	NULL
DEGREE	-	VARCHAR2(40)	NULL
INSTANCES	-	VARCHAR2(40)	NULL
PARTITIONED	-	VARCHAR2(3)	NULL
TEMPORARY	-	VARCHAR2(1)	NULL
GENERATED	-	VARCHAR2(1)	NULL
SECONDARY	-	VARCHAR2(1)	NULL
BUFFER_POOL	-	VARCHAR2(7)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
DURATION	-	VARHCAR2(15)	NULL
PCT_DIRECT_ACCESS	-	NUMBER	0
ITYP_OWNER	-	VARCHAR2(30)	NULL
ITYP_NAME	-	VARCHAR2(30)	NULL
PARAMETERS	-	VARCHAR2(1000)	NULL
GLOBAL_STATS	-	VARCHAR2(3)	NULL
DOMIDX_STATUS	-	VARCHAR2(12)	NULL
DOMIDX_OPSTATUS	-	VARCHAR2(6)	NULL
FUNCIDX_STATUS	-	VARCHAR2(8)	NULL

USER_OBJECTS

Name	Null?	Type	Value
OBJECT_NAME	-	VARCHAR2(128)	-
SUBOBJECT_NAME	-	VARCHAR2(30)	NULL
OBJECT_ID	-	NUMBER	0
DATA_OBJECT_ID	-	NUMBER	0

Name	Null?	Type	Value
OBJECT_TYPE	-	VARCHAR2(18)	"TABLE" or "VIEW" or "SYNONYM" or "INDEX" or "PROCEDURE"
CREATED	-	DATE	NULL
LAST_DDL_TIME	-	DATE	NULL
TIMESTAMP	-	VARCHAR2(19)	NULL
STATUS	-	VARCHAR2(7)	NULL
TEMPORARY	-	VARCHAR2(1)	NULL
GENERATED	-	VARCHAR2(1)	NULL
SECONDARY	-	VARCHAR2(1)	NULL

USER_TAB_COLUMNS

Name	Null?	Type	Value
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
COLUMN_NAME	NOT NULL	VARCHAR2(30)	-
DATA_TYPE	-	VARCHAR2(106)	-
DATA_TYPE_MOD	-	VARCHAR2(3)	NULL
DATA_TYPE_OWNER	-	VARCHAR2(30)	NULL
DATA_LENGTH	NOT NULL	NUMBER	-
DATA_PRECISION	-	NUMBER	-
DATA_SCALE	-	NUMBER	-
NULLABLE	-	VARCHAR2(1)	"Y" or "N"
COLUMN_ID	NOT NULL	NUMBER	-
DEFAULT_LENGTH	-	NUMBER	NULL
DATA_DEFAULT	-	LONG	NULL
NUM_DISTINCT	-	NUMBER	NULL
LOW_VALUE	-	RAW(32)	NULL

Name	Null?	Type	Value
HIGH_VALUE	-	RAW(32)	NULL
DENSITY	-	NUMBER	0
NUM_NULLS	-	NUMBER	0
NUM_BUCKETS	-	NUMBER	0
LAST_ANALYZED	-	DATE	NULL
SAMPLE_SIZE	-	NUMBER	0
CHARACTER_SET_NAME	-	VARCHAR2(44)	NULL
CHAR_COL_DECL_LENGTH	-	NUMBER	0
GLOBAL_STATS	-	VARCHAR2(3)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
AVG_COL_LEN	-	NUMBER	0

USER_TAB_COMMENTS

Name	Null?	Type	Value
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLE_TYPE	-	VARCHAR2(11)	"TABLE" or "VIEW"
COMMENTS	-	VARCHAR2(4000)	NULL

USER_TABLES

Name	Null?	Type	Value
TABLE_NAME	NOT NULL	VARCHAR2(30)	-
TABLESPACE_NAME	-	VARCHAR2(30)	NULL
CLUSTER_NAME	-	VARCHAR2(30)	NULL
IOT_NAME	-	VARCHAR2(30)	NULL
PCT_FREE	-	NUMBER	0
PCT_USED	-	NUMBER	0

Name	Null?	Type	Value
INI_TRANS	-	NUMBER	0
MAX_TRANS	-	NUMBER	0
INITIAL_EXTENT	-	NUMBER	0
NEXT_EXTENT	-	NUMBER	0
MIN_EXTENTS	-	NUMBER	0
MAX_EXTENTS	-	NUMBER	0
PCT_INCREASE	-	NUMBER	0
FREELISTS	-	NUMBER	0
FREELIST_GROUPS	-	NUMBER	0
LOGGING	-	VARCHAR2(3)	NULL
BACKED_UP	-	VARCHAR2(1)	NULL
NUM_ROWS	-	NUMBER	-
BLOCKS	-	NUMBER	-
EMPTY_BLOCKS	-	NUMBER	0
AVG_SPACE	-	NUMBER	0
CHAIN_CNT	-	NUMBER	0
AVG_ROW_LEN	-	NUMBER	0
AVG_SPACE_FREELIST_BLOCKS	-	NUMBER	0
NUM_FREELIST_BLOCKS	-	NUMBER	0
DEGREE	-	VARCHAR2(10)	NULL
INSTANCES	-	VARCHAR2(10)	NULL
CACHE	-	VARCHAR2(5)	NULL
TABLE_LOCK	-	VARCHAR2(8)	NULL
SAMPLE_SIZE	-	NUMBER	0
LAST_ANALYZED	-	DATE	NULL
PARTITIONED	-	VARCHAR2(3)	NULL
IOT_TYPE	-	VARCHAR2(12)	NULL
TEMPORARY	-	VARHCAR2(1)	NULL

Name	Null?	Type	Value
SECONDARY	-	VARCHAR2(1)	NULL
NESTED	-	VARCHAR2(3)	NULL
BUFFER_POOL	-	VARCHAR2(7)	NULL
ROW_MOVEMENT	-	VARCHAR2(8)	NULL
GLOBAL_STATS	-	VARCHAR2(3)	NULL
USER_STATS	-	VARCHAR2(3)	NULL
DURATION	-	VARCHAR2(15)	NULL
SKIP_CORRUPT	-	VARCHAR2(8)	NULL
MONITORING	-	VARCHAR2(3)	NULL

USER_USERS

Name	Null?	Type	Value
USERNAME	NOT NULL	VARCHAR2(30)	-
USER_ID	NOT NULL	NUMBER	0
ACCOUNT_STATUS	NOT NULL	VARCHAR2(32)	OPEN
LOCK_DATE	-	DATE	NULL
EXPIRY_DATE	-	DATE	NULL
DEFAULT_TABLESPACE	NOT NULL	VARCHAR2(30)	NULL
TEMPORARY_TABLESPACE	NOT NULL	VARCHAR2(30)	NULL
CREATED	NOT NULL	DATE	NULL
INITIAL_RSRC_CONSUMER_GROUP	-	VARCHAR2(30)	NULL
EXTERNAL_NAME	-	VARCHAR2(4000)	NULL

USER_VIEWS

Name	Null?	Type	Value
VIEW_NAME	NOT NULL	VARCHAR2(30)	-
TEXT_LENGTH	-	NUMBER	0

Name	Null?	Type	Value
TEXT	-	LONG	NULL
TYPE_TEXT_LENGTH	-	NUMBER	0
TYPE_TEXT	-	VARCHAR2(4000)	NULL
OID_TEXT_LENGTH	-	NUMBER	0
OID_TEXT	-	VARCHAR2(4000)	NULL
VIEW_TYPE_OWNER	-	VARCHAR2(30)	NULL
VIEW_TYPE	-	VARCHAR2(30)	NULL

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V