Oracle8*i*

Distributed Database Systems

Release 2 (8.1.6)

December 1999

A76960-01

ORACLE®

Oracle8*i* Distributed Database Systems, Release 2 (8.1.6)

A76960-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: Jason Durbin, Lance Ashdown

Contributing Authors: William Creekbaum, Steve Bobrowski, Katherine Hughes, Pavna Jain, Peter Vasterd

Contributors: John Bellemore, Anupam Bhide, Roger Bodamer, Jacco Draaijer, Diana Foch-Laurentz, Nina Lewis, Raghu Mani, Basab Maulik, Denise Oertel, Paul Raveling, Kendall Scott, Gordon Smith, Katia Tarkhanov, Randy Urbano, Sandy Venning, Eric Voss, and others

Graphic Designer: Valarie Moore

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

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the Programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and Net8, SQL*Plus, Oracle Call Interface, Oracle Transparent Gateway, Oracle7, Oracle7 Server, Oracle8, Oracle8*i*, PL/SQL, Pro*C, Pro*C/C++, and Enterprise Manager are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments xi

Preface xiii

What's New in Oracle8*i*? xiii
Release 8.1.6 xiii
Release 8.1.5 xiv
Structure xv
Changes to This Book xvi
Audience xvi
Knowledge Assumed of the Reader xvii
How to Use This Guide xvii
Conventions Used in This Guide xviii
Your Comments Are Welcome xviii

Part I Distributed Database Systems Concepts and Administration

1 Distributed Database Concepts

Distributed Database Architecture 1-2
Homogenous Distributed Database Systems 1-2
Heterogeneous Distributed Database Systems 1-5
Client/Server Database Architecture 1-7
Database Links 1-9
What Are Database Links? 1-10
Why Use Database Links? 1-13

Global Database Names in Database Links 1-13
Names for Database Links 1-15
Types of Database Links 1-16
Users of Database Links 1-17
Creation of Database Links: Examples 1-20
Schema Objects and Database Links 1-21
Database Link Restrictions 1-23
Distributed Database Administration 1-23
Site Autonomy 1-24
Distributed Database Security 1-25
Auditing Database Links 1-31
Administration Tools 1-31
Transaction Processing in a Distributed System 1-33
Remote SQL Statements 1-33
Distributed SQL Statements 1-34
Shared SQL for Remote and Distributed Statements 1-34
Remote Transactions 1-35
Distributed Transactions 1-35
Two-Phase Commit Mechanism 1-35
Database Link Name Resolution 1-36
Schema Object Name Resolution 1-38
Global Name Resolution in Views, Synonyms, and Procedures 1-41
Distributed Database Application Development 1-44
Transparency in a Distributed Database System 1-44
Remote Procedure Calls (RPCs) 1-46
Distributed Query Optimization 1-47
National Language Support 1-47
Client/Server Environment 1-48
Homogeneous Distributed Environment 1-48
Heterogeneous Distributed Environment 1-49

2 Managing a Distributed Database

Managing Global Names in a Distributed System 2-2
Understanding How Global Database Names Are Formed 2-2
Determining Whether Global Naming Is Enforced 2-3

Viewing a Global Database Name 2-4
Changing the Domain in a Global Database Name........ 2-4
Changing a Global Database Name: Scenario........ 2-5
Creating Database Links 2-8
Obtaining Privileges Necessary for Creating Database Links 2-8
Specifying Link Types........ 2-9
Specifying Link Users 2-11
Using Connection Qualifiers to Specify Service Names Within Link Names 2-13
Creating Shared Database Links 2-14
Determining Whether to Use Shared Database Links........ 2-14
Creating Shared Database Links 2-15
Configuring Shared Database Links........ 2-16
Managing Database Links 2-18
Closing Database Links 2-18
Dropping Database Links........ 2-19
Limiting the Number of Active Database Link Connections........ 2-20
Viewing Information About Database Links........ 2-21
Determining Which Links Are in the Database 2-21
Determining Which Link Connections Are Open 2-24
Creating Location Transparency 2-26
Using Views to Create Location Transparency........ 2-26
Using Synonyms to Create Location Transparency 2-28
Using Procedures to Create Location Transparency........ 2-30
Managing Statement Transparency........ 2-32
Understanding Transparency Restrictions 2-33
Managing a Distributed Database: Scenarios........ 2-34
Creating a Public Fixed User Database Link 2-34
Creating a Public Fixed User Shared Database Link........ 2-35
Creating a Public Connected User Database Link........ 2-36
Creating a Public Connected User Shared Database Link........ 2-36
Creating a Public Current User Database Link........ 2-37

3 Developing Applications for a Distributed Database System

Managing the Distribution of an Application's Data 3-2
Controlling Connections Established by Database Links 3-2
Maintaining Referential Integrity in a Distributed System 3-3
Tuning Distributed Queries 3-3
Using Collocated Inline Views 3-4
Using Cost-Based Optimization 3-5
Using Hints 3-8
Analyzing the Execution Plan 3-10
Handling Errors in Remote Procedures 3-12

Part II Distributed Transactions Concepts and Administration

4 Distributed Transactions Concepts

What Are Distributed Transactions? 4-2
Supported Types of Distributed Transactions 4-3
Session Trees for Distributed Transactions 4-4
Two-Phase Commit Mechanism 4-4
Session Trees for Distributed Transactions 4-5
Clients 4-6
Database Servers 4-6
Local Coordinators 4-6
Global Coordinator 4-7
Commit Point Site 4-7
Two-Phase Commit Mechanism 4-11
Prepare Phase 4-11
Commit Phase 4-14
Forget Phase 4-16
In-Doubt Transactions 4-16
Automatic Resolution of In-Doubt Transactions 4-17
Manual Resolution of In-Doubt Transactions 4-19
Relevance of System Change Numbers for In-Doubt Transactions 4-19
Distributed Transaction Processing: Case Study 4-20
Stage 1: Client Application Issues DML Statements 4-20

Stage 2: Oracle Determines Commit Point Site 4-22
Stage 3: Global Coordinator Sends Prepare Response 4-22
Stage 4: Commit Point Site Commits 4-23
Stage 5: Commit Point Site Informs Global Coordinator of Commit 4-24
Stage 6: Global and Local Coordinators Tell All Nodes to Commit 4-24
Stage 7: Global Coordinator and Commit Point Site Complete the Commit 4-25

5 Managing Distributed Transactions

Setting Distributed Transaction Initialization Parameters 5-2
Limiting the Number of Distributed Transactions 5-2
Specifying the Lock Timeout Interval 5-4
Specifying the Interval for Holding Open Connections 5-5
Specifying the Commit Point Strength of a Node 5-5
Viewing Information About Distributed Transactions 5-6
Determining the ID Number and Status of Prepared Transactions 5-6
Tracing the Session Tree of In-Doubt Transactions 5-9
Deciding How to Handle In-Doubt Transactions 5-10
Discovering Problems with a Two-Phase Commit 5-11
Determining Whether to Perform a Manual Override 5-12
Analyzing the Transaction Data 5-12
Manually Overriding In-Doubt Transactions 5-13
Manually Committing an In-Doubt Transaction 5-14
Manually Rolling Back an In-Doubt Transaction 5-15
Purging Pending Rows from the Data Dictionary 5-15
Executing the PURGE_LOST_DB_ENTRY Procedure 5-16
Determining When to Use DBMS_TRANSACTION 5-16
Manually Committing an In-Doubt Transaction: Example 5-17
Step 1: Record User Feedback 5-19
Step 2: Query DBA_2PC_PENDING 5-19
Step 3: Query DBA_2PC_NEIGHBORS on Local Node 5-21
Step 4: Querying Data Dictionary Views on All Nodes 5-22
Step 5: Commit the In-Doubt Transaction 5-25
Step 6: Check for Mixed Outcome Using DBA_2PC_PENDING 5-25
Simulating Distributed Transaction Failure 5-26
Forcing a Distributed Transaction to Fail 5-26

Disabling and Enabling RECO 5-27
Managing Read Consistency 5-28

Part III Heterogeneous Services Concepts and Administration

6 Oracle Heterogeneous Services Concepts

What is Heterogeneous Services? 6-2
Database Links to a Non-Oracle System 6-2
Heterogeneous Services Agents 6-3
Types of Heterogeneous Services 6-3
Transaction Service 6-3
SQL Service 6-4
Heterogeneous Services Process Architecture 6-4
Transparent Gateways 6-5
Generic Connectivity 6-6
Architecture of the Heterogeneous Services Data Dictionary 6-6
Classes and Instances 6-7
Data Dictionary Views 6-8

7 Managing Oracle Heterogeneous Services Using Transparent Gateways

Setting Up Access to Non-Oracle Systems 7-2
Step 1: Install the Heterogeneous Services Data Dictionary 7-2
Step 2: Set Up the Environment to Access Heterogeneous Services Agents 7-2
Step 3: Create the Database Link to the Non-Oracle System 7-4
Step 4: Test the Connection 7-4
Registering Agents 7-5
Enabling Agent Self-Registration 7-5
Disabling Agent Self-Registration 7-9
Using the Heterogeneous Services Data Dictionary Views 7-9
Understanding the Types of Views 7-9
Understanding the Sources of Data Dictionary Information 7-11
Using the General Views 7-11
Using the Transaction Service Views 7-12
Using the SQL Service Views 7-13

Using the Heterogeneous Services Dynamic Performance Views 7-15
Determining Which Agents Are Running on a Host 7-15
Determining the Open HS Sessions 7-16
Determining the HS Parameters 7-16
Using the DBMS_HS Package 7-17
Specifying Initialization Parameters 7-17
Unspecifying Initialization Parameters 7-18

8 Managing Heterogeneous Services Using Generic Connectivity

What Is Generic Connectivity? 8-2
Types of Agents 8-2
Generic Connectivity Architecture 8-3
SQL Execution 8-5
Datatype Mapping 8-5
Generic Connectivity Restrictions 8-5
Supported Oracle SQL Statements 8-5
Functions Supported by Generic Connectivity 8-6
Configuring Generic Connectivity Agents 8-6
Creating the Initialization File 8-7
Editing the Initialization File 8-7
Setting Initialization Parameters for an ODBC-based Data Source 8-9
Setting Initialization Parameters for an OLE DB-based Data Source 8-11
ODBC Connectivity Requirements 8-12
OLE DB (SQL) Connectivity Requirements 8-14
Data Provider Requirements 8-14
OLE DB (FS) Connectivity Requirements 8-15
Bookmarks 8-15
OLE DB Interfaces 8-16
Data Source Properties 8-17

9 Developing Applications with Heterogeneous Services

Developing Applications with Heterogeneous Services: Overview 9-2
Developing Using Pass-Through SQL 9-2
Using the DBMS_HS_PASSTHROUGH package 9-2
Considering the Implications of Using Pass-Through SQL 9-3

Executing Pass-Through SQL Statements 9-3
Optimizing Data Transfers Using Bulk Fetch 9-9
Using OCI, an Oracle Precompiler, or Another Tool for Array Fetches 9-10
Controlling the Array Fetch Between Oracle Database Server and Agent 9-11
Controlling the Array Fetch Between Agent and Non-Oracle Server 9-11
Controlling the Reblocking of Array Fetches 9-11
Researching the Locking Behavior of Non-Oracle Systems 9-12

A Heterogeneous Services Initialization Parameters

HS_COMMIT_POINT_STRENGTH A-2
HS_DB_DOMAIN A-2
HS_DB_INTERNAL_NAME A-3
HS_DB_NAME A-3
HS_DESCRIBE_CACHE_HWM A-3
HS_FDS_CONNECT_INFO A-4
HS_FDS_SHAREABLE_NAME A-5
HS_FDS_TRACE_LEVEL A-5
HS_FDS_TRACE_FILE_NAME A-5
HS_LANGUAGE A-6
HS_NLS_DATE_FORMAT A-7
HS_NLS_DATE_LANGUAGE A-7
HS_NLS_NCHAR A-8
HS_OPEN_CURSORS A-8
HS_ROWID_CACHE_SIZE A-9
HS_RPC_FETCH_REBLOCKING A-9
HS_RPC_FETCH_SIZE A-10

B Data Dictionary Views Available Through Heterogeneous Services

C Data Dictionary Translation for Generic Connectivity

Data Dictionary Translation Support C-2
Accessing the Non-Oracle Data Dictionary C-2
Supported Views and Tables C-3
Data Dictionary Mapping C-4

Default Column Values C-5
Generic Connectivity Data Dictionary Descriptions C-6
ALL_CATALOG C-6
ALL_COL_COMMENTS C-6
ALL_CONS_COLUMNS C-6
ALL_CONSTRAINTS C-7
ALL_IND_COLUMNS C-7
ALL_INDEXES C-8
ALL_OBJECTS C-10
ALL_TAB_COLUMNS C-11
ALL_TAB_COMMENTS C-12
ALL_TABLES C-12
ALL_USERS C-14
ALL_VIEWS C-14
DICTIONARY C-14
USER_CATALOG C-15
USER_COL_COMMENTS C-15
USER_CONS_COLUMNS C-15
USER_CONSTRAINTS C-15
USER_IND_COLUMNS C-16
USER_INDEXES C-17
USER_OBJECTS C-19
USER_TAB_COLUMNS C-19
USER_TAB_COMMENTS C-20
USER_TABLES C-21
USER_USERS C-22
USER_VIEWS C-23

D Datatype Mapping

Mapping ODBC Datatypes to Oracle Datatypes D-2
Mapping OLE DB Datatypes to Oracle Datatypes D-3

Index

Send Us Your Comments

Oracle8*i* Distributed Database Systems, Release 2 (8.1.6)

A76960-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

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

- E-mail - infodev@us.oracle.com
- FAX - (650) 506-7228 Attn: Oracle Server Documentation
- Postal service:
  Oracle Corporation
  Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

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

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

Preface

This manual describes implementation issues for an Oracle8*i* distributed database system. It also introduces the tools and utilities available to assist you in implementing and maintaining your distributed system.

Oracle8i Distributed Database Systems contains information that describes the features and functionality of the Oracle8*i* and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features, but several advanced features are available only with the Enterprise Edition, and some of these are optional.

> **See Also:** *Getting to Know Oracle8i* for information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition.

What's New in Oracle8*i*?

This section describes new features in Oracle8*i*.

Release 8.1.6

The following distributed database features are new in release 8.1.6:

- Oracle now enables generic connectivity to heterogeneous systems through ODBC and OLE DB standards. See:
  - Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity"
  - Appendix C, "Data Dictionary Translation for Generic Connectivity"
  - Appendix D, "Datatype Mapping"

For transparent gateways, Oracle supports and certifies the configuration from the Oracle application tools to the target system. For generic connectivity, Oracle supports the HS ODBC agent—for any issues with non-Oracle components you should contact appropriate vendors.

- Oracle8*i* supports functionality that allows a global user to be centrally managed by an enterprise directory (see "Centralized User and Privilege Management" on page 1-27). Consequently, the administrator of each database is not required to create a new schema for each new global user in the distributed system. The central LDAP-based directory contains information about:
  - Which databases in a distributed system an enterprise user can access
  - Which role on each database an enterprise user can use
  - Which schema on each database an enterprise user can connect to
- You can execute DML and DDL statements in parallel, and INSERT direct load statements serially (see "DML and DDL Transactions" on page 4-3).
- The Kerberos authentication tool works with connected user database links (see "Distributed Database Security" on page 1-25).
- As part of the Oracle Advanced Security option, SSL (single sign-on) allows users to connect to multiple databases without repeatedly typing in passwords (see "Distributed Database Security" on page 1-25)
- The following Heterogeneous Services initialization parameters are new for release 8.1.6:
  - HS_FDS_CONNECT_INFO
  - HS_FDS_SHAREABLE_NAME
  - HS_FDS_TRACE
  - HS_FDS_TRACE_FILE_NAME

Release 8.1.5

The following distributed database features are new in release 8.1.5:

- The distributed SQL optimization capabilities of the Oracle server are extended to account for non-Oracle systems. This extension means that Oracle determines the best method for executing the SQL statement. For example, Oracle may determine that the greatest distributed performance can be achieved by performing a join at the remote non-Oracle site. In this case, only the rows that

satisfy the SELECT statement are returned to the originating Oracle site, greatly reducing the amount of data transmitted through the network. Additionally, collocated in-line views can improve the performance of a distributed query by accessing multiple tables of a non-Oracle system at once, thereby reducing the number of round trips.

- The following heterogeneous services views are new (see "Using the Heterogeneous Services Dynamic Performance Views" on page 7-15):
  - V$HS_AGENT
  - V$HS_SESSION

Structure

This book contains the following parts and chapters:

Part / Chapter	Contents
PART 1	**Distributed Database Systems Concepts and Administration**
Chapter 1, "Distributed Database Concepts"	Describes the basic concepts and terminology of Oracle's distributed database architecture. It is recommended reading for anyone planning to implement or maintain a distributed database system.
Chapter 2, "Managing a Distributed Database"	Discusses issues of concern to the database administrator (DBA) implementing or maintaining distributed databases.
Chapter 3, "Developing Applications for a Distributed Database System"	Describes the special considerations that are necessary if you are designing an application to run in a distributed system.
PART 2	**Distributed Transactions Concepts and Administration**
Chapter 4, "Distributed Transactions Concepts"	Describes how Oracle maintains the integrity of distributed transactions using the two-phase commit mechanism.
Chapter 5, "Managing Distributed Transactions"	Explains how to administer distributed transactions.
PART 3	**Heterogeneous Services Concepts and Administration**
Chapter 6, "Oracle Heterogeneous Services Concepts"	Provides an overview of Oracle Heterogeneous Services.

Part / Chapter	Contents
Chapter 7, "Managing Oracle Heterogeneous Services Using Transparent Gateways"	Explains how to implement and maintain Heterogeneous Services using an Oracle Transparent Gateway.
Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity"	Provides the information you need to connect to non-Oracle data stores through ODBC or OLE DB.
Chapter 9, "Developing Applications with Heterogeneous Services"	Provides the information you will need to develop applications that use Oracle Heterogeneous Services.
Appendix A, "Heterogeneous Services Initialization Parameters"	Lists all Heterogeneous Services-specific initialization parameters and their values.
Appendix B, "Data Dictionary Views Available Through Heterogeneous Services"	Lists the data dictionary views that are available through heterogeneous services mapping.
Appendix C, "Data Dictionary Translation for Generic Connectivity"	Lists translations for non-Oracle data dictionary information. Generic connectivity agents translate queries to the Oracle8*i* data dictionary table into queries that retrieve data from a non-Oracle data dictionary.
Appendix D, "Datatype Mapping"	Lists datatypes mapped from ODBC and OLE DB compliant data sources to supported Oracle datatypes.

Changes to This Book

The following aspects of this manual are new in 8.1.6:

- Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity", Appendix C, "Data Dictionary Translation for Generic Connectivity", and Appendix D, "Datatype Mapping" describe generic connectivity.
- The distributed transactions documentation now constitutes its own Part of the book.
- Heterogeneous packages are now documented in *Oracle8i Supplied PL/SQL Packages Reference* rather than in the appendices of this manual.

Audience

This guide is for DBAs who administer or plan to implement a distributed database system involving either Oracle to Oracle database links or Oracle to non-Oracle database links.

Knowledge Assumed of the Reader

Readers of this guide are assumed to be familiar with:

- Relational database concepts and basic database administration as described in *Oracle8i Concepts* and the *Oracle8i Administrator's Guide*.
- The operating system environment under which they are running Oracle.

How to Use This Guide

This manual contains these basic types of information:

Information Type	Chapter
Conceptual	Chapter 1, "Distributed Database Concepts" Chapter 4, "Distributed Transactions Concepts" Chapter 6, "Oracle Heterogeneous Services Concepts"
Administrative	Chapter 2, "Managing a Distributed Database" Chapter 7, "Managing Oracle Heterogeneous Services Using Transparent Gateways" Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity"
Application Development	Chapter 3, "Developing Applications for a Distributed Database System" Chapter 9, "Developing Applications with Heterogeneous Services"
Reference	Appendix A, "Heterogeneous Services Initialization Parameters" Appendix B, "Data Dictionary Views Available Through Heterogeneous Services" Appendix C, "Data Dictionary Translation for Generic Connectivity" Appendix D, "Datatype Mapping"

To acquaint yourself with the basic features of distributed databases in homogeneous and heterogeneous systems, read Chapter 1, "Distributed Database Concepts" and then Chapter 6, "Oracle Heterogeneous Services Concepts".

Refer to the administrative chapters to learn how to perform specific tasks related to management of distributed systems. Refer to the application development chapters if your interest is in designing application that work with distributed systems.

Finally, refer to the reference chapters for information about heterogeneous services initialization parameters and generic connectivity mapping.

Conventions Used in This Guide

The following conventions are used in code fragments in this guide:

<table>
<tr><th>Format</th><th>Indicates</th></tr>
<tr><td>UPPERCASE TEXT</td><td>Text that must be entered exactly as shown. For example:
SQLPLUS username/password
INTO TABLENAME 'table'</td></tr>
<tr><td>lowercase italicized text</td><td>Emphasized term or glossary term. It also identifies a variable for which you should substitute an appropriate value. Parentheses should be entered as shown. For example:
VARCHAR (length)</td></tr>
<tr><td>Vertical bars |</td><td>Alternate choices. For example:
ASC | DESC</td></tr>
<tr><td>Braces { }</td><td>The enclosed items are required, that is, you must choose one of the alternatives. For example:
{column_name | array_def}</td></tr>
<tr><td>Square brackets [ ]</td><td>The enclosed items are optional. For example:
DECIMAL (digits [, precision])</td></tr>
<tr><td><<i>operator</i>></td><td>A SQL operator. For example:
WHERE <i>x</i> <<i>operator</i>> x</td></tr>
<tr><td>Ellipses ...</td><td>A repeated item. For example:
WHERE <i>column_1</i> <<i>operator</i>> x
  AND <i>column_2</i> <<i>operator</i>> <i>y</i>
 [AND ...]</td></tr>
</table>

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of our references. As we write, revise, and evaluate, your opinions are the most important input we receive. At the front of this reference is a reader's comment form that we encourage you to use to tell us both what you like and what you dislike about this

(or other) Oracle manuals. If the form is missing, or you would like to contact us, please use the following address or fax number:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
FAX: 650-506-7228

You can also e-mail your comments to the Information Development department at the following e-mail address: infodev@us.oracle.com

Part I

Distributed Database Systems Concepts and Administration

1 Distributed Database Concepts

This chapter describes the basic concepts and terminology of Oracle's distributed database architecture. The chapter includes:

- Distributed Database Architecture
- Database Links
- Distributed Database Administration
- Transaction Processing in a Distributed System
- Distributed Database Application Development
- National Language Support

> **See Also:** *Getting to Know Oracle8i* for information about features new to the current Oracle8*i* release.

Distributed Database Architecture

A *distributed database system* allows applications to access data from local and remote databases. In a *homogenous distributed system*, each database is an Oracle database. In a *heterogeneous distributed system*, at least one of the databases is a non-Oracle database. Distributed databases use a *client/server* architecture to process information requests.

This section contains the following topics:

- Homogenous Distributed Database Systems
- Heterogeneous Distributed Database Systems
- Client/Server Database Architecture

Homogenous Distributed Database Systems

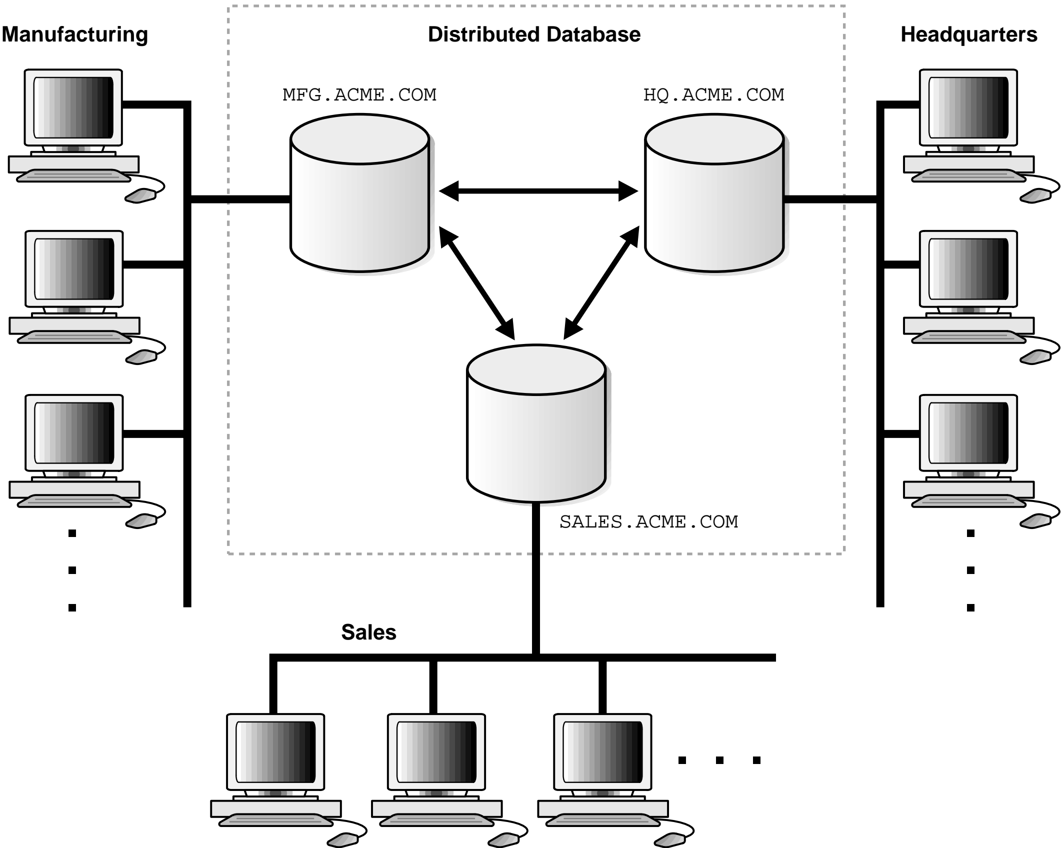
A *homogenous distributed database system* is a network of two or more Oracle databases that reside on one or more machines. Figure 1–1 illustrates a distributed system that connects three databases: HQ, MFG, and SALES. An application can simultaneously access or modify the data in several databases in a single distributed environment. For example, a single query from a Manufacturing client on local database MFG can retrieve joined data from the PRODUCTS table on the local database and the DEPT table on the remote HQ database.

For a client application, the location and platform of the databases are transparent. You can also create *synonyms* for remote objects in the distributed system so that users can access them with the same syntax as local objects. For example, if you are connected to database MFG yet want to access data on database HQ, creating a synonym on MFG for the remote DEPT table allows you to issue this query:

```
SELECT * FROM dept;
```

In this way, a distributed system gives the appearance of native data access. Users on MFG do not have to know that the data they access resides on remote databases.

Figure 1–1 Homogeneous Distributed Database



An Oracle distributed database system can incorporate Oracle databases of different versions. All supported releases of Oracle can participate in a distributed database system. Nevertheless, the applications that work with the distributed database must understand the functionality that is available at each node in the system—for example, a distributed database application cannot expect an Oracle7 database to understand the object SQL extensions that are only available with Oracle8*i*.

Distributed Databases Vs. Distributed Processing

The terms *distributed database* and *distributed processing* are closely related, yet have distinct meanings.

Distributed database	A set of databases in a distributed system that can appear to applications as a single data source.
Distributed processing	The operations that occurs when an application distributes its tasks among different computers in a network. For example, a database application typically distributes front-end presentation tasks to client computers and allows a back-end database server to manage shared access to a database. Consequently, a distributed database application processing system is more commonly referred to as a *client/server* database application system.

Oracle distributed database systems employ a distributed processing architecture. For example, an Oracle database server acts as a client when it requests data that another Oracle database server manages.

Distributed Databases Vs. Replicated Databases

The terms *distributed database system* and *database replication* are related, yet distinct. In a *pure* (that is, non-replicated) distributed database, the system manages a single copy of all data and supporting database objects. Typically, distributed database applications use distributed transactions to access both local and remote data and modify the global database in real-time.

Note: This book discusses only pure distributed databases.

The term *replication* refers to the operation of copying and maintaining database objects in multiple databases belonging to a distributed system. While replication relies on distributed database technology, database replication offers applications benefits that are not possible within a pure distributed database environment.

Most commonly, replication is used to improve local database performance and protect the availability of applications because alternate data access options exist. For example, an application may normally access a local database rather than a remote server to minimize network traffic and achieve maximum performance. Furthermore, the application can continue to function if the local server experiences a failure, but other servers with replicated data remain accessible.

See Also: *Oracle8i Replication* for more information about Oracle's replication features.

Heterogeneous Distributed Database Systems

In a *heterogeneous distributed database system*, at least one of the databases is a non-Oracle system. To the application, the heterogeneous distributed database system appears as a single, local, Oracle database; the local Oracle database server hides the distribution and heterogeneity of the data.

The Oracle database server accesses the non-Oracle system using Oracle8*i* Heterogeneous Services in conjunction with an *agent*. If you access the non-Oracle data store using an Oracle Transparent Gateway, then the agent is a system-specific application. For example, if you include a Sybase database in an Oracle distributed system, then you need to obtain a Sybase-specific transparent gateway so that the Oracle databases in the system can communicate with it.

Alternatively, you can use *generic connectivity* to access non-Oracle data stores so long as the non-Oracle system supports the ODBC or OLE DB protocols. If you use the generic agent included with the Oracle database server, then you do not need to purchase a separate transparent gateway.

> **See Also:** Part III of this manual for more information about Heterogeneous Services.

Heterogeneous Services

Heterogeneous Services (HS) is an integrated component within the Oracle8*i* server and the enabling technology for the current suite of Oracle Transparent Gateway products. HS provides the common architecture and administration mechanisms for Oracle gateway products and other heterogeneous access facilities. Also, it provides upwardly compatible functionality for users of most of the earlier Oracle Transparent Gateway releases.

> **See Also:** Chapter 6, "Oracle Heterogeneous Services Concepts" for an overview of heterogeneous services.

Transparent Gateway Agents

For each non-Oracle system that you access, Heterogeneous Services can use a transparent gateway agent to interface with the specified non-Oracle system. The agent is specific to the non-Oracle system, so each type of system requires a different agent.

The transparent gateway agent facilitates communication between Oracle and non-Oracle databases and uses the Heterogeneous Services component in the

Oracle database server. The agent executes SQL and transactional requests at the non-Oracle system on behalf of the Oracle database server.

> **See Also:** *Oracle Transparent Gateway Installation and User's Guide version 8.1* for detailed information on installation and configuration.

Generic Connectivity

Generic connectivity allows you to connect to non-Oracle8*i* data stores by using either a Heterogeneous Services ODBC agent or a Heterogeneous Services OLE DB agent—both are included with your Oracle8*i* product as a standard feature. Any data source compatible with the ODBC or OLE DB standards can be accessed using a generic connectivity agent.

The advantage to generic connectivity is that it does not require you to purchase and configure a separate system-specific agent. You simply need an ODBC or OLE DB driver that can interface with the agent.

> **See Also:** Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity" for detailed information on installation and configuration of generic connectivity.

Heterogeneous Services Features

The features of the Heterogeneous Services include the following:

Feature	Purpose
Distributed transactions	Allows a transaction to span both Oracle and non-Oracle systems, while still guaranteeing transaction consistency.
SQL translations	Integrates data from non-Oracle systems into the Oracle environment as if the data were stored in one local database. SQL statements are transparently transformed into SQL statement understood by the non-Oracle system.
Data dictionary translations	Makes a non-Oracle system appear as an Oracle database server. SQL statements containing references to Oracle's data dictionary tables are transformed into SQL statements containing references to a non-Oracle system's data dictionary tables.
Pass-Through SQL	Gives application programmers direct access to a non-Oracle system from an Oracle application using the non-Oracle system's SQL dialect.
Stored procedure access	Allows access to stored procedures in SQL-based non-Oracle systems as if they were PL/SQL remote procedures.

Feature	Purpose
NLS support	Supports multi-byte character sets, and translates character sets between a non-Oracle system and the Oracle8*i* server.
Multi-Threaded agents	Takes advantage of your operating system's threading capabilities by reducing the number of required processes.
Agent self-registration	Automates the updating of Heterogeneous Services configuration data on remote hosts, ensuring correct operation over heterogeneous database links.
Generic connectivity	Allows Oracle8*i* to connect to a non-Oracle data store using the ODBC or OLE DB protocol.

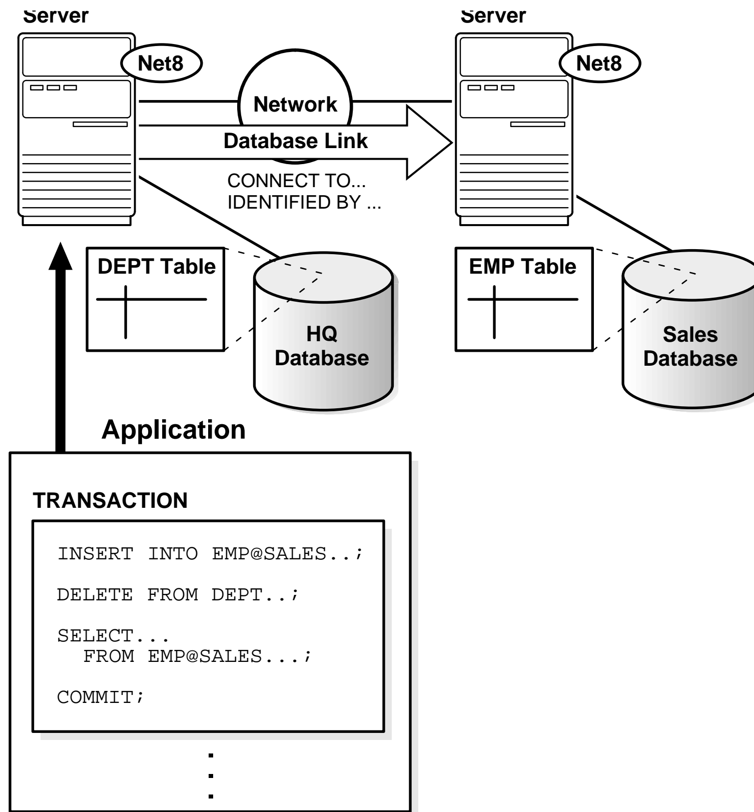
Note: Not all listed features are necessarily supported by your Heterogeneous Services agent or Oracle Transparent Gateway. See your system-specific documentation for supported features.

Client/Server Database Architecture

A database server is the Oracle software managing a database, and a client is an application that requests information from a server. Each computer in a network is a node that can host one or more databases. Each node in a distributed database system can act as a client, a server, or both, depending on the situation.

In Figure 1–2, the host for the HQ database is acting as a database server when a statement is issued against its local data (for example, the second statement in each transaction issues a statement against the local DEPT table), but is acting as a client when it issues a statement against remote data (for example, the first statement in each transaction is issued against the remote table EMP in the SALES database).

Figure 1–2 An Oracle Distributed Database System



Direct and Indirect Connections

A client can connect *directly* or *indirectly* to a database server. A direct connection occurs when a client connects to a server and accesses information from a database contained on that server. For example, if you connect to the HQ database and access the DEPT table on this database as in Figure 1–2, you can issue the following:

```
SELECT * FROM dept;
```

This query is direct because you are not accessing an object on a remote database.

In contrast, an indirect connection occurs when a client connects to a server and then accesses information contained in a database on a different server. For example, if you connect to the HQ database but access the EMP table on the remote SALES database as in Figure 1–2, you can issue the following:

```
SELECT * FROM emp@sales;
```

This query is indirect because the object you are accessing is not on the database to which you are directly connected.

Database Links

The central concept in distributed database systems is a *database link*. A database links is a connection between two physical database servers that allows a client to access them as one logical database.

This section contains the following topics:

- What Are Database Links?
- Why Use Database Links?
- Global Database Names in Database Links
- Names for Database Links
- Types of Database Links
- Users of Database Links
- Creation of Database Links: Examples
- Schema Objects and Database Links
- Database Link Restrictions

What Are Database Links?

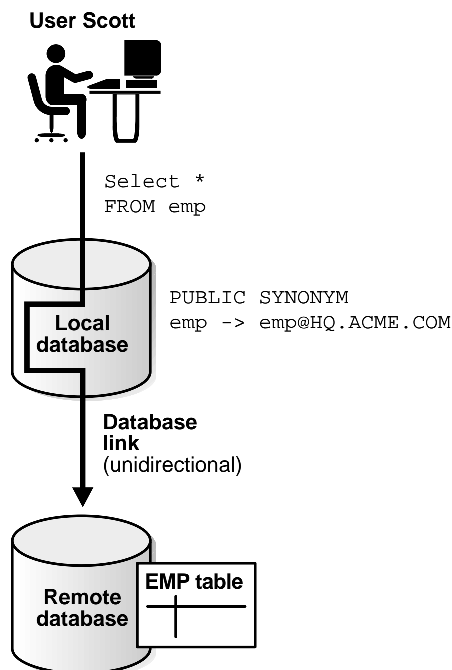
A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, then they must define a link that is stored in the data dictionary of database B.

A database link connection allows local users to access data on a remote database. For this connection to occur, each database in the distributed system must have a unique *global database name* in the network domain. The global database name uniquely identifies a database server in a distributed system.

Figure 1–3 shows an example of user SCOTT accessing the EMP table on the remote database with the global name HQ.ACME.COM:

Figure 1–3 Database Link



Database links are either *private* or *public*. If they are private, then only the user who created the link has access; if they are public, then all database users have access.

One principal difference among database links is the way that connections to a remote database occur. Users accessing a remote database through a:

- *Connected user link* connect as themselves, which means that they must have an account on the remote database with the same username as their account on the local database.
- *Fixed user link* connect using the username and password referenced in the link. For example, if JANE uses a fixed user link that connects to the HR database with the username and password SCOTT/TIGER, then she connects as SCOTT. JANE has all the privileges in HR granted to SCOTT directly, and all the default roles that SCOTT has been granted in the HR database.
- *Current user link* connect as a global user. A local user can connect as a global user in the context of a stored procedure—without storing the global user's password in a link definition. For example, JANE can access a procedure that

SCOTT wrote, accessing SCOTT's account and SCOTT's schema on the HR database. Current user links are an aspect of the Oracle Advanced Security option (formerly called Advanced Networking Option).

Create database links using the CREATE DATABASE LINK statement. After a link is created, you can use it to specify schema objects in SQL statements.

What Are Shared Database Links?

A shared database link is a link between a local server process and the remote database. The link is shared because multiple client processes can use the same link simultaneously.

When a local database is connected to a remote database through a database link, either database can run in dedicated or multi-threaded server (MTS) mode. The following table illustrates the possibilities:

Local Database Mode	Remote Database Mode
Dedicated	Dedicated
Dedicated	Multi-threaded
Multi-threaded	Dedicated
Multi-threaded	Multi-threaded

A shared database link can exist in any of these four configurations. Shared links differ from standard database links in the following ways:

- Different users accessing the same schema object through a database link can share a network connection.
- When a user needs to establish a connection to a remote server from a particular server process, the process can reuse connections already established to the remote server. The reuse of the connection can occur if the connection was established on the same server process with the same database link—possibly in a different session. In a non-shared database link, a connection is not shared across multiple sessions.
- When you use a shared database link in an MTS configuration, a network connection is established directly out of the shared server process in the local server. For a non-shared database link on a local multi-threaded server, this connection would have been established through the local dispatcher, requiring context switches for the local dispatcher, and requiring data to go through the dispatcher.

See Also: *Net8 Administrator's Guide* for information about the multi-threaded server option.

Why Use Database Links?

The great advantage of database links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner. In other words, a local user can access a link to a remote database without having to be a user on the remote database.

For example, assume that employees submit expense reports to Accounts Payable (A/P), and further suppose that a user using an A/P application needs to retrieve information about employees from the HR database. The A/P users should be able to connect to the HR database and execute a stored procedure in the remote HR database that retrieves the desired information. The A/P users should not need to be HR database users to do their jobs; they should only be able to access HR information in a controlled way as limited by the procedure.

Database links allow you to grant limited access on remote databases to local users. By using current user links, you can create centrally managed global users whose password information is hidden from both administrators *and* non-administrators. For example, A/P users can access the HR database as SCOTT, but unlike fixed user links, SCOTT's credentials are not stored where database users can see them.

By using fixed user links, you can create non-global users whose password information is stored in unencrypted form in the LINK$ data dictionary table. Fixed user links are easy to create and require low overhead because there are no SSL or directory requirements, but a security risk results from the storage of password information in the data dictionary.

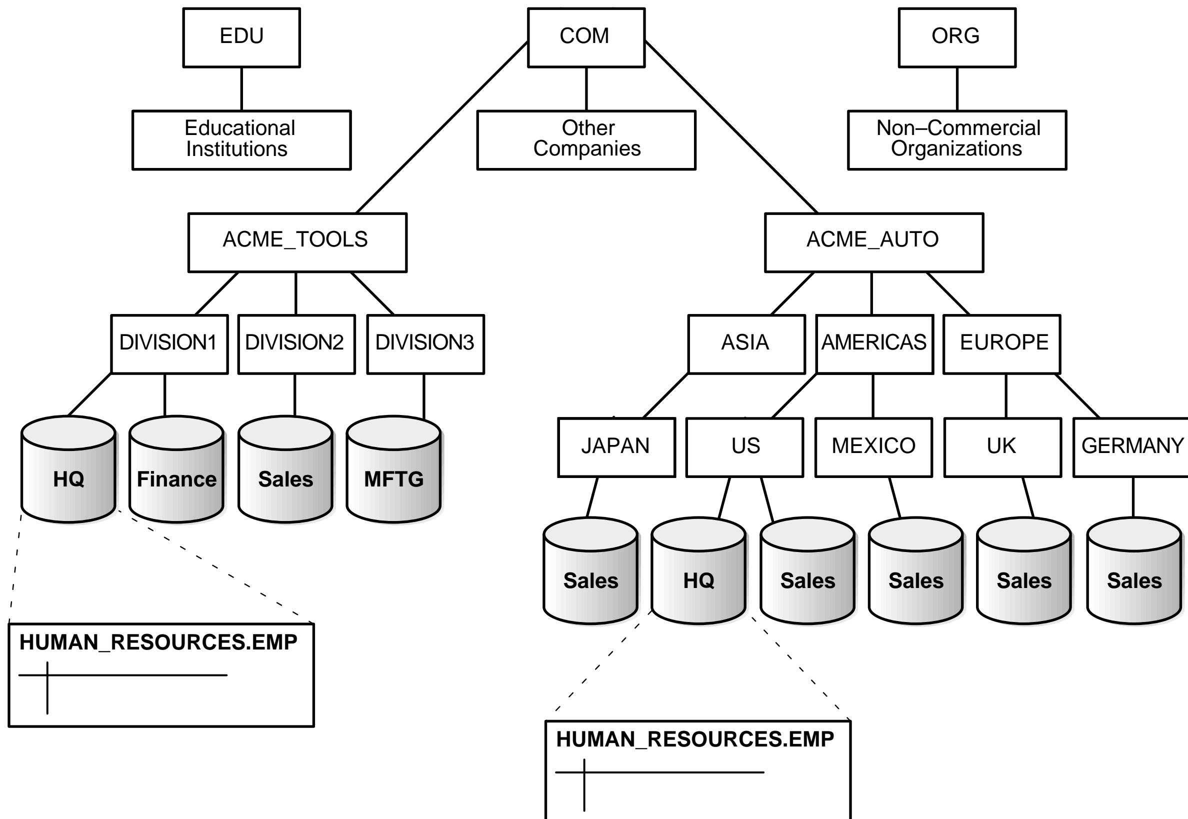
See Also: "Users of Database Links" on page 1-17 for an explanation of database link users, and "Viewing Information About Database Links" for an explanation of how to hide passwords from non-administrators.

Global Database Names in Database Links

To understand how a database link works, you must first understand what a global database name is. Each database in a distributed database is uniquely identified by its global database name. Oracle forms a database's global database name by prefixing the database's network domain, specified by the DB_DOMAIN initialization parameter at database creation, with the individual database name, specified by the DB_NAME initialization parameter.

For example, Figure 1–4 illustrates a representative hierarchical arrangement of databases throughout a network.

Figure 1–4 Network Directories and Global Database Names



The name of a database is formed by starting at the leaf of the tree and following a path to the root. For example, the MFTG database is in DIVISION3 of the ACME_TOOLS branch of the COM domain. The global database name for MFTG is created by concatenating the nodes in the tree as follows:

```
MFTG.DIVISION3.ACME_TOOLS.COM
```

While several databases can share an individual name, each database must have a unique global database name. For example, the network domains US.AMERICAS.ACME_AUTO.COM and UK.EUROPE.ACME_AUTO.COM each

contain a SALES database. The global database naming system distinguishes the SALES database in the AMERICAS division from the SALES database in the EUROPE division as follows:

```
SALES.US.AMERICAS.ACME_AUTO.COM
SALES.UK.EUROPE.ACME_AUTO.COM
```

See Also: "Managing Global Names in a Distributed System" on page 2-2 to learn how to specify and change global database names.

Names for Database Links

Typically, a database link has the same name as the global database name of the remote database that it references. For example, if the global database name of a database is SALES.US.ORACLE.COM, then the database link is also called SALES.US.ORACLE.COM.

When you set the initialization parameter GLOBAL_NAMES to TRUE, Oracle ensures that the name of the database link is the same as the global database name of the remote database. For example, if the global database name for HQ is HQ.ACME.COM, and GLOBAL_NAMES is TRUE, then the link name must be called HQ.ACME.COM. Note that Oracle checks the domain part of the global database name as stored in the data dictionary, *not* the DB_DOMAIN setting in the initialization parameter file (see "Changing the Domain in a Global Database Name" on page 2-4).

If you set the initialization parameter GLOBAL_NAMES to FALSE, then you are not required to use global naming. You can then name the database link whatever you want. For example, you can name a database link to HQ.ACME.COM as FOO.

Note: Oracle Corporation recommends that you use global naming because many useful features, including Oracle Advanced Replication, require global naming.

After you have enabled global naming, database links are essentially transparent to users of a distributed database because the name of a database link is the same as the global name of the database to which the link points. For example, the following statement creates a database link in the local database to remote database SALES:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com USING 'sales1';
```

> **See Also:** *Oracle8i Reference* for more information about specifying the initialization parameter GLOBAL_NAMES.

Types of Database Links

Oracle allows you to create *private*, *public*, and *global* database links. These basic link types differ according to which users are allowed access to the remote database:

Type	Owner	Description
Private	User who created the link. Access ownership data through DBA_DB_LINKS or ALL_DB_LINKS.	Creates link in a specific schema of the local database. Only the owner of a private database link or PL/SQL subprograms in the schema can use this link to access database objects in the corresponding remote database.
Public	User called PUBLIC. Access ownership data through DBA_DB_LINKS or ALL_DB_LINKS.	Creates a database-wide link. All users and PL/SQL subprograms in the database can use the link to access database objects in the corresponding remote database.
Global	User called PUBLIC. Access ownership data through DBA_DB_LINKS or ALL_DB_LINKS.	Creates a network-wide link. When an Oracle network uses Oracle Names, the names servers in the system automatically create and manage global database links for every Oracle database in the network. Users and PL/SQL subprograms in any database can use a global link to access objects in the corresponding remote database.

Determining the type of database links to employ in a distributed database depends on the specific requirements of the applications using the system. Consider these advantages and disadvantages:

Private Database Link	This link is more secure than a public or global link, because only the owner of the private link, or subprograms within the same schema, can use the link to access the remote database.
Public Database Link	When many users require an access path to a remote Oracle database, you can create a single public database link for all users in a database.
Global Database Link	When an Oracle network uses Oracle Names, an administrator can conveniently manage global database links for all databases in the system. Database link management is centralized and simple.

See Also: T"Specifying Link Types" on page 2-9 to learn how to create different types of database links, and "Viewing Information About Database Links" on page 2-21 to learn how to access information about links.

Users of Database Links

When creating the link, you determine which user should connect to the remote database to access the data. The following table explains the differences among the categories of users involved in database links:

User Type	Meaning	Sample Link Creation Syntax
Connected user	A local user accessing a database link in which no fixed username and password have been specified. If SYSTEM accesses a public link in a query, then the connected user is SYSTEM, and Oracle connects to the SYSTEM schema in the remote database. **Note:** A connected user does not have to be the user who created the link, but is any user who is accessing the link.	CREATE PUBLIC DATABASE LINK hq USING 'hq';
Current user	A global user in a CURRENT_USER database link. The global user must be authenticated by an X.509 certificate and a private key and be a user on both databases involved in the link. Current user links are an aspect of the Oracle Advanced Security option. **See Also:** *Oracle Advanced Security Administrator's Guide* for more information about global security.	CREATE PUBLIC DATABASE LINK hq CONNECT TO CURRENT_USER using 'hq';
Fixed user	A user whose username/password is part of the link definition. If a link includes a fixed user, then the fixed user's username and password are used to connect to the remote database.	CREATE PUBLIC DATABASE LINK hq CONNECT TO jane IDENTIFIED BY doe USING 'hq';

See Also: "Specifying Link Users" on page 2-11 to learn how to specify users where creating links.

Connected User Database Links

Connected user links have no connect string associated with them. The advantage of a connected user link is that a user referencing the link connects to the remote database as the same user. Furthermore, because no connect string is associated with the link, no password is stored in clear text in the data dictionary.

Connected user links have some disadvantages. Because these links require users to have accounts and privileges on the remote databases to which they are attempting to connect, they require more privilege administration for administrators. Also, giving users more privileges than they need violates the fundamental security concept of least privilege: users should only be given the privileges they need to perform their jobs.

The ability to use connected user database link depends on several factors, chief among them whether the user is authenticated by Oracle using a password, or externally authenticated by the operating system or a network authentication service. If the user is externally authenticated, then the ability to use a connected user link also depends on whether the remote database accepts remote authentication of users, which is set by the REMOTE_OS_AUTHENT initialization parameter.

The REMOTE_OS_AUTHENT parameter operates as follows:

If REMOTE_OS_AUTHENT is...	Then...
TRUE for the remote database	An externally-authenticated user can connect to the remote database using a connected user database link.
FALSE for the remote database	An externally-authenticated user cannot connect to the remote database using a connected user database link unless a secure protocol or a network authentication service supported by the Oracle Advanced Security option is used.

Fixed User Database Links

A benefit of a named link is that it connects a user in a primary database to a remote database with the security context of the user in the connect string. For example, local user JOE can create a public database link in JOE's schema that specifies the fixed user SCOTT with password TIGER. If JANE uses the fixed user link in a query, then JANE is the user on the local database, but she connects to the remote database as SCOTT/TIGER.

Fixed user links have a username and password associated with the connect string. The username and password are stored in unencrypted form in the data dictionary in the LINK$ table. This fact creates a possible security weakness of fixed user database links: a user with the SELECT ANY TABLE privilege has access to the data dictionary so long as the O7_DICTIONARY_ACCESSIBILITY initialization parameter is set to TRUE, and thus the authentication associated with a fixed user is compromised.

Note: The default value for O7_DICTIONARY_ACCESSIBILITY is TRUE.

For an example of this security problem, assume that JANE does not have privileges to use a private link that connects to the HR database as SCOTT/TIGER, but has SELECT ANY TABLE privilege on a database in which the O7_DICTIONARY_ACCESSIBILITY initialization parameter is set to TRUE. She can select from LINK$ and read that the connect string to HR is SCOTT/TIGER. If JANE has an account on the host on which HR resides, then she can connect to the host and then connect to HR as SCOTT using the password TIGER. She will have all SCOTT's privileges if she connects locally and any audit records will be recorded as if she were SCOTT.

Current User Database Links

Current user database links make use of a global user. A global user must be authenticated by an X.509 certificate and a private key and be a user on both databases involved in the link.

The user invoking the CURRENT_USER link does not have to be a global user. For example, if JANE is authenticated by password to the Accounts Payable database, she can access a stored procedure to retrieve data from the HR database. The procedure uses a current user database link, which connects her to HR as global user SCOTT. SCOTT is a global user and thereby authenticated through a certificate and private key over SSL, but JANE is not.

Note that current user database links have these consequences:

- If the current user database link is *not* accessed from within a stored object, then the current user is the same as the connected user accessing the link. For example, if SCOTT issues a SELECT statement through a current user link, then the current user is SCOTT.
- When executing a stored object such as a procedure, view, or trigger that accesses a database link, the current user is the user that *owns* the stored object, and not the user that *calls* the object. For example, if JANE calls procedure SCOTT.P (created by SCOTT), and a current user link appears *within* the called procedure, then SCOTT is the current user of the link.
- If the stored object is an invoker-rights function, procedure, or package, then the invoker's authorization ID is used to connect as a remote user. For example, if user JANE calls procedure SCOTT.P (an invoker-rights procedure created by

SCOTT), and the link appears inside procedure SCOTT.P, then JANE is the current user.

- You cannot connect to a database as an enterprise user and then use a current user link in a stored procedure that exists in a shared, global schema. For example, you are user JANE and access a stored procedure in the shared schema GUEST on database DB1. You cannot use a current user link in this schema to log on to a remote database.

See Also: "Distributed Database Security" on page 1-25 for more information about security issues relating to database links.

Creation of Database Links: Examples

Create database links using the CREATE DATABASE LINK statement. The table gives examples of SQL statements that create database links in a *local* database to the *remote* SALES.US.AMERICAS.ACME_AUTO.COM database:

SQL Statement	Connects To Database	Connects As	Link Type
`CREATE DATABASE LINK sales.us.americas.acme_auto.com USING 'sales_us';`	SALES using net service name SALES_US	Connected user	Private connected user
`CREATE DATABASE LINK foo CONNECT TO CURRENT_USER USING 'am_sls';`	SALES using service name AM_SLS	Current global user	Private current user
`CREATE DATABASE LINK sales.us.americas.acme_auto.com CONNECT TO scott IDENTIFIED BY tiger USING 'sales_us';`	SALES using net service name SALES_US	SCOTT using password TIGER	Private fixed user
`CREATE PUBLIC DATABASE LINK sales CONNECT TO scott IDENTIFIED BY tiger USING 'rev';`	SALES using net service name REV	SCOTT using password TIGER	Public fixed user
`CREATE SHARED PUBLIC DATABASE LINK sales.us.americas.acme_auto.com CONNECT TO scott IDENTIFIED BY tiger AUTHENTICATED BY anupam IDENTIFIED BY bhide USING 'sales';`	SALES using net service name SALES	SCOTT using password TIGER, authenticated as ANUPAM using password BHIDE	Shared public fixed user

See Also: "Creating Database Links" on page 2-8 to learn how to create links. For CREATE DATABASE LINK syntax, see the *Oracle8i SQL Reference.*

Schema Objects and Database Links

After you have created a database link, you can execute SQL statements that access objects on the remote database. For example, to access remote object EMP using database link FOO, you can issue:

```
SELECT * FROM emp@foo;
```

Constructing properly formed object names using database links is an essential aspect of data manipulation in distributed systems.

Naming of Schema Objects Using Database Links

Oracle uses the global database name to name the schema objects globally using the following scheme:

schema.schema_object@global_database_name

where:

schema	is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.
schema_object	is a logical data structure like a table, index, view, synonym, procedure, package, or a database link.
global_database_name	is the name that uniquely identifies a remote database. This name must be the same as the concatenation of the remote database's initialization parameters DB_NAME and DB_DOMAIN, unless the parameter GLOBAL_NAMES is set to FALSE, in which case any name is acceptable.

For example, using a database link to database SALES.DIVISION3.ACME.COM, a user or application can reference remote data as follows:

```
SELECT * FROM scott.emp@sales.division3.acme.com;  # emp table in scott's schema
SELECT loc FROM scott.dept@sales.division3.acme.com;
```

If GLOBAL_NAMES is set to FALSE, then you can use any name for the link to SALES.DIVISION3.ACME.COM. For example, you can call the link FOO. Then, you can access database as follows:

```
SELECT name FROM scott.emp@foo;  # link name different from global name
```

Synonyms for Schema Objects

Oracle allows you to create *synonyms* so that you can hide the database link name from the user. A synonym allows access to a table on a remote database using the same syntax that you would use to access a table on a local database. For example, assume you issue the following query against a table in a remote database:

```
SELECT * FROM emp@hq.acme.com;
```

You can create the synonym EMP for EMP@HQ.ACME.COM so that you can issue the following query instead to access the same data:

```
SELECT * FROM emp;
```

> **See Also:** "Using Synonyms to Create Location Transparency" on page 2-28 to learn how to create synonyms for objects specified using database links.

Schema Object Name Resolution

To resolve application references to schema objects (a process called *name resolution*), Oracle forms object names hierarchically. For example, Oracle guarantees that each schema within a database has a unique name, and that within a schema each object has a unique name. As a result, a schema object's name is always unique within the database. Furthermore, Oracle resolves application references to an object's local name.

In a distributed database, a schema object such as a table is accessible to all applications in the system. Oracle extends the hierarchical naming model with global database names to effectively create *global object names* and resolve references to the schema objects in a distributed database system. For example, a query can reference a remote table by specifying its fully qualified name, including the database in which it resides.

For example, assume that you connect to the local database as user SYSTEM:

```
CONNECT system/manager@sales1
```

You then issue the following statements using database link HQ.ACME.COM to access objects in the SCOTT and JANE schemas on remote database HQ:

```
SELECT * FROM scott.emp@hq.acme.com;
INSERT INTO jane.accounts@hq.acme.com (acc_no, acc_name, balance)
  VALUES (5001, 'BOWER', 2000);
UPDATE jane.accounts@hq.acme.com
  SET balance = balance + 500;
DELETE FROM jane.accounts@hq.acme.com
  WHERE acc_name = 'BOWER';
```

Database Link Restrictions

You *cannot* perform the following operations using database links:

- Grant privileges on remote objects.
- Execute DESCRIBE operations on some remote objects. The following remote objects, however, do support DESCRIBE operations:
  - Tables
  - Views
  - Procedures
  - Functions
- ANALYZE remote objects.
- Define or enforce referential integrity.
- Grant roles to users in a remote database.
- Obtain non-default roles on a remote database. For example, if JANE connects to the local database and executes a stored procedure that uses a fixed user link connecting as SCOTT, JANE receives SCOTT's default roles on the remote database. Jane cannot issue SET ROLE to obtain a non-default role.
- Execute hash query joins that use MTS connections.
- Use a current user link without authentication through SSL or NT native authentication.

Distributed Database Administration

The following sections explain some of the topics relating to database management in an Oracle distributed database system:

- Site Autonomy

- Distributed Database Security
- Auditing Database Links
- Administration Tools

> **See Also:** TChapter 2, "Managing a Distributed Database" to learn how to administer homogenous systems, and Chapter 7, "Managing Oracle Heterogeneous Services Using Transparent Gateways" to learn how to administer heterogeneous systems.

Site Autonomy

Site autonomy means that each server participating in a distributed database is administered independently from all other databases. Although several databases can work together, each database is a separate repository of data that is managed individually. Some of the benefits of site autonomy in an Oracle distributed database include:

- Nodes of the system can mirror the logical organization of companies or groups that need to maintain independence.
- Local administrators control corresponding local data. Therefore, each database administrator's domain of responsibility is smaller and more manageable.
- Independent failures are less likely to disrupt other nodes of the distributed database. No single database failure need halt all distributed operations or be a performance bottleneck.
- Administrators can recover from isolated system failures independently from other nodes in the system.
- A data dictionary exists for each local database—a global catalog is not necessary to access local data.
- Nodes can upgrade software independently.

Although Oracle allows you to manage each database in a distributed database system independently, you should not ignore the global requirements of the system. For example, you may need to:

- Create additional user accounts in each database to support the links that you create to facilitate server-to-server connections.
- Set additional initialization parameters such as DISTRIBUTED_LOCK_TIMEOUT, DISTRIBUTED_TRANSACTIONS, COMMIT_POINT_STRENGTH, and so forth.

Distributed Database Security

Oracle supports all of the security features that are available with a non-distributed database environment for distributed database systems, including:

- Password authentication for users and roles
- Some types of external authentication for users and roles including:
  - Kerberos version 5 for connected user links
  - DCE for connected user links
- Login packet encryption for client-to-server and server-to-server connections

The following sections explain some additional topics to consider when configuring an Oracle distributed database system:

- Authentication Through Database Links
- Authentication Without Passwords
- Supporting User Accounts and Roles
- Centralized User and Privilege Management
- Data Encryption

> **See Also:** *Oracle Advanced Security Administrator's Guide* for more information about external authentication.

Authentication Through Database Links

Database links are either *private* or *public*, *authenticated* or *non-authenticated*. You create public links by specifying the PUBLIC keyword in the link creation statement. For example, you can issue:

```
CREATE PUBLIC DATABASE LINK foo USING 'sales';
```

You create authenticated links by specifying the CONNECT TO clause, AUTHENTICATED BY clause, or both clauses together in the database link creation statement. For example, you can issue:

```
CREATE DATABASE LINK sales CONNECT TO scott IDENTIFIED BY tiger USING 'sales';
CREATE SHARED PUBLIC DATABASE LINK sales CONNECT TO mick IDENTIFIED BY jagger
     AUTHENTICATED BY david IDENTIFIED BY bowie USING 'sales';
```

This table describes how users access the remote database through the link:

Link Type	Authenticated?	Security Access
Private	No	When connecting to the remote database, Oracle uses security information (userid/password) taken from the local session. Hence, the link is a connected user database link. Passwords must be synchronized between the two databases.
Private	Yes	The userid/password is taken from the link definition rather than from the local session context. Hence, the link is a fixed user database link. This configuration allows passwords to be different on the two databases, but the local database link password must match the remote database password. The password is stored in clear text on the local system catalog, adding a security risk.
Public	No	Works the same as a private non-authenticated link, except that all users can reference this pointer to the remote database.
Public	Yes	All users on the local database can access the remote database and all use the same userid/password to make the connection. Also, the password is stored in clear text in the local catalog, so you can see the password if you have sufficient privileges in the local database.

Authentication Without Passwords

When using a connected user or current user database link, you can use an external authentication source such as Kerberos to obtain *end-to-end security*. In end-to-end authentication, credentials are passed from server to server and can be authenticated by a database server belonging to the same domain. For example, if JANE is authenticated externally on a local database, and wants to use a connected user link to connect as herself to a remote database, the local server passes the security ticket to the remote database.

Supporting User Accounts and Roles

In a distributed database system, you must carefully plan the user accounts and roles that are necessary to support applications using the system. Note that:

- The user accounts necessary to establish server-to-server connections must be available in all databases of the distributed database system.

- The roles necessary to make available application privileges to distributed database application users must be present in all databases of the distributed database system.

As you create the database links for the nodes in a distributed database system, determine which user accounts and roles each site needs to support server-to-server connections that use the links.

In a distributed environment, users typically require access to many network services. When you must configure separate authentications for each user to access each network service, security administration can become unwieldy, especially for large systems.

> **See Also:** "Creating Database Links" on page 2-8 for more information about the user accounts that must be available to support different types of database links in the system.

Centralized User and Privilege Management

Oracle provides different ways for you to manage the users and privileges involved in a distributed system. For example, you have these options:

- Enterprise user management. You can create global users that are authenticated through SSL, then manage these users and their privileges in a directory through an independent enterprise directory service.
- Network authentication service. This common technique simplifies security management for distributed environments. You can use the Net8 Oracle Advanced Security option to enhance Net8 and the security of an Oracle distributed database system. Windows NT native authentication is an example of a non-Oracle authentication solution.

> **See Also:** *Net8 Administrator's Guide* and *Oracle Advanced Security Administrator's Guide* for more information about global user security.

Schema-Dependent Global Users One option for centralizing user and privilege management is to create the following:

- A global user in a centralized directory.
- A user in every database that the global user must connect to.

For example, you can create a global user called FRED with the following SQL statement:

```
CREATE USER fred IDENTIFIED GLOBALLY AS 'CN=fred adams,O=Oracle,C=England';
```

This solution allows a single global user to be authenticated by a centralized directory.

The schema-dependent global user solution has the consequence that you must create a user called FRED on every database that this user must access. Because most users need permission to access an application schema but do not need their own schemas, the creation of a separate account in each database for every global user creates significant overhead. Because of this problem, Oracle also supports schema-independent users, which are global users that an access a single, generic schema in every database.

Schema-Independent Global Users Oracle8*i* supports functionality that allows a global user to be centrally managed by an enterprise directory service. Users who are managed in the directory are called *enterprise users*. This directory contains information about:

- Which databases in a distributed system an enterprise user can access
- Which role on each database an enterprise user can use
- Which schema on each database an enterprise user can connect to

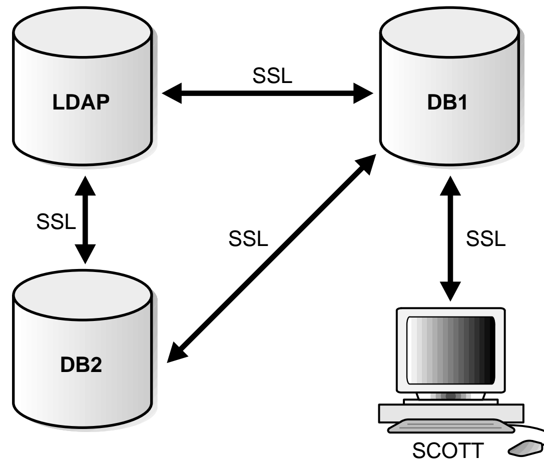
The administrator of each database is not required to create a global user account for each enterprise user on each database to which the enterprise user needs to connect. Instead, multiple enterprise users can connect to the same database schema, called a *shared schema*.

Note: You cannot access a current user database link in a shared schema.

For example, suppose JANE, BILL, and SCOTT all use a human resources application. The HR application objects are all contained in the GUEST schema on the HR database. In this case, you can create a local global user account to be used as a shared schema. This global username, that is, shared schema name, is GUEST. JANE, BILL, and SCOTT are all created as enterprise users in the directory service. They are also mapped to the GUEST schema in the directory, and can be assigned different authorizations in the HR application.

Figure 1–5 illustrates an example of global user security using the enterprise directory service:

Figure 1–5 Global User Security



Assume that the enterprise directory service contains the following information on enterprise users for DB1 and DB2:

Database	Role	Schema	Enterprise Users
DB1	clerk1	guest	bill scott
DB2	clerk2	guest	jane scott

Also, assume that the local administrators for DB1 and DB2 have issued statements as follows:

Database	CREATE Statements

DB1	`CREATE USER guest IDENTIFIED GLOBALLY AS '';`
	`CREATE ROLE clerk1 GRANT select ON emp;`
	`CREATE PUBLIC DATABASE LINK db2_link CONNECT AS CURRENT_USER USING 'db2';`

DB2	`CREATE USER guest IDENTIFIED GLOBALLY;` `CREATE ROLE clerk2 GRANT select ON dept;`

Assume that enterprise user SCOTT requests a connection to local database DB1 in order to execute a distributed transaction involving DB2. The following steps occur (not necessarily in this exact order):

1. Enterprise user SCOTT and database DB1 mutually authenticate one another using SSL.
2. SCOTT issues the following statement:

```
SELECT e.ename, d.loc
FROM emp e, dept@db2_link d
WHERE e.deptno=d.deptno
```

3. Databases DB1 and DB2 mutually authenticate one another using SSL.
4. DB1 queries the enterprise directory service to determine whether enterprise user SCOTT has access to DB1, and discovers SCOTT can access local schema GUEST using role CLERK1.
5. Database DB2 queries the enterprise directory service to determine whether enterprise user SCOTT has access to DB2, and discovers SCOTT can access local schema GUEST using role CLERK2.
6. Enterprise user SCOTT logs into DB2 to schema GUEST with role CLERK2 and issues a SELECT to obtain the required information and transfer it to DB1.
7. DB1 receives the requested data from DB2 and returns it to the client SCOTT.

See Also: *Net8 Administrator's Guide* and *Oracle Advanced Security Administrator's Guide* for more information about enterprise user security.

Data Encryption

The Oracle Advanced Security option also enables Net8 and related products to use network data encryption and checksumming so that data cannot be read or altered. It protects data from unauthorized viewing by using the RSA Data Security RC4 or the Data Encryption Standard (DES) encryption algorithm.

To ensure that data has not been modified, deleted, or replayed during transmission, the security services of the Oracle Advanced Security option can generate a cryptographically secure message digest and include it with each packet sent across the network.

See Also: *Net8 Administrator's Guide* and *Oracle Advanced Security Administrator's Guide* for more information about these and other features of the Oracle Advanced Security option.

Auditing Database Links

You must always perform auditing operations locally. That is, if a user acts in a local database and accesses a remote database through a database link, the local actions are audited in the local database, and the remote actions are audited in the remote database—provided that appropriate audit options are set in the respective databases.

The remote database cannot determine whether a successful connect request and subsequent SQL commands come from another server or from a locally connected client. For example, assume the following:

- Fixed user link HR.ACME.COM connects local user JANE to the remote HR database as remote user SCOTT
- SCOTT is audited on the remote database

Actions performed during the remote database session are audited as if SCOTT were connected locally to HR and performing the same actions there. You must set audit options in the remote database to capture the actions of the username—in this case, SCOTT on the HR database—embedded in the link if the desired effect is to audit what JANE is doing in the remote database.

Note: You can audit the global username for global users.

You cannot set local auditing options on remote objects. Therefore, you cannot audit use of a database link, although access to remote objects can be audited on the remote database.

Administration Tools

The database administrator has several choices for tools to use when managing an Oracle distributed database system:

- Enterprise Manager
- Third-Party Administration Tools
- SNMP Support

Enterprise Manager

Enterprise Manager is Oracle's database administration tool. The graphical component of Enterprise Manager allows you to perform database administration tasks with the convenience of a graphical user interface (GUI). The non-graphical component of Enterprise Manager provides a command line interface.

Enterprise Manager provides administrative functionality for distributed databases through an easy-to-use interface. You can use Enterprise Manager to:

- Administer multiple databases. You can use Enterprise Manager to administer a single database or to simultaneously administer multiple databases.
- Centralize database administration tasks. You can administer both local and remote databases running on any Oracle platform in any location worldwide. In addition, these Oracle platforms can be connected by any network protocols supported by Net8.
- Dynamically execute SQL, PL/SQL, and Enterprise Manager commands. You can use Enterprise Manager to enter, edit, and execute statements. Enterprise Manager also maintains a history of statements executed.

  Thus, you can re-execute statements without retyping them, a particularly useful feature if you need to execute lengthy statements repeatedly in a distributed database system.
- Manage security features such as global users, global roles, and the enterprise directory service.

Third-Party Administration Tools

Currently more than 60 companies produce more than 150 products that help manage Oracle databases and networks, providing a truly open environment.

SNMP Support

Besides its network administration capabilities, Oracle *Simple Network Management Protocol* (*SNMP*) support allows an Oracle database server to be located and queried by any SNMP-based network management system. SNMP is the accepted standard underlying many popular network management systems such as:

- HP's OpenView
- Digital's POLYCENTER Manager on NetView
- IBM's NetView/6000
- Novell's NetWare Management System

- SunSoft's SunNet Manager

See Also: *Oracle SNMP Support Reference Guide* for more information about SNMP.

Transaction Processing in a Distributed System

A transaction is a logical unit of work constituted by one or more SQL statements executed by a single user. A transaction begins with the user's first executable SQL statement and ends when it is committed or rolled back by that user.

A *remote transaction* contains only statements that access a single remote node. A *distributed transaction* contains statements that access more than one node.

The following sections define important concepts in transaction processing and explain how transactions access data in a distributed database:

- Remote SQL Statements
- Distributed SQL Statements
- Shared SQL for Remote and Distributed Statements
- Remote Transactions
- Distributed Transactions
- Two-Phase Commit Mechanism
- Database Link Name Resolution
- Schema Object Name Resolution

Remote SQL Statements

A *remote query* statement is a query that selects information from one or more remote tables, all of which reside at the same remote node. For example, the following query accesses data from the DEPT table in the SCOTT schema of the remote SALES database:

```
SELECT * FROM scott.dept@sales.us.americas.acme_auto.com;
```

A *remote update* statement is an update that modifies data in one or more tables, all of which are located at the same remote node. For example, the following query updates the DEPT table in the SCOTT schema of the remote SALES database:

```
UPDATE scott.dept@mktng.us.americas.acme_auto.com
```

```
SET loc = 'NEW YORK'
WHERE deptno = 10;
```

Note: A remote update can include a subquery that retrieves data from one or more remote nodes, but because the update happens at only a single remote node, the statement is classified as a remote update.

Distributed SQL Statements

A *distributed query* statement retrieves information from two or more nodes. For example, the following query accesses data from the local database as well as the remote SALES database:

```
SELECT ename, dname
  FROM scott.emp e, scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

A *distributed update* statement modifies data on two or more nodes. A distributed update is possible using a PL/SQL subprogram unit such as a procedure or trigger that includes two or more remote updates that access data on different nodes. For example, the following PL/SQL program unit updates tables on the local database and the remote SALES database:

```
BEGIN
  UPDATE scott.dept@sales.us.americas.acme_auto.com
    SET loc = 'NEW YORK'
    WHERE deptno = 10;
  UPDATE scott.emp
    SET deptno = 11
    WHERE deptno = 10;
END;
COMMIT;
```

Oracle sends statements in the program to the remote nodes, and their execution succeeds or fails as a unit.

Shared SQL for Remote and Distributed Statements

The mechanics of a remote or distributed statement using shared SQL are essentially the same as those of a local statement. The SQL text must match, and the referenced objects must match. If available, shared SQL areas can be used for the local and remote handling of any statement or decomposed query.

See Also: *Oracle8i Concepts* for more information about shared SQL.

Remote Transactions

A *remote transaction* contains one or more remote statements, all of which reference a single remote node. For example, the following transaction contains two statements, each of which accessing the remote SALES database:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
UPDATE scott.emp@sales.us.americas.acme_auto.com
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

Distributed Transactions

A *distributed transaction* is a transaction that includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database. For example, this transaction updates the local database and the remote SALES database:

```
UPDATE scott.dept@sales.us.americas.acme_auto.com
  SET loc = 'NEW YORK'
  WHERE deptno = 10;
UPDATE scott.emp
  SET deptno = 11
  WHERE deptno = 10;
COMMIT;
```

Note: If all statements of a transaction reference only a single remote node, the transaction is remote, not distributed.

Two-Phase Commit Mechanism

An database must guarantee that all statements in a transaction, distributed or non-distributed, either commit or roll back as a unit. The effects of an ongoing transaction should be invisible to all other transactions at all nodes; this transparency should be true for transactions that include any type of operation, including queries, updates, or remote procedure calls.

The general mechanisms of transaction control in a non-distributed database are discussed in the *Oracle8i Concepts*. In a distributed database, Oracle must coordinate transaction control with the same characteristics over a network and maintain data consistency, even if a network or system failure occurs.

Oracle's *two-phase commit* mechanism guarantees that *all* database servers participating in a distributed transaction either all commit or all roll back the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

See Also: Chapter 4, "Distributed Transactions Concepts" for more information about Oracle's two-phase commit mechanism.

Database Link Name Resolution

A *global object name* is an object specified using a database link. The essential components of a global object name are:

- Object name
- Database name
- Domain

The following table shows the components of an explicitly specified global database object name:

Statement	Object	Database	Domain
`SELECT * FROM joan.dept@sales.acme.com`	`dept`	`sales`	`acme.com`
`SELECT * FROM emp@mktg.us.acme.com`	`emp`	`mktg`	`us.acme.com`

Whenever a SQL statement includes a reference to a global object name, Oracle searches for a database link with a name that matches the database name specified in the global object name. For example, if you issue the following statement:

```
SELECT * FROM scott.emp@orders.us.acme.com;
```

Oracle searches for a database link called ORDERS.US.ACME.COM. Oracle performs this operation to determine the path to the specified remote database.

Oracle always searches for matching database links in the following order:

1. Private database links in the schema of the user who issued the SQL statement.
2. Public database links in the local database.
3. Global database links (only if an Oracle Names Server is available).

Name Resolution When the Global Database Name Is Complete

Assume that you issue the following SQL statement, which specifies a complete global database name:

```
SELECT * FROM emp@prod1.us.oracle.com
```

In this case, both the database name (PROD1) and domain components (US.ORACLE.COM) are specified, so Oracle searches for private, public, and global database links. Oracle searches only for links that match the specified global database name.

Name Resolution When the Global Database Name Is Partial

If any part of the domain is specified, Oracle assumes that a complete global database name is specified. If a SQL statement specifies a partial global database name (that is, only the database component is specified), Oracle appends the value in the DB_DOMAIN parameter to the value in the DB_NAME parameter to construct a complete name. For example, assume you issue the following statements:

```
CONNECT scott/tiger@locdb
SELECT * FROM scott.emp@orders;
```

If the network domain for LOCDB is US.ACME.COM, then Oracle appends this domain to ORDERS to construct the complete global database name of ORDERS.US.ACME.COM. Oracle searches for database links that match only the constructed global name. If a matching link is not found, Oracle returns an error and the SQL statement cannot execute.

Name Resolution When No Global Database Name Is Specified

If a global object name references an object in the local database and a database link name is *not* specified using the @ symbol, then Oracle automatically detects that the object is local and does not search for or use database links to resolve the object reference. For example, assume that you issue the following statements:

```
CONNECT scott/tiger@locdb
SELECT * from scott.emp;
```

Because the second statement does not specify a global database name using a database link connect string, Oracle does not search for database links.

Terminating the Search for Name Resolution

Oracle does not necessarily stop searching for matching database links when it finds the first match. Oracle must search for matching private, public, and network database links until it determines a complete path to the remote database (both a remote account and service name).

The first match determines the remote schema as illustrated in the following table:

If you...	Then Oracle...	As in the example...
Do *not* specify the CONNECT clause	Uses a *connected user* database link	`CREATE DATABASE LINK k1 USING 'prod'`
Do specify the CONNECT TO ... IDENTIFIED BY clause	Uses a *fixed user* database link	`CREATE DATABASE LINK k2 CONNECT TO scott IDENTIFIED BY tiger USING 'prod'`
Specify the CONNECT TO CURRENT_USER clause	Uses a *current user* database link	`CREATE DATABASE LINK k3 CONNECT TO CURRENT_USER USING 'prod'`
Do *not* specify the USING clause	Searches until it finds a link specifying a database string. If matching database links are found and a string is never identified, Oracle returns an error.	`CREATE DATABASE LINK k4 CONNECT TO CURRENT_USER`

After Oracle determines a complete path, it creates a remote session—assuming that an identical connection is not already open on behalf of the same local session. If a session already exists, Oracle reuses it.

Schema Object Name Resolution

After the local Oracle database connects to the specified remote database on behalf of the local user that issued the SQL statement, object resolution continues as if the remote user had issued the associated SQL statement. The first match determines the remote schema according to the following rules:

If you use...	Then object resolution proceeds in the...
A fixed user database link	Schema specified in the link creation statement.
A connected user database link	Connected user's remote schema.
A current user database link	Current user's schema.

If Oracle cannot find the object, then it checks public objects of the remote database. If it cannot resolve the object, then the established remote session remains but the SQL statement cannot execute and returns an error.

Examples of Global Object Name Resolution

The following are examples of global object name resolution in a distributed database system. For all the following examples, assume that:

- The remote database is named SALES.DIVISION3.ACME.COM.
- The local database is named HQ.DIVISION3.ACME.COM.
- An Oracle Names Server (and therefore, global database links) is not available.

Example: Resolving a Complete Object Name This example illustrates how Oracle resolves a complete global object name and determines the appropriate path to the remote database using both a private and public database link. For this example, assume that a remote table EMP is contained in the schema TSMITH.

Consider the following statements issued by SCOTT at the local database:

```
CONNECT scott/tiger@hq

CREATE PUBLIC DATABASE LINK sales.division3.acme.com
CONNECT TO guest IDENTIFIED BY network
  USING 'dbstring';
```

Later, JWARD connects and issues the following statements:

```
CONNECT jward/bronco@hq

CREATE DATABASE LINK sales.division3.acme.com
  CONNECT TO tsmith IDENTIFIED BY radio;

UPDATE tsmith.emp@sales.division3.acme.com
  SET deptno = 40
  WHERE deptno = 10;
```

Oracle processes the final statement as follows:

1. Oracle determines that a complete global object name is referenced in JWARD's update statement. Therefore, the system begins searching in the local database for a database link with a matching name.
2. Oracle finds a matching private database link in the schema JWARD. Nevertheless, the private database link JWARD.SALES.DIVISION3.ACME.COM does not indicate a complete path to the remote SALES database, only a remote account. Therefore, Oracle now searches for a matching public database link.
3. Oracle finds the public database link in SCOTT's schema. From this public database link, Oracle takes the service name DBSTRING.
4. Combined with the remote account taken from the matching private fixed user database link, Oracle determines a complete path and proceeds to establish a connection to the remote SALES database as user TSMITH/RADIO.
5. The remote database can now resolve the object reference to the EMP table. Oracle searches in the TSMITH schema and finds the referenced EMP table.
6. The remote database completes the execution of the statement and returns the results to the local database.

Example: Resolving a Partial Object Name This example illustrates how Oracle resolves a partial global object name and determines the appropriate path to the remote database using both a private and public database link.

For this example, assume that:

- A table EMP on the remote database SALES is contained in the schema TSMITH, but not in schema SCOTT.
- A public synonym named EMP resides at local database HQ and points to TSMITH.EMP at the remote database SALES.
- The public database link in "Example: Resolving a Complete Object Name" on page 1-39 is already created on local database HQ:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com
  CONNECT TO guest IDENTIFIED BY network
  USING 'dbstring';
```

Consider the following statements issued at local database HQ:

```
CONNECT scott/tiger@hq
```

```
CREATE DATABASE LINK sales.division3.acme.com;

DELETE FROM emp@sales
  WHERE empno = 4299;
```

Oracle processes the final DELETE statement as follows:

1. Oracle notices that a partial global object name is referenced in SCOTT's DELETE statement. It expands it to a complete global object name using the domain of the local database as follows:

   ```
   DELETE FROM emp@sales.division3.acme.com
     WHERE empno = 4299;
   ```

2. Oracle searches the local database for a database link with a matching name.
3. Oracle finds a matching *private* connected user link in the schema SCOTT, but the private database link indicates no path at all. Oracle uses the connected username/password as the remote account portion of the path and then searches for and finds a matching *public* database link:

   ```
   CREATE PUBLIC DATABASE LINK sales.division3.acme.com
     CONNECT TO guest IDENTIFIED BY network
     USING 'dbstring';
   ```

4. Oracle takes the database net service name DBSTRING from the public database link. At this point, Oracle has determined a complete path.
5. Oracle connects to the remote database as SCOTT/TIGER and searches for and does not find an object named EMP in the schema SCOTT.
6. The remote database searches for a public synonym named EMP and finds it.
7. The remote database executes the statement and returns the results to the local database.

Global Name Resolution in Views, Synonyms, and Procedures

A view, synonym, or PL/SQL program unit (for example, a procedure, function, or trigger) can reference a remote schema object by its global object name. If the global object name is complete, then Oracle stores the definition of the object without expanding the global object name. If the name is partial, however, Oracle expands the name using the domain of the local database name.

The following table explains when Oracle completes the expansion of a partial global object name for views, synonyms, and program units:

If you...	Then Oracle...
Create a view	Does *not* expand partial global names—the data dictionary stores the exact text of the defining query. Instead, Oracle expands a partial global object name each time a statement that uses the view is parsed.
Create a synonym	Expands partial global names. The definition of the synonym stored in the data dictionary includes the expanded global object name.
Compile a program unit	Expands partial global names.

What Happens When Global Names Change

Global name changes can affect views, synonyms, and procedures that reference remote data using partial global object names. If the global name of the referenced database changes, views and procedures may try to reference a nonexistent or incorrect database. On the other hand, synonyms do not expand database link names at runtime, so they do not change.

For example, consider two databases named SALES.UK.ACME.COM and HR.UK.ACME.COM. Also, assume that the SALES database contains the following view and synonym:

```
CREATE VIEW employee_names AS
        SELECT ename FROM scott.emp@hr;

CREATE SYNONYM employee FOR scott.emp@hr;
```

Oracle expands the EMPLOYEE synonym definition and stores it as:

```
scott.emp@hr.uk.acme.com
```

Scenario 1: Both Databases Change Names First, consider the situation where both the Sales and Human Resources departments are relocated to the United States. Consequently, the corresponding global database names are both changed as follows:

Old Global Name	New Global Name
SALES.UK.ACME.COM	SALES.US.ORACLE.COM

Old Global Name	New Global Name
HR.UK.ACME.COM	HR.US.ACME.COM

The following table describes query expansion before and after the change in global names:

Query on SALES	Expansion Before Change	Expansion After Change
SELECT * FROM employee_names	SELECT * FROM scott.emp@hr.uk.acme.com	SELECT * FROM scott.emp@hr.us.acme.com
SELECT * FROM employee	SELECT * FROM scott.emp@hr.uk.acme.com	SELECT * FROM scott.emp@hr.uk.acme.com

Scenario 2: One Database Changes Names Now consider that only the Sales department is moved to the United States; Human Resources remains in the UK. Consequently, the corresponding global database names are both changed as follows:

Old Global Name	New Global Name
SALES.UK.ACME.COM	SALES.US.ORACLE.COM
HR.UK.ACME.COM	no change

The following table describes query expansion before and after the change in global names:

Query on SALES	Expansion Before Change	Expansion After Change
SELECT * FROM employee_names	SELECT * FROM scott.emp@hr.uk.acme.com	SELECT * FROM scott.emp@hr.us.acme.com
SELECT * FROM employee	SELECT * FROM scott.emp@hr.uk.acme.com	SELECT * FROM scott.emp@hr.uk.acme.com

In this case, the defining query of the EMPLOYEE_NAMES view expands to a non-existent global database name. On the other hand, the EMPLOYEE synonym continues to reference the correct database, HR.UK.ACME.COM.

Distributed Database Application Development

Application development in a distributed system raises issues that are not applicable in a non-distributed system. This section contains the following topics relevant for distributed application development:

- Transparency in a Distributed Database System
- Remote Procedure Calls (RPCs)
- Distributed Query Optimization

> **See Also:** Chapter 3, "Developing Applications for a Distributed Database System" to learn how to develop applications for distributed systems.

Transparency in a Distributed Database System

With minimal effort, you can develop applications that make an Oracle distributed database system transparent to users that work with the system. The goal of transparency is to make a distributed database system appear as though it is a single Oracle database. Consequently, the system does not burden developers and users of the system with complexities that would otherwise make distributed database application development challenging and detract from user productivity.

The following sections explain more about transparency in a distributed database system.

Location Transparency

An Oracle distributed database system has features that allow application developers and administrators to hide the physical location of database objects from applications and users. *Location transparency* exists when a user can universally refer to a database object such as a table, regardless of the node to which an application connects. Location transparency has several benefits, including:

- Access to remote data is simple, because database users do not need to know the physical location of database objects.
- Administrators can move database objects with no impact on end-users or existing database applications.

Most typically, administrators and developers use synonyms to establish location transparency for the tables and supporting objects in an application schema. For example, the following statements create synonyms in a database for tables in another, remote database.

```
CREATE PUBLIC SYNONYM emp
  FOR scott.emp@sales.us.americas.acme_auto.com
CREATE PUBLIC SYNONYM dept
  FOR scott.dept@sales.us.americas.acme_auto.com
```

Now, rather than access the remote tables with a query such as:

```
SELECT ename, dname
  FROM scott.emp@sales.us.americas.acme_auto.com e,
       scott.dept@sales.us.americas.acme_auto.com d
  WHERE e.deptno = d.deptno;
```

An application can issue a much simpler query that does not have to account for the location of the remote tables.

```
SELECT ename, dname
  FROM emp e, dept d
  WHERE e.deptno = d.deptno;
```

In addition to synonyms, developers can also use views and stored procedures to establish location transparency for applications that work in a distributed database system.

SQL and COMMIT Transparency

Oracle's distributed database architecture also provides query, update, and transaction transparency. For example, standard SQL commands such as SELECT, INSERT, UPDATE, and DELETE work just as they do in a non-distributed database environment. Additionally, applications control transactions using the standard SQL commands COMMIT, SAVEPOINT, and ROLLBACK—there is no requirement for complex programming or other special operations to provide distributed transaction control.

- The statements in a single transaction can reference any number of local or remote tables.
- Oracle guarantees that all nodes involved in a distributed transaction take the same action: they either all commit or all roll back the transaction.
- If a network or system failure occurs during the commit of a distributed transaction, the transaction is automatically and transparently resolved globally; that is, when the network or system is restored, the nodes either all commit or all roll back the transaction.

Internal Operations Each committed transaction has an associated *system change number* (*SCN*) to uniquely identify the changes made by the statements within that transaction. In a distributed database, the SCNs of communicating nodes are coordinated when:

- A connection is established using the path described by one or more database links.
- A distributed SQL statement is executed.
- A distributed transaction is committed.

Among other benefits, the coordination of SCNs among the nodes of a distributed database system allows global distributed read-consistency at both the statement and transaction level. If necessary, global distributed time-based recovery can also be completed.

Replication Transparency

Oracle also provide many features to transparently replicate data among the nodes of the system. For more information about Oracle's replication features, see *Oracle8i Replication.*

Remote Procedure Calls (RPCs)

Developers can code PL/SQL packages and procedures to support applications that work with a distributed database. Applications can make local procedure calls to perform work at the local database and *remote procedure calls (RPCs)* to perform work at a remote database.

When a program calls a remote procedure, the local server passes all procedure parameters to the remote server in the call. For example, the following PL/SQL program unit calls the packaged procedure DEL_EMP located at the remote SALES database and passes it the parameter 1257:

```
BEGIN
 emp_mgmt.del_emp@sales.us.americas.acme_auto.com(1257);
END;
```

In order for the RPC to succeed, the called procedure must exist at the remote site.

When developing packages and procedures for distributed database systems, developers must code with an understanding of what program units should do at remote locations, and how to return the results to a calling application.

Distributed Query Optimization

Distributed query optimization is a default Oracle8*i* feature that reduces the amount of data transfer required between sites when a transaction retrieves data from remote tables referenced in a distributed SQL statement.

Distributed query optimization uses Oracle's cost-based optimization to find or generate SQL expressions that extract only the necessary data from remote tables, process that data at a remote site or sometimes at the local site, and send the results to the local site for final processing. This operation reduces the amount of required data transfer when compared to the time it takes to transfer all the table data to the local site for processing.

Using various cost-based optimizer hints such as DRIVING_SITE, NO_MERGE, and INDEX, you can control where Oracle processes the data and how it accesses the data.

See Also: "Using Cost-Based Optimization" on page 3-5 for more information about cost-based optimization.

National Language Support

Oracle supports environments in which clients, Oracle database servers, and non-Oracle servers use different character sets. In Oracle8*i*, NCHAR support is provided for heterogeneous environments. You can set a variety of NLS and HS parameters to control data conversion between different character sets.

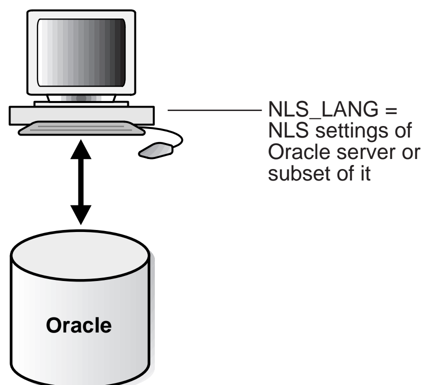
Character settings are defined by the following NLS and HS parameters:

Parameters	**Environment**	**Defined For**
NLS_LANG	Client-Server	Client
NLS_LANGUAGE NLS_CHARACTERSET NLS_TERRITORY	Client-Server Non-Heterogeneous Distributed Heterogeneous Distributed	Oracle database server
HS_LANGUAGE	Heterogeneous Distributed	Non-Oracle server Transparent gateway
NLS_NCHAR HS_NLS_NCHAR	Heterogeneous Distributed	Oracle database server Transparent gateway

Client/Server Environment

In a client/server environment, set the client character set to be the same as or a subset of the Oracle database server character set, as illustrated in Figure 1–6:

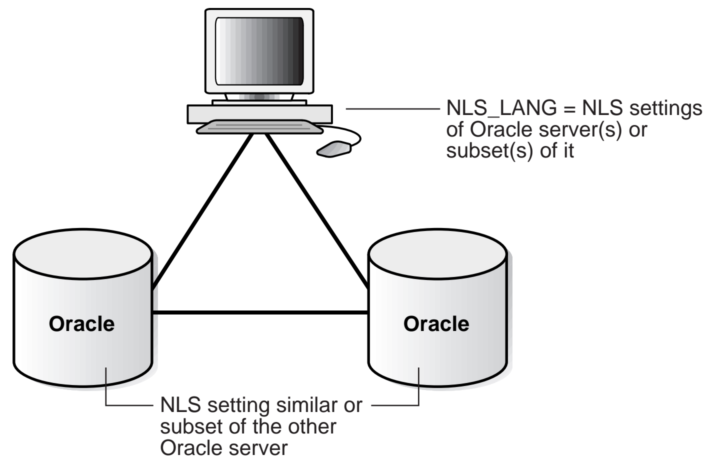
Figure 1–6 NLS Settings in a Client-Server Environment



Homogeneous Distributed Environment

In a non-heterogeneous environment, the client and server character sets should be either the same as or subsets of the main server character set, as illustrated in Figure 1–7:

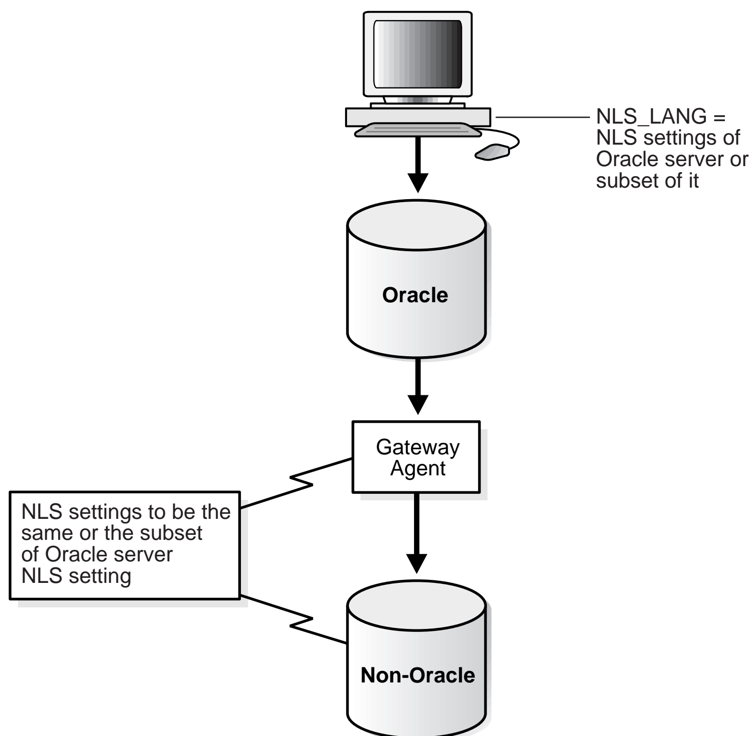
Figure 1–7 NLS Settings in a Homogeneous Environment



Heterogeneous Distributed Environment

In a heterogeneous environment, the NLS settings of the client, the transparent gateway, and the non-Oracle data source should be either the same or a subset of the Oracle database server NLS character set as illustrated in Figure 1–8. Transparent gateways have full NLS support.

Figure 1–8 NLS Settings in a Heterogeneous Environment



In a heterogeneous environment, only transparent gateways built with HS technology support complete NCHAR capabilities. Whether a specific transparent gateway supports NCHAR depends on the non-Oracle data source it is targeting. For further information on how a particular transparent gateway handles NCHAR support, consult the particular transparent gateway documentation.

> **See Also:** *Oracle8i Reference* for more information about National Language Support features.

2

Managing a Distributed Database

This chapter describes how to manage and maintain a distributed database system. Topics include:

- Managing Global Names in a Distributed System
- Creating Database Links
- Creating Shared Database Links
- Managing Database Links
- Viewing Information About Database Links
- Creating Location Transparency
- Managing Statement Transparency
- Managing a Distributed Database: Scenarios

Managing Global Names in a Distributed System

In a distributed database system, each database should have a unique *global database name*. Global database names uniquely identify a database in the system. A primary administration task in a distributed system is managing the creation and alteration of global database names.

This section contains the following topics:

- Understanding How Global Database Names Are Formed
- Determining Whether Global Naming Is Enforced
- Viewing a Global Database Name
- Changing the Domain in a Global Database Name
- Changing a Global Database Name: Scenario

Understanding How Global Database Names Are Formed

A global database name is formed from two components: a database name and a domain. The database name and the domain name are determined by the following initialization parameters at database creation:

Component	Parameter	Requirements	Example
Database name	DB_NAME	Must be eight characters or less.	SALES
Domain containing the database	DB_DOMAIN	Must follow standard Internet conventions. Levels in domain names must be separated by dots and the order of domain names is from leaf to root, left to right.	US.ACME.COM

These are examples of valid global database names:

DB_NAME	DB_DOMAIN	Global Database Name
SALES	AU.ORACLE.COM	SALES.AU.ORACLE.COM
SALES	US.ORACLE.COM	SALES.US.ORACLE.COM
MKTG	US.ORACLE.COM	MKTG.US.ORACLE.COM
PAYROLL	NONPROFIT.ORG	PAYROLL.NONPROFIT.ORG

The DB_DOMAIN initialization parameter is only important at database creation time when it is used, together with the DB_NAME parameter, to form the database's global name. At this point, the database's global name is stored in the data dictionary. You must change the global name using an ALTER DATABASE statement, *not* by altering the DB_DOMAIN parameter in the initialization parameter file. It is good practice, however, to change the DB_DOMAIN parameter to reflect the change in the domain name before the next database startup.

Determining Whether Global Naming Is Enforced

The name that you give to a link on the local database depends on whether the remote database that you want to access enforces global naming. If the remote database enforces global naming, then you must use the remote database's global database name as the name of the link. For example, if you are connected to the local HQ server and want to create a link to the remote MNFG database, and MNFG enforces global naming, then you must use MNFG's global database name as the link name.

You can also use service names as part of the database link name. For example, if you use the service names SN1 and SN2 to connect to database HQ.ACME.COM, and HQ enforces global naming, then you can create the following link names to HQ:

```
HQ.ACME.COM@SN1
HQ.ACME.COM@SN2
```

See Also: "Using Connection Qualifiers to Specify Service Names Within Link Names" on page 2-13 for more information about using services names in link names.

To determine whether global naming on a database is enforced on a database, either examine the database's initialization parameter file or query V$PARAMETER. For example, to see whether global naming is enforced on MNFG, you could start a session on MNFG and then create and execute the following `globalnames.sql` script (sample output included):

```
COL name FORMAT a12
COL value FORMAT a6
SELECT name, value FROM v$parameter
WHERE name = 'global_names'
/
```

```
SQL> @globalnames

NAME         VALUE
------------ ------
global_names FALSE
```

Viewing a Global Database Name

Use the data dictionary view GLOBAL_NAME to view the database's global name. For example, issue the following:

```
SELECT * FROM global_name;

GLOBAL_NAME
-------------------------------------------------------------------------------
SALES.AU.ORACLE.COM
```

Changing the Domain in a Global Database Name

Use the ALTER DATABASE statement to change the domain in a database's global name. Note that after the database is created, changing the initialization parameter DB_DOMAIN has no effect on the global database name or on the resolution of database link names.

The following example shows the syntax for the renaming statement, where *database* is a database name and *domain* is the network domain:

```
ALTER DATABASE RENAME GLOBAL_NAME database.domain;
```

To change the domain in a global database name:

1. Determine the current global database name. For example, issue:

   ```
   SELECT * FROM global_name;

   GLOBAL_NAME
   ----------------------------------------------------------------------------
   SALES.AU.ORACLE.COM
   ```

2. Rename the global database name using an ALTER DATABASE statement. For example, enter:

   ```
   ALTER DATABASE RENAME GLOBAL_NAME sales.us.oracle.com;
   ```

3. Query the GLOBAL_NAME table to check the new name. For example, enter:

```
SELECT * FROM global_name;

GLOBAL_NAME
----------------------------------------------------------------------------
SALES.US.ORACLE.COM
```

Changing a Global Database Name: Scenario

In this scenario, you change the domain part of the global database name of the local database. You also create database links using partially-specified global names to test how Oracle resolves the names. You discover that Oracle resolves the partial names using the domain part of the current global database name of the local database, not the value for the initialization parameter DB_DOMAIN.

1. You connect to SALES.US.ACME.COM and query the GLOBAL_NAME data dictionary view to determine the database's current global name:

   ```
   CONNECT sys/sys_pwd@sales.us.acme.com
   SELECT * FROM global_name;

   GLOBAL_NAME
   ----------------------------------------------------------------------------
   SALES.US.ACME.COM
   ```

2. You query V$PARAMETER to determine the current setting for the DB_DOMAIN initialization parameter:

   ```
   SELECT name, value FROM v$parameter WHERE name = 'db_domain';

   NAME       VALUE
   ---------  -----------
   db_domain  US.ACME.COM
   ```

3. You then create a database link to a database called HQ, using only a partially-specified global name:

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

   Oracle expands the global database name for this link by appending the domain part of the global database name of the *local* database to the name of the database specified in the link.

4. You query USER_DB_LINKS to determine which domain name Oracle uses to resolve the partially specified global database name:

   ```
   SELECT db_link FROM user_db_links;
   ```

```
DB_LINK
------------------
HQ.US.ACME.COM
```

This result indicates that the domain part of the global database name of the local database is US.ACME.COM. Oracle uses this domain in resolving partial database link names when the database link is created.

5. Because you have received word that the SALES database will move to Japan, you rename the SALES database to SALES.JP.ACME.COM:

```
ALTER DATABASE RENAME GLOBAL_NAME TO sales.jp.acme.com;
SELECT * FROM global_name;

GLOBAL_NAME
----------------------------------------------------------------------------
SALES.JP.ACME.COM
```

6. You query V$PARAMETER again and discover that the value of DB_DOMAIN is *not* changed, despite the fact that you renamed the domain part of the global database name:

```
SELECT name, value FROM v$parameter
  WHERE name = 'db_domain';

NAME       VALUE
---------  -----------
db_domain  US.ACME.COM
```

This result indicates that the value of the DB_DOMAIN parameter is independent of the ALTER DATABASE RENAME GLOBAL_NAME statement. The ALTER DATABASE statement determines the domain of the global database name, not the DB_DOMAIN initialization parameter (although it is good practice to alter DB_DOMAIN to reflect the new domain name).

7. You create another database link to database SUPPLY, and then query USER_DB_LINKS to see how Oracle resolves the domain part of SUPPLY's global database name:

```
CREATE DATABASE LINK supply USING 'supply';
SELECT db_link FROM user_db_links;

DB_LINK
------------------
HQ.US.ACME.COM
SUPPLY.JP.ACME.COM
```

This result indicates that Oracle resolves the partially specified link name by using the domain JP.ACME.COM. This domain is used when the link is created because it is the domain part of the global database name of the local database. Oracle does *not* use the DB_DOMAIN parameter setting when resolving the partial link name.

8. You then receive word that your previous information was faulty: SALES will be in the ASIA.JP.ACME.COM domain, not the JP.ACME.COM domain. Consequently, you rename the global database name as follows:

```
ALTER DATABASE RENAME GLOBAL_NAME TO sales.asia.jp.acme.com;
SELECT * FROM global_name;

GLOBAL_NAME
----------------------------------------------------------------------------
SALES.ASIA.JP.ACME.COM
```

9. You query V$PARAMETER to again check the setting for the parameter DB_DOMAIN:

```
SELECT name, value FROM v$parameter
  WHERE name = 'db_domain';

NAME       VALUE
---------- -----------
db_domain  US.ACME.COM
```

The result indicates that the domain setting in the parameter file is exactly the same as it was before you issued *either* of the ALTER DATABASE RENAME statements.

10. Finally, you create a link to the WAREHOUSE database and again query USER_DB_LINKS to determine how Oracle resolves the partially-specified global name:

```
CREATE DATABASE LINK warehouse USING 'warehouse';
SELECT db_link FROM user_db_links;

DB_LINK
------------------
HQ.US.ACME.COM
SUPPLY.JP.ACME.COM
WAREHOUSE.ASIA.JP.ACME.COM
```

Again, you see that Oracle uses the domain part of the global database name of the local database to expand the partial link name during link creation.

> **See Also:** *Oracle8i Reference* for more information about specifying the DB_NAME and DB_DOMAIN initialization parameters.

Creating Database Links

To support application access to the data and schema objects throughout a distributed database system, you must create all necessary database links. This section contains the following topics:

- Obtaining Privileges Necessary for Creating Database Links
- Specifying Link Types
- Specifying Link Users
- Using Connection Qualifiers to Specify Service Names Within Link Names

Obtaining Privileges Necessary for Creating Database Links

A database link is a pointer in the local database that allows you to access objects on a remote database. To create a private database link, you must have been granted the proper privileges. The following table illustrates which privileges are required on which database for which type of link:

Privilege	Database	Required For
CREATE DATABASE LINK	Local	Creation of a private database link.
CREATE PUBLIC DATABASE LINK	Local	Creation of a public database link.
CREATE SESSION	Remote	Creation of any type of database link.

To see which privileges you currently have available, query ROLE_SYS_PRIVS. For example, you could create and execute the following `privs.sql` script (sample output included):

```
SELECT DISTINCT privilege AS "Database Link Privileges"
FROM role_sys_privs
WHERE privilege IN ( 'CREATE SESSION','CREATE DATABASE LINK',
                     'CREATE PUBLIC DATABASE LINK')
/
```

```
SQL> @privs

Database Link Privileges
----------------------------------------
CREATE DATABASE LINK
CREATE PUBLIC DATABASE LINK
CREATE SESSION
```

Specifying Link Types

When you create a database link, you must decide who will have access to it. The following sections describe how to create the three basic types of links:

- Creating Private Database Links
- Creating Public Database Links
- Creating Global Database Links

Creating Private Database Links

To create a private database link, specify the following (where *link_name* is the global database name or an arbitrary link name):

```
CREATE DATABASE LINK link_name ...;
```

Following are examples of private database links:

This SQL Statement...	Creates...
`CREATE DATABASE LINK supply.us.acme.com;`	A private link using the global database name to the remote SUPPLY database. The link uses the userid/password of the connected user. So if SCOTT (identified by TIGER) uses the link in a query, the link establishes a connection to the remote database as SCOTT/TIGER.
`CREATE DATABASE LINK link_2 CONNECT TO jane IDENTIFIED BY doe USING 'us_supply';`	A private fixed user link called LINK_2 to the database with service name US_SUPPLY. The link connects to the remote database with the userid/password of JANE/DOE regardless of the connected user.

This SQL Statement...	Creates...
`CREATE DATABASE LINK link_1 CONNECT TO CURRENT_USER USING 'us_supply';`	A private link called LINK_1 to the database with service name US_SUPPLY. The link uses the userid/password of the current user to log onto the remote database. **Note:** The current user may not be the same as the connected user, and must be a global user on both databases involved in the link (see "Users of Database Links" on page 1-17). Current user links are part of the Oracle Advanced Security option.

See Also: *Oracle8i SQL Reference* for CREATE DATABASE LINK syntax.

Creating Public Database Links

To create a public database link, use the keyword PUBLIC (where *link_name* is the global database name or an arbitrary link name):

```
CREATE PUBLIC DATABASE LINK link_name ...;
```

Following are examples of public database links:

This SQL Statement...	Creates...
`CREATE PUBLIC DATABASE LINK supply.us.acme.com;`	A public link to the remote SUPPLY database. The link uses the userid/password of the connected user. So if SCOTT (identified by TIGER) uses the link in a query, the link establishes a connection to the remote database as SCOTT/TIGER.
`CREATE PUBLIC DATABASE LINK pu_link CONNECT TO CURRENT_USER USING 'supply';`	A public link called PU_LINK to the database with service name SUPPLY. The link uses the userid/password of the current user to log onto the remote database. **Note:** The current user may not be the same as the connected user, and must be a global user on both databases involved in the link (see "Users of Database Links" on page 1-17).
`CREATE PUBLIC DATABASE LINK sales.us.acme.com CONNECT TO jane IDENTIFIED BY doe;`	A public fixed user link to the remote SALES database. The link connects to the remote database with the userid/password of JANE/DOE.

See Also: *Oracle8i SQL Reference* for CREATE PUBLIC DATABASE LINK syntax.

Creating Global Database Links

You must define *global database links* in the Oracle Names Server. See the *Net8 Administrator's Guide* to learn how to create global database links.

Specifying Link Users

A database link defines a communication path from one database to another. When an application uses a database link to access a remote database, Oracle establishes a database session in the remote database on behalf of the local application request.

When you create a private or public database link, you can determine which schema on the remote database the link will establish connections to by creating fixed user, current user, and connected user database links.

Creating Fixed User Database Links

To create a *fixed user database link*, you embed the credentials (in this case, a username and password) required to access the remote database in the definition of the link:

```
CREATE DATABASE LINK ... CONNECT TO username IDENTIFIED BY password ...;
```

Following are examples of fixed user database links:

This SQL Statement...	Creates...
`CREATE PUBLIC DATABASE LINK supply.us.acme.com CONNECT TO scott AS tiger;`	A public link using the global database name to the remote SUPPLY database. The link connects to the remote database with the userid/password SCOTT/TIGER.
`CREATE DATABASE LINK foo CONNECT TO jane IDENTIFIED BY doe USING 'finance';`	A private fixed user link called FOO to the database with service name FINANCE. The link connects to the remote database with the userid/password JANE/DOE.

When an application uses a fixed user database link, the local server always establishes a connection to a fixed remote schema in the remote database. The local server also sends the fixed user's credentials across the network when an application uses the link to access the remote database.

Creating Connected User and Current User Database Links

Connected user and *current user* database links do not include credentials in the definition of the link. The credentials used to connect to the remote database can change depending on the user that references the database link and the operation performed by the application.

Note: For many distributed applications, you do not want a user to have privileges in a remote database. One simple way to achieve this result is to create a procedure that contains a fixed user or current user database link within it. In this way, the user accessing the procedure temporarily assumes someone else's privileges.

For an extended conceptual discussion of the distinction between connected users and current users, see "Users of Database Links" on page 1-17.

Creating a Connected User Database Link To create a connected user database link, omit the CONNECT TO clause. The following syntax creates a connected user database link, where *dblink* is the name of the link and *net_service_name* is an optional connect string:

```
CREATE [SHARED] [PUBLIC] DATABASE LINK dblink ... [USING 'net_service_name'];
```

For example, to create a connected user database link, use the following syntax:

```
CREATE DATABASE LINK sales.division3.acme.com USING 'sales';
```

Creating a Current User Database Link To create a current user database link, use the CONNECT TO CURRENT_USER clause in the link creation statement. Current user links are only available through the Oracle Advanced Security option.

The following syntax creates a current user database link, where *dblink* is the name of the link and *net_service_name* is an optional connect string:

```
CREATE [SHARED] [PUBLIC] DATABASE LINK dblink CONNECT TO CURRENT_USER
[USING 'net_service_name'];
```

For example, to create a connected user database link to the SALES database, you might use the following syntax:

```
CREATE DATABASE LINK sales CONNECT TO CURRENT_USER USING 'sales';
```

Note: To use a current user database link, the current user must be a global user on both databases involved in the link.

See Also: *Oracle8i SQL Reference* for more syntax information about creating database links.

Using Connection Qualifiers to Specify Service Names Within Link Names

In some situations, you may want to have several database links of the same type (for example, public) that point to the same remote database, yet establish connections to the remote database using different communication pathways. Some cases in which this strategy is useful are:

- A remote database is configured using the Oracle Parallel Server, so you define several public database links at your local node so that connections can be established to specific instances of the remote database.
- Some clients connect to the Oracle server using TCP/IP while others use DECNET.

To facilitate such functionality, Oracle allows you to create a database link with an optional service name in the database link name. When creating a database link, a service name is specified as the trailing portion of the database link name, separated by an `@` sign, as in `@sales`. This string is called a *connection qualifier*.

For example, assume that remote database HQ.ACME.COM is managed by the Oracle Parallel Server. The HQ database has two instances named HQ_1 and HQ_2. The local database can contain the following public database links to define pathways to the remote instances of the HQ database:

```
CREATE PUBLIC DATABASE LINK hq.acme.com@hq_1
  USING 'string_to_hq_1';
CREATE PUBLIC DATABASE LINK hq.acme.com@hq_2
  USING 'string_to_hq_2';
CREATE PUBLIC DATABASE LINK hq.acme.com
  USING 'string_to_hq';
```

Notice in the first two examples that a service name is simply a part of the database link name. The text of the service name does not necessarily indicate how a connection is to be established; this information is specified in the service name of the USING clause. Also notice that in the third example, a service name is not specified as part of the link name. In this case, just as when a service name is specified as part of the link name, the instance is determined by the USING string.

To use a service name to specify a particular instance, include the service name at the end of the global object name:

```
SELECT * FROM scott.emp@hq.acme.com@hq_1
```

Note that in this example, there are two @ symbols.

Creating Shared Database Links

Every application that references a remote server using a standard database link establishes a connection between the local database and the remote database. Many users running applications simultaneously can cause a high number of connections between the local and remote databases.

Shared database links enable you to limit the number of network connections required between the local server and the remote server.

This section contains the following topics:

- Determining Whether to Use Shared Database Links
- Creating Shared Database Links
- Configuring Shared Database Links

See Also: "What Are Shared Database Links?" on page 1-12 for a conceptual overview of shared database links.

Determining Whether to Use Shared Database Links

Look carefully at your application and multi-threaded server configuration to determine whether to use shared links. A simple guideline is to use shared database links when the number of users accessing a database link is expected to be much larger than the number of server processes in the local database.

The following table illustrates three possible configurations involving database links:

Link Type	Server Mode	Consequences
Non-Shared	Dedicated/MTS	If your application uses a standard public database link, and 100 users simultaneously require a connection, then 100 direct network connections to the remote database are required.
Shared	MTS	If 10 shared server processes exist in the local MTS-mode database, then 100 users that use the same database link require 10 or fewer network connections to the remote server. Each local shared server process may only need one connection to the remote server.

Link Type	Server Mode	Consequences
Shared	Dedicated	If 10 clients connect to a local dedicated server, and each client has 10 sessions on the same connection (thus establishing 100 sessions overall), and each session references the same remote database, then only 10 connections are needed. With a non-shared database link, 100 connections are needed.

Shared database links are not useful in all situations. Assume that only one user accesses the remote server. If this user defines a shared database link and 10 shared server processes exist in the local database, then this user can require up to 10 network connections to the remote server. Because the user can use each shared server process, each process can establish a connection to the remote server.

Clearly, a non-shared database link is preferable in this situation because it requires only one network connection. Shared database links lead to more network connections in single-user scenarios, so use shared links only when many users need to use the same link. Typically, shared links are used for public database links, but can also be used for private database links when many clients access the same local schema (and therefore the same private database link).

Creating Shared Database Links

To create a shared database link, use the keyword SHARED in the CREATE DATABASE LINK statement:

```
CREATE SHARED DATABASE LINK dblink_name
[CONNECT TO username IDENTIFIED BY password]|[CONNECT TO CURRENT_USER]
AUTHENTICATED BY schema_name IDENTIFIED BY password
[USING 'service_name'];
```

The following example creates a fixed user, shared link to database SALES, connecting as SCOTT and authenticated as JANE:

```
CREATE SHARED DATABASE LINK link2sales
CONNECT TO scott IDENTIFIED BY tiger
AUTHENTICATED BY keith IDENTIFIED BY richards
USING 'sales';
```

Whenever you use the keyword SHARED, the clause AUTHENTICATED BY is required. The schema specified in the AUTHENTICATED BY clause is only used for security reasons and can be considered a dummy schema. It is not affected when using shared database links, nor does it affect the users of the shared database link.

The AUTHENTICATED BY clause is required to prevent unauthorized clients from masquerading as a database link user and gaining access to privileges information.

See Also: *Oracle8i SQL Reference* for information about the CREATE DATABASE LINK statement.

Configuring Shared Database Links

You can configure shared database links in the following ways:

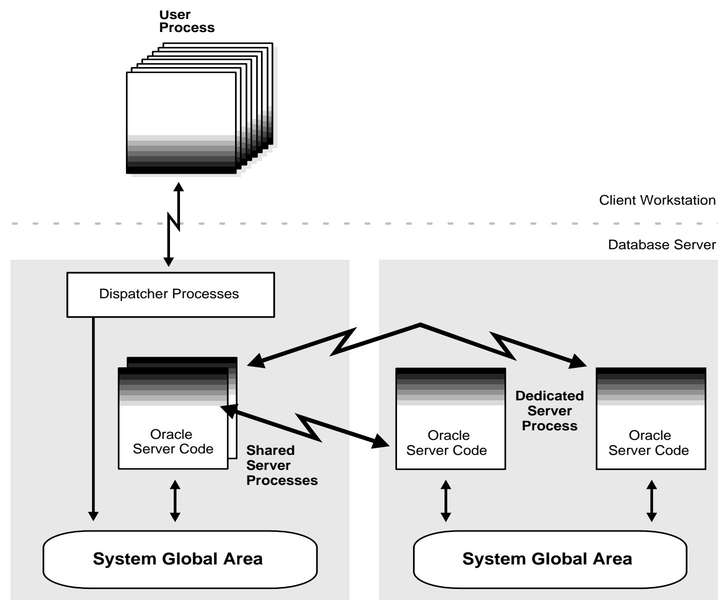
- Creating Shared Links to Dedicated Servers
- Creating Shared Links to Multi-Threaded Servers

Creating Shared Links to Dedicated Servers

In the configuration illustrated in Figure 2–1, a shared server process in the local server owns a dedicated remote server process. The advantage is that a direct network transport exists between the local shared server and the remote dedicated server. A disadvantage is that extra back-end server processes are needed.

Note: The remote server can either be a multi-threaded server or dedicated server. There is a dedicated connection between the local and remote servers. When the remote server is a multi-threaded server, you can force a dedicated server connection by using the (SERVER=DEDICATED) clause in the definition of the service name.

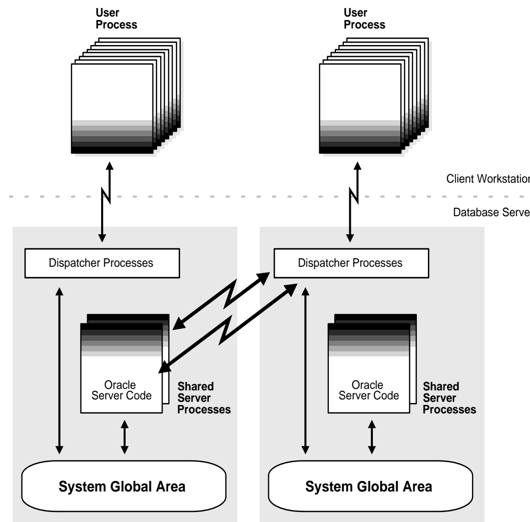
Figure 2–1 A Shared Database Link to Dedicated Server Processes



Creating Shared Links to Multi-Threaded Servers

The configuration illustrated in Figure 2–2 uses shared server processes on the remote server. This configuration eliminates the need for more dedicated servers, but requires the connection to go through the dispatcher on the remote server. Note that both the local and the remote server must be configured as multi-threaded servers.

Figure 2–2 Shared Database Link to Multi-Threaded Server



> **See Also:** *Net8 Administrator's Guide* for information about the multi-threaded server option.

Managing Database Links

This section contains the following topics:

- Closing Database Links
- Dropping Database Links
- Limiting the Number of Active Database Link Connections

Closing Database Links

If you access a database link in a session, then the link remains open until you close the session. A link is open in the sense that a process is active on each of the remote databases accessed through the link. This situation has the following consequences:

- If 20 users open sessions and access the same public link in a local database, then 20 database link connections are open.
- If 20 users open sessions and each user accesses a private link, then 20 database link connections are open.

- If one user starts a session and accesses 20 different links, then 20 database link connections are open.

After you close a session, the links that were active in the session are automatically closed. You may have occasion to close the link manually. For example, close links when:

- The network connection established by a link is used infrequently in an application.
- The user session must be terminated.

If you want to close a link, issue the following statement, where *linkname* refers to the name of the link:

```
ALTER SESSION CLOSE DATABASE LINK linkname;
```

Note that this statement only closes the links that are active in your current session.

Dropping Database Links

You can drop a database link just as you can drop a table or view. If the link is private, then it must be in your schema. If the link is public, then you must have the DROP PUBLIC DATABASE LINK system privilege.

The command syntax is as follows, where *dblink* is the name of the link:

```
DROP [PUBLIC] DATABASE LINK dblink;
```

To drop a private database link:

1. Connect to the local database using SQL*Plus. For example, enter:

   ```
   CONNECT scott/tiger@local_db
   ```

2. Query USER_DB_LINKS to view the links that you own. For example, enter:

   ```
   SELECT db_link FROM user_db_links;

   DB_LINK
   -----------------------------------
   SALES.US.ORACLE.COM
   MKTG.US.ORACLE.COM
   2 rows selected.
   ```

3. Drop the desired link using the DROP DATABASE LINK statement. For example, enter:

```
DROP DATABASE LINK sales.us.oracle.com;
```

To drop a public database link:

1. Connect to the local database as a user with the DROP PUBLIC DATABASE LINK privilege. For example, enter:

```
CONNECT sys/change_on_install@local_db AS SYSDBA
```

2. Query DBA_DB_LINKS to view the public links. For example, enter:

```
SELECT db_link FROM user_db_links
  WHERE owner = 'PUBLIC';

DB_LINK
-----------------------------------
DBL1.US.ORACLE.COM
SALES.US.ORACLE.COM
INST2.US.ORACLE.COM
RMAN2.US.ORACLE.COM
4 rows selected.
```

3. Drop the desired link using the DROP PUBLIC DATABASE LINK statement. For example, enter:

```
DROP PUBLIC DATABASE LINK sales.us.oracle.com;
```

Limiting the Number of Active Database Link Connections

You can limit the number of connections from a user process to remote databases using the static initialization parameter OPEN_LINKS. This parameter controls the number of remote connections that a single user session can use concurrently in distributed transactions.

Note the following considerations for setting this parameter:

- The value should be greater than or equal to the number of databases referred to in a single SQL statement that references multiple databases.
- Increase the value if several distributed databases are accessed over time. Thus, if you regularly access three database, set OPEN_LINKS to 3 or greater.
- The default value for OPEN_LINKS is 4. If OPEN_LINKS is set to 0, then no distributed transactions are allowed.

> **See Also:** *Oracle8i SQL Reference* for more information about OPEN_LINKS.

Viewing Information About Database Links

The data dictionary of each database stores the definitions of all the database links in the database. You can use data dictionary tables and views to gain information about the links. This section contains the following topics:

- Determining Which Links Are in the Database
- Determining Which Link Connections Are Open

Determining Which Links Are in the Database

The following views show the database links that have been defined at the local database and stored in the data dictionary:

View	Purpose
DBA_DB_LINKS	Lists all database links in the database.
ALL_DB_LINKS	Lists all database links accessible to the connected user.
USER_DB_LINKS	Lists all database links owned by the connected user.

These data dictionary views contain the same basic information about database links, with some exceptions:

Column	Which Views?	Description
OWNER	All except USER_*	The user who created the database link. If the link is public, then the user is listed as PUBLIC.
DB_LINK	All	The name of the database link.
USERNAME	All	If the link definition includes a fixed user, then this column displays the username of the fixed user. If there is no fixed user, the column is NULL.
PASSWORD	Only USER_*	The password for logging into the remote database.
HOST	All	The net service name used to connect to the remote database.
CREATED	All	Creation time of the database link.

Any user can query USER_DB_LINKS to determine which database links are available to that user. If you have the DBA privilege, then you can access information about all links in the database. For example, you can create and run the following script access link information (sample output is below):

```
COL owner FORMAT a10
COL username FORMAT a8 HEADING "USER"
COL db_link FORMAT a30
COL host FORMAT a7 HEADING "SERVICE"
SELECT * FROM dba_db_links
/

SQL>@link_script

OWNER      DB_LINK                        USER     SERVICE CREATED
---------- ------------------------------ -------- ------- ----------
SYS        TARGET.US.ACME.COM             SYS      inst1   23-JUN-99
PUBLIC     DBL1.UK.ACME.COM               BLAKE    ora51   23-JUN-99
PUBLIC     RMAN2.US.ACME.COM                       inst2   23-JUN-99
PUBLIC     DEPT.US.ACME.COM                        inst2   23-JUN-99
JANE       DBL.UK.ACME.COM                BLAKE    ora51   23-JUN-99
SCOTT      EMP.US.ACME.COM                SCOTT    inst2   23-JUN-99
6 rows selected.
```

Viewing Password Information

Only USER_DB_LINKS contains a column for password information. Nevertheless, if you have the DBA role, then you can view passwords for all links in the database by querying the LINK$ table. If you do *not* have the DBA role but have the SELECT ANY TABLE privilege, *and* the initialization parameter O7_DICTIONARY_ ACCESSIBILITY is set to TRUE (default), then you can also access this information.

> **Note:** Oracle corporation recommends that you set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE so that only DBA-privileged connections can see password information in the LINK$ table.

You can create and run the following script in SQL*Plus to obtain password information (sample output included):

```
col userid format a10
col password format a10
SELECT userid,password
  FROM sys.link$
  WHERE password IS NOT NULL
/

SQL>@linkpwd
```

```
USERID     PASSWORD
---------- ----------
SYS        ORACLE
BLAKE      TYGER
SCOTT      TIGER
3 rows selected.
```

Viewing Authentication Passwords If you have the DBA role, then you can view AUTHENTICATED BY ... IDENTIFIED BY ... usernames and passwords for all links in the database by querying the LINK$ table. If you do *not* have the DBA role but have the SELECT ANY TABLE privilege, *and* the initialization parameter O7_DICTIONARY_ACCESSIBILITY is set to TRUE (default), then you can also access this information.

Note: Oracle corporation recommends that you set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE so that only DBA-privileged connections can see password information in the LINK$ table.

You can create and run the following script in SQL*Plus to obtain password information (sample output included):

```
col authusr format a10
col authpwd format a10
SELECT authusr as userid, authpwd as password
  FROM sys.link$
  WHERE password IS NOT NULL
/

SQL> @authpwd

USERID     PASSWORD
---------- ----------
ELLIE      MAY
1 row selected.
```

See Also: *Oracle8i Concepts* or *Oracle8i SQL Reference* for information on viewing the data dictionary.

You can also view the link and password information together in a join by creating and executing the following script (sample output included):

```
COL owner FORMAT a8
COL db_link FORMAT a15
COL username FORMAT a8 HEADING "CON_USER"
COL password FORMAT a8 HEADING "CON_PWD"
COL authusr FORMAT a8 HEADING "AUTH_USER"
COL authpwd format a8 HEADING "AUTH_PWD"
COL host FORMAT a7 HEADING "SERVICE"
COL created FORMAT a10

SELECT DISTINCT d.owner,d.db_link,d.username,l.password,
                l.authusr,l.authpwd,d.host,d.created
FROM dba_db_links d, sys.link$ l
WHERE password IS NOT NULL
AND d.username = l.userid
/

SQL> @user_and_pwd

OWNER    DB_LINK        CON_USER CON_PWD AUTH_USE AUTH_PWD SERVICE CREATED
-------- --------------- -------- -------- -------- -------- ------- ----------
JANE     DBL.ACME.COM    BLAKE    TYGER    ELLIE    MAY      ora51   23-JUN-99
PUBLIC   DBL1.ACME.COM   SCOTT    TIGER                      ora51   23-JUN-99
SYS      TARGET.ACME.COM SYS      ORACLE                     inst1   23-JUN-99
```

Determining Which Link Connections Are Open

You may find it useful to determine which database link connections are currently open in your session. Note that if you connect as SYSDBA, you cannot query a view to determine all the links open for all sessions: you can only access the link information in the session within which you are working.

The following views show the database link connections that are currently open in your current session:

View	Purpose
V$DBLINK	Lists all open database links in your session, that is, all database links with the IN_TRANSACTION column set to YES.
GV$DBLINK	Lists all open database links in your session along with their corresponding instances. This view is useful in an Oracle Parallel Server configuration.

These data dictionary views contain the same basic information about database links, with one exception:

Column	Which Views?	Description
DB_LINK	All	The name of the database link.
OWNER_ID	All	The owner of the database link.
LOGGED_ON	All	Whether the database link is currently logged on.
HETEROGENEOUS	All	Whether the database link is homogeneous (NO) or heterogeneous (YES).
PROTOCOL	All	The communication protocol for the database link.
OPEN_CURSORS	All	Whether cursors are open for the database link.
IN_TRANSACTION	All	Whether the database link is accessed in a transaction that has not yet been committed or rolled back.
UPDATE_SENT	All	Whether there was an update on the database link.
COMMIT_POINT_ STRENGTH	All	The commit point strength of the transactions using the database link.
INST_ID	GV$DBLINK only	The instance from which the view information was obtained.

For example, you can create and execute the script below to determine which links are open (sample output included):

```
COL db_link FORMAT a25
COL owner_id FORMAT 99999 HEADING "OWNID"
COL logged_on FORMAT a5 HEADING "LOGON"
COL heterogeneous FORMAT a5 HEADING "HETER"
COL protocol FORMAT a8
COL open_cursors FORMAT 999 HEADING "OPN_CUR"
COL in_transaction FORMAT a3 HEADING "TXN"
COL update_sent FORMAT a6 HEADING "UPDATE"
COL commit_point_strength FORMAT 99999 HEADING "C_P_S"

SELECT * FROM v$dblink
/

SQL> @dblink
```

```
DB_LINK                   OWNID LOGON HETER PROTOCOL OPN_CUR TXN UPDATE  C_P_S
------------------------- ------ ----- ----- -------- ------- --- ------ ------
INST2.ACME.COM                 0 YES   YES    UNKN           0 YES YES       255
```

Creating Location Transparency

After you have configured the necessary database links, you can use various tools to hide the distributed nature of the database system from users. In other words, users can access remote objects as if they were local objects. The following sections explain how to hide distributed functionality form users:

- Using Views to Create Location Transparency
- Using Synonyms to Create Location Transparency
- Using Procedures to Create Location Transparency

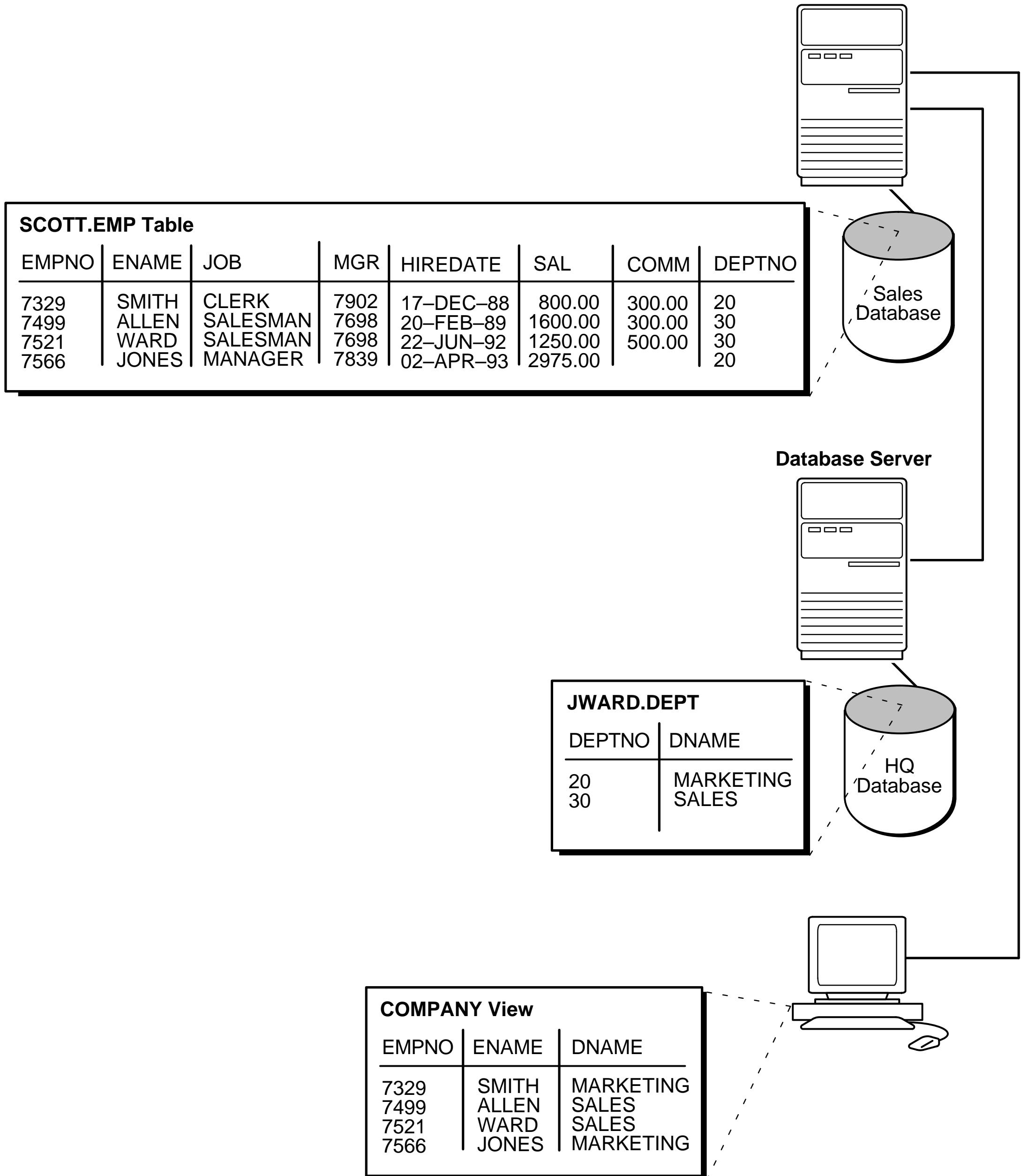
Using Views to Create Location Transparency

Local views can provide location transparency for local and remote tables in a distributed database system.

For example, assume that table EMP is stored in a local database and table DEPT is stored in a remote database. To make these tables transparent to users of the system, you can create a view in the local database that joins local and remote data:

```
CREATE VIEW company
AS
SELECT a.empno, a.ename, b.dname
FROM emp a, dept@hq.acme.com b
WHERE a.deptno = b.deptno;
```

Figure 2–3 Views and Location Transparency



SCOTT.EMP Table

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CLERK	7902	17–DEC–88	800.00	300.00	20
7499	ALLEN	SALESMAN	7698	20–FEB–89	1600.00	300.00	30
7521	WARD	SALESMAN	7698	22–JUN–92	1250.00	500.00	30
7566	JONES	MANAGER	7839	02–APR–93	2975.00		20

JWARD.DEPT

DEPTNO	DNAME
20	MARKETING
30	SALES

COMPANY View

EMPNO	ENAME	DNAME
7329	SMITH	MARKETING
7499	ALLEN	SALES
7521	WARD	SALES
7566	JONES	MARKETING

When users access this view, they do not need to know where the data is physically stored, or if data from more than one table is being accessed. Thus, it is easier for them to get required information. For example, the following query provides data from both the local and remote database table:

```
SELECT * FROM company;
```

Managing Privileges in Views

Assume a local view references a remote table or view. The owner of the local view can grant only those object privileges on her view that have been granted by the remote user. (The remote user is implied by the type of database link). This is similar to privilege management for views that reference local data.

Using Synonyms to Create Location Transparency

Synonyms are useful in both distributed and non-distributed environments because they hide the identity of the underlying object, including its location in a distributed database system. If you must rename or move the underlying object, you only need to redefine the synonym; applications based on the synonym continue to function normally. Synonyms also simplify SQL statements for users in a distributed database system.

Creating Synonyms

You can create synonyms for the following:

- Tables
- Types
- Views
- Snapshots
- Sequences
- Procedures
- Functions
- Packages

All synonyms are schema objects that are stored in the data dictionary of the database in which they are created. To simplify remote table access through database links, a synonym can allow single-word access to remote data, hiding the specific object name and the location from users of the synonym.

The syntax to create a synonym is:

```
CREATE [PUBLIC] synonym_name
FOR [schema.]object_name[@database_link_name]
```

where:

PUBLIC	is a keyword specifying that this synonym is available to all users. Omitting this parameter makes a synonym private, and usable only by the creator. Public synonyms can be created only by a user with CREATE PUBLIC SYNONYM system privilege.
synonym_name	specifies the alternate object name to be referenced by users and applications.
schema	specifies the schema of the object specified in *object_name*. Omitting this parameter uses the creator's schema as the schema of the object.
object_name	specifies either a table, view, sequence, snapshot, type, procedure, function or package as appropriate.
database_link_name	specifies the database link identifying the remote database and schema in which the object specified in object_name is located.

A synonym must be a uniquely named object for its schema. If a schema contains a schema object and a public synonym exists with the same name, then Oracle always finds the schema object when the user that owns the schema references that name.

Example: Creating a Public Synonym Assume that in every database in a distributed database system, a public synonym is defined for the SCOTT.EMP table stored in the HQ database:

```
CREATE PUBLIC SYNONYM emp FOR scott.emp@hq.acme.com;
```

You can design an employee management application without regard to where the application is used because the location of the table SCOTT.EMP@HQ.ACME.COM is hidden by the public synonyms. SQL statements in the application access the table by referencing the public synonym EMP.

Furthermore, if you move the EMP table from the HQ database to the HR database, then you only need to change the public synonyms on the nodes of the system. The employee management application continues to function properly on all nodes.

Managing Privileges and Synonyms

A synonym is a reference to an actual object. A user who has access to a synonym for a particular schema object must also have privileges on the underlying schema object itself. For example, if the user attempts to access a synonym but does not have privileges on the table it identifies, an error occurs indicating that the table or view does not exist.

Assume SCOTT creates local synonym EMP as an alias for remote object SCOTT.EMP@SALES.ACME.COM. SCOTT *cannot* grant object privileges on the synonym to another local user. SCOTT cannot grant local privileges for the synonym because this operation amounts to granting privileges for the remote EMP table on the SALES database, which is not allowed. This behavior is different from privilege management for synonyms that are aliases for local tables or views.

Therefore, you cannot manage local privileges when synonyms are used for location transparency. Security for the base object is controlled entirely at the remote node. For example, user ADMIN cannot grant object privileges for the EMP_SYN synonym.

Unlike a database link referenced in a view or procedure definition, a database link referenced in a synonym is resolved by first looking for a private link owned by the schema in effect at the time the reference to the synonym is parsed. Therefore, to ensure the desired object resolution, it is especially important to specify the underlying object's schema in the definition of a synonym.

Using Procedures to Create Location Transparency

PL/SQL program units called *procedures* can provide location transparency. You have these options:

- Using Local Procedures to Reference Remote Data
- Using Local Procedures to Call Remote Procedures
- Using Local Synonyms to Reference Remote Procedures

Using Local Procedures to Reference Remote Data

Procedures or functions (either stand-alone or in packages) can contain SQL statements that reference remote data. For example, consider the procedure created by the following statement:

```
CREATE PROCEDURE fire_emp (enum NUMBER) AS
BEGIN
  DELETE FROM emp@hq.acme.com
```

```
  WHERE empno = enum;
END;
```

When a user or application calls the FIRE_EMP procedure, it is not apparent that a remote table is being modified.

A second layer of location transparency is possible when the statements in a procedure indirectly reference remote data using local procedures, views, or synonyms. For example, the following statement defines a local synonym:

```
CREATE SYNONYM emp FOR emp@hq.acme.com;
```

Given this synonym, you can create the FIRE_EMP procedure using the following statement:

```
CREATE PROCEDURE fire_emp (enum NUMBER) AS
BEGIN
  DELETE FROM emp WHERE empno = enum;
END;
```

If you rename or move the table EMP@HQ, then you only need to modify the local synonym that references the table. None of the procedures and applications that call the procedure require modification.

Using Local Procedures to Call Remote Procedures

You can use a local procedure to call a remote procedure. The remote procedure can then execute the required DML. For example, assume that SCOTT connects to LOCAL_DB and creates the following procedure:

```
CONNECT scott/tiger@local_db

CREATE PROCEDURE fire_emp (enum NUMBER)
AS
BEGIN
  EXECUTE term_emp@hq.acme.com;
END;
```

Now, assume that SCOTT connects to the remote database and creates the remote procedure:

```
CONNECT scott/tiger@hq.acme.com

CREATE PROCEDURE term_emp (enum NUMBER)
AS
BEGIN
```

```
  DELETE FROM emp WHERE empno = enum;
END;
```

When a user or application connected to LOCAL_DB calls the FIRE_EMP procedure, this procedure in turn calls the remote TERM_EMP procedure on HQ.ACME.COM.

Using Local Synonyms to Reference Remote Procedures

For example, SCOTT connects to the local SALES.ACME.COM database and creates the following procedure:

```
CREATE PROCEDURE fire_emp (enum NUMBER) AS
BEGIN
DELETE FROM emp@hq.acme.com
WHERE empno = enum;
END;
```

PEGGY then connects to the SUPPLY.ACME.COM database and creates the following synonym for the procedure that SCOTT created on the remote SALES database:

```
SQL> CONNECT peggy/hill@supply
SQL> CREATE PUBLIC SYNONYM emp FOR scott.fire_emp@sales.acme.com;
```

A local user on SUPPLY can use this synonym to execute the procedure on SALES.

Managing Procedures and Privileges

Assume a local procedure includes a statement that references a remote table or view. The owner of the local procedure can grant the EXECUTE privilege to any user, thereby giving that user the ability to execute the procedure and, indirectly, access remote data.

In general, procedures aid in security. Privileges for objects referenced within a procedure do not need to be explicitly granted to the calling users.

Managing Statement Transparency

Oracle allows the following standard DML statements to reference remote tables:

- SELECT (queries)
- INSERT
- UPDATE

- DELETE
- SELECT ... FOR UPDATE (not always supported in Heterogeneous Systems)
- LOCK TABLE

Queries including joins, aggregates, subqueries, and SELECT ... FOR UPDATE can reference any number of local and remote tables and views. For example, the following query joins information from two remote tables:

```
SELECT e.empno, e.ename, d.dname
  FROM scott.emp@sales.division3.acme.com e, jward.dept@hq.acme.com d
  WHERE e.deptno = d.deptno;
```

UPDATE, INSERT, DELETE, and LOCK TABLE statements can reference both local and remote tables. No programming is necessary to update remote data. For example, the following statement inserts new rows into the remote table EMP in the SCOTT.SALES schema by selecting rows from the EMP table in the JWARD schema in the local database:

```
INSERT INTO scott.emp@sales.division3.acme.com
  SELECT * FROM jward.emp;
```

Understanding Transparency Restrictions

Several restrictions apply to statement transparency:

- Within a single SQL statement, all referenced LONG and LONG RAW columns, sequences, updated tables, and locked tables must be located at the same node.
- Oracle does not allow remote DDL statements (for example, CREATE, ALTER, and DROP) in homogeneous systems except through remote execution of members of the DBMS_SQL package, as in this example:

  ```
  DBMS_SQL.PARSE@link_name(crs, 'drop table emp', v7);
  ```

  Note that in Heterogeneous Systems, a pass-through facility allows you to execute DDL.
- The LIST CHAINED ROWS clause of an ANALYZE statement cannot reference remote tables.
- In a distributed database system, Oracle always evaluates environmentally-dependent SQL functions such as SYSDATE, USER, UID, and USERENV with respect to the local server, no matter where the statement (or portion of a statement) executes.

Note: Oracle supports the USERENV function for queries only.

- A number of performance restrictions relate to access of remote objects:
  - Remote views do not have statistical data.
  - Queries on partitioned tables may not be optimized.
  - No more than 20 indexes are considered for a remote table.
  - No more than 20 columns are used for a composite index.

  See Also: *Oracle8i Supplied PL/SQL Packages Reference* for more information about the DBMS_SQL package.

- There is a restriction in Oracle's implementation of distributed read consistency that can cause one node to be in the past with respect to another node. In accordance with read consistency, a query may end up retrieving consistent, but out-of-date data. See "Managing Read Consistency" on page 5-28 to learn how to manage this problem.

Managing a Distributed Database: Scenarios

This section gives examples of various types of statements involving management of database links:

- Creating a Public Fixed User Database Link
- Creating a Public Fixed User Shared Database Link
- Creating a Public Connected User Database Link
- Creating a Public Connected User Shared Database Link
- Creating a Public Current User Database Link

Creating a Public Fixed User Database Link

The following example connects to the local database as JANE and creates a public fixed user database link to database SALES for SCOTT. The database is accessed through its net service name SLDB:

```
CONNECT jane/doe@local

CREATE PUBLIC DATABASE LINK sales.division3.acme.com
```

```
  CONNECT TO SCOTT IDENTIFIED BY TIGER
  USING 'sldb';
```

Consequences

Any user connected to the local database can use the SALES.DIVISION3.ACME.COM database link to connect to the remote database. Each user connects to the remote schema SCOTT in the remote database.

To access the table EMP table in SCOTT's remote schema, a user can issue the following SQL query:

```
SELECT * FROM emp@sales.division3.acme.com;
```

Note that each application or user session creates a separate connection to the common account on the server. The connection to the remote database remains open for the duration of the application or user session.

Creating a Public Fixed User Shared Database Link

The following examples connects to the local database as DANA and creates a public link to the SALES database (using its net service name SLDB). The link allows a connection to the remote database as SCOTT and authenticates this user as SCOTT:

```
CONNECT dana/sculley@local

CREATE SHARED PUBLIC DATABASE LINK sales.division3.acme.com
  CONNECT TO scott IDENTIFIED BY tiger
  AUTHENTICATED BY scott IDENTIFIED BY tiger
  USING 'sldb';
```

Consequences

Any user connected to the local MTS-mode server can use this database link to connect to the remote SALES database through a shared server process. The user can then query tables in the SCOTT schema.

In the above example, each local shared server can establish one connection to the remote server. Whenever a local shared server process needs to access the remote server through the SALES.DIVISION3.ACME.COM database link, the local shared server process reuses established network connections.

Creating a Public Connected User Database Link

The following example connects to the local database as LARRY and creates a public link to the database with the net service name SLDB:

```
CONNECT larry/oracle@local

CREATE PUBLIC DATABASE LINK redwood
  USING 'sldb';
```

Consequences

Any user connected to the local database can use the REDWOOD database link. The connected user in the local database who uses the database link determines the remote schema.

If SCOTT is the connected user and uses the database link, then the database link connects to the remote schema SCOTT. If FOX is the connected user and uses the database link, then the database link connects to remote schema FOX.

The following statement fails for local user FOX in the local database when the remote schema FOX cannot resolve the EMP schema object. That is, if the FOX schema in the SALES.DIVISION3.ACME.COM does not have EMP as a table, view, or (public) synonym, an error will be returned.

```
CONNECT fox/mulder@local

SELECT * FROM emp@redwood;
```

Creating a Public Connected User Shared Database Link

The following example connects to the local database as NEIL and creates a shared, public link to the SALES database (using its net service name SLDB). The user is authenticated by the userid/password of CRAZY/HORSE. The following statement creates a public, connected user, shared database link:

```
CONNECT neil/young@local

CREATE SHARED PUBLIC DATABASE LINK sales.division3.acme.com
  AUTHENTICATED BY crazy IDENTIFIED BY horse
  USING 'sldb';
```

Consequences

Each user connected to the local server can use this shared database link to connect to the remote database and query the tables in the corresponding remote schema.

Each local, shared server process establishes one connection to the remote server. Whenever a local server process needs to access the remote server through the SALES.DIVISION3.ACME.COM database link, the local process reuses established network connections, even if the connected user is a different user.

If this database link is used frequently, eventually every shared server in the local database will have a remote connection. At this point, no more physical connections are needed to the remote server, even if new users use this shared database link.

Creating a Public Current User Database Link

The following example connects to the local database as the connected user and creates a public link to the SALES database (using its net service name SLDB). The following statement creates a public current user database link:

```
CONNECT bart/simpson@local

CREATE PUBLIC DATABASE LINK sales.division3.acme.com
  CONNECT TO CURRENT_USER
  USING 'sldb';
```

Note: To use this link, the current user must be a global user.

Consequences

Assume SCOTT creates local procedure FIRE_EMP that deletes a row from the remote EMP table, and grants execute privilege to FORD.

```
CONNECT scott/tiger@local_db

CREATE PROCEDURE fire_emp (enum NUMBER)
AS
BEGIN
   DELETE FROM emp@sales.division3.acme.com
   WHERE empno=enum;
END;

GRANT EXECUTE ON FIRE_EMP TO FORD;
```

Now, assume that FORD connects to the local database and runs SCOTT's procedure:

```
CONNECT ford/fairlane@local_db
```

```
EXECUTE PROCEDURE scott.fire_emp (enum 10345);
```

When FORD executes the procedure SCOTT.FIRE_EMP, the procedure runs under SCOTT's privileges. Because a current user database link is used, the connection is established to SCOTT's remote schema—not FORD's remote schema. Note that SCOTT must be a global user while FORD does not have to be a global user.

Note: If a connected user database link were used instead, the connection would be to FORD's remote schema. For more information about invoker's-rights and privileges, see the *PL/SQL User's Guide and Reference*.

You can accomplish the same result by using a fixed user database link to SCOTT's remote schema. With fixed user database links, however, security can be compromised because SCOTT's username and password are available in readable format in the database.

3

Developing Applications for a Distributed Database System

This chapter describes considerations important when developing an application to run in a distributed database system. *Oracle8i Concepts* describes how Oracle eliminates much of the need to design applications specifically to work in a distributed environment.

The topics covered include:

- Managing the Distribution of an Application's Data
- Controlling Connections Established by Database Links
- Maintaining Referential Integrity in a Distributed System
- Tuning Distributed Queries
- Handling Errors in Remote Procedures

> **See Also:** *Oracle8i Administrator's Guide* for a complete discussion of implementing Oracle8*i* applications, and *Oracle8i Application Developer's Guide - Fundamentals* for more information about application development in an Oracle environment.

Managing the Distribution of an Application's Data

In a distributed database environment, coordinate with the database administrator to determine the best location for the data. Some issues to consider are:

- Number of transactions posted from each location
- Amount of data (portion of table) used by each node
- Performance characteristics and reliability of the network
- Speed of various nodes, capacities of disks
- Importance of a node or link when it is unavailable
- Need for referential integrity among tables

Controlling Connections Established by Database Links

When a global object name is referenced in a SQL statement or remote procedure call, database links establish a connection to a session in the remote database on behalf of the local user. The remote connection and session are only created if the connection has not already been established previously for the local user session.

The connections and sessions established to remote databases persist for the duration of the local user's session, unless the application or user explicitly terminates them. Note that when you issue a SELECT statement across a database link, a transaction lock is placed on the rollback segments. To re-release the segment, you must issue a COMMIT or ROLLBACK statement.

Terminating remote connections established using database links is useful for disconnecting high cost connections that are no longer required by the application. You can terminate a remote connection and session using the ALTER SESSION command with the CLOSE DATABASE LINK parameter. For example, assume you issue the following transactions:

```
SELECT * FROM emp@sales;
COMMIT;
```

The following statement terminates the session in the remote database pointed to by the SALES database link:

```
ALTER SESSION CLOSE DATABASE LINK sales;
```

To close a database link connection in your user session, you must have the ALTER SESSION system privilege.

Note: Before closing a database link, first close all cursors that use the link and then end your current transaction if it uses the link.

See Also: *Oracle8i SQL Reference* for more information about the ALTER SESSION statement.

Maintaining Referential Integrity in a Distributed System

If a part of a distributed statement fails, for example, due to an integrity constraint violation, Oracle returns error number `ORA-02055`. Subsequent statements or procedure calls return error number `ORA-02067` until a rollback or rollback to savepoint is issued.

Design your application to check for any returned error messages that indicate that a portion of the distributed update has failed. If you detect a failure, you should roll back the entire transaction before allowing the application to proceed.

Oracle does not permit declarative referential integrity constraints to be defined across nodes of a distributed system. In other words, a declarative referential integrity constraint on one table cannot specify a foreign key that references a primary or unique key of a remote table. Nevertheless, you can maintain parent/child table relationships across nodes using triggers.

If you decide to define referential integrity across the nodes of a distributed database using triggers, be aware that network failures can limit the accessibility of not only the parent table, but also the child table. For example, assume that the child table is in the SALES database and the parent table is in the HQ database. If the network connection between the two databases fails, some DML statements against the child table (those that insert rows into the child table or update a foreign key value in the child table) cannot proceed because the referential integrity triggers must have access to the parent table in the HQ database.

See Also: *Oracle8i Concepts* for more information about using triggers to enforce referential integrity.

Tuning Distributed Queries

The local Oracle database server breaks the distributed query into a corresponding number of remote queries, which it then sends to the remote nodes for execution. The remote nodes execute the queries and send the results back to the local node.

The local node then performs any necessary post-processing and returns the results to the user or application.

You have several options for designing your application to optimize query processing. This section contains the following topics:

- Using Collocated Inline Views
- Using Cost-Based Optimization
- Using Hints
- Analyzing the Execution Plan

Using Collocated Inline Views

The most effective way of optimizing distributed queries is to access the remote databases as little as possible and to retrieve only the required data.

For example, assume you reference five remote tables from two different remote databases in a distributed query and have a complex filter (for example, `WHERE r1.salary + r2.salary > 50000`). You can improve the performance of the query by rewriting the query to access the remote databases once and to apply the filter at the remote site. This rewrite causes less data to be transferred to the query execution site.

Rewriting your query to access the remote database once is achieved by using *collocated inline views*. The following terms need to be defined:

collocated	Two or more tables located in the same database.
inline view	A SELECT statement that is substituted for a table in a parent SELECT statement. The embedded SELECT statement (in bold) is an example of an inline view: SELECT e.empno, e.ename, d.deptno, d.dname FROM (**SELECT empno, ename from emp@orc1.world**) e, dept d;
collocated inline view	An inline view that selects data from multiple tables from a single database only. It reduces the amount of times that the remote database is accessed, improving the performance of a distributed query.

Oracle Corporation recommends that you form your distributed query using collocated inline views to increase the performance of your distributed query. Oracle's cost-based optimization can transparently rewrite many of your

distributed queries to take advantage of the performance gains offered by collocated inline views.

Using Cost-Based Optimization

In addition to rewriting your queries with collocated inline views, the cost-based optimization method optimizes distributed queries according to the gathered statistics of the referenced tables and the computations performed by the optimizer.

For example, cost-based optimization analyzes the following query. The example assumes that table statistics are available. Note that it analyzes the query inside a CREATE TABLE statement:

```
CREATE TABLE AS (
                 SELECT l.a, l.b, r1.c, r1.d, r1.e, r2.b, r2.c
                 FROM local l, remote1 r1, remote2 r2
                    WHERE l.c = r.c
                    AND r1.c = r2.c
                    AND r.e > 300
                );
```

and rewrites it as:

```
CREATE TABLE AS (
                 SELECT l.a, l.b, v.c, v.d, v.e
                 FROM (
                        SELECT r1.c, r1.d, r1.e, r2.b, r2.c
                         FROM remote1 r1, remote2 r2
                          WHERE r1.c = r2.c
                          AND r1.e > 300
                      ) v, local l
                 WHERE l.c = r1.c
                );
```

The alias v is assigned to the inline view, which can then be referenced as a table in the above SELECT statement. Creating a collocated inline view reduces the amount of queries performed at a remote site, thereby reducing costly network traffic.

How Does Cost-Based Optimization Work?

The optimizer's main task is to rewrite a distributed query to use collocated inline views. This optimization is performed in three steps:

1. All mergeable views are merged.
2. Optimizer performs collocated query block test.

3. Optimizer rewrites query using collocated inline views.

After the query is rewritten, it is executed and the data set is returned to the user.

Cost-Based Optimization Restrictions While cost-based optimization is performed transparently to the user, it is unable to improve the performance of several distributed query scenarios. Specifically, if your distributed query contains any of the following, cost-based optimization is not effective:

- Aggregates
- Subqueries
- Complex SQL

If your distributed query contains one of the above, see "Using Hints" on page 3-8 to learn how you can modify your query and use hints to improve the performance of your distributed query.

Setting Up Cost-Based Optimization

After you have set up your system to use cost-based optimization to improve the performance of distributed queries, the operation is transparent to the user. In other words, the optimization occurs automatically when the query is issued.

You need to complete the following tasks to set up your system to take advantage of Oracle's optimizer:

- Setting Up the Environment
- Analyzing Tables

Setting Up the Environment To enable cost-based optimization, set the OPTIMIZER_MODE initialization parameter to CHOOSE or COST. You can persistently set this parameter by:

- Modifying the OPTIMZER_MODE parameter in the initialization parameter file.
- Setting it on a session-level by issuing an ALTER SESSION statement.

Issue one of the following statements to set the OPTIMIZER_MODE parameter at the session level:

```
ALTER SESSION OPTIMIZER_MODE = CHOOSE;
ALTER SESSION OPTIMIZER_MODE = COST;
```

See Also: *Oracle8i Designing and Tuning for Performance* manual for information on setting the `OPTIMZER_MODE` parameter in the parameter file and for configuring your system to use a cost-based optimization method.

Analyzing Tables In order for cost-based optimization to select the most efficient path for a distributed query, you must provide accurate statistics for the tables involved.

One way to generate statistics for a table is to execute an ANALYZE statement. Note that when you execute this statement, Oracle locks the tables being analyzed. For example, if you reference the EMP and DEPT tables in your distributed query, execute the following to generate the necessary statistics:

```
ANALYZE TABLE emp COMPUTE STATISTICS;
ANALYZE TABLE dept COMPUTE STATISTICS;
```

The statistics are stored in the following locations:

Statistic Type	Tables
Tables	DBA/ALL/USER_TABLES
Columns	DBA/ALL/USER_TAB_COL_STATISTICS DBA/ALL/USER_TAB_COLUMNS
Histograms	DBA/ALL/USER_TAB_HISTOGRAMS DBA/ALL/USER_PART_HISTOGRAMS DBA/ALL/USER_SUBPART_HISTOGRAMS
User-defined statistics	DBA/ALL/USER_USTATS

Note: You must connect locally with respect to the tables to execute the ANALYZE statement. You cannot execute the following:

```
ANALYZE TABLE remote@remote.com COMPUTE STATISTICS;
```

You must first connect to the remote site and then execute the above ANALYZE statement.

> **See Also:** *Oracle8i SQL Reference* for additional information on using the ANALYZE statement, and *Oracle8i Designing and Tuning for Performance* to learn how to generate statistics for more than one object at a time and to learn how to automate the process of keeping statistics current.

You can also collect statistics using the DBMS_ STATS package. The following procedures enable the gathering of certain classes of optimizer statistics, with possible performance improvements over the ANALYZE command:

- GATHER_INDEX_STATS
- GATHER_TABLE_STATS
- GATHER_SCHEMA_STATS
- GATHER_DATABASE_STATS

For example, assume that distributed transactions routinely access the SCOTT.DEPT table. To ensure that the cost-based optimizer is still picking the best plan, execute the following:

```
BEGIN
   DBMS_STATS.GATHER_TABLE_STATS ('scott', 'dept');
END;
```

> **See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for additional information on using the DBMS_STATS package.

Using Hints

If a statement is not sufficiently optimized, then you can use hints to extend the capability of cost-based optimization. Specifically, if you write your own query to utilize collocated inline views, instruct the cost-based optimizer not to rewrite your distributed query.

Additionally, if you have special knowledge about the database environment (that is, statistics, load, network and CPU limitations, distributed queries, etc.), you can specify a hint to guide cost-based optimization. For example, if you have written your own optimized query using collocated inline views that are based on your knowledge of the database environment, specify the NO_MERGE hint to prevent the optimizer from rewriting your query.

This technique is especially helpful if your distributed query contains an aggregate, subquery, or complex SQL. Because this type of distributed query cannot be

rewritten by the optimizer, specifying NO_MERGE causes the optimizer to skip the steps described in the "Using Hints" section on page 3-8.

The DRIVING_SITE hint allows you to define a remote site to act as the query execution site. This hint is especially helpful when the remote site contains the majority of the data. In this way, the query executes on the remote site, which then returns the data to the local site.

Using the NO_MERGE Hint

The NO_MERGE hint prevents Oracle from merging an inline view into a potentially non-collocated SQL statement (see "Using Hints" on page 3-8). This hint is embedded in the SELECT statement and can appear either at the beginning of the SELECT statement with the inline view as an argument or in the query block that defines the inline view.

```
/* with argument */

SELECT /*+NO_MERGE(v)*/ t1.x, v.avg_y
   FROM t1, (SELECT x, AVG(y) AS avg_y FROM t2 GROUP BY x) v,
   WHERE t1.x = v.x AND t1.y = 1;

/* in query block */

SELECT t1.x, v.avg_y
   FROM t1, (SELECT /*+NO_MERGE*/ x, AVG(y) AS avg_y FROM t2 GROUP BY x) v,
   WHERE t1.x = v.x AND t1.y = 1;
```

Typically, you use this hint when you have developed an optimized query based on your knowledge of your database environment.

> **See Also:** *Oracle8i Designing and Tuning for Performance* for more information about the NO_MERGE hint.

Using the DRIVING_SITE Hint

The DRIVING_SITE hint allows you to specify the site where the query execution is performed. It is best to let the cost-based optimization determine where the execution should be performed, but if you want to override the optimizer, you can specify the execution site manually.

Following is an example of a SELECT statement with a DRIVING_SITE hint:

```
SELECT /*+DRIVING_SITE(dept)*/ * FROM emp, dept@remote.com
   WHERE emp.deptno = dept.deptno;
```

See Also: *Oracle8i Designing and Tuning for Performance* for more information about the DRIVING_SITE hint.

Analyzing the Execution Plan

An important aspect to tuning distributed queries is analyzing the execution plan. The feedback that you receive from your analysis is an important element to testing and verifying your database. Verification is increasingly important when you want to compare the execution plan for a distributed query optimized by cost-based optimization versus the plan for a query manually optimized using hints, collocated inline views, etc.

See Also: *Oracle8i Designing and Tuning for Performance* for detailed information about execution plans, the EXPLAIN PLAN command, and how to interpret the results.

Preparing the Database to Store the Plan

Before you can view the execution plan for the distributed query, prepare the database to store the execution plan. You can perform this preparation by executing a script. Execute the following script to prepare your database to store an execution plan:

```
SQL> @utlxplan.sql
```

Note: The location of the `utlxplan.sql` file depends on your operating system.

After you execute the `utlxplan.sql` file, a PLAN_TABLE is created in the current schema to temporarily store the execution plan.

Generating the Execution Plan

After you have prepared the database to store the execution plan, you are ready to view the plan for a specified query. Instead of directly executing a SQL statement, append the statement to the EXPLAIN PLAN FOR clause. For example, you can execute the following:

```
EXPLAIN PLAN FOR
   SELECT d.dname
   FROM dept d
      WHERE d.deptno
      IN (SELECT deptno
```

```
            FROM emp@orc2.world
            GROUP BY deptno
            HAVING COUNT (deptno) >3
            )
/
```

Viewing the Execution Plan

After you have executed the above SQL statement, the execution plan is stored temporarily in the PLAN_TABLE that you created earlier. To view the results of the execution plan, execute the following script:

```
@utlxpls.sql
```

Note: The location of the `utlxpls.sql` file depends on your operating system.

Executing the `utlxpls.sql` script displays the execution plan for the SELECT statement that you specified. The results are formatted as follows:

```
Plan Table
--------------------------------------------------------------------------------
| Operation                 |  Name     |  Rows | Bytes|  Cost  | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT          |           |       |      |        |       |       |
|  NESTED LOOPS             |           |       |      |        |       |       |
|   VIEW                    |           |       |      |        |       |       |
|    REMOTE                 |           |       |      |        |       |       |
|   TABLE ACCESS BY INDEX RO|DEPT       |       |      |        |       |       |
|    INDEX UNIQUE SCAN      |PK_DEPT    |       |      |        |       |       |
--------------------------------------------------------------------------------
```

If you are manually optimizing distributed queries by writing your own collocated inline views or using hints, it is best to generate an execution plan before and after your manual optimization. With both execution plans, you can compare the effectiveness of your manual optimization and make changes as necessary to improve the performance of the distributed query.

To view the SQL statement that will be executed at the remote site, execute the following select statement:

```
SELECT other
FROM plan_table
   WHERE operation = 'REMOTE';
```

Following is sample output:

```
SELECT DISTINCT "A1"."DEPTNO" FROM "EMP" "A1"
   GROUP BY "A1"."DEPTNO" HAVING COUNT("A1"."DEPTNO")>3
```

Note: If you are having difficulty viewing the entire contents of the OTHER column, you may need to execute the following:

```
SET LONG 9999999
```

Handling Errors in Remote Procedures

When Oracle executes a procedure locally or at a remote location, four types of exceptions can occur:

- PL/SQL user-defined exceptions, which must be declared using the keyword EXCEPTION.
- PL/SQL predefined exceptions such as the NO_DATA_FOUND keyword.
- SQL errors such as `ORA-00900` and `ORA-02015`.
- Application exceptions generated using the RAISE_APPLICATION_ERROR() procedure.

When using local procedures, you can trap these messages by writing an exception handler such as the following:

```
BEGIN
  ...
EXCEPTION
   WHEN ZERO_DIVIDE THEN
   /* ... handle the exception */
END;
```

Notice that the WHEN clause requires an exception name. If the exception does not have a name, for example, exceptions generated with RAISE_APPLICATION_ERROR, you can assign one using PRAGMA_EXCEPTION_INIT. For example:

```
DECLARE
  null_salary EXCEPTION;
  PRAGMA EXCEPTION_INIT(null_salary, -20101);
BEGIN
  ...
  RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
...
```

```
EXCEPTION
  WHEN null_salary THEN
  ...
END;
```

When calling a remote procedure, exceptions can be handled by an exception handler in the local procedure. The remote procedure must return an error number to the local, calling procedure, which then handles the exception as shown in the previous example. Note that PL/SQL user-defined exceptions always return `ORA-06510` to the local procedure.

Therefore, it is not possible to distinguish between two different user-defined exceptions based on the error number. All other remote exceptions can be handled in the same manner as local exceptions.

Part II

Distributed Transactions Concepts and Administration

4

Distributed Transactions Concepts

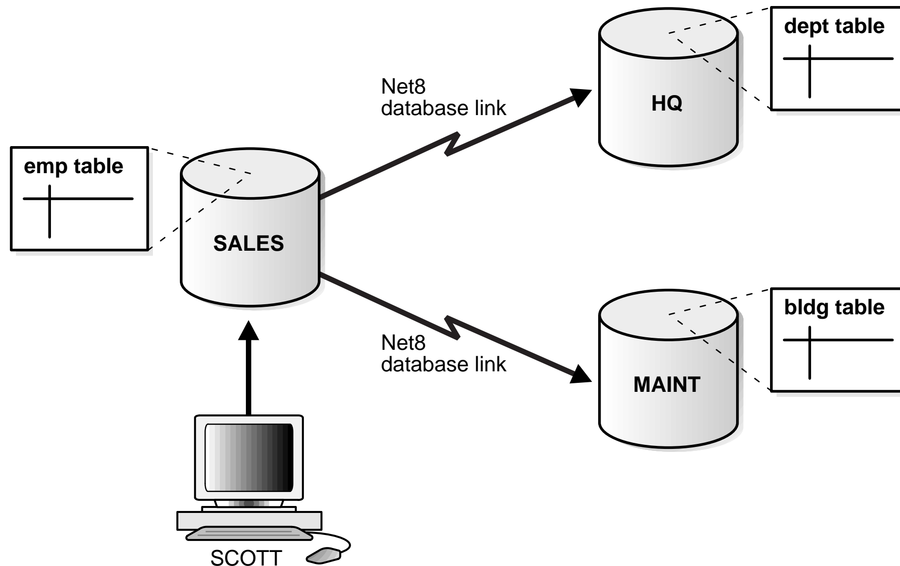
This chapter describes what distributed transactions are and how Oracle8*i* maintains their integrity. Topics include:

- What Are Distributed Transactions?
- Session Trees for Distributed Transactions
- Two-Phase Commit Mechanism
- In-Doubt Transactions
- Distributed Transaction Processing: Case Study

What Are Distributed Transactions?

A *distributed transaction* includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database. For example, assume the database configuration depicted in Figure 4–1:

Figure 4–1 Distributed System



The following distributed transaction executed by SCOTT updates the local SALES database, the remote HQ database, and the remote MAINT database:

```
UPDATE scott.dept@hq.us.acme.com
  SET loc = 'REDWOOD SHORES'
  WHERE deptno = 10;
UPDATE scott.emp
  SET deptno = 11
  WHERE deptno = 10;
UPDATE scott.bldg@maint.us.acme.com
  SET room = 1225
  WHERE room = 1163;
COMMIT;
```

Note: If all statements of a transaction reference only a single remote node, then the transaction is remote, not distributed.

The section contains the following topics:

- Supported Types of Distributed Transactions
- Session Trees for Distributed Transactions
- Two-Phase Commit Mechanism

Supported Types of Distributed Transactions

This section describes permissible operations in distributed transactions:

- DML and DDL Transactions
- Transaction Control Statements

DML and DDL Transactions

The following list describes DML and DDL operations supported in a distributed transaction:

- CREATE TABLE AS SELECT
- DELETE
- INSERT (default and direct load)
- LOCK TABLE
- SELECT
- SELECT FOR UPDATE

You can execute DML and DDL statements in parallel, and INSERT direct load statements serially, but note the following restrictions:

- All remote operations must be SELECT statements.
- These statements must not be clauses in another distributed transaction.
- If the table referenced in the *table_expression_clause* of an INSERT, UPDATE, or DELETE statement is remote, then execution is serial rather than parallel.
- You cannot perform remote operations after issuing parallel DML/DDL or direct load INSERT.

- If the transaction begins using XA or OCI, it executes serially.
- No loopback operations can be performed on the transaction originating the parallel operation. For example, you cannot reference a remote object that is actually a synonym for a local object.
- If you perform a distributed operation other than a SELECT in the transaction, no DML is parallelized.

Transaction Control Statements

The following list describes supported transaction control statements:

- COMMIT
- ROLLBACK
- SAVEPOINT

> **See Also:** *Oracle8i SQL Reference* for more information about these SQL statements.

Session Trees for Distributed Transactions

Oracle8*i* defines a *session tree* of all nodes participating in a distributed transaction. A session tree is a hierarchical model of the transaction that describes the relationships among the nodes that are involved. Each node plays a role in the transaction. For example, the node that originates the transaction is the *global coordinator*, and the node in charge of initiating a commit or rollback is called the *commit point site*.

> **See Also:** "Session Trees for Distributed Transactions" on page 4-5 for an explanation of session trees, and "Distributed Transaction Processing: Case Study" on page 4-20 for an example of a session tree.

Two-Phase Commit Mechanism

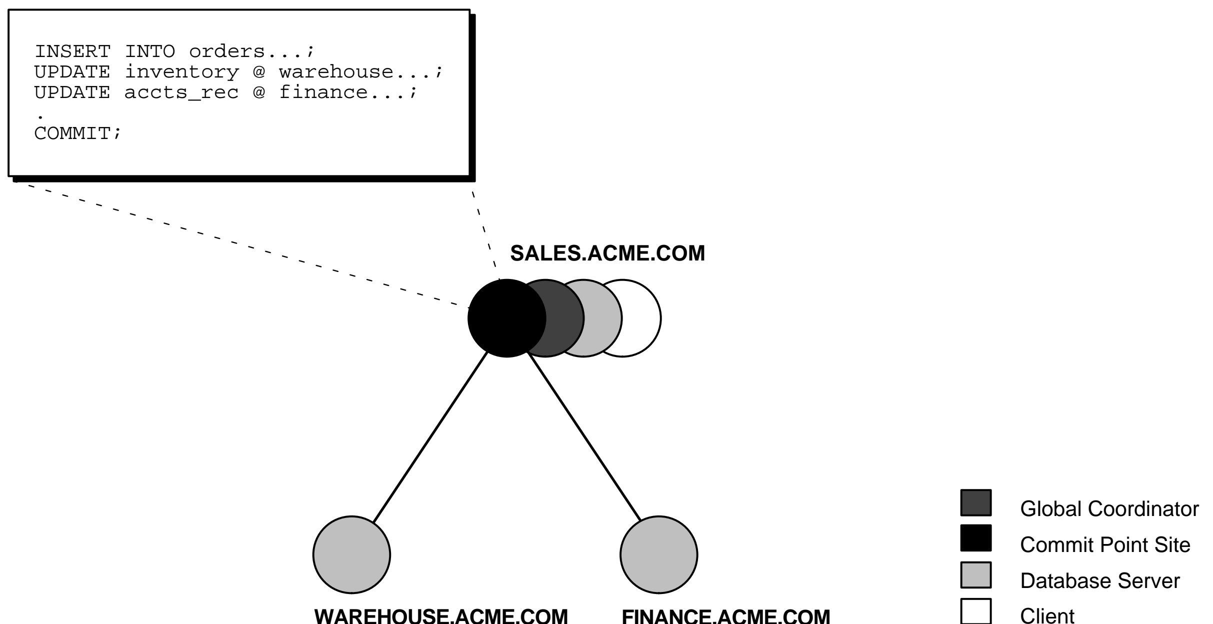
Unlike a transaction on a local database, a distributed transaction involves altering data on multiple databases. Consequently, distributed transaction processing is more complicated, because Oracle must coordinate the committing or rolling back of the changes in a transaction as a self-contained unit. In other words, the entire transaction commits, or the entire transactions rolls back.

Oracle ensures the integrity of data in a distributed transaction using the *two-phase commit mechanism*. In the *prepare phase*, the initiating node in the transaction asks the other participating nodes to promise to commit or roll back the transaction. During the *commit phase*, the initiating node asks all participating nodes to commit the transaction; if this outcome is not possible, then all nodes are asked to roll back.

Session Trees for Distributed Transactions

As the statements in a distributed transaction are issued, Oracle8*i* defines a *session tree* of all nodes participating in the transaction. A session tree is a hierarchical model that describes the relationships among sessions and their roles. Figure 4–2 illustrates a session tree:

Figure 4–2 Example of a Session Tree



All nodes participating in the session tree of a distributed transaction assume one or more of the following roles:

client	A node that references information in a database belonging to a different node.
database server	A node that receives a request for information from another node.
global coordinator	The node that originates the distributed transaction.
local coordinator	A node that is forced to reference data on other nodes to complete its part of the transaction.
commit point site	The node that commits or rolls back the transaction as instructed by the global coordinator.

The role a node plays in a distributed transaction is determined by:

- Whether the transaction is local or remote
- The *commit point strength* of the node ("Commit Point Site" on page 4-7)
- Whether all requested data is available at a node, or whether other nodes need to be referenced to complete the transaction
- Whether the node is read-only

Clients

A node acts as a client when it references information from another node's database. The referenced node is a *database server*. In Figure 4–2, the node SALES is a client of the nodes that host the WAREHOUSE and FINANCE databases.

Database Servers

A *database server* is a node that hosts a database from which a client requests data.

In Figure 4–2, an application at the SALES node initiates a distributed transaction that accesses data from the WAREHOUSE and FINANCE nodes. Therefore, SALES.ACME.COM has the role of client node, and WAREHOUSE and FINANCE are both database servers. In this example, SALES is a database server *and* a client because the application also requests a change to the SALES database.

Local Coordinators

A node that must reference data on other nodes to complete its part in the distributed transaction is called a *local coordinator*. In Figure 4–2, SALES is a local coordinator because it coordinates the nodes it directly references: WAREHOUSE

and FINANCE. SALES also happens to be the global coordinator because it coordinates all the nodes involved in the transaction.

A local coordinator is responsible for coordinating the transaction among the nodes it communicates directly with by:

- Receiving and relaying transaction status information to and from those nodes.
- Passing queries to those nodes.
- Receiving queries from those nodes and passing them on to other nodes.
- Returning the results of queries to the nodes that initiated them.

Global Coordinator

The node where the distributed transaction originates is called the *global coordinator*. The database application issuing the distributed transaction is directly connected to the node acting as the global coordinator. For example, in Figure 4–2, the transaction issued at the node SALES references information from the database servers WAREHOUSE and FINANCE. Therefore, SALES.ACME.COM is the global coordinator of this distributed transaction.

The global coordinator becomes the parent or root of the session tree. The global coordinator performs the following operations during a distributed transaction:

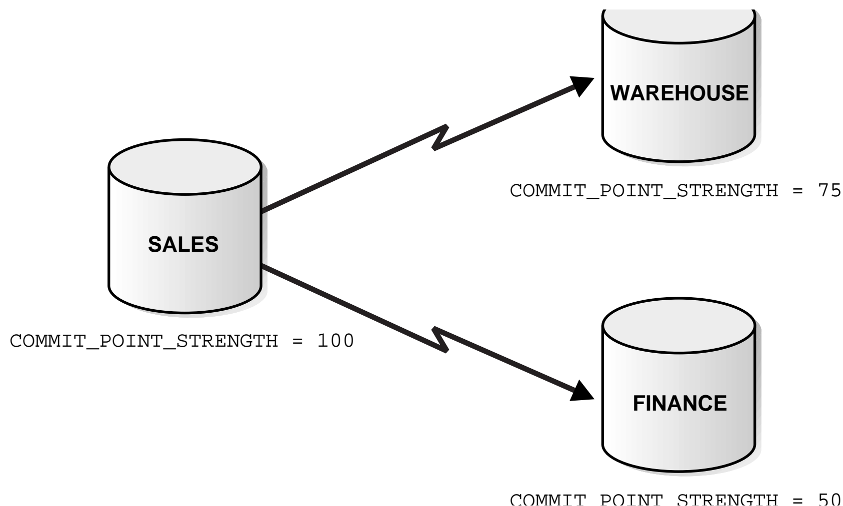
- Sends all of the distributed transaction's SQL statements, remote procedure calls, etc. to the directly referenced nodes, thus forming the session tree.
- Instructs all directly referenced nodes other than the commit point site to prepare the transaction.
- Instructs the commit point site to initiate the global commit of the transaction if all nodes prepare successfully.
- Instructs all nodes to initiate a global rollback of the transaction if there is an abort response.

Commit Point Site

The job of the *commit point site* is to initiate a commit or roll back operation as instructed by the global coordinator. The system administrator always designates one node to be the commit point site in the session tree by assigning all nodes a commit point strength. The node selected as commit point site should be the node that stores the most critical data.

Figure 4–3 illustrates an example of distributed system, with SALES serving as the commit point site:

Figure 4–3 Commit Point Site



The commit point site is distinct from all other nodes involved in a distributed transaction in these ways:

- The commit point site never enters the prepared state. Consequently, if the commit point site stores the most critical data, this data never remains in-doubt, even if a failure occurs. In failure situations, failed nodes remain in a prepared state, holding necessary locks on data until in-doubt transactions are resolved.
- The commit point site commits before the other nodes involved in the transaction. In effect, the outcome of a distributed transaction at the commit point site determines whether the transaction at all nodes is committed or rolled back: the other nodes follow the lead of the commit point site. The global coordinator ensures that all nodes complete the transaction in the same manner as the commit point site.

How a Distributed Transaction Commits

A distributed transaction is considered committed after all non-commit point sites are prepared, and the transaction has been actually committed at the commit point site. The online redo log at the commit point site is updated as soon as the distributed transaction is committed at this node.

Because the commit point log contains a record of the commit, the transaction is considered committed even though some participating nodes may still be only in the prepared state and the transaction not yet actually committed at these nodes. In the same way, a distributed transaction is considered *not* committed if the commit has not been logged at the commit point site.

Commit Point Strength

Every database server must be assigned a *commit point strength*. If a database server is referenced in a distributed transaction, the value of its commit point strength determines which role it plays in the two-phase commit. Specifically, the commit point strength determines whether a given node is the commit point site in the distributed transaction and thus commits before all of the other nodes. This value is specified using the initialization parameter COMMIT_POINT_STRENGTH.

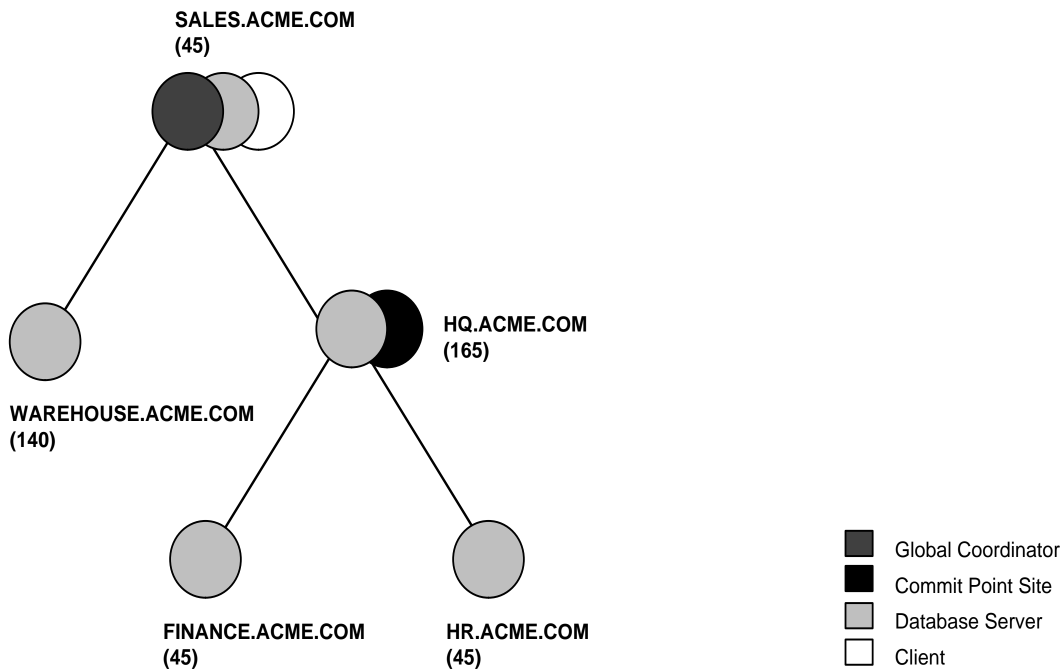
> **See Also:** *Oracle8i Reference* for more information about COMMIT_POINT_STRENGTH.

How Oracle Determines the Commit Point Site The commit point site, which is determined at the beginning of the prepare phase, is selected only from the nodes participating in the transaction. The following sequence of events occurs:

1. Of the nodes directly referenced by the global coordinator, Oracle selects the node with the highest commit point strength as the commit point site.
2. The initially-selected node determines if any of the nodes from which it has to obtain information for this transaction has a higher commit point strength.
3. Either the node with the highest commit point strength directly referenced in the transaction or one of its servers with a higher commit point strength becomes the commit point site.
4. After the final commit point site has been determined, the global coordinator sends prepare responses to all nodes participating in the transaction.

Figure 4–4 shows in a sample session tree the commit point strengths of each node (in parentheses) and shows the node chosen as the commit point site:

Figure 4–4 Commit Point Strengths and Determination of the Commit Point Site



The following conditions apply when determining the commit point site:

- A read-only node cannot be the commit point site.
- If multiple nodes directly referenced by the global coordinator have the same commit point strength, then Oracle designates one of these as the commit point site.
- If a distributed transaction ends with a rollback, then the prepare and commit phases are not needed. Consequently, Oracle never determines a commit point site. Instead, the global coordinator sends a ROLLBACK statement to all nodes and ends the processing of the distributed transaction.

As Figure 4–4 illustrates, the commit point site and the global coordinator can be different nodes of the session tree. The commit point strength of each node is communicated to the coordinators when the initial connections are made. The coordinators retain the commit point strengths of each node they are in direct communication with so that commit point sites can be efficiently selected during

two-phase commits. Therefore, it is not necessary for the commit point strength to be exchanged between a coordinator and a node each time a commit occurs.

> **See Also:** "Specifying the Commit Point Strength of a Node" on page 5-5 to learn how to set the commit point strength of a node, and *Oracle8i Reference* for more information about the initialization parameter COMMIT_POINT_STRENGTH.

Two-Phase Commit Mechanism

All participating nodes in a distributed transaction should perform the same action: they should either all commit or all perform a rollback of the transaction. Oracle8*i* automatically controls and monitors the commit or rollback of a distributed transaction and maintains the integrity of the *global database* (the collection of databases participating in the transaction) using the two-phase commit mechanism. This mechanism is completely transparent, requiring no programming on the part of the user or application developer.

The commit mechanism has the following distinct phases, which Oracle performs automatically whenever a user commits a distributed transaction:

prepare phase	The initiating node, called the *global coordinator*, asks participating nodes other than the commit point site to promise to commit or roll back the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.
commit phase	If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.
forget phase	The global coordinator forgets about the transaction.

This section contains the following topics:

- Prepare Phase
- Commit Phase
- Forget Phase

Prepare Phase

The first phase in committing a distributed transaction is the *prepare phase*. In this phase, Oracle does not actually commit or roll back the transaction. Instead, all

nodes referenced in a distributed transaction (except the commit point site, described in the "Commit Point Site" on page 4-7) are told to prepare to commit. By preparing, a node:

- Records information in the online redo logs so that it can subsequently either commit or roll back the transaction, regardless of intervening failures.
- Places a distributed lock on modified tables, which prevents reads.

When a node responds to the global coordinator that it is prepared to commit, the prepared node *promises* to either commit or roll back the transaction later—but does not make a unilateral decision on whether to commit or roll back the transaction. The promise means that if an instance failure occurs at this point, the node can use the redo records in the online log to recover the database back to the prepare phase.

Note: Queries that start after a node has prepared cannot access the associated locked data until all phases complete. The time is insignificant unless a failure occurs (see "Deciding How to Handle In-Doubt Transactions" on page 5-10).

Types of Responses in the Prepare Phase

When a node is told to prepare, it can respond in the following ways:

prepared	Data on the node has been modified by a statement in the distributed transaction, and the node has successfully prepared.
read-only	No data on the node has been, or can be, modified (only queried), so no preparation is necessary.
abort	The node cannot successfully prepare.

Prepared Response When a node has successfully prepared, it issues a *prepared message*. The message indicates that the node has records of the changes in the online log, so it is prepared either to commit or perform a rollback. The message also guarantees that locks held for the transaction can survive a failure.

Read-Only Response When a node is asked to prepare, and the SQL statements affecting the database do not change the node's data, the node responds with a *read-only message*. The message indicates that the node will not participate in the commit phase.

There are three cases in which all or part of a distributed transaction is read-only:

Case	Conditions	Consequence
Partially read-only	Any of the following occurs: ▪ Only queries are issued at one or more nodes. ▪ No data is changed. ▪ Changes rolled back due to triggers firing or constraint violations.	The read-only nodes recognize their status when asked to prepare. They give their local coordinators a read-only response. Thus, the commit phase completes faster because Oracle eliminates read-only nodes from subsequent processing.
Completely read-only with prepare phase	All of following occur: ▪ No data changes. ▪ Transaction is *not* started with SET TRANSACTION READ ONLY statement.	All nodes recognize that they are read-only during prepare phase, so no commit phase is required. The global coordinator, not knowing whether all nodes are read-only, must still perform the prepare phase.
Completely read-only without two-phase commit	All of following occur: ▪ No data changes. ▪ Transaction *is* started with SET TRANSACTION READ ONLY statement.	Only queries are allowed in the transaction, so global coordinator does not have to perform two-phase commit. Changes by other transactions do not degrade global transaction-level read consistency because of global SCN coordination among nodes. The transaction does not use rollback segments.

Note that if a distributed transaction is set to read-only, then it does not use rollback segments. If many users connect to the database and their transactions are *not* set to READ ONLY, then they allocate rollback space even if they are only performing queries.

Abort Response When a node cannot successfully prepare, it performs the following actions:

1. Releases resources currently held by the transaction and rolls back the local portion of the transaction.
2. Responds to the node that referenced it in the distributed transaction with an *abort message*.

These actions then propagate to the other nodes involved in the distributed transaction so that they can roll back the transaction and guarantee the integrity of the data in the global database. This response enforces the primary rule of a distributed transaction: *all nodes involved in the transaction either all commit or all roll back the transaction at the same logical time.*

Steps in the Prepare Phase

To complete the prepare phase, each node excluding the commit point site performs the following steps:

1. The node requests that its *descendants*, that is, the nodes subsequently referenced, prepare to commit.
2. The node checks to see whether the transaction changes data on itself or its descendants. If there is no change to the data, then the node skips the remaining steps and returns a *read-only response* (see "Read-Only Response" on page 4-12).
3. The node allocates the resources it needs to commit the transaction if data is changed.
4. The node saves redo records corresponding to changes made by the transaction to its online redo log.
5. The node guarantees that locks held for the transaction are able to survive a failure.
6. The node responds to the initiating node with a *prepared response* (see "Prepared Response" on page 4-12) or, if its attempt or the attempt of one of its descendents to prepare was unsuccessful, with an *abort response* (see "Abort Response" on page 4-13).

These actions guarantee that the node can subsequently commit or roll back the transaction on the node. The prepared nodes then wait until a COMMIT or ROLLBACK request is received from the global coordinator.

After the nodes are prepared, the distributed transaction is said to be *in-doubt* (see "In-Doubt Transactions" on page 4-16). It retains in-doubt status until all changes are either committed or rolled back.

Commit Phase

The second phase in committing a distributed transaction is the commit phase. Before this phase occurs, *all* nodes other than the commit point site referenced in the distributed transaction have guaranteed that they are prepared, that is, they have the necessary resources to commit the transaction.

Steps in the Commit Phase

The commit phase consists of the following steps:

1. The global coordinator instructs the commit point site to commit.
2. The commit point site commits.
3. The commit point site informs the global coordinator that it has committed.
4. The global and local coordinators send a message to all nodes instructing them to commit the transaction.
5. At each node, Oracle8*i* commits the local portion of the distributed transaction and releases locks.
6. At each node, Oracle8*i* records an additional redo entry in the local redo log, indicating that the transaction has committed.
7. The participating nodes notify the global coordinator that they have committed.

When the commit phase is complete, the data on all nodes of the distributed system is consistent with one another.

Guaranteeing Global Database Consistency

Each committed transaction has an associated system change number (SCN) to uniquely identify the changes made by the SQL statements within that transaction. The SCN functions as an internal Oracle timestamp that uniquely identifies a committed version of the database.

In a distributed system, the SCNs of communicating nodes are coordinated when all of the following actions occur:

- A connection occurs using the path described by one or more database links.
- A distributed SQL statement executes.
- A distributed transaction commits.

Among other benefits, the coordination of SCNs among the nodes of a distributed system ensures global read-consistency at both the statement and transaction level. If necessary, global time-based recovery can also be completed.

> **See Also:** "Managing Read Consistency" on page 5-28 for information about managing time lag issues in read consistency.

During the prepare phase, Oracle8*i* determines the highest SCN at all nodes involved in the transaction. The transaction then commits with the high SCN at the

commit point site. The commit SCN is then sent to all prepared nodes with the commit decision.

Forget Phase

After the participating nodes notify the commit point site that they have committed, the commit point site can forget about the transaction. The following steps occur:

1. After receiving notice from the global coordinator that all nodes have committed, the commit point site erases status information about this transaction.
2. The commit point site informs the global coordinator that it has erased the status information.
3. The global coordinator erases its own information about the transaction.

In-Doubt Transactions

The two-phase commit mechanism ensures that all nodes either commit or perform a rollback together. What happens if any of the three phases fails because of a system or network error? The transaction becomes *in-doubt*.

Distributed transactions can become in-doubt in the following ways:

- A server machine running Oracle software crashes.
- A network connection between two or more Oracle databases involved in distributed processing is disconnected.
- An unhandled software error occurs.

The RECO process automatically resolves in-doubt transactions when the machine, network, or software problem is resolved. Until RECO can resolve the transaction, the data is locked for both reads and writes. Oracle blocks reads because it cannot determine which version of the data to display for a query.

This section contains the following topics:

- Automatic Resolution of In-Doubt Transactions
- Manual Resolution of In-Doubt Transactions
- Relevance of System Change Numbers for In-Doubt Transactions

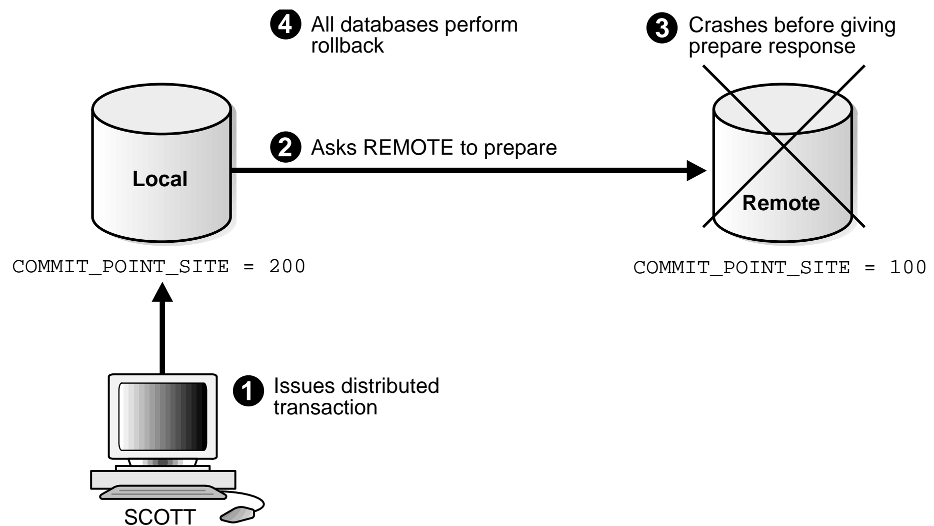
Automatic Resolution of In-Doubt Transactions

In the majority of cases, Oracle resolves the in-doubt transaction automatically. Assume that there are two nodes, LOCAL and REMOTE, in the following scenarios. The local node is the commit point site. User SCOTT connects to LOCAL and executes and commits a distributed transaction that updates LOCAL and REMOTE.

Failure During the Prepare Phase

Figure 4–5 illustrates the sequence of events when there is a failure during the prepare phase of a distributed transaction:

Figure 4–5 Failure During Prepare Phase



The following steps occur:

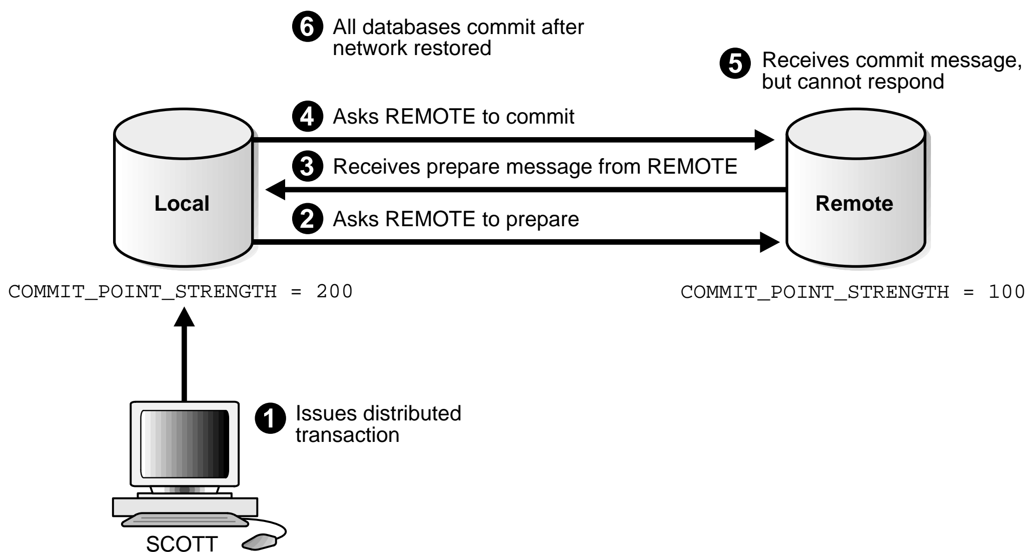
1. Scott connects to LOCAL and executes a distributed transaction.
2. The global coordinator, which in this example is also the commit point site, requests all databases other than the commit point site to promise to commit or roll back when told to do so.
3. The REMOTE database crashes before issuing the prepare response back to LOCAL.

4. The transaction is ultimately rolled back on each database by the RECO process when the remote site is restored.

Failure During the Commit Phase

Figure 4–5 illustrates the sequence of events when there is a failure during the commit phase of a distributed transaction:

Figure 4–6 Failure During Prepare Phase



The following steps occur:

1. Scott connects to LOCAL and executes a distributed transaction.
2. The global coordinator, which in this case is also the commit point site, requests all databases other than the commit point site to promise to commit or roll back when told to do so.
3. The commit point site receives a prepare message from REMOTE saying that it will commit.
4. The commit point site commits the transaction locally, then sends a commit message to REMOTE asking it to commit.
5. The REMOTE database receives the commit message, but cannot respond because of a network failure.

6. The transaction is ultimately committed on the remote database by the RECO process after the network is restored.

> **See Also:** "Deciding How to Handle In-Doubt Transactions" on page 5-10 for a description of failure situations and how Oracle8*i* resolves intervening failures during two-phase commit.

Manual Resolution of In-Doubt Transactions

You should only need to resolve an in-doubt transaction in the following cases:

- The in-doubt transaction has locks on critical data or rollback segments.
- The cause of the machine, network, or software failure cannot be repaired quickly.

Resolution of in-doubt transactions can be complicated. The procedure requires that you do the following:

- Identify the transaction identification number for the in-doubt transaction.
- Query the DBA_2PC_PENDING and DBA_2PC_NEIGHBORS views to determine whether the databases involved in the transaction have committed.
- If necessary, force a commit using the COMMIT FORCE statement or a rollback using the ROLLBACK FORCE statement.

> **See Also:** "Deciding How to Handle In-Doubt Transactions" on page 5-10 and "Manually Overriding In-Doubt Transactions" on page 5-13 to learn how to resolve in-doubt transactions.

Relevance of System Change Numbers for In-Doubt Transactions

A *system change number* (SCN) is an internal timestamp for a committed version of the database. The Oracle database server uses the SCN clock value to guarantee transaction consistency. For example, when a user commits a transaction, Oracle records an SCN for this commit in the online redo log.

Oracle uses SCNs to coordinate distributed transactions among different databases. For example, Oracle uses SCNs in the following way:

1. An application establishes a connection using a database link.
2. The distributed transaction commits with the highest global SCN among all the databases involved.
3. The commit global SCN is sent to all databases involved in the transaction.

SCNs are important for distributed transactions because they function as a synchronized commit timestamp of a transaction—even if the transaction fails. If a transaction becomes in-doubt, an administrator can use this SCN to coordinate changes made to the global database. The global SCN for the transaction commit can also be used to identify the transaction later, for example, in distributed recovery.

Distributed Transaction Processing: Case Study

In this scenario, a company has separate Oracle8*i* database servers, SALES.ACME.COM and WAREHOUSE.ACME.COM. As users insert sales records into the SALES database, associated records are being updated at the WAREHOUSE database.

This case study of distributed processing illustrates:

- The definition of a session tree
- How a commit point site is determined
- When prepare messages are sent
- When a transaction actually commits
- What information is stored locally about the transaction

Stage 1: Client Application Issues DML Statements

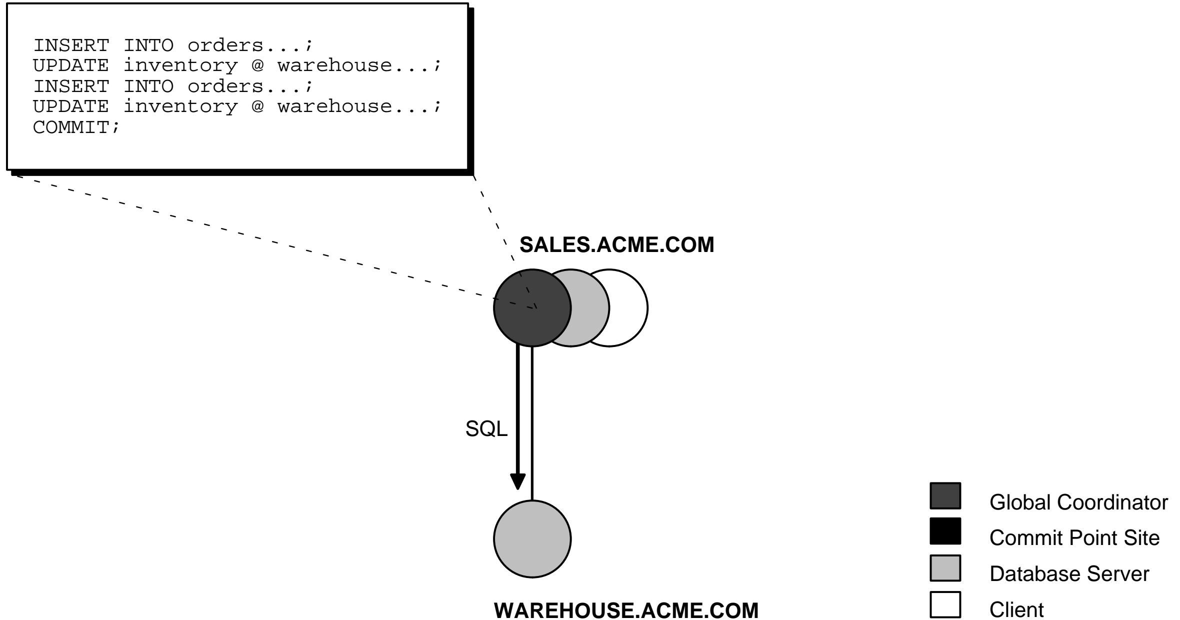
At the Sales department, a salesperson uses SQL*Plus to enter a sales order and then commit it. The application issues a number of SQL statements to enter the order into the SALES database and update the inventory in the WAREHOUSE database:

```
CONNECT scott/tiger@sales.acme.com ...;
INSERT INTO orders ...;
UPDATE inventory@warehouse.acme.com ...;
INSERT INTO orders ...;
UPDATE inventory@warehouse.acme.com ...;
COMMIT;
```

These SQL statements are part of a single distributed transaction, guaranteeing that all issued SQL statements succeed or fail as a unit. Treating the statements as a unit prevents the possibility of an order being placed and then inventory not being updated to reflect the order. In effect, the transaction guarantees the consistency of data in the global database.

As each of the SQL statements in the transaction executes, the session tree is defined, as shown in Figure 4–7.

Figure 4–7 Defining the Session Tree



Note the following aspects of the transaction:

- An order entry application running with the SALES database initiates the transaction. Therefore, SALES.ACME.COM is the global coordinator for the distributed transaction.
- The order entry application inserts a new sales record into the SALES database and updates the inventory at the warehouse. Therefore, the nodes SALES.ACME.COM and WAREHOUSE.ACME.COM are both database servers.
- Because SALES.ACME.COM updates the inventory, it is a client of WAREHOUSE.ACME.COM.

This stage completes the definition of the session tree for this distributed transaction. Each node in the tree has acquired the necessary data locks to execute the SQL statements that reference local data. These locks remain even after the SQL statements have been executed until the two-phase commit is completed.

Stage 2: Oracle Determines Commit Point Site

Oracle determines the commit point site immediately following the COMMIT statement. SALES.ACME.COM, the global coordinator, is determined to be the commit point site, as shown in Figure 4–8.

> **See Also:** "Commit Point Strength" on page 4-9 for more information about how the commit point site is determined.

Figure 4–8 Determining the Commit Point Site

Stage 3: Global Coordinator Sends Prepare Response

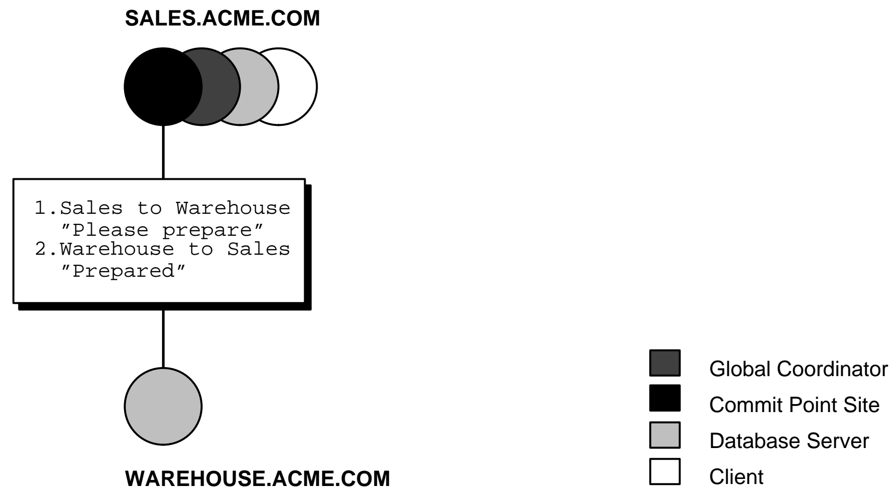
The prepare stage involves the following steps:

1. After Oracle determines the commit point site, the global coordinator sends the prepare message to all directly referenced nodes of the session tree, *excluding* the commit point site. In this example, WAREHOUSE.ACME.COM is the only node asked to prepare.
2. WAREHOUSE.ACME.COM tries to prepare. If a node can guarantee that it can commit the locally dependent part of the transaction and can record the commit information in its local redo log, then the node can successfully prepare. In this example, only WAREHOUSE.ACME.COM receives a prepare message because SALES.ACME.COM is the commit point site.
3. WAREHOUSE.ACME.COM responds to SALES.ACME.COM with a prepared message.

As each node prepares, it sends a message back to the node that asked it to prepare. Depending on the responses, one of the following can happen:

- If *any* of the nodes asked to prepare respond with an abort message to the global coordinator, then the global coordinator tells all nodes to roll back the transaction, and the operation is completed.
- If *all* nodes asked to prepare respond with a prepared or a read-only message to the global coordinator, that is, they have successfully prepared, then the global coordinator asks the commit point site to commit the transaction.

Figure 4–9 Sending and Acknowledging the Prepare Message



Stage 4: Commit Point Site Commits

The committing of the transaction by the commit point site involves the following steps:

1. SALES.ACME.COM, receiving acknowledgment that WAREHOUSE.ACME.COM is prepared, instructs the commit point site to commit the transaction.
2. The commit point site now commits the transaction locally and records this fact in its local redo log.

Even if WAREHOUSE.ACME.COM has not yet committed, the outcome of this transaction is pre-determined. In other words, the transaction *will* be committed at all nodes even if a given node's ability to commit is delayed.

Stage 5: Commit Point Site Informs Global Coordinator of Commit

This stage involves the following steps:

1. The commit point site tells the global coordinator that the transaction has committed. Because the commit point site and global coordinator are the same node in this example, no operation is required. The commit point site knows that the transaction is committed because it recorded this fact in its online log.
2. The global coordinator confirms that the transaction has been committed on all other nodes involved in the distributed transaction.

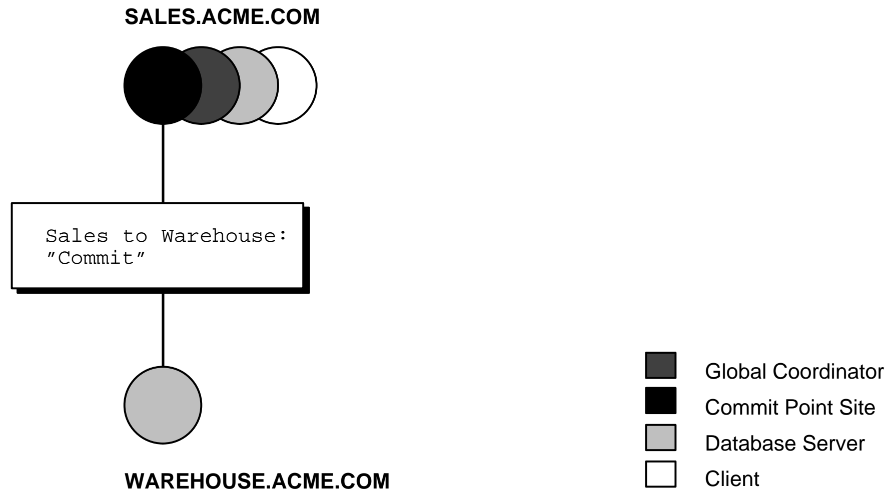
Stage 6: Global and Local Coordinators Tell All Nodes to Commit

The committing of the transaction by all the nodes in the transaction involves the following steps:

1. After the global coordinator has been informed of the commit at the commit point site, it tells all other directly referenced nodes to commit.
2. In turn, any local coordinators instruct their servers to commit, and so on.
3. Each node, including the global coordinator, commits the transaction and records appropriate redo log entries locally. As each node commits, the resource locks that were being held locally for that transaction are released.

In Figure 4–10, SALES.ACME.COM, which is both the commit point site and the global coordinator, has already committed the transaction locally. SALES now instructs WAREHOUSE.ACME.COM to commit the transaction.

Figure 4–10 Instructing Nodes to Commit



Stage 7: Global Coordinator and Commit Point Site Complete the Commit

The completion of the commit of the transaction occurs in the following steps:

1. After all referenced nodes and the global coordinator have committed the transaction, the global coordinator informs the commit point site of this fact.
2. The commit point site, which has been waiting for this message, erases the status information about this distributed transaction.
3. The commit point site informs the global coordinator that it is finished. In other words, the commit point site forgets about committing the distributed transaction. This action is permissible because all nodes involved in the two-phase commit have committed the transaction successfully, so they will never have to determine its status in the future.
4. The global coordinator finalizes the transaction by forgetting about the transaction itself.

   After the completion of the COMMIT phase, the distributed transaction is itself complete. The steps described above are accomplished automatically and in a fraction of a second.

5

Managing Distributed Transactions

This chapter describes how to manage and troubleshoot distributed transactions. Topics include:

- Setting Distributed Transaction Initialization Parameters
- Viewing Information About Distributed Transactions
- Deciding How to Handle In-Doubt Transactions
- Manually Overriding In-Doubt Transactions
- Purging Pending Rows from the Data Dictionary
- Manually Committing an In-Doubt Transaction: Example
- Simulating Distributed Transaction Failure
- Managing Read Consistency

Setting Distributed Transaction Initialization Parameters

You can set initialization parameters that control the behavior of distributed transaction processing. The following tables describes initialization parameters relevant for distributed transaction processing:

Parameter	Description
DISTRIBUTED_TRANSACTIONS	Specifies the maximum number of distributed transactions in which this database can concurrently participate.
DISTRIBUTED_LOCK_TIMEOUT	Specifies the number of seconds that a distributed transaction waits for locked resources.
DISTRIBUTED_RECOVERY_ CONNECTION_HOLD_TIME	Specifies the number of seconds that Oracle holds open a remote connection after a distributed transaction fails.
COMMIT_POINT_STRENGTH	Specifies the value used to determine the commit point site in a distributed transaction.

This section includes the following topics:

- Limiting the Number of Distributed Transactions
- Specifying the Lock Timeout Interval
- Specifying the Interval for Holding Open Connections
- Specifying the Commit Point Strength of a Node

Limiting the Number of Distributed Transactions

The initialization parameter DISTRIBUTED_TRANSACTIONS limits the number of distributed transactions in which a given instance can concurrently participate, both as a client and a server. The default value for this parameter is operating system-dependent.

If Oracle reaches this limit and a subsequent user issues a SQL statement referencing a remote database, then the system rolls back the statement and returns the following error message:

```
ORA-2042: too many global transactions
```

For example, assume that you set the parameter as follows for a given instance:

```
DISTRIBUTED_TRANSACTIONS = 10
```

In this case, a maximum of 10 sessions can concurrently process a distributed transaction. If an additional session attempts to issue a DML statement requiring distributed access, then Oracle returns an error message to the session and rolls back the statement.

Note: Oracle recommends setting the value for DISTRIBUTED_TRANSACTIONS equal to the total number of distributed database sites in your environment.

Increasing the Transaction Limit

Consider increasing the value of the DISTRIBUTED_TRANSACTIONS when an instance regularly participates in numerous distributed transactions and the `ORA-2042` is frequently returned. Increasing the limit allows more users to concurrently issue distributed transactions.

Decreasing the Transaction Limit

If your site is experiencing an abnormally high number of network failures, you can temporarily decrease the value of DISTRIBUTED_TRANSACTIONS. This operation limits the number of in-doubt transactions in which your site takes part, and thereby limits the amount of locked data at your site, and the number of in-doubt transactions you might have to resolve

Disabling Distributed Transaction Processing

If DISTRIBUTED_TRANSACTIONS is set to zero, no distributed SQL statements can be issued in any session. Also, the RECO background process is not started at startup of the local instance. In-doubt distributed transactions that may be present cannot be automatically resolved by Oracle*8i*. Therefore, only set this initialization parameter to zero to prevent distributed transactions when a new instance is started and when it is certain that no in-doubt distributed transactions remained after the last instance shut down.

See Also: *Oracle8i Reference* for more information about the DISTRIBUTED_TRANSACTIONS initialization parameter.

Specifying the Lock Timeout Interval

When you issue a SQL statement, Oracle8*i* attempts to lock the resources needed to successfully execute the statement. If the requested data is currently held by statements of other uncommitted transactions, however, and remains locked for a long time, a timeout occurs.

Consider the following scenarios involving data access failure:

- Transaction Timeouts
- Locks From In-Doubt Transactions

Transaction Timeouts

A DML statement that requires locks on a remote database can be blocked if another transaction own locks on the requested data. If these locks continue to block the requesting SQL statement, then the following sequence of events occurs:

1. A timeout occurs.
2. Oracle rolls back the statement.
3. Oracle returns this error message to the user:

```
ORA-02049: time-out: distributed transaction waiting for lock
```

Because the transaction did not modify data, no actions are necessary as a result of the timeout. Applications should proceed as if a deadlock has been encountered. The user who executed the statement can try to re-execute the statement later. If the lock persists, then the user should contact an administrator to report the problem.

Setting the Timeout Interval Use the initialization parameter DISTRIBUTED_LOCK_TIMEOUT to control the timeout interval, which is set in seconds (see "Specifying the Lock Timeout Interval" on page 5-4). For example, to set the timeout interval for an instance to 30 seconds, include the following line in the associated parameter file:

```
DISTRIBUTED_LOCK_TIMEOUT = 30
```

The default value for this parameter is 60 seconds. Normally, Oracle waits indefinitely for a lock to be released. With the above timeout interval, the timeout errors discussed in the previous section occur if a transaction cannot proceed after 30 seconds of waiting for unavailable resources.

> **See Also:** *Oracle8i Reference* for more information about the DISTRIBUTED_LOCK_TIMEOUT initialization parameter.

Locks From In-Doubt Transactions

A query or DML statement that requires locks on a local database can be blocked indefinitely due to the locked resources of an in-doubt distributed transaction. In this case, Oracle issues the following error message:

```
ORA-01591: lock held by in-doubt distributed transaction identifier
```

In this case, Oracle rolls back the SQL statement immediately. The user who executed the statement can try to re-execute the statement later. If the lock persists, the user should contact an administrator to report the problem, *including* the ID of the in-doubt distributed transaction.

The chances of the above situations occurring are rare considering the low probability of failures during the critical portions of the two-phase commit. Even if such a failure occurs, and assuming quick recovery from a network or system failure, problems are automatically resolved without manual intervention. Thus, problems usually resolve before they can be detected by users or database administrators.

Specifying the Interval for Holding Open Connections

If a distributed transaction fails, then the connection from the local site to the remote site may not close immediately. Instead, it remains open in case communication can be restored quickly, without having to re-establish the connection. Use the following initialization parameter to specify the length of time to hold open a remote connection after a distributed transaction fails:

```
DISTRIBUTED_RECOVERY_CONNECTION_HOLD_TIME
```

The default value for this parameter is 200 seconds. If you set larger values, then you minimize reconnection time but also consume local resources for a longer time period. The range of values is between 0 and 1800. You can set this parameter to a value greater than 1800, however, which simply means that the connection never closes.

Specifying the Commit Point Strength of a Node

The database with the highest commit point strength determines which node commits first in a distributed transaction. When specifying a commit point strength for each node, ensure that the most critical server will be non-blocking if a failure occurs during a prepare or commit phase. The following initialization parameter determines a node's commit point strength:

```
COMMIT_POINT_STRENGTH
```

The default value is operating system-dependent. The range of values is any integer from 0 to 255. For example, to set the commit point strength of a database to 200, include the following line in that database's initialization parameter file:

```
COMMIT_POINT_STRENGTH = 200
```

The commit point strength only determines the commit point site in a distributed transaction.

> **See Also:** "Commit Point Site" on page 4-7 for a conceptual overview of commit points.

Suggestions for Setting the Commit Point Strength

When setting the commit point strength for a database, note the following considerations:

- Because the commit point site stores information about the status of the transaction, the commit point site should not be a node that is frequently unreliable or unavailable in case other nodes need information about the transaction's status.
- Set the commit point strength for a database relative to the amount of critical shared data in the database. For example, a database on a mainframe computer usually shares more data among users than a database on a PC. Therefore, set the commit point strength of the mainframe to a higher value than the PC.

Viewing Information About Distributed Transactions

The data dictionary of each database stores information about all open distributed transactions. You can use data dictionary tables and views to gain information about the transactions. This section contains the following topics:

- Determining the ID Number and Status of Prepared Transactions
- Tracing the Session Tree of In-Doubt Transactions

Determining the ID Number and Status of Prepared Transactions

The following view show the database links that have been defined at the local database and stored in the data dictionary:

View	Purpose
DBA_2PC_PENDING	Lists all in-doubt distributed transactions. The view is empty until populated by an in-doubt transaction. After the transaction is resolved, the view is purged.

Use this view to determine the global commit number for a particular transaction ID. You can use this global commit number when manually resolving an in-doubt transaction.

The following table shows the most relevant columns (for a description of all the columns in the view, see *Oracle8i Reference*):

Table 5–1 DBA_2PC_PENDING

Column	Description
LOCAL_TRAN_ID	Local transaction identifier in the format *integer.integer.integer*. **Note:** When the LOCAL_TRAN_ID and the GLOBAL_TRAN_ID for a connection are the same, the node is the global coordinator of the transaction.
GLOBAL_TRAN_ID	Global database identifier in the format *global_db_name*.*db_hex_id.local_tran_id*, where *db_hex_id* is an eight-character hexadecimal value used to uniquely identify the database. This common transaction ID is the same on every node for a distributed transaction. **Note:** When the LOCAL_TRAN_ID and the GLOBAL_TRAN_ID for a connection are the same, the node is the global coordinator of the transaction.
STATE	*See* Table 5–2, "STATE Column of DBA_2PC_PENDING".
MIXED	YES means that part of the transaction was committed on one node and rolled back on another node.
HOST	Name of the host machine.
COMMIT#	Global commit number for committed transactions.

Table 5–2 STATE Column of DBA_2PC_PENDING

collecting	This category normally applies only to the global coordinator or local coordinators. The node is currently collecting information from other database servers before it can decide whether it can prepare.

Table 5–2 STATE Column of DBA_2PC_PENDING

prepared	The node has prepared and may or may not have acknowledged this to its local coordinator with a prepared message. However, no commit request has been received. The node remains prepared, holding any local resource locks necessary for the transaction to commit.
committed	The node (any type) has committed the transaction, but other nodes involved in the transaction may not have done the same. That is, the transaction is still pending at one or more nodes.
forced commit	A pending transaction can be forced to commit at the discretion of a database administrator. This entry occurs if a transaction is manually committed at a local node by a database administrator.
forced abort (rollback)	A pending transaction can be forced to roll back at the discretion of a database administrator. This entry occurs if this transaction is manually rolled back at a local node by a database administrator.

Execute the following script to query pertinent information in DBA_2PC_PENDING (sample output included):

```
COL local_tran_id FORMAT a13
COL global_tran_id FORMAT a30
COL state FORMAT a8
COL mixed FORMAT a3
COL host FORMAT a10
COL commit# FORMAT a10

SELECT local_tran_id, global_tran_id, state, mixed, host, commit#
FROM dba_2pc_pending
/

SQL> @pending_txn_script

LOCAL_TRAN_ID GLOBAL_TRAN_ID                 STATE    MIX HOST       COMMIT#
------------- ------------------------------ -------- --- ---------- ----------
1.15.870      HQ.ACME.COM.ef192da4.1.15.870  commit   no  dlsun183   115499
```

This output indicates that local transaction 1.15.870 has been committed on this node, but it may be pending on one or more other nodes. Because LOCAL_TRAN_ID and the local part of GLOBAL_TRAN_ID are the same, the node is the global coordinator of the transaction.

Tracing the Session Tree of In-Doubt Transactions

The following view shows which in-doubt transactions are incoming from a remote client and which are outgoing to a remote server:

View	Purpose
DBA_2PC_NEIGHBORS	Lists all incoming (from remote client) and outgoing (to remote server) in-doubt distributed transactions. It also indicates whether the local node is the commit point site in the transaction. The view is empty until populated by an in-doubt transaction. After the transaction is resolved, the view is purged.

When a transaction is in-doubt, you may need to determine which nodes performed which roles in the session tree. Use to this view to determine:

- All the incoming and outgoing connections for a given transaction.
- Whether the node is the commit point site in a given transaction.
- Whether the node is a global coordinator in a given transaction (because its local transaction ID and global transaction ID are the same).

The following table shows the most relevant columns (for an account of all the columns in the view, see *Oracle8i Reference*):

Table 5–3 DBA_2PC_NEIGHBORS

Column	Description
LOCAL_TRAN_ID	Local transaction identifier with the format *integer.integer.integer*. **Note:** When LOCAL_TRAN_ID and GLOBAL_TRAN_ID.DBA_2PC_PENDING for a connection are the same, the node is the global coordinator of the transaction.
IN_OUT	IN for incoming transactions; OUT for outgoing transactions.
DATABASE	For incoming transactions, the name of the client database that requested information from this local node; for outgoing transactions, the name of the database link used to access information on a remote server.
DBUSER_OWNER	For incoming transactions, the local account used to connect by the remote database link; for outgoing transactions, the owner of the database link.

Table 5–3 DBA_2PC_NEIGHBORS

Column	Description
INTERFACE	C is a commit message; N is either a message indicating a prepared state or a request for a read-only commit. When IN_OUT is OUT, C means that the child at the remote end of the connection is the commit point site and knows whether to commit or abort. N means that the local node is informing the remote node that it is prepared. When IN_OUT is IN, C means that the local node or a database at the remote end of an outgoing connection is the commit point site. N means that the remote node is informing the local node that it is prepared.

Execute the following script to query pertinent information in DBA_2PC_PENDING (sample output included):

```
COL local_tran_id FORMAT a13
COL in_out FORMAT a6
COL database FORMAT a25
COL dbuser_owner FORMAT a15
COL interface FORMAT a3

SELECT local_tran_id, in_out, database, dbuser_owner, interface
FROM dba_2pc_neighbors
/

SQL> CONNECT sys/sys_pwd@hq.acme.com
SQL> @neighbors_script

LOCAL_TRAN_ID IN_OUT DATABASE                  DBUSER_OWNER    INT
------------- ------ ------------------------- --------------- ---
1.15.870      out    SALES.ACME.COM            SYS             C
```

This output indicates that the local node sent an outgoing request to remote server SALES to commit transaction 1.15.870. If SALES committed the transaction but no other node did, then you know that SALES is the commit point site—because the commit point site always commits first.

Deciding How to Handle In-Doubt Transactions

A transaction is in-doubt when there is a failure during any aspect of the two-phase commit. Distributed transactions become in-doubt in the following ways:

- A server machine running Oracle software crashes.
- A network connection between two or more Oracle databases involved in distributed processing is disconnected.
- An unhandled software error occurs.

> **See Also:** "In-Doubt Transactions" on page 4-16 for a conceptual overview of in-doubt transactions.

You can manually force the commit or rollback of a local, in-doubt distributed transaction. Because this operation can generate consistency problems, perform it only when specific conditions exist.

This section contains the following topics:

- Discovering Problems with a Two-Phase Commit
- Determining Whether to Perform a Manual Override
- Analyzing the Transaction Data

Discovering Problems with a Two-Phase Commit

The user application that commits a distributed transaction is informed of a problem by one of the following error messages:

```
ORA-02050: transaction ID rolled back,
           some remote dbs may be in-doubt
ORA-02051: transaction ID committed,
           some remote dbs may be in-doubt
ORA-02054: transaction ID in-doubt
```

A robust application should save information about a transaction if it receives any of the above errors. This information can be used later if manual distributed transaction recovery is desired.

No action is required by the administrator of any node that has one or more in-doubt distributed transactions due to a network or system failure. The automatic recovery features of Oracle8*i* transparently complete any in-doubt transaction so that the same outcome occurs on all nodes of a session tree (that is, all commit or all roll back) after the network or system failure is resolved.

In extended outages, however, you can force the commit or rollback of a transaction to release any locked data. Applications must account for such possibilities.

Determining Whether to Perform a Manual Override

Override a specific in-doubt transaction manually *only* when one of the following situations exists:

- The in-doubt transaction locks data that is required by other transactions. This situation occurs when the `ORA-01591` error message interferes with user transactions.
- An in-doubt transaction prevents the extents of a rollback segment from being used by other transactions. The first portion of an in-doubt distributed transaction's local transaction ID corresponds to the ID of the rollback segment, as listed by the data dictionary views DBA_2PC_PENDING and DBA_ROLLBACK_SEGS.
- The failure preventing the two-phase commit phases to complete cannot be corrected in an acceptable time period. Examples of such cases include a telecommunication network that has been damaged or a damaged database that requires a long recovery time.

Normally, you should make a decision to locally force an in-doubt distributed transaction in consultation with administrators at other locations. A wrong decision can lead to database inconsistencies that can be difficult to trace and that you must manually correct.

If the conditions above do not apply, *always* allow the automatic recovery features of Oracle8*i* to complete the transaction. If any of the above criteria are met, however, consider a local override of the in-doubt transaction.

Analyzing the Transaction Data

If you decide to force the transaction to complete, analyze available information with the following goals in mind.

Find a Node That Committed or Rolled Back

Use the DBA_2PC_PENDING view to find a node that has either committed or rolled back the transaction. If you can find a node that has already resolved the transaction, then you can follow the action taken at that node.

Look For Transaction Comments

See if any information is given in the TRAN_COMMENT column of DBA_2PC_PENDING for the distributed transaction. Comments are included in the COMMENT parameter of the COMMIT command.

For example, an in-doubt distributed transaction's comment can indicate the origin of the transaction and what type of transaction it is:

```
COMMIT COMMENT 'Finance/Accts_pay/Trans_type 10B';
```

Look For Transaction Advice

See if any information is given in the ADVICE column of DBA_2PC_PENDING for the distributed transaction. An application can prescribe advice about whether to force the commit or force the rollback of separate parts of a distributed transaction with the ADVISE parameter of the SQL command ALTER SESSION.

The advice sent during the prepare phase to each node is the advice in effect at the time the most recent DML statement executed at that database in the current transaction.

For example, consider a distributed transaction that moves an employee record from the EMP table at one node to the EMP table at another node. The transaction can protect the record—even when administrators independently force the in-doubt transaction at each node—by including the following sequence of SQL statements:

```
ALTER SESSION ADVISE COMMIT;
INSERT INTO emp@hq ... ;    /*advice to commit at HQ */
ALTER SESSION ADVISE ROLLBACK;
DELETE FROM emp@sales ... ; /*advice to roll back at SALES*/

ALTER SESSION ADVISE NOTHING;
```

If you manually force the in-doubt transaction, the worst that can happen is that each node has a copy of the employee record; the record cannot disappear.

Manually Overriding In-Doubt Transactions

Use the COMMIT or ROLLBACK statement with the FORCE option and a text string that indicates either the local or global transaction ID of the in-doubt transaction to commit.

Note: In all examples, the transaction is committed or rolled back on the local node, and the local pending transaction table records a value of forced commit or forced abort for the STATE column of this transaction's row.

This section contains the following topics:

- Manually Committing an In-Doubt Transaction
- Manually Rolling Back an In-Doubt Transaction

Manually Committing an In-Doubt Transaction

Before attempting to commit the transaction, ensure that you have the proper privileges. Note the following requirements:

If the transaction was committed by...	Then you must have this privilege...
You	FORCE TRANSACTION
Another user	FORCE ANY TRANSACTION

Committing Using Only the Transaction ID

The following SQL statement is the command to commit an in-doubt transaction:

```
COMMIT FORCE 'transaction_id';
```

The variable *transaction_id* is the identifier of the transaction as specified in either the LOCAL_TRAN_ID or GLOBAL_TRAN_ID columns of the DBA_2PC_PENDING data dictionary view.

For example, assume that you query DBA_2PC_PENDING and determine the local transaction ID for a distributed transaction:

```
LOCAL_TRAN_ID          1.45.13
```

You then issue the following SQL statement to force the commit of this in-doubt transaction:

```
COMMIT FORCE '1.45.13';
```

Committing Using an SCN

Optionally, you can specify the SCN for the transaction when forcing a transaction to commit. This feature allows you to commit an in-doubt transaction with the SCN assigned when it was committed at other nodes.

Consequently, you maintain the synchronized commit time of the distributed transaction even if there is a failure. Specify an SCN only when you can determine the SCN of the same transaction already committed at another node.

For example, assume you want to manually commit a transaction with the following global transaction ID:

```
SALES.ACME.COM.55d1c563.1.93.29
```

First, query the DBA_2PC_PENDING view of a remote database also involved with the transaction in question. Note the SCN used for the commit of the transaction at that node. Specify the SCN when committing the transaction at the local node. For example, if the SCN is 829381993, issue:

```
COMMIT FORCE 'SALES.ACME.COM.55d1c563.1.93.29', 829381993;
```

Manually Rolling Back an In-Doubt Transaction

Before attempting to roll back the in-doubt distributed transaction, ensure that you have the proper privileges. Note the following requirements:

If the transaction was committed by...	Then you must have this privilege...
You	FORCE TRANSACTION
Another user	FORCE ANY TRANSACTION

The following SQL statement is the command to roll back an in-doubt transaction:

```
ROLLBACK FORCE 'transaction_id';
```

The variable *transaction_id* is the identifier of the transaction as specified in either the LOCAL_TRAN_ID or GLOBAL_TRAN_ID columns of the DBA_2PC_PENDING data dictionary view.

For example, to roll back the in-doubt transaction with the local transaction ID of 2.9.4, use the following statement:

```
ROLLBACK FORCE '2.9.4';
```

Note: You cannot roll back an in-doubt transaction to a savepoint.

Purging Pending Rows from the Data Dictionary

Before RECO recovers an in-doubt transaction, the transaction appears in DBA_2PC_PENDING.STATE as COLLECTING, COMMITTED, or PREPARED. If you

force an in-doubt transaction using COMMIT FORCE or ROLLBACK FORCE, then the states FORCED COMMIT or FORCED ROLLBACK may appear.

Automatic recovery normally deletes entries in these states. The only exception is when recovery discovers a forced transaction that is in a state inconsistent with other sites in the transaction; in this case, the entry can be left in the table and the MIXED column in DBA_2PC_PENDING has a value of YES.

If automatic recovery is not possible because a remote database has been permanently lost, then recovery cannot identify the re-created database because it receives a new database ID when it is re-created. In this case, you must use the PURGE_LOST_DB_ENTRY procedure in the DBMS_TRANSACTION package to clean up the entries. The entries do not hold up database resources, so there is no urgency in cleaning them up.

Executing the PURGE_LOST_DB_ENTRY Procedure

To manually remove an entry from the data dictionary, use the following syntax (where *trans_id* is the identifier for the transaction):

```
DBMS_TRANSACTION.PURGE_LOST_DB_ENTRY('trans_id');
```

For example, to purge pending distributed transaction 1.44.99, enter the following command in SQL*Plus:

```
EXECUTE DBMS_TRANSACTION.PURGE_LOST_DB_ENTRY('1.44.99');
```

Execute this procedure only if significant reconfiguration has occurred so that automatic recovery cannot resolve the transaction. Examples include:

- Total loss of the remote database
- Reconfiguration in software resulting in loss of two-phase commit capability
- Loss of information from an external transaction coordinator such as a TPMonitor

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for more information about the DBMS_TRANSACTION package.

Determining When to Use DBMS_TRANSACTION

The following tables indicates what the various states indicate about the distributed transaction what the administrator's action should be:

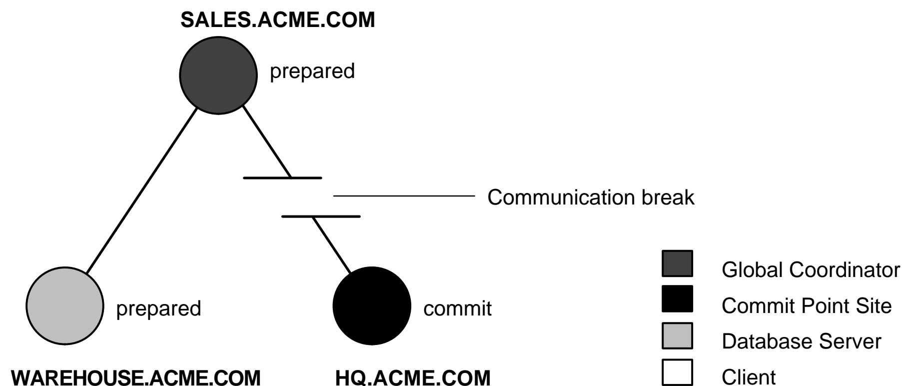
STATE Column	State of Global Transaction	State of Local Transaction	Normal Action	Alternative Action
Collecting	Rolled back	Rolled back	None	PURGE_LOST_DB_ENTRY (only if autorecovery cannot resolve transaction)
Committed	Committed	Committed	None	PURGE_LOST_DB_ENTRY (only if autorecovery cannot resolve transaction)
Prepared	Unknown	Prepared	None	Force commit or rollback
Forced commit	Unknown	Committed	None	PURGE_LOST_DB_ENTRY (only if autorecovery cannot resolve transaction)
Forced rollback	Unknown	Rolled back	None	PURGE_LOST_DB_ENTRY (only if autorecovery cannot resolve transaction)
Forced commit	Mixed	Committed	Manually remove inconsistencies then use PURGE_MIXED	
Forced rollback	Mixed	Rolled back	Manually remove inconsistencies then use PURGE_MIXED	

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for more information about the DBMS_TRANSACTION package.

Manually Committing an In-Doubt Transaction: Example

The following example, illustrated in Figure 5–1, shows a failure during the commit of a distributed transaction. It explains how to go about gaining information before manually forcing the commit or rollback of the local portion of an in-doubt distributed transaction.

Figure 5–1 Example of an In-Doubt Distributed Transaction



In this failure case, the prepare phase completes. During the commit phase, however, the commit point site's commit confirmation never reaches the global coordinator, even though the commit point site committed the transaction.

You are the WAREHOUSE database administrator. The inventory data locked because of the in-doubt transaction is critical to other transactions. The data cannot be accessed, however, because the locks must be held until the in-doubt transaction either commits or rolls back. Furthermore, you understand that the communication link between sales and headquarters cannot be resolved immediately.

Therefore, you decide to manually force the local portion of the in-doubt transaction using the following steps:

1. Record user feedback.
2. Query the local DBA_2PC_PENDING view to obtain the global transaction ID and get other information about the in-doubt transaction.
3. Query the local DBA_2PC_NEIGHBORS view to begin tracing the session tree so that you can find a node that resolved the in-doubt transaction.
4. Check the mixed outcome flag after normal communication is re-established.

The following sections explain each step in detail for this example:

- Step 1: Record User Feedback
- Step 2: Query DBA_2PC_PENDING
- Step 3: Query DBA_2PC_NEIGHBORS on Local Node

- Step 4: Querying Data Dictionary Views on All Nodes
- Step 5: Commit the In-Doubt Transaction
- Step 6: Check for Mixed Outcome Using DBA_2PC_PENDING

Step 1: Record User Feedback

The users of the local database system that conflict with the locks of the in-doubt transaction receive the following error message:

```
ORA-01591: lock held by in-doubt distributed transaction 1.21.17
```

In this case, 1.21.17 is the local transaction ID of the in-doubt distributed transaction. You should request and record this ID number from users that report problems to identify which in-doubt transactions should be forced.

Step 2: Query DBA_2PC_PENDING

After connecting with SQL*Plus to WAREHOUSE, query the local DBA_2PC_PENDING data dictionary view to gain information about the in-doubt transaction:

```
CONNECT sys/sys_pwd@warehouse.acme.com
SELECT * FROM sys.dba_2pc_pending WHERE local_tran_id = '1.21.17';
```

Oracle returns the following information:

```
Column Name            Value
---------------------- --------------------------------------
LOCAL_TRAN_ID          1.21.17
GLOBAL_TRAN_ID         SALES.ACME.COM.55d1c563.1.93.29
STATE                  prepared
MIXED                  no
ADVICE
TRAN_COMMENT           Sales/New Order/Trans_type 10B
FAIL_TIME              31-MAY-91
FORCE_TIME
RETRY_TIME             31-MAY-91
OS_USER                SWILLIAMS
OS_TERMINAL            TWA139:
HOST                   system1
DB_USER                SWILLIAMS
COMMIT#
```

Determining the Global Transaction ID

The global transaction ID is the common transaction ID that is the same on every node for a distributed transaction. It is of the form:

```
global_database_name.hhhhhhhh.local_transaction_id
```

where:

`global_database_name`	is the database name of the global coordinator.
`hhhhhhh`	is the internal database identifier of the global coordinator (in hexadecimal).
`local_transaction_id`	is the corresponding local transaction ID assigned on the global coordinator.

Note that the last portion of the global transaction ID and the local transaction ID match at the global coordinator. In the example, you can tell that WAREHOUSE is *not* the global coordinator because these numbers do not match:

```
LOCAL_TRAN_ID          1.21.17
GLOBAL_TRAN_ID         ... 1.93.29
```

Determining the State of the Transaction

The transaction on this node is in a prepared state:

```
STATE                  prepared
```

Therefore, WAREHOUSE waits for its coordinator to send either a commit or a rollback request.

Looking For Comments or Advice

The transaction's comment or advice can include information about this transaction. If so, use this comment to your advantage. In this example, the origin and transaction type is in the transaction's comment:

```
TRAN_COMMENT           Sales/New Order/Trans_type 10B
```

This information can reveal something that helps you decide whether to commit or rollback the local portion of the transaction. If useful comments do not accompany an in-doubt transaction, you must complete some extra administrative work to trace the session tree and find a node that has resolved the transaction.

Step 3: Query DBA_2PC_NEIGHBORS on Local Node

The purpose of this step is to climb the session tree so that you find coordinators, eventually reaching the global coordinator. Along the way, you may find a coordinator that has resolved the transaction. If not, you can eventually work your way to the commit point site, which will always have resolved the in-doubt transaction. To trace the session tree, query the DBA_2PC_NEIGHBORS view on each node.

In this case, you query this view on the WAREHOUSE database:

```
CONNECT sys/sys_pwd@warehouse.acme.com
SELECT * FROM dba_2pc_neighbors
  WHERE local_tran_id = '1.21.17'
  ORDER BY sess#, in_out;

Column Name            Value
---------------------- --------------------------------------
LOCAL_TRAN_ID          1.21.17
IN_OUT                 in
DATABASE               SALES.ACME.COM
DBUSER_OWNER           SWILLIAMS
INTERFACE              N
DBID                   000003F4
SESS#                  1
BRANCH                 0100
```

Obtaining Database Role and Database Link Information

The DBA_2PC_NEIGHBORS view provides information about connections associated with an in-doubt transaction. Information for each connection is different, based on whether the connection is *inbound* (IN_OUT = in) or *outbound* (IN_OUT = out):

IN_OUT	Meaning	DATABASE	DBUSER_OWNER
in	Your node is a server of another node.	Lists the name of the client database that connected to your node.	Lists the local account for the database link connection that corresponds to the in-doubt transaction.
out	Your node is a client of other servers.	Lists the name of the database link that connects to the remote node.	Lists the owner of the database link for the in-doubt transaction.

In this example, the IN_OUT column reveals that the WAREHOUSE database is a server for the SALES client, as specified in the DATABASE column:

```
IN_OUT                 in
DATABASE               SALES.ACME.COM
```

The connection to WAREHOUSE was established through a database link from the SWILLIAMS account, as shown by the DBUSER_OWNER column:

```
DBUSER_OWNER           SWILLIAMS
```

Determining the Commit Point Site

Additionally, the INTERFACE column tells whether the local node or a subordinate node is the commit point site:

```
INTERFACE              N
```

Neither WAREHOUSE nor any of its descendants is the commit point site, as shown by the INTERFACE column.

Step 4: Querying Data Dictionary Views on All Nodes

At this point, you can contact the administrator at the located nodes and ask each person to repeat Steps 2 and 3 using the global transaction ID.

> **Note:** If you can directly connect to these nodes with another network, you can repeat Steps 2 and 3 yourself.

For example, the following results are returned when Steps 2 and 3 are performed at SALES and HQ.

Checking the Status of Pending Transactions at SALES

At this stage, the SALES administrator queries the DBA_2PC_PENDING data dictionary view:

```
SQL> CONNECT sys/sys_pwd@sales.acme.com
SQL> SELECT * FROM sys.dba_2pc_pending
   > WHERE global_tran_id = 'SALES.ACME.COM.55d1c563.1.93.29';

Column Name            Value
---------------------- --------------------------------------
LOCAL_TRAN_ID          1.93.29
```

```
GLOBAL_TRAN_ID          SALES.ACME.COM.55d1c563.1.93.29
STATE                   prepared
MIXED                   no
ADVICE
TRAN_COMMENT            Sales/New Order/Trans_type 10B
FAIL_TIME               31-MAY-91
FORCE_TIME
RETRY_TIME              31-MAY-91
OS_USER                 SWILLIAMS
OS_TERMINAL             TWA139:
HOST                    system1
DB_USER                 SWILLIAMS
COMMIT#
```

Determining the Coordinators and Commit Point Site at SALES

Next, the SALES administrator queries DBA_2PC_NEIGHBORS to determine the global and local coordinators as well as the commit point site:

```
SELECT * FROM dba_2pc_neighbors
   WHERE global_tran_id = 'SALES.ACME.COM.55d1c563.1.93.29'
   ORDER BY sess#, in_out;
```

This query returns three rows:

- The connection to WAREHOUSE
- The connection to HQ
- The connection established by the user

Reformatted information corresponding to the rows for the WAREHOUSE connection appears below:

```
Column Name             Value
----------------------  --------------------------------------
LOCAL_TRAN_ID           1.93.29
IN_OUT                  OUT
DATABASE                WAREHOUSE.ACME.COM
DBUSER_OWNER            SWILLIAMS
INTERFACE               N
DBID                    55d1c563
SESS#                   1
BRANCH                  1
```

Reformatted information corresponding to the rows for the HQ connection appears below:

```
Column Name            Value
---------------------- --------------------------------------
LOCAL_TRAN_ID          1.93.29
IN_OUT                 OUT
DATABASE               HQ.ACME.COM
DBUSER_OWNER           ALLEN
INTERFACE              C
DBID                   00000390
SESS#                  1
BRANCH                 1
```

The information from the previous query reveals the following:

- SALES is the global coordinator because the local transaction ID and global transaction ID match.
- Two outbound connections are established from this node, but no inbound connections. SALES is not the server of another node.
- HQ or one of its servers is the commit point site.

Checking the Status of Pending Transactions at HQ:

At this stage, the HQ administrator queries the DBA_2PC_PENDING data dictionary view:

```
SELECT * FROM dba_2pc_pending
   WHERE global_tran_id = 'SALES.ACME.COM.55d1c563.1.93.29';

Column Name            Value
---------------------- --------------------------------------
LOCAL_TRAN_ID          1.45.13
GLOBAL_TRAN_ID         SALES.ACME.COM.55d1c563.1.93.29
STATE                  COMMIT
MIXED                  NO
ACTION
TRAN_COMMENT           Sales/New Order/Trans_type 10B
FAIL_TIME              31-MAY-91
FORCE_TIME
RETRY_TIME             31-MAY-91
OS_USER                SWILLIAMS
OS_TERMINAL            TWA139:
HOST                   SYSTEM1
DB_USER                SWILLIAMS
COMMIT#                129314
```

At this point, you have found a node that resolved the transaction. As the view reveals, it has been committed and assigned a commit ID number:

```
STATE                COMMIT
COMMIT#              129314
```

Therefore, you can force the in-doubt transaction to commit at your local database. It is a good idea to contact any other administrators you know that could also benefit from your investigation.

Step 5: Commit the In-Doubt Transaction

You contact the administrator of the SALES database, who manually commits the in-doubt transaction using the global ID:

```
SQL> CONNECT sys/sys_pwd@sales.acme.com
SQL> COMMIT FORCE 'SALES.ACME.COM.55d1c563.1.93.29';
```

As administrator of the WAREHOUSE database, you manually commit the in-doubt transaction using the global ID:

```
SQL> CONNECT sys/sys_pwd@warehouse.acme.com
SQL> COMMIT FORCE 'SALES.ACME.COM.55d1c563.1.93.29';
```

Step 6: Check for Mixed Outcome Using DBA_2PC_PENDING

After you manually force a transaction to commit or roll back, the corresponding row in the pending transaction table remains. The state of the transaction is changed depending on how you forced the transaction.

Every Oracle8*i* database has a *pending transaction table*. This is a special table that stores information about distributed transactions as they proceed through the two-phase commit phases. You can query a database's pending transaction table through the DBA_2PC_PENDING data dictionary view (see Table 5–1, "DBA_2PC_PENDING").

Also of particular interest in the pending transaction table is the mixed outcome flag as indicated in DBA_2PC_PENDING.MIXED. You can make the wrong choice if a pending transaction is forced to commit or roll back. For example, the local administrator rolls back the transaction, but the other nodes commit it. Incorrect decisions are detected automatically, and the damage flag for the corresponding pending transaction's record is set (MIXED=yes).

The RECO (Recoverer) background process uses the information in the pending transaction table to finalize the status of in-doubt transactions. You can also use the

information in the pending transaction table to manually override the automatic recovery procedures for pending distributed transactions.

All transactions automatically resolved by RECO are automatically removed from the pending transaction table. Additionally, all information about in-doubt transactions correctly resolved by an administrator (as checked when RECO reestablishes communication) are automatically removed from the pending transaction table. However, all rows resolved by an administrator that result in a mixed outcome across nodes remain in the pending transaction table of all involved nodes until they are manually deleted.

Simulating Distributed Transaction Failure

You can force the failure of a distributed transaction for the following reasons:

- To observe RECO automatically resolving the local portion of the transaction.
- To practice manually resolving in-doubt distributed transactions and observing the results.

The following sections describes the features available and the steps necessary to perform such operations.

Forcing a Distributed Transaction to Fail

You can include comments in the COMMENT parameter of the COMMIT statement. To intentionally induce a failure during the two-phase commit phases of a distributed transaction, include the following comment in the COMMENT parameter:

```
COMMIT COMMENT 'ORA-2PC-CRASH-TEST-n';
```

where *n* is one of the following integers:

n	Effect
1	Crash commit point site after collect
2	Crash non-commit point site after collect
3	Crash before prepare (non-commit point site)
4	Crash after prepare (non-commit point site)
5	Crash commit point site before commit
6	Crash commit point site after commit
7	Crash non-commit point site before commit
8	Crash non-commit point site after commit
9	Crash commit point site before forget
10	Crash non-commit point site before forget

For example, the following statement returns the following messages if the local commit point strength is greater than the remote commit point strength and both nodes are updated:

```
COMMIT COMMENT 'ORA-2PC-CRASH-TEST-7';

ORA-02054: transaction 1.93.29 in-doubt
ORA-02059: ORA-CRASH-TEST-7 in commit comment
```

At this point, the in-doubt distributed transaction appears in the DBA_2PC_PENDING view. If enabled, RECO automatically resolves the transaction.

Disabling and Enabling RECO

The RECO background process of an Oracle8*i* instance automatically resolves failures involving distributed transactions. At exponentially growing time intervals, the RECO background process of a node attempts to recover the local portion of an in-doubt distributed transaction.

RECO can use an existing connection or establish a new connection to other nodes involved in the failed transaction. When a connection is established, RECO automatically resolves all in-doubt transactions. Rows corresponding to any resolved in-doubt transactions are automatically removed from each database's pending transaction table.

You can enable and disable RECO using the ALTER SYSTEM statement with the ENABLE/DISABLE DISTRIBUTED RECOVERY options. For example, you can temporarily disable RECO to force the failure of a two-phase commit and manually resolve the in-doubt transaction.

The following statement disables RECO:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

Alternatively, the following statement enables RECO so that in-doubt transactions are automatically resolved:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

Note: Single-process instances (for example, a PC running MS-DOS) have no separate background processes, and therefore no RECO process. Therefore, when a single-process instance that participates in a distributed system is started, you must manually enable distributed recovery using the statement above.

See Also: Your Oracle operating system-specific documentation for more information about distributed transaction recovery for single-process instances.

Managing Read Consistency

An important restriction exists in Oracle's implementation of distributed read consistency. The problem arises because each system has its own SCN, which you can view as the database's internal timestamp. The Oracle database server uses the SCN to decide which version of data is returned from a query.

The SCNs in a distributed transaction are synchronized at the end of each remote SQL statement and at the start and end of each transaction. Between two nodes that have heavy traffic and especially distributed updates, the synchronization is frequent. Nevertheless, no practical way exists to keep SCNs in a distributed system absolutely synchronized: a window always exists in which one node may have an SCN that is somewhat in the past with respect to the SCN of another node.

Because of the SCN gap, you can execute a query that uses a slightly old snapshot, so that the most recent changes to the remote database are not seen. In accordance with read consistency, a query can therefore retrieve consistent, but out-of-date data. Note that all data retrieved by the query will be from the old SCN, so that if a

locally executed update transaction updates two tables at a remote node, then data selected from both tables in the next remote access contain data prior to the update.

One consequence of the SCN gap is that two consecutive SELECT statements can retrieve different data even though no DML has been executed between the two statements. For example, you can issue an update statement and then commit the update on the remote database. When you issue a SELECT statement on a view based on this remote table, the view does not show the update to the row. The next time that you issue the SELECT statement, the update is present.

You can use the following techniques to ensure that the SCNs of the two machines are synchronized just before a query:

- Because SCNs are synchronized at the end of a remote query, precede each remote query with a dummy remote query to the same site, for example, SELECT * FROM dual@remote.
- Because SCNs are synchronized at the start of every remote transaction, commit or roll back the current transaction before issuing the remote query.

Part III

Heterogeneous Services Concepts and Administration

6

Oracle Heterogeneous Services Concepts

This chapter describes the basic concepts of the Oracle Heterogeneous Services. Topics include:

- What is Heterogeneous Services?
- Types of Heterogeneous Services
- Heterogeneous Services Process Architecture
- Architecture of the Heterogeneous Services Data Dictionary

See Also: *Getting to Know Oracle8i* for information about features new to the current release.

What is Heterogeneous Services?

Heterogeneous Services (HS) is an integrated component within the Oracle8*i* database server, and provides the generic technology for accessing non-Oracle systems from the Oracle database server. Heterogeneous Services enables you to use:

- Oracle SQL statements to transparently access data stored in non-Oracle systems as if the data resided within an Oracle database server.
- Oracle procedure calls to transparently access non-Oracle systems, services, or application programming interfaces (APIs) from your Oracle distributed environment.

To access a non-Oracle system, you need to use a complementary Heterogeneous Services agent. You can connect to a non-Oracle system through an *Oracle Transparent Gateway*, which is an agent that is tailored specifically for the system that you are accessing. If you connect to the non-Oracle system using generic connectivity through the ODBC or OLE DB interfaces, however, then the agent is an executable that it automatically installed with the Oracle database server.

> **Note:** The phrase *non-Oracle system* denotes both non-Oracle data stores (or databases) that are accessed using SQL, and systems that are accessed procedurally.

Database Links to a Non-Oracle System

Heterogeneous Services makes a non-Oracle system appear as a remote Oracle database server. To access or manipulate tables or to execute procedures in the non-Oracle system, create a database link that specifies the connect descriptor for the non-Oracle database. Use the following syntax to create a link to a non-Oracle system (variables in italics):

```
CREATE DATABASE LINK link_name
  CONNECT TO user IDENTIFIED BY password
  USING 'non_oracle_system';
```

If a non-Oracle system is referenced, then HS translates the SQL statement or PL/SQL remote procedure call into the appropriate statement at the non-Oracle system.

You can access tables and procedures at the non-Oracle system by qualifying the tables and procedures with the database link. This operation is identical to

accessing tables and procedures at a remote Oracle database server. Consider the following example that accesses a non-Oracle system through a database link:

```
SELECT * FROM EMP@non_oracle_system;
```

Heterogeneous Services translates the Oracle SQL statement into the SQL dialect of the target system and then executes the translated SQL statement at the non-Oracle system.

Heterogeneous Services Agents

While Heterogeneous Services provides the generic technology in the Oracle8*i* server, a Heterogeneous Services agent is required to access a particular non-Oracle system such as Informix or Sybase. Oracle Corporation provides Heterogeneous Services agents in the form of Oracle Transparent Gateways version 8 and higher.

Oracle Transparent Gateways is one family of products that uses the Heterogeneous Services. Generic connectivity is another family of agents based on Heterogeneous Services. The phrase *Heterogeneous Services agents* denotes all products that are based on Heterogeneous Services, including Oracle Transparent Gateways and the generic connectivity agents.

> **See Also:** Chapter 7, "Managing Oracle Heterogeneous Services Using Transparent Gateways" to learn to configure Oracle Transparent Gateway agents, and Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity" to learn to configure generic connectivity agents.

Types of Heterogeneous Services

Heterogeneous Services provides the following services:

- Transaction Service
- SQL Service

Transaction Service

The *transaction service* allows non-Oracle systems to be integrated into Oracle transactions and sessions. Users transparently set up an authenticated session in the non-Oracle system when it is accessed for the first time over a database link within an Oracle user session. At the end of the Oracle user session, the session is transparently closed 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 system to transparently coordinate the distributed transaction. If the non-Oracle system supports some but not all aspects of the two-phase commit protocol, then the Oracle database server typically supports distributed transactions with the non-Oracle system (with possible restrictions).

The SQL service uses the transaction service. Oracle's object transaction service uses agents that implement only the transaction service.

See Also: "Using the Transaction Service Views" on page 7-12 for more information on heterogeneous distributed transactions.

SQL Service

The *SQL service* transparently accesses the non-Oracle system using SQL. If an application's SQL request requires data from a non-Oracle system, HS performs the following steps:

1. Translates the Oracle SQL request into an equivalent SQL request understood by the non-Oracle system.
2. Accesses the non-Oracle data.
3. Makes the data available to the Oracle database server for post-processing.

The SQL service provides capabilities to:

- Transform Oracle's SQL into a SQL dialect understood by the non-Oracle system
- Transform SQL requests on Oracle's data dictionary tables to requests on the non-Oracle system's data dictionary
- Map non-Oracle system datatypes onto Oracle's datatypes

Heterogeneous Services Process Architecture

The basic architecture for HS involves a client accessing an Oracle database server, which in turn sends a request to an agent residing on a non-Oracle server. This section contains the following topics:

- Transparent Gateways
- Generic Connectivity

Transparent Gateways

A transparent gateway, also called an *agent*, is required to access a specific non-Oracle system from an Oracle8*i* server. The Oracle database server communicates with the agent, which in turn communicates with the non-Oracle system. Unless you are using the generic connectivity feature of Oracle, you need to purchase and configure a system-specific agent to connect to the non-Oracle data store. For example, a dBASE data store requires a dBASE agent, and a Sybase data store requires a Sybase agent.

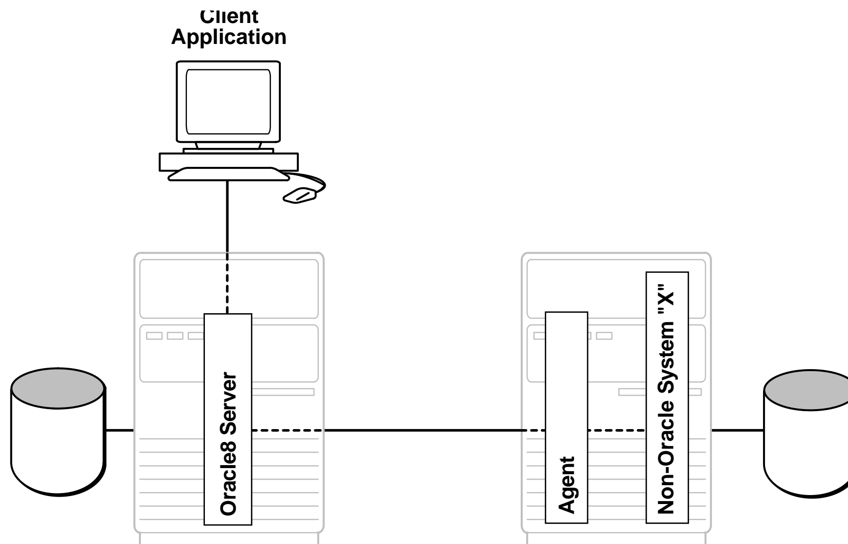
With transparent gateways, you can easily access data anywhere in a distributed system without knowing the location of the data or how it is stored. The term *transparent* indicates that the network, location, operating system, data storage format, and access methods are hidden from the user application.

As illustrated in Figure 6–1, agents can reside on the same machine as the non-Oracle system, but are not required to. The agent can also reside on the same machine as the Oracle8*i* server, or even on a third machine. The agent must be:

- Accessible by the Oracle8*i* server through Net8.
- Able to access the non-Oracle system using a non-Oracle system-specific communication mechanism.

When a user session accesses a non-Oracle system through a database link on the Oracle8*i* server, a Net8 Listener starts an agent process. This agent process remains running until the user session is disconnected or until the database link is explicitly closed.

Figure 6–1 Accessing Heterogeneous Non-Oracle Systems



Generic Connectivity

If you connect to a non-Oracle data store using generic connectivity, the process architecture is essentially the same as in the non-generic case. The difference is that Oracle provides generic ODBC and OLE DB agents with the server—no transparent gateway is required. As long as the non-Oracle system supports these protocols, you can use them without purchasing a system-specific agent. To make the generic agents work, however, you must also configure a driver that can interface with the agent.

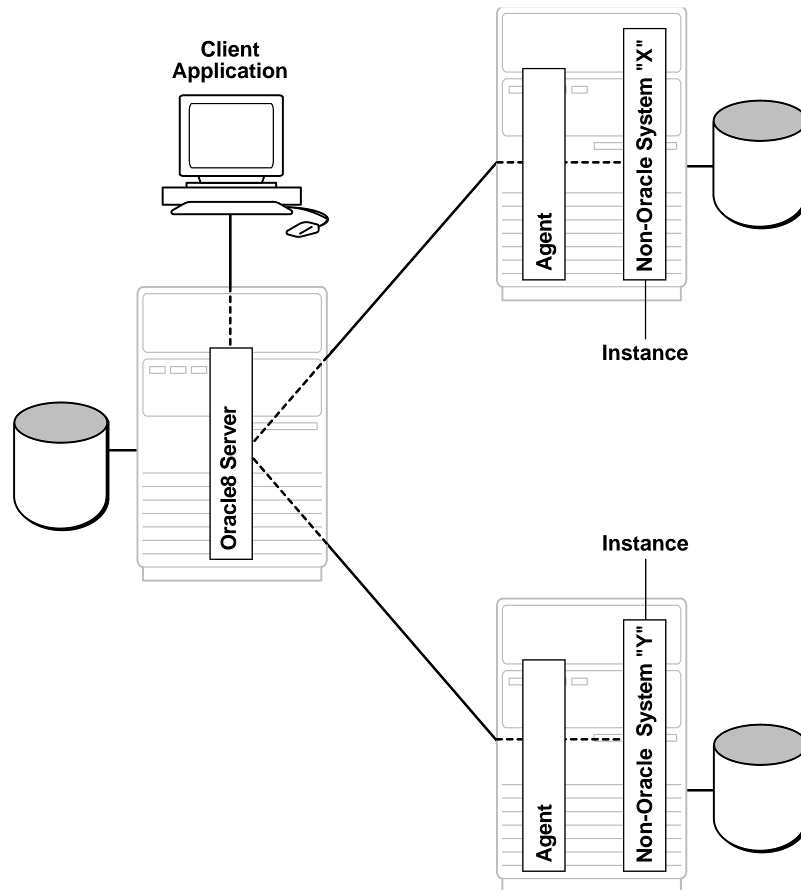
> **See Also:** Chapter 8, "Managing Heterogeneous Services Using Generic Connectivity" for information about installation and configuration of generic connectivity.

Architecture of the Heterogeneous Services Data Dictionary

Each non-Oracle system accessed from an Oracle8*i* server is considered a non-Oracle system *instance and class*. You can access multiple non-Oracle systems from the same Oracle8*i* server, as illustrated in Figure 6–2.

The Oracle8*i* server must know the non-Oracle system capabilities (SQL translations, data dictionary translations) for each non-Oracle system that it accesses. This information is stored in the Oracle8*i* data dictionary.

Figure 6–2 Accessing Multiple Non-Oracle Instances



Classes and Instances

If this information were stored separately for each non-Oracle system you access, the amount of stored data dictionary information could become large and sometimes redundant. For example, when you access three non-Oracle system instances of the same type, the same capabilities, SQL translations, and data dictionary translations are stored.

To avoid unnecessary redundancy, Oracle organizes this data by *classes and instances* in the data dictionary. A *class* defines a type of non-Oracle system. An *instance*

defines specializations of a class for a specific non-Oracle system. Note that instance information takes precedence over class information, and class information takes precedence over server-supplied defaults.

If you access multiple non-Oracle systems of the same class, then you may want to set certain information, such as initialization parameters, at the instance level. Heterogeneous Services stores both class and instance information. Multiple instances can share the same class information, but each non-Oracle system instance has its own instance information.

Consider a case where the Oracle8*i* server accesses three instances of type Megabase release 5 and two instances of Megabase release 6. Suppose Megabase release 5 and Megabase release 6 have different capabilities. The data dictionary contains two class definitions, one for release 5 and one for release 6, and five instance definitions.

Data Dictionary Views

The Heterogeneous Services data dictionary views contain information about:

- Names of instances and classes uploaded into the Oracle8*i* 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
- Distributed external procedures accessible from the Oracle8*i* server

You can access information from the Oracle data dictionary used fixed views. The views can be divided into four main types:

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

See Also: "Using the Heterogeneous Services Data Dictionary Views" on page 7-9 to learn how to use these views, and Appendix B, "Data Dictionary Views Available Through Heterogeneous Services" for reference information on the views.

7

Managing Oracle Heterogeneous Services Using Transparent Gateways

This chapter teaches you how to maintain a heterogeneous distributed environment when using a transparent gateway. Topics include:

- Setting Up Access to Non-Oracle Systems
- Registering Agents
- Using the Heterogeneous Services Data Dictionary Views
- Using the Heterogeneous Services Dynamic Performance Views
- Using the DBMS_HS Package

Setting Up Access to Non-Oracle Systems

This section explains the generic steps to configure access to a non-Oracle system. Please see the *Installation and User's Guide* for your agent for more installation information. The instructions for configuring your agent may slightly differ from the following.

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 Oracle8*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 Oracle8*i* server starts an agent process through the Net8 listener. For the Oracle8*i* server to be able to connect to the agent, you must:

1. Set up a Net8 service name for the agent that can be used by the Oracle8*i* server. The Net8 service name descriptor includes protocol-specific information needed to access the Net8 listener. The service name descriptor must include the (HS=OK) clause to make sure the connection uses Oracle8*i* Heterogeneous Services.
2. Set up the listener to listen for incoming request from the Oracle8*i* server and spawn HS agents. Modify the `listener.ora` file so that the listener can start Heterogeneous Services agents, and then restart the listener.

A Sample Entry for a Net8 Service Name

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

```
MegaBase6_sales= (DESCRIPTION=
                     (ADDRESS=(PROTOCOL=tcp)
                              (HOST=dlsun206)
                              (PORT=1521))

                     (CONNECT_DATA = (SID=SalesDB))

                     (HS = OK))
```

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. See the *Installation and User's Guide* for your agent for more information about how to define the Net8 service name.

A Sample Listener Entry

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/8.1.3)
                  (PROGRAM=tg4mb80)
      )
  )
```

The value associated with PROGRAM keyword defines the name of the agent executable. The agent executable must reside in the `$ORACLE_HOME/bin` directory. Typically, you use SID_NAME to define the initialization parameter file for the agent.

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 command is the Net8 service name.

For example, to create a database link to the SALES database on a MegaBase release 6 server, you might issue:

```
CREATE DATABASE LINK sales
USING 'MegaBase6_sales';
```

See Also: Chapter 2, "Managing a Distributed Database" for more information on creating database links.

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 HS 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 Oracle8*i* data dictionary automatically the first time they are accessed. Some agent vendors may provide scripts, however, that you must run on the Oracle8*i* server.

See Also: "Using the Heterogeneous Services Data Dictionary Views" on page 7-9.

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 release 8.1, self-registration occurs only when the HS_AUTOREGISTER initialization parameter is set to TRUE (default).

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 HS 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 HS connection. These three sets of information are collectively called *HS configuration data*:

HS Configuration Data	Description
HS 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 7-8.

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 HS 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 HS agent at a remote site can be upgraded to a new version or patched, requiring modification of the HS configuration data.
- A 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 HS 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 HS 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 Oracle8*i* servers.
- Upload capability definitions and data dictionary translations, if available, from an HS 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:** "Using the Heterogeneous Services Data Dictionary Views" on page 7-9 to learn how to use the HS 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 HS 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 MegaBase8.0.4 to class MegaBase8.1.3, 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 HS 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 HS connections.

> **See Also:** *Oracle8i 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 HS data dictionary views are not useful sources of information. Nevertheless, Oracle still requires information about the class and instance of each agent. If agent self-registration is disabled, Oracle stores this information in local memory.

Using the Heterogeneous Services Data Dictionary Views

You can use the HS 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 HS 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 7–1 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 Oracle8*i* server
HS_FDS_INST	General	Instances accessible from this Oracle8*i* server
HS_INST_CAPS	Transaction service, SQL service	Capabilities for each instance
HS_INST_DD	SQL service	Data dictionary translations for each instance
HS_INST_INIT	General	Initialization parameters for each instance

Like all Oracle data dictionary tables, these views are read-only. Do not use SQL to change the content of any of the underlying tables. To make changes to any of the underlying tables, use the procedures available in the DBMS_HS package.

See Also:

- "Architecture of the Heterogeneous Services Data Dictionary" on page 6-6 for more information about classes and instances
- *Oracle8i Reference* for information about the HS views
- "Using the DBMS_HS Package" on page 7-17 for more information about the DBMS_HS package

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 HS 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 HS 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 HS 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 HS_INST_INIT	Information about the HS initialization parameters

For example, you can access both MegaBase release 5 and release 6 from an Oracle8*i* server. After accessing the agents for the first time, the information uploaded into the Oracle8*i* server could look like:

```
SQL> SELECT * FROM hs_fds_class;

FDS_CLASS_NAME        FDS_CLASS_COMMENTS             FDS_CLASS_ID
--------------------- ------------------------------ ------------
MegaBase5             Uses ODBC HS driver, R1.0                 1
MegaBase6             Uses ODBC HS driver, R1.0                21
```

Two classes are uploaded: one class to access MegaBase release 5 servers and one class to access MegaBase release 6 servers. The data dictionary in the Oracle8*i* server now contains capability information, SQL translations, and data dictionary translations for both MegaBase5 and MegaBase6.

In addition to this information, the Oracle8*i* 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 and HS_INST_CAPS capability 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.

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 can not participate in the two-phase commit protocol.

Commit Confirm (CC) — The non-Oracle system can participate in distributed transactions. It can participate in Oracle's two-phase commit protocol but only as the Commit Point Site. That is, it can *not* 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 Oracle'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 can *not* remember the outcome of a particular transaction if asked to by the global coordinator.
Two-Phase Commit Confirm	The non-Oracle system can participate in distributed transactions. It can participate in Oracle'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 views HS_CLASS_CAPS and HS_INST_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 Oracle8*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.

> **See Also:** Chapter 4, "Distributed Transactions Concepts" for more information about distributed transactions.

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 Oracle8*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 Oracle8*i* server, HS 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 HS views whose names are suffixed with _DD.

For example, the following SELECT statement transforms into a MegaBase query that retrieves information about EMP tables from the MegaBase 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, HS causes it to appear as if the data dictionary table is available, but the table contains no information.

To retrieve information for which Oracle8*i* 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 or HS_INST_DD views view:

```
SELECT DD_TABLE_NAME, TRANSLATION_TYPE
FROM   HS_CLASS_DD
WHERE  FDS_CLASS_NAME='MegaBase6';

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 B, "Data Dictionary Views Available Through Heterogeneous Services" 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 HS 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 HS agents currently running on a given host, using one row per 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 *Oracle8i Reference*):

Table 7–2 V$HS_AGENT

Column	Description
AGENT_ID	Net8 session identifier used for connections to agent (`listener.ora` SID)
MACHINE	Operating system machine name
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 HS Sessions

The following view shows which HS 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 *Oracle8i Reference*):

Table 7–3 V$HS_SESSION

Column	Description
HS_SESSION_ID	Unique HS session identifier
AGENT_ID	Net8 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 (eg, when using external procedures)
DB_LINK_OWNER	Owner of the database link in DB_LINK

Determining the HS Parameters

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

View	Purpose
V$HS_PARAMETER	Lists HS 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 *Oracle8i Reference*):

Table 7–4 V$HS_SESSION

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

Using the DBMS_HS Package

The DBMS_HS package contains functions and procedures that allow you to specify and unspecify Heterogeneous Services initialization parameters, capabilities, instance names, class names, etc. These parameters are configured in the gateway initialization file—not the Oracle initialization parameter file. The only exceptions is HS_AUTOREGISTER, which is set in the Oracle initialization parameter file.

> **See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for a reference listing off all DBMS_HS package interface information for HS administration.

Specifying Initialization Parameters

Set initialization parameters either in the Oracle8*i* server or in the Heterogeneous Services agent. To set initialization parameters in the Oracle8*i* server, use the DBMS_HS package. Please see the agent's *Installation and User's Guide* for more information. If the same initialization parameter is set both in the agent and the Oracle8*i* server, then the value of initialization parameter in the Oracle8*i* server takes precedence.

The following types of initialization parameters exist:

Type	Description
Generic	Defined by Heterogeneous Services. See Appendix A, "Heterogeneous Services Initialization Parameters" for more information on generic initialization parameters.
Non-Oracle class-specific	Defined by the agent vendor. Some non-Oracle data store class-specific parameters may be mandatory. For example, a parameter may include connection information required to connect to a non-Oracle system. These parameters are documented in the *Installation and User's Guide* for your agent.

You can set both generic and non-Oracle data store class-specific HS initialization parameters in the Oracle database server using the CREATE_INST_INIT procedure in the DBMS_HS package.

For example, set the HS_DB_DOMAIN initialization parameter as follows

```
DBMS_HS.CREATE_INST_INIT
          (FDS_INST_NAME   => 'SalesDB',
           FDS_CLASS_NAME  => 'MegaBase6',
           INIT_VALUE_NAME => 'HS_DB_DOMAIN',
           INIT_VALUE      => 'US.SALES.COM');
```

See Also: Appendix A, "Heterogeneous Services Initialization Parameters" for more information about initialization parameters.

Unspecifying Initialization Parameters

To unspecify an HS initialization parameter in the Oracle8*i* server, use the DROP_INST_INIT procedure. For example, to delete the HS_DB_DOMAIN entry, enter:

```
DBMS_HS.DROP_INST_INIT
          (FDS_INST_NAME   => 'SalesDB',
           FDS_CLASS_NAME  => 'MegaBase6',
           INIT_VALUE_NAME => 'HS_DB_DOMAIN');
```

See Also: *Oracle8i Supplied PL/SQL Packages Reference* for a full description of the DBMS_HS package.

8

Managing Heterogeneous Services Using Generic Connectivity

The following topics describe the configuration and usage of generic connectivity agents:

- What Is Generic Connectivity?
- Supported Oracle SQL Statements
- Configuring Generic Connectivity Agents
- ODBC Connectivity Requirements
- OLE DB (SQL) Connectivity Requirements
- OLE DB (FS) Connectivity Requirements

What Is Generic Connectivity?

Generic connectivity is intended for low-end data integration solutions requiring the ad hoc query capability to connect from Oracle8*i* to non-Oracle database systems. Generic connectivity is enabled by Oracle Heterogeneous Services, 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 Oracle8*i* system. Be sure to use the agents shipped with your particular Oracle system and 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 Architecture
- SQL Execution
- Datatype Mapping
- Generic Connectivity Restrictions

Types of Agents

Generic connectivity is implemented as one of the following types of HS 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.

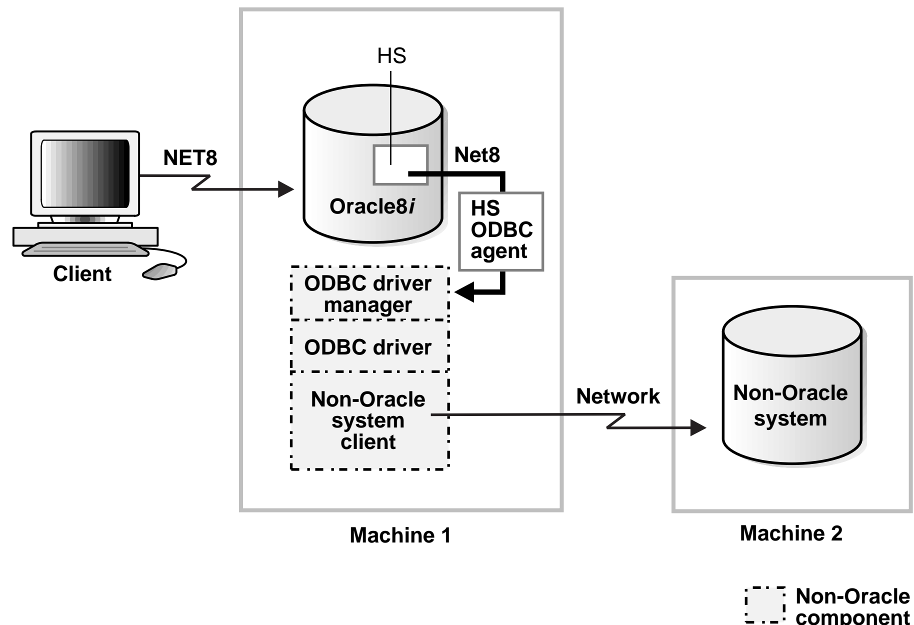
Generic Connectivity Architecture

To access the non-Oracle data store using generic connectivity, the agents work with an ODBC or OLE DB driver. Oracle8*i* 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 Oracle8*i* or a different machine.

Oracle and Non-Oracle Systems on Separate Machines

Figure 8–1 shows an example of one configuration in which an Oracle and non-Oracle database are on separate machines, communicating through an HS ODBC agent:

Figure 8–1 Non-Oracle System on Separate Computer



In this configuration, a client connects to Oracle8*i* through Net8. The HS part of the Oracle database server then connects through Net8 to the Heterogeneous Services ODBC agent. This agent communicates with the following non-Oracle components:

- An ODBC driver manager

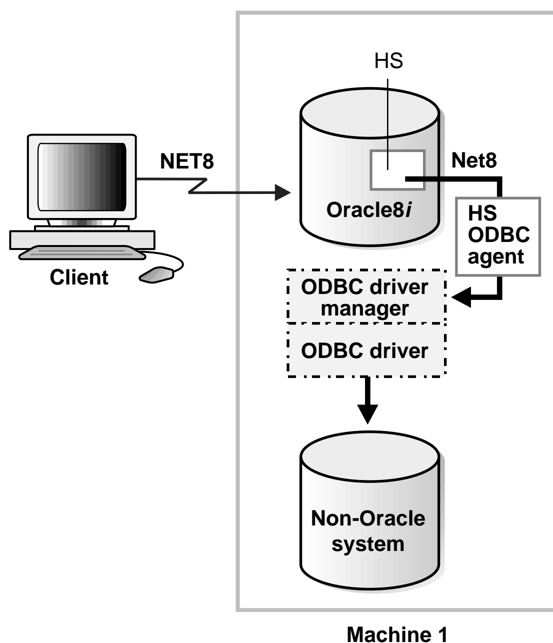
- An ODBC driver
- A non-Oracle client application

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

Oracle and Non-Oracle Systems on Same Machine

Figure 8–2 shows an example of a different configuration in which an Oracle and non-Oracle database are on the same machine, again communicating through an HS ODBC agent:

Figure 8–2 Accessing Heterogeneous Non-Oracle Systems



In this configuration, a client connects to Oracle8*i* through Net8. The HS part of the Oracle database server then connects through Net8 to the Heterogeneous Services ODBC agent. This 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.

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 may be manipulated as if it were a table in a relational database. The naming conventions used at the non-Oracle system may also depend on whether you are using an ODBC or OLE DB agent.

Datatype Mapping

Oracle8*i* maps the datatypes used in ODBC and OLE DB compliant data sources to supported Oracle datatypes. When the results of a query are returned, Oracle8*i* converts the ODBC or OLE DB datatypes to Oracle datatypes. For example, the ODBC datatype SQL_TIMESTAMP and the OLE DB datatype DBTYPE_DBTIMESTAMP are converted to Oracle's DATE datatype.

> **See Also:** Appendix D, "Datatype Mapping" for information on how the datatypes are mapped for each data source.

Generic Connectivity Restrictions

Following are some restrictions for generic connectivity:

- 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 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

Generic connectivity supports the following statements, but only if your 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, many Oracle functions are not supported by generic connectivity, although post-processing is performed by Oracle8*i*, 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 Oracle8*i* 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
```

Functions Supported by Generic Connectivity

Generic connectivity assumes that this 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 to a non-Oracle data source, you need to set the agent parameters. This section contains the following topics:

- Creating the Initialization File
- Editing the Initialization File

- Setting Initialization Parameters for an ODBC-based Data Source
- Setting Initialization Parameters for an OLE DB-based Data Source

Creating the Initialization File

You must create and customize an initialization file for your generic connectivity agent. Oracle supplies sample initialization files named `init`*`agent`*`.ora`, where *`agent`* might be `odbc`, `olesql`, or `olefs`, to indicate which agent the sample file can be used for, as in the following:

```
initodbc.ora
initolesql.ora
initolefs.ora
```

The sample files are stored in the `/admin` directory for that particular agent, in the `$ORACLE_HOME/rdbms/hs` path.

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 the agent connects to.

The *HS_SID* is also used to identify how to connect to the agent when you configure the listener by modifying your `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 `/admin` directory for that particular agent in the `$ORACLE_HOME/rdbms/hs` path.

> **See Also:** "Step 2: Set Up the Environment to Access Heterogeneous Services Agents" for more information on configuring the listener.

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 your `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: Appendix A, "Heterogeneous Services Initialization Parameters" for more information on parameters.

To set the parameter values, use the syntax:

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

where:

[SET][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 Appendix A, "Heterogeneous Services Initialization Parameters" for a description of all HS parameters and their possible values. The parameter is case-sensitive.
value	is the value you want to specify for the HS 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 your drivers or non-Oracle system need as environment variables.

PRIVATE is only supported for these Heterogeneous Services parameters:

- HS_FDS_CONNECT_INFO
- HS_FDS_SHAREABLE_NAME
- HS_FDS_TRACE_LEVEL
- HS_FDS_TRACE_FILE_NAME

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 DSN or a System DSN which has previously been defined using the ODBC Driver Manager.

When connecting using a File DSN, specify the value using the following syntax:

```
HS_FDS_CONNECT_INFO=FILEDSN=file_dsn
```

When connecting using a System DSN, specify the value using:

```
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 your ODBC driver. For example, for SQL Server enter the SQL Server machine.

> **Note:** The name entered for the DSN must match the name used for the gateway in `init`*`HS_SID`*`.ora`.

7. Continue clicking **Next** and answering the prompts until you reach the end (that is, 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 Appendix A, "Heterogeneous Services Initialization Parameters" for detailed information on all the initialization parameters.

Setting Parameters on UNIX: Example Assume that the `odbc.ini` file to connect 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=Informix7
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 will have 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"
```

Or, specify the connection details directly:

```
<SET|PRIVATE|SET PRIVATE> HS_FDS_CONNECT_INFO="provider;db[,CATALOG=catalog]"
```

where:

`provider`	is the name of the provider as it appears in the registry. This value is case sensitive.
`db`	is the name of the database.
`catalog`	is the name of the catalog

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. Before deciding to accept the default values or change them, see Appendix A, "Heterogeneous Services Initialization Parameters" 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 Oracle8*i*. On Windows NT, you must have an ODBC driver manager also located on the same machine. The ODBC driver manager and driver must meet these 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 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 datatypes and expose the following ODBC APIs:

Table 8–1 ODBC Functions **(Page 1 of 3)**

ODBC Function	Comment
SQLAllocConnect	
SQLAllocEnv	
SQLAllocStmt	
SQLBindCol	
SQLBindParameter	

Table 8–1 ODBC Functions (Page 2 of 3)

ODBC Function	Comment
SQLColumns	
SQLConnect	
SQLDescribeCol	
SQLDisconnect	
SQLDriverConnect	
SQLError	
SQLExecDirect	
SQLExecute	
SQLExtendedFetch	Recommended if used by your non-Oracle system.
SQLFetch	
SQLForeignKeys	Recommended if used by your non-Oracle system.
SQLFreeConnect	
SQLFreeEnv	
SQLFreeStmt	
SQLGetConnectOption	
SQLGetData	
SQLGetFunctions	
SQLGetInfo	
SQLGetTypeInfo	
SQLNumParams	Recommended if used by your non-Oracle system.
SQLNumResultCols	
SQLParamData	
SQLPrepare	
SQLPrimaryKeys	Recommended if used by your non-Oracle system.
SQLProcedureColumns	Recommended if used by your non-Oracle system.
SQLProcedures	Recommended if used by your non-Oracle system.
SQLPutData	

Table 8–1 ODBC Functions **(Page 3 of 3)**

ODBC Function	Comment
SQLRowCount	
SQLSetConnectOption	
SQLSetStmtOption	
SQLStatistics	
SQLTables	
SQLTransact	Recommended if used by your 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 OLD DB interfaces. The data providers in this case are the non-Oracle system you want to connect to using generic connectivity and OLE DB (SQL).

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

Data Provider Requirements

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

Table 8–2 OLE DB (SQL) Interfaces **(Page 1 of 2)**

Interface	Methods
IAccessor	CreateAccessor, ReleaseAccessor
IColumnsInfo	GetColumnsInfo (Command and Rowset objects)
ICommand	Execute
ICommandPrepare	Prepare
ICommandProperties	SetProperties
ICommandText	SetCommandText
ICommandWithParameters	GetParameterInfo
IDBCreateCommand	CreateCommand

Table 8–2 OLE DB (SQL) Interfaces **(Page 2 of 2)**

Interface	Methods
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)[b]	Read, Seek, SetSize, Stat, Write
ISupportErrorInfo	InterfaceSupportsErrorInfo
ITransactionLocal (optional)	StartTransaction, Commit, Abort

1 You can use IErrorLookup with the GetErrorDescription method as well.

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. The data providers in this case are the non-Oracle systems you want to connect to using a generic connectivity agent and OLE DB (FS). OLE DB (FS) connectivity uses OLE DB Index interfaces, if the provider exposes them.

Required usernames and passwords are passed to the provider when the application calls `IDBInitialize::Initialize()`.

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.

Bookmarks

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

OLE DB Interfaces

The data provider must provide the following OLE DB interfaces:

Table 8–3 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
IRowsetIndex[2]	SetRange
IErrorInfo[3]	GetDescription, GetSource
IErrorRecords	GetErrorInfo
IRowset	GetData, GetNextRows, ReleaseRows, RestartPosition
IStream (OLE)[a]	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.

Data Source Properties

The OLE DB data source must support these initialization properties:

- DBPROP_INIT_DATASOURCE
- DBPROP_AUTH_USERID

  required if the userid has been supplied in the security file
- DBPROP_AUTH_PASSWORD

  required if the userid and password have been supplied in the security file

The OLE DB data source must also support these 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

9 Developing Applications with Heterogeneous Services

This chapter provides information for application developers who want to use Heterogeneous Services.

Topics covered include:

- Developing Applications with Heterogeneous Services: Overview
- Developing Using Pass-Through SQL
- Optimizing Data Transfers Using Bulk Fetch
- Researching the Locking Behavior of Non-Oracle Systems

Developing Applications with Heterogeneous Services: Overview

When writing applications, you do not need to worry when a non-Oracle database is part of the distributed system. Heterogeneous Services makes the non-Oracle system appear as if it were another Oracle8*i* server.

Nevertheless, you may occasionally need to access a non-Oracle system using the non-Oracle system's SQL dialect. To make access possible, Heterogeneous Services provides a *pass-through SQL* feature that allows you to directly execute a native SQL statement at the non-Oracle system.

Additionally, Heterogeneous Services supports bulk fetches to optimize the data transfers for large data sets between a non-Oracle system, agent and Oracle database server. This chapter also discusses how to tune such data transfers.

Developing Using Pass-Through SQL

The pass-through SQL feature allows you to send a statement directly to a non-Oracle system without being interpreted by the Oracle8*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:

- Using the DBMS_HS_PASSTHROUGH package
- Considering the Implications of Using Pass-Through SQL
- Executing Pass-Through SQL Statements

Using the DBMS_HS_PASSTHROUGH package

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

The DBMS_HS_PASSTHROUGH package conceptually resides at the non-Oracle system. You must invoke procedures and functions in the package by using the appropriate database link to the non-Oracle system.

> **See Also:** *Oracle8i Supplied PL/SQL Packages 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 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
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

This section contains these topics:

- Executing Non-Queries
- Executing Queries

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.

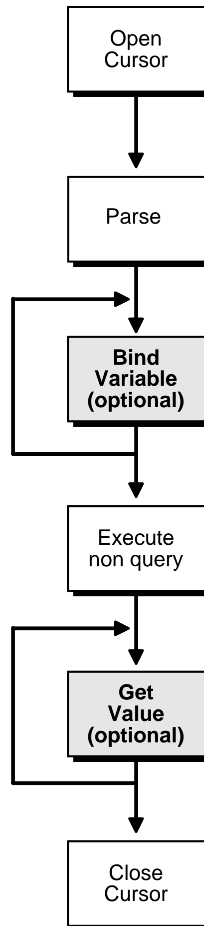
To execute pass-through SQL statements with bind variables, you must:

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 9–1 shows the flow diagram for executing non-queries with bind variables.

Figure 9–1 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.

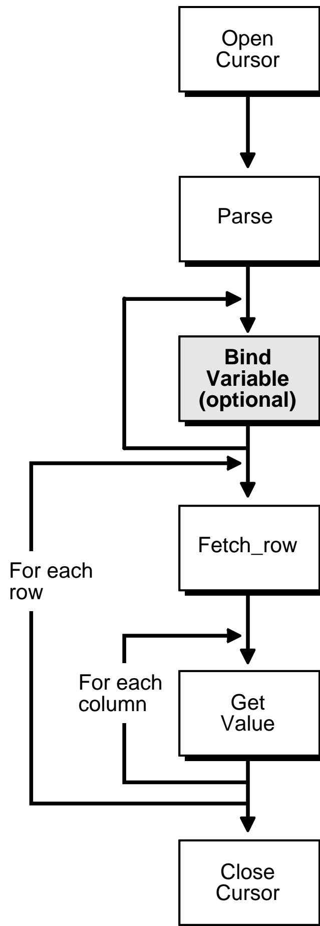
For IN OUT bind variables, you must use the BIND_INOUT_VARIABLE procedure to provide a value *before* the SQL statement is executed. *After* the SQL statement is executed, you must use the GET_VALUE procedure to retrieve the new value of the bind variable.

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 9–2 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 9–2 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, HS optimizes the round trips between the Oracle8*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.

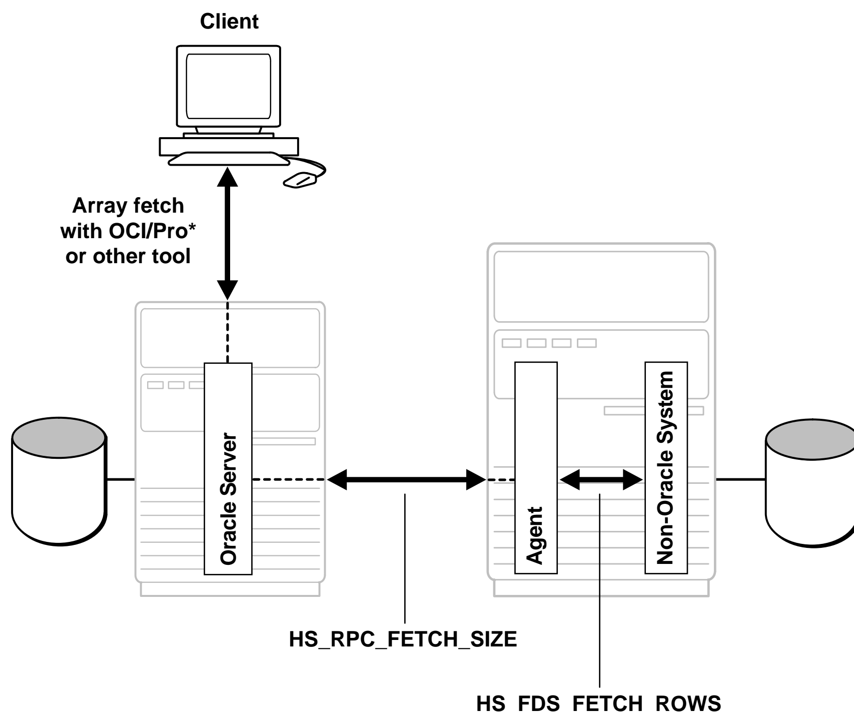
Optimizing Data Transfers Using Bulk Fetch

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

- From the non-Oracle system to the agent process
- From the agent process to the Oracle database server
- From the Oracle database server to the application

Oracle allows you to optimize all three data transfers, as illustrated in Figure 9–3.

Figure 9–3 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 Oracle8*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 per network round trip.

Controlling the Array Fetch Between Oracle Database Server and Agent

When Oracle retrieves data from a non-Oracle system, the HS initialization parameter HS_RPC_FETCH_SIZE defines the number of bytes sent per fetch between the agent and the Oracle8*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 64K 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 60K. When HS_RPC_FETCH_REBLOCKING is set to ON, the agent starts fetching 100 rows from the non-Oracle system.

Because there is only 60K bytes 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 120K of data in the agent, the first 64K can be sent back to the Oracle database server.

Now there is 56K of data left in the agent. The agent fetches another 100 rows from the non-Oracle system before sending the next 64K 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 Oracle8*i* server.

Researching the Locking Behavior of Non-Oracle Systems

When designing applications with Heterogeneous Services, be aware that the Oracle database server and non-Oracle data sources can have different locking behaviors. For example, some non-Oracle data sources differ from the Oracle database server in how they set read/write locks on records in affected tables.

Oracle cannot change any aspect of the locking behavior of a non-Oracle data source. In order to avoid adverse effects on other users of the non-Oracle data source, all applications that access a non-Oracle data source must always adhere to the programming standards of that data source.

> **See Also:** Your non-Oracle system's documentation for information about locking behavior.

A

Heterogeneous Services Initialization Parameters

Oracle database server initialization parameters are distinct from HS parameters. Set HS parameters by editing the Oracle Transparent Gateway initialization file, or by using the DBMS_HS package to set them in the data dictionary. String values for HS parameters must be lowercase. This appendix contains information on:

- HS_COMMIT_POINT_STRENGTH
- HS_DB_DOMAIN
- HS_DB_INTERNAL_NAME
- HS_DB_NAME
- HS_DESCRIBE_CACHE_HWM
- HS_FDS_CONNECT_INFO
- HS_FDS_SHAREABLE_NAME
- HS_FDS_TRACE_LEVEL
- HS_FDS_TRACE_FILE_NAME
- HS_LANGUAGE
- HS_NLS_DATE_FORMAT
- HS_NLS_DATE_LANGUAGE
- HS_NLS_NCHAR
- HS_OPEN_CURSORS
- HS_ROWID_CACHE_SIZE
- HS_RPC_FETCH_REBLOCKING
- HS_RPC_FETCH_SIZE

HS_COMMIT_POINT_STRENGTH

Default value:	0
Range of values:	0 to 255

HS_COMMIT_POINT_STRENGTH has the same function as the Oracle8*i* parameter COMMIT_POINT_STRENGTH.

Set HS_COMMIT_POINT_STRENGTH to a value relative to the importance of the site that will be 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).

See Also: Chapter 7, "Managing Oracle Heterogeneous Services Using Transparent Gateways" for more information about heterogeneous distributed transactions, and Chapter 4, "Distributed Transactions Concepts" for more information about distributed transactions and commit point sites.

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 *Oracle8i Administrator's Guide* and the *Oracle8i 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 datastore 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 datastore.

Increase the high water mark to improve performance, especially when you are accessing many mapped tables. 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 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 data link (UDL) with the following formats:

- HS_FDS_CONNECT_INFO="UDLFILE=*data_link*"
- HS_FDS_CONNECT_INFO="*provider*;db[,CATALOG=*catalog*]"

  which allows you to specify the connection details directly, and where:

  – *provider* is the case-sensitive name of the provider as it appears in the registry.

  – *db* is the name of the database.

  – *catalog* is the name of the catalog.

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

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 turned on or off for generic connectivity. Turn on the tracing to see which error messages occurred when you encounter problems. The results are written to a generic connectivity log file, in the `/log` directory under the `$ORACLE_HOME` directory.

HS_FDS_TRACE_FILE_NAME

Default value:	none
Range of values:	not applicable

Specifies the name of the trace file to which generic connectivity error messages are written, if TRACE is enabled. The trace file is located 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 Oracle8*i* 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 Oracle8*i* character set, and vice versa. This translation can degrade performance. In some cases, HS 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 part 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 HS_LANGUAGE 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 Oracle8*i* server and the non-Oracle data source depends on how the driver is implemented. See the *Installation and Users' Guide* for your platform for more information about the level of National Language Support.

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 *Oracle8i Reference*, but must match the date format of the target system. For example, if the target system stores the date "February 14, 1995" as "1995/02/14", 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. This 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 Oracle's CREATE DATABASE statement. The HS_NLS_NCHAR value should be the character set ID of a character set supported by Oracle's NLSRTL library.

See Also: "HS_LANGUAGE" on page A-6.

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 HS 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.

> **See Also:** Chapter 9, "Developing Applications with Heterogeneous Services" for more information.

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.

See Also: Chapter 9, "Developing Applications with Heterogeneous Services" for more information about array fetches.

B

Data Dictionary Views Available Through Heterogeneous Services

This appendix lists the data dictionary views that are supported through heterogeneous services mapping:

- 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_SNAPSHOTS
- 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_SNAPSHOT_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

C

Data Dictionary Translation for Generic Connectivity

Generic connectivity agents translate a query that refers to an Oracle8*i* data dictionary table into a query that retrieves the data from a non-Oracle data dictionary. You perform queries on data dictionary tables over the database link in the same way you query data dictionary tables in Oracle8*i*. The generic connectivity data dictionary is similar to the Oracle8*i* data dictionary in appearance and use. Non-Oracle data dictionary information is supplied to the user in Oracle8*i* data dictionary format.

To better understand the data dictionary support provided by generic connectivity, read these sections:

- Data Dictionary Translation Support
- Data Dictionary Mapping
- Generic Connectivity Data Dictionary Descriptions

Data Dictionary Translation Support

Data dictionary information is stored in the non-Oracle system as system tables and accessed through ODBC or OLE DB application programming interfaces (APIs). This section contains the following topics:

- Accessing the Non-Oracle Data Dictionary
- Supported Views and Tables

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 Oracle8*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.

Supported Views and Tables

Generic connectivity supports only these views and tables:

- ALL_CATALOG
- ALL_COL_COMMENTS
- ALL_CONS_COLUMNS
- ALL_CONSTRAINTS
- ALL_IND_COLUMNS
- ALL_INDEXES
- ALL_OBJECTS
- ALL_TAB_COLUMNS
- ALL_TAB_COMMENTS
- ALL_TABLES
- ALL_USERS
- ALL_VIEWS
- DICTIONARY
- USER_CATALOG
- USER_COL_COMMENTS
- USER_CONS_COLUMNS
- USER_CONSTRAINTS
- USER_IND_COLUMNS
- USER_INDEXES
- USER_OBJECTS
- USER_TAB_COLUMNS
- USER_TAB_COMMENTS
- USER_TABLES
- USER_USERS
- USER_VIEWS

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 9–1 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
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

Table 9–1 Generic Connectivity Data Dictionary Mapping

View	ODBC API	OLE DB API
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

Default Column Values

The generic connectivity data dictionary differs from a typical Oracle database server data dictionary. The Oracle database server columns that are missing in a non-Oracle data dictionary table are filled with the following, depending on the column type:

- Zeros
- Spaces
- NULL values
- Default values

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 Oracle8*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)	

Name	Null?	Type	Value
INDEX_NAME	NOT NULL	VARCHAR2(30)	
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

Name	Null?	Type	Value
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
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

Name	Null?	Type	Value
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

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)	
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)	

Name	Null?	Type	Value
SEARCH_CONDITION		LONG	NULL
R_OWNER		VARCHAR2(30)	
R_CONSTRAINT_NAME		VARCHAR2(30)	
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)	
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	

Name	Null?	Type	Value
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		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
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	

Name	Null?	Type	Value
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
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
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

Name	Null?	Type	Value
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		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

Name	Null?	Type	Value
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
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

D

Datatype Mapping

Oracle8*i* maps the datatypes used in ODBC and OLE DB compliant data sources to supported Oracle datatypes. When the results of a query are returned, Oracle8*i* converts the ODBC or OLE DB datatypes to Oracle datatypes. For information on how the datatypes are mapped for each data source, see the following:

- Mapping ODBC Datatypes to Oracle Datatypes
- Mapping OLE DB Datatypes to Oracle Datatypes

Mapping ODBC Datatypes to Oracle Datatypes

This table shows the mapping from ODBC datatypes to Oracle datatypes:

ODBC	Oracle
SQL_BIGINT	NUMBER(19,0)
SQL_BINARY	RAW
SQL_CHAR	CHAR
SQL_DATE	DATE
SQL_DECIMAL(p,s)	NUMBER(p,s)
SQL_DOUBLE	FLOAT(49)
SQL_FLOAT	FLOAT(49)
SQL_INTEGER	NUMBER(10)
SQL_LONGVARBINARY	LONG RAW
SQL_LONGVARCHAR	LONG
SQL_NUMERIC(p,s)	NUMBER(p,s)
SQL_REAL	FLOAT(23)
SQL_SMALLINT	NUMBER(5)
SQL_TIME	DATE
SQL_TIMESTAMP	DATE
SQL_TINYINT	NUMBER(3)
SQL_VARCHAR	VARCHAR

Mapping OLE DB Datatypes to Oracle Datatypes

This table shows the mapping from OLE DB datatypes to Oracle datatypes:

OLE DB	Oracle
DBTYPE_UI1	NUMBER(3)
DBTYPE_I1	NUMBER(3)
DBTYPE_UI2	NUMBER(5)
DBTYPE_I2	NUMBER(5)
DBTYPE_BOOL	NUMBER(5)
DBTYPE_UI4	NUMBER(10)
DBTYPE_I4	NUMBER(10)
DBTYPE_UI8	NUMBER(19,0)
DBTYPE_I8	NUMBER(19,0)
DBTYPE_NUMERIC(p,s)	NUMBER(p,s)
DBTYPE_R4	FLOAT(23)
DBTYPE_R8	FLOAT(49)
DBTYPE_DECIMAL	FLOAT(49)
DBTYPE_STR	VARCHAR2
DBTYPE_WSTR	VARCHAR2
DBTYPE_CY	NUMBER(19,0)
DBTYPE_DBDATE	DATE
DBTYPE_DBTIME	DATE
DBTYPE_DBTIMESTAMP	DATE
DBTYPE_BYTES	RAW
DTYPE_BYTES (long attribute)	LONG RAW
DBTYPE_STRING (long attribute)	LONG

Index

A

abort response, 4-13
two-phase commit, 4-13
administration
distributed databases, 2-1
tools, 1-31
agents
generic connectivity, 6-6
Heterogeneous Services, 6-3, 6-5
definition, 1-5
disabling self-registration, 7-9
registering, 7-5, 7-6, 7-7
specifying initialization parameters for, 7-4
aggregate functions, 2-33
ALL_DB_LINKS view, 2-21
ALTER SESSION
system privilege, 3-2
ALTER SESSION statement
ADVISE clause, 5-13
CLOSE DATABASE LINK clause, 3-2
ALTER SYSTEM statement
DISABLE DISTRIBUTED RECOVERY clause, 5-28
ENABLE DISTRIBUTED RECOVERY clause, 5-28
ANALYZE TABLE statement, 3-7
analyzing tables
cost-based optimization, 3-7
application development
constraints, 3-3
database links
controlling connections, 3-2
distributed databases, 3-1
analyzing execution plan, 3-10
controlling connections, 3-2
handling errors, 3-3
handling RPC errors, 3-12
managing distribution of data, 3-2
managing referential integrity, 3-3
optimizing distributed queries, 1-47
overview, 1-44
remote procedure calls, 1-46
tuning distributed queries, 3-3
tuning using collocated inline views, 3-4
using cost-based optimization, 3-5
using hints to tune queries, 3-8
distributing data, 3-2
Heterogeneous Services, 9-1, 9-2
controlling array fetches between non-Oracle server and agent, 9-11
controlling array fetches between Oracle server and agent, 9-11
controlling reblocking of array fetches, 9-11
DBMS_HS_PASSTHROUGH package, 9-2
pass-through SQL, 9-2
using bulk fetches, 9-9
using OCI for bulk fetches, 9-10
referential integrity, 3-3
remote connections
terminating, 3-2
using Heterogeneous Services, 9-1
applications
errors
RAISE_APPLICATION_ERROR() procedure, 3-12
array fetches, 9-10
agents, 9-11

auditing
database links, 1-31
AUTHENTICATED BY clause
CREATE DATABASE LINK statement, 2-16
authentication
database links, 1-25

B

bind queries
executing using pass-through SQL, 9-7
BIND_INOUT_VARIABLE procedure, 9-3, 9-7
BIND_OUT_VARIABLE procedure, 9-3, 9-6
BIND_VARIABLE procedure, 9-3
buffers
multiple rows, 9-8
bulk fetches
optimizing data transfers using, 9-9

C

calls
remote procedure, 1-46
CATHO.SQL script
installing data dictionary for Heterogeneous Services, 7-2
centralized user management
distributed systems, 1-27
character sets
Heterogeneous Services, A-6
client/server architectures
distributed databases, 1-7
direct and indirect connections, 1-9
NLS, 1-47
CLOSE DATABASE LINK clause
ALTER SESSION statement, 3-2
CLOSE_CURSOR function, 9-3
closing database links, 2-18
collocated inline views
tuning distributed queries, 3-4
comments
in COMMIT statements, 5-13
commit phase, 4-12, 4-24
two-phase commit, 4-14
commit point site, 4-7
commit point strength, 4-9, 5-5, A-2
determining, 4-10
distributed transactions, 4-7, 4-9
how Oracle determines, 4-9
commit point strength
definition, 4-9
specifying, 5-5
COMMIT statement
COMMENT parameter, 5-13, 5-26
FORCE clause, 5-13, 5-14, 5-15
forcing, 5-12
two-phase commit and, 1-36
COMMIT_POINT_STRENGTH initialization parameter, 4-9, 5-6
committing transactions
distributed
commit point site, 4-7
configuring generic connectivity, 8-6
configuring transparent gateways, 7-2
connected user database links, 2-12
advantages and disadvantages, 1-17
creating, 2-12
definition, 1-17
example, 1-20
REMOTE_OS_AUTHENT initialization parameter, 1-18
connection qualifiers
database links and, 2-13
connections
remote
terminating, 3-2
constraints
application development issues, 3-3
ORA-02055
constrain violation, 3-3
cost-based optimization, 3-5
distributed databases, 1-47
hints, 3-8
using for distributed queries, 3-5
CREATE DATABASE LINK statement, 2-9
CREATE_INST_INIT procedure, 7-18
creating connected user links
scenario, 2-36
creating current user links
scenario, 2-37

creating database links, 2-8
connected user, 2-12
current user, 2-12
example, 1-20
fixed user, 2-11
private, 2-9
public, 2-10
service names within link names, 2-13
specifying types, 2-9
creating fixed user links
scenario, 2-34, 2-35
current user database links, 2-12
advantages and disadvantages, 1-19
cannot access in shared schema, 1-28
creating, 2-12
definition, 1-17
example, 1-20
schema independence, 1-28
cursors
and closing database links, 3-2

D

data dictionary
contents with generic connectivity, C-3
installing for Heterogeneous Services, 7-2
mapping for generic connectivity, C-4
Oracle server name/SQL Server name, C-4
purging pending rows from, 5-15, 5-16
tables, 6-4
translation support for generic connectivity, C-2
data dictionary views
DBA_DB_LINKS, 2-21, 5-6, 5-9
generic connectivity, C-3
Heterogeneous Services, 7-9, B-1
USER, 5-6, 5-9
data encryption
distributed systems, 1-30
data manipulation language
statements allowed in distributed transactions, 1-33
database links
advantages, 1-13
auditing, 1-31
authentication, 1-25
without passwords, 1-26
closing, 2-18, 3-2
connected user, 2-12, 2-36
advantages and disadvantages, 1-17
definition, 1-17
connections
controlling, 3-2
determining open, 2-24
creating, 2-8
connected user, 2-12, 2-36
connected user, shared, 2-36
current user, 2-12, 2-37
example, 1-20
fixed user, 2-11, 2-34
fixed user, shared, 2-35
obtaining necessary privileges, 2-8
private, 2-9
public, 2-10
scenarios, 2-34
shared, 2-14, 2-15
specifying types, 2-9
current user, 1-16, 2-12
advantages and disadvantages, 1-19
definition, 1-17
data dictionary views
ALL, 5-6, 5-9
DBA_DB_LINKS, 5-6, 5-9
USER, 2-21, 5-6, 5-9
definition, 1-10
distributed queries, 1-34
distributed transactions, 1-35
dropping, 2-19
enforcing global naming, 2-3
enterprise users and, 1-28
fixed user, 2-34
advantages and disadvantages, 1-18
definition, 1-17
global
definition, 1-16
global names, 1-13
global object names, 1-36
handling errors, 3-3
heterogeneous systems, 6-2, 7-4
limiting number of connections, 2-20
listing, 2-21, 5-6, 5-9

- managing, 2-18
- minimizing network connections, 2-14
- name resolution, 1-36
  - schema objects, 1-38
  - views, synonyms, and procedures, 1-41
  - when global database name is complete, 1-37
  - when global database name is partial, 1-37
  - when no global database name is specified, 1-37
- names for, 1-15
- passwords
  - viewing, 2-22
- private
  - definition, 1-16
- public
  - definition, 1-16
- referential integrity in, 3-3
- remote queries, 1-33
- remote transactions, 1-33, 1-35
- resolution, 1-36
- restrictions, 1-23
- roles on remote database, 1-23
- schema objects, 1-21
  - name resolution, 1-22
  - synonyms for, 1-22
- service names used within link names, 2-13
- shared, 1-12
  - configuring, 2-16
  - creating, 2-14
  - creating links to dedicated servers, 2-16
  - creating links to multi-threaded (MTS) servers, 2-17
  - determining whether to use, 2-14
- shared SQL, 1-34
- tuning distributed queries, 3-3
- tuning queries with hints, 3-8
- tuning using collocated inline views, 3-4
- types of links, 1-16
- types of users, 1-17
- users
  - specifying, 2-11
- using cost-based optimization, 3-5
- viewing, 2-21

databases
- administration, 2-1
- distributed
  - site autonomy of, 1-24

datatypes
- mapping, 6-4
  - ODBC, D-2
  - ODBC to Oracle, D-2
  - OLE DB, D-3
  - OLE DB to Oracle, D-3

date formats
- Heterogeneous Services, A-7

DBA_2PC_NEIGHBORS view, 5-9
- using to trace session tree, 5-9

DBA_2PC_PENDING view, 5-6, 5-15, 5-25
- using to list in-doubt transactions, 5-7

DBA_DB_LINKS view, 2-21, 5-6, 5-9

DBMS_HS package
- specifying HS parameters, 7-17

DBMS_HS_PASSTHROUGH package, 9-2
- list of functions and procedures, 9-3

DBMS_TRANSACTION package
- PURGE_LOST_DB_ENTRY procedure, 5-16

declarative referential integrity constraints, 3-3

describe cache high water mark
- definition, A-3

Digital's POLYCENTER Manager on NetView, 1-32

disabling recoverer process
- distributed transactions, 5-28

distributed applications
- distributing data, 3-2

distributed databases
- administration
  - overview, 1-23
- application development
  - analyzing execution plan, 3-10
  - controlling connections, 3-2
  - handling errors, 3-3
  - handling RPC errors, 3-12
  - managing distribution of data, 3-2
  - managing referential integrity, 3-3
  - tuning distributed queries, 3-3
  - tuning using collocated inline views, 3-4
  - using cost-based optimization, 3-5
  - using hints to tune queries, 3-8
- client/server architectures, 1-7

commit point strength, 4-9
cost-based optimization, 1-47
distributed processing, 1-3
distributed queries, 1-34
distributed updates, 1-34
distributing an application's data, 3-2
global database names
how they are formed, 2-2
global object names, 1-22, 2-2
global users
schema-dependent, 1-27
schema-independent, 1-28
location transparency, 1-44
creating, 2-26
creating using procedures, 2-30
creating using synonyms, 2-28
creating using views, 2-26
restrictions, 2-33
management tools, 1-31
managing read consistency, 5-28
NLS support, 1-47
nodes of, 1-7
overview, 1-2
referential integrity
application development, 3-3
remote object security, 2-28
remote queries and updates, 1-33
replicated databases and, 1-4
scenarios, 2-34
security, 1-25
site autonomy, 1-24
SQL transparency, 1-45
testing features, 5-26
transaction processing, 1-33
transparency, 1-44
queries, 2-32
updates, 2-32
distributed processing
distributed databases, 1-3
distributed queries, 1-34
analyzing tables, 3-7
application development issues, 3-3
cost-based optimization, 3-5
optimizing, 1-47
distributed systems
data encryption, 1-30
distributed transactions, 1-35
case study, 4-20
commit point site, 4-7
commit point strength, 4-9
committing, 4-9
database server role, 4-6
decreasing limit for, 5-3
defined, 4-2
disabling processing of, 5-3
DML and DDL, 4-3
failure during, 5-4
forcing to fail, 5-26
global coordinator, 4-7
increasing limit for, 5-3
initialization parameters influencing, 5-2
limiting number, 5-2
local coordinator, 4-7
lock timeout interval, 5-4
locked resources, 5-4
locks for in-doubt, 5-5
management, 4-1, 5-1
manually overriding in-doubt, 5-12
recovery in single-process systems, 5-28
session trees, 4-4, 4-5, 4-6
clients, 4-6
commit point site, 4-7, 4-9
database servers, 4-6
global coordinators, 4-7
local coordinators, 4-6
setting advice, 5-13
specifying
commit point strength, 5-5
interval for open connections, 5-5
tracing session tree, 5-9
transaction control statements, 4-4
transaction timeouts, 5-4
two-phase commit, 4-4
discovering problems, 5-11
example, 4-20
viewing information about, 5-6
distributed updates, 1-34
DISTRIBUTED_LOCK_TIMEOUT initialization parameter
controlling time-outs with, 5-4

DISTRIBUTED_RECOVERY_CONNECTION_HOLD_TIME initialization parameter, 5-5
DISTRIBUTED_TRANSACTIONS initialization parameter
  recoverer process (RECO), 5-3
  setting, 5-2
  when to alter, 5-3
DML. *See* data manipulation language
drivers
  ODBC, 8-12
  OLEFS, 8-15
  OLESQL, 8-14
DRIVING_SITE hint, 3-9
dropping database links, 2-19
dynamic performance views
  Heterogeneous Services, 7-15
    determining open sessions, 7-16
    determining which agents are on host, 7-15

E

enabling recoverer process
  distributed transactions, 5-28
enterprise users
  definition, 1-28
errors
  messages
    trapping, 3-12
  ORA-00900, 3-12
  ORA-01591, 5-5
  ORA-02015, 3-12
  ORA-02049, 5-4
  ORA-02050, 5-11
  ORA-02051, 5-11
  ORA-02054, 5-11
  ORA-02055
    integrity constrain violation, 3-3
  ORA-02067
    rollback required, 3-3
  ORA-06510
    PL/SQL error, 3-13
  remote procedures, 3-12
examples
  manual transaction override, 5-17
exception handler, 3-12
  local, 3-13
EXCEPTION keyword, 3-12
exceptions
  assigning names
    PRAGMA_EXCEPTION_INIT, 3-12
  user-defined
    PL/SQL, 3-12
EXECUTE_IMMEDIATE procedure, 9-3
  restrictions, 9-4
EXECUTE_NON_QUERY procedure, 9-3
execution plans
  analyzing for distributed queries, 3-10

F

FDS_CLASS, 7-7
FDS_CLASS_VERSION, 7-7
FDS_INST_NAME, 7-8
features, new
  centralized user and privilege management, xiv
  generic connectivity, xiii
  Heterogeneous Services initialization parameters, xiv
  parallel DML and DDL, xiv
  SQL optimization for heterogeneous systems, xiv
  V$HS_AGENT and V$HS_SESSION, xv
FETCH_ROW procedure, 9-3
  executing queries using pass-through SQL, 9-7
fetches
  bulk, 9-9
  optimizing round-trips, 9-8
fixed user database links
  07_DICTIONARY_ACCESSIBILITY initialization parameter, 1-18
  advantages and disadvantages, 1-18
  creating, 2-11
  definition, 1-17
  example, 1-20
FORCE clause
  COMMIT statement, 5-13
  ROLLBACK statement, 5-13
forcing
  COMMIT or ROLLBACK, 5-8, 5-12
forget phase

two-phase commit, 4-16

G

generic connectivity
architecture, 8-3
Oracle and non-Oracle on same machine, 8-4
Oracle and non-Oracle on separate machines, 8-3
configuration, 8-6
creating initialization file, 8-7
data dictionary
translation support, C-2
datatype mapping, 8-5
definition, 1-6, 8-2
DELETE statement, 8-6
editing initialization file, 8-7
error tracing, A-5
Heterogeneous Services, 6-6
INSERT statement, 8-6
non-Oracle data dictionary access, C-2
ODBC connectivity requirements, 8-12
OLE DB (FS) connectivity requirements, 8-15
interfaces, 8-16
OLE DB (SQL) connectivity requirements, 8-14
restrictions, 8-5
setting parameters for ODBC source, 8-9
UNIX, 8-10
Windows NT, 8-9
setting parameters for OLE DB source, 8-11
SQL execution, 8-5
supported functions, 8-6
supported SQL syntax, 8-5
types of agents, 8-2
UPDATE statement, 8-6
GET_VALUE procedure, 9-3, 9-6, 9-7
global coordinators, 4-7
distributed transactions, 4-7
global database consistency
distributed databases and, 4-15
global database links, 1-16
creating, 2-11
global database names
changing the domain, 2-4
database links, 1-13
distributed databases
how they are formed, 2-2
enforcing for database links, 1-15
enforcing global naming, 2-3
impact of changing, 1-42
querying, 2-4
global object names
database links, 1-36
distributed databases, 2-2
global users, 2-37
distributed systems
schema-dependent, 1-27
schema-independent, 1-28
GLOBAL_NAME view
using to determine global database name, 2-4
GLOBAL_NAMES initialization parameter, 1-15
GV$DBLINK view, 2-24

H

heterogeneous distributed systems
accessing, 7-2
definition, 1-5
Heterogeneous Services
agent registration, 7-5
avoiding configuration mismatches, 7-6
disabling, 7-9
enabling, 7-5
agents, 6-3, 6-5
self-registration, 7-7
application development, 9-1, 9-2
controlling array fetches between non-Oracle server and agent, 9-11
controlling array fetches between Oracle server and agent, 9-11
controlling reblocking of array fetches, 9-11
DBMS_HS_PASSTHOUGH package, 9-2
locking behavior of non-Oracle systems, 9-12
pass-through SQL, 9-2
using bulk fetches, 9-9
using OCI for bulk fetches, 9-10
creating database links, 7-4
data dictionary, 6-6
classes and instances, 6-7
data dictionary views, 7-9, B-1

types, 7-9
understanding sources, 7-11
using general views, 7-11
using SQL service views, 7-13
using transaction service views, 7-12
database links to non-Oracle systems, 6-2
DBMS_HS package
using to specify initialization parameters, 7-17
using to unspecify initialization parameters, 7-18
defining maximum number of open cursors, A-8
dynamic performance views, 7-15
V$HS_AGENT view, 7-15
V$HS_SESSION view, 7-16
features, 1-6
generic connectivity
architecture, 8-3
creating initialization file, 8-7
datatype mapping, 8-5
definition, 8-2
editing initialization file, 8-7
non-Oracle data dictionary access, C-2
ODBC connectivity requirements, 8-12
OLE DB (FS) connectivity requirements, 8-15
OLE DB (FS) interfaces, 8-16
OLE DB (SQL) connectivity requirements, 8-14
restrictions, 8-5
setting parameters for ODBC source, 8-9
setting parameters for OLE DB source, 8-11
SQL execution, 8-5
supported functions, 8-6
supported SQL syntax, 8-5
supported tables, C-3
types of agents, 8-2
initialization parameters
specifying, 7-17
unspecifying, 7-18
installing data dictionary, 7-2
locking behavior of non-Oracle systems, 9-12
optimizing data transfer, A-9
overview, 1-5, 6-2
process architecture, 6-4, 6-5
setting global name, A-3
setting up access using transparent gateway, 7-2
setting up environment, 7-2
specifying cache high water mark, A-3
specifying cache size, A-9
specifying commit point strength, A-2
specifying domain, A-2
specifying instance identifier, A-3
SQL service, 6-4
testing connections, 7-4
transaction service, 6-3
tuning internal data buffering, A-10
types, 6-3
hints, 3-8
DRIVING_SITE, 3-9
NO_MERGE, 3-9
using to tune distributed queries, 3-8
HP's OpenView, 1-32
HS_AUTOREGISTER initialization parameter, 7-17
using to enable agent self-registration, 7-8
HS_BASE_CAPS view, 7-10
HS_BASE_DD view, 7-10
HS_CLASS_CAPS view, 7-10
HS_CLASS_DD view, 7-10
HS_CLASS_INIT view, 7-10
HS_COMMIT_POINT_STRENGTH initialization parameter, A-2
HS_DB_DOMAIN initialization parameter, 7-18, A-2
HS_DB_INTERNAL_NAME initialization parameter, A-3
HS_DB_NAME initialization parameter, A-3
HS_DESCRIBE_CACHE_HWM initialization parameter, A-3
HS_FDS_CLASS view, 7-10
HS_FDS_CONNECT_INFO initialization parameter, A-4
specifying connection information, 8-7
HS_FDS_FETCH_ROWS initialization parameter, 9-11
HS_FDS_INST view, 7-10
HS_FDS_SHAREABLE_NAME initialization parameter, A-5

HS_FDS_TRACE initialization parameter, A-5
HS_FDS_TRACE_FILE_NAME initialization parameter, A-5
HS_FDS_TRACE_LEVEL initialization parameter
  enabling agent tracing, 8-8
HS_INST_CAPS view, 7-10
HS_INST_DD view, 7-10
HS_INST_INIT view, 7-10
HS_LANGUAGE initialization parameter, A-6
HS_NLS_DATE_FORMAT initialization parameter, A-7
HS_NLS_DATE_LANGUAGE initialization parameter, A-7
HS_NLS_NCHAR initialization parameter, A-8
HS_OPEN_CURSORS initialization parameter, A-8
HS_ROWID_CACHE_SIZE initialization parameter, A-9
HS_RPC_FETCH_REBLOCKING initialization parameter, 9-11, A-9
HS_RPC_FETCH_SIZE initialization parameter, 9-11, A-10

I

IBM's NetView/6000, 1-32
in-doubt transactions, 4-14
  after a system failure, 5-11
  automatic resolution, 4-17
    failure during commit phase, 4-18
    failure during prepare phase, 4-17
  deciding how to handle, 5-10
  deciding whether to perform manual override, 5-12
  intentionally creating, 5-26
  manually committing, 5-14
  manually overriding, 4-19, 5-13
    scenario, 5-17
  manually rolling back, 5-15
  overriding manually, 5-12
  overview, 4-16
  pending transactions table, 5-25
  purging rows from data dictionary, 5-15
    deciding when necessary, 5-16
  recoverer process, 5-27
  rollback segments, 5-12
  rolling back, 5-13, 5-14, 5-15
  SCNs and, 4-19
  simulating, 5-26
  tracing session tree, 5-9
  viewing information about, 5-6
integrity constraints
  ORA-02055
    constraint violation, 3-3

J

joins
  distributed databases
    managing statement transparency, 2-33

L

listeners, 7-2
listing database links, 2-21, 5-6, 5-9
local coordinators, 4-7
  distributed transactions, 4-6
location transparency
  distributed databases
    creating using procedures, 2-30
    creating using synonyms, 2-28
    creating using views, 2-26
  using procedures, 2-30, 2-31, 2-32
lock timeout interval
  distributed transactions, 5-4
locks
  in distributed transactions, 5-4
  in non-Oracle systems, 9-12
  in-doubt distributed transactions, 5-4, 5-5
LONG columns, 2-33
LONG RAW columns, 2-33

M

manual overrides
  in-doubt transactions, 5-13
messages
  error
    trapping, 3-12
multiple rows

buffering, 9-8

N

name resolution
distributed databases, 1-22
impact of global name changes, 1-42
schema objects, 1-38
when global database name is complete, 1-37
when global database name is partial, 1-37
when no global database name is specified, 1-37
National Language Support (NLS)
client/server architectures, 1-48
distributed databases
clients and servers may diverge, 1-47
heterogeneous systems, 1-49
homogeneous systems, 1-48
Heterogeneous Services, A-6
character set of non-Oracle source, A-8
date format, A-7
languages in character date values, A-7
Net8 listener, 6-5, 7-2
network connections
minimizing, 2-14
networks
distributed databases use of, 1-2
new features
centralized user and privilege management, xiv
generic connectivity, xiii
Heterogeneous Services initialization parameters, xiv
parallel DML and DDL, xiv
SQL optimization for heterogeneous systems, xiv
V$HS_AGENT and V$HS_SESSION, xv
NO_DATA_FOUND keyword, 3-12
NO_MERGE hint, 3-9
Novell's NetWare Management System, 1-32

O

objects
referencing with synonyms, 2-29
OCI
optimizing data transfers using, 9-10
ODBC agents
connectivity requirements, 8-12
functions, 8-12
ODBC connectivity
data dictionary mapping, C-4
mapping OBDC datatypes, D-2
mapping Oracle datatypes, D-2
ODBC driver, 8-12
requirements, 8-12
specifying connection information
UNIX, A-4
Windows NT, A-4
specifying path to library, A-5
OLE DB agents
connectivity requirements, 8-14, 8-15
interfaces, 8-16
OLE DB connectivity
data dictionary mapping, C-4
mapping to Oracle datatypes, D-3
setting connection information, A-4
OLEFS drivers, 8-15
data provider requirements, 8-15
initialization properties, 8-17
rowset properties, 8-17
security, 8-15
OLESQL drivers, 8-14
data provider requirements, 8-14
security, 8-14
OPEN_CURSOR procedure, 9-3
OPEN_LINKS initialization parameter, 2-20
ORA-00900 error, 3-12
ORA-02015 error, 3-12
ORA-02055 error
integrity constraint violation, 3-3
ORA-02067 error
rollback required, 3-3
ORA-06510 error
PL/SQL error, 3-13
Oracle Call Interface. *See* OCI
Oracle precompiler
optimizing data transfers using, 9-10
Oracle Transparent Gateways
Heterogeneous Services and, 6-3
OUT bind variables, 9-6

P

PARSE procedure, 9-3
pass-through SQL, 9-2
avoiding SQL interpretation, 9-2
executing statements, 9-3
non-queries, 9-4
queries, 9-7
with bind variables, 9-4
with IN bind variables, 9-5
with IN OUT bind variables, 9-6
with OUT bind variables, 9-6
implications of using, 9-3
overview, 9-2
restrictions, 9-3
passwords
database links
viewing, 2-22
pending transaction tables, 5-25
PL/SQL
development environment, 9-2
errors
ORA-06510, 3-13
user-defined exceptions, 3-12
PRAGMA_EXCEPTION_INIT procedure
assigning exception names, 3-12
prepare phase, 4-12
recognizing read-only nodes, 4-13
two-phase commit, 4-11, 4-12
prepare/commit phases
abort response, 4-12
effects of failure, 5-4
failures during, 5-11
forcing to fail, 5-26
locked resources, 5-4
pending transaction table, 5-25
prepared response, 4-12
read-only response, 4-12
testing recovery, 5-26
prepared response
two-phase commit, 4-12
private database links, 1-16
privileges
closing a database link, 3-2
creating database links, 2-8
managing with procedures, 2-32
managing with synonyms, 2-30
managing with views, 2-28
procedures
location transparency using, 2-30, 2-31, 2-32
remote calls, 1-46
process architecture for distributed external procedures, 6-6
public database links
connected user, 2-36
fixed user, 2-34
public fixed user database links, 2-34
PURGE_LOST_DB_ENTRY procedure
DBMS_TRANSACTION package, 5-16
purging pending rows
from data dictionary, 5-15
when necessary, 5-16

Q

queries
distributed, 1-34
application development issues, 3-3
distributed or remote, 1-33
location transparency and, 1-45
pass-through SQL, 9-7
post-processing, 3-4
remote, 3-4
transparency, 2-32

R

read consistency
managing in distributed databases, 5-28
read-only response
two-phase commit, 4-12
reblocking, 9-11
recoverer process (RECO)
disabling, 5-27, 5-28
distributed transaction recovery, 5-27
DISTRIBUTED_TRANSACTIONS initialization parameter, 5-3
enabling, 5-27, 5-28
pending transaction table, 5-27
recovery

testing distributed transactions, 5-26
referential integrity
distributed database systems
application development, 3-3
remote data
querying, 2-33
updating, 2-33
remote procedure calls, 1-46
distributed databases and, 1-46
remote queries, 3-4
distributed databases and, 1-33
execution, 3-4
post-processing, 3-4
remote transactions, 1-35
defined, 1-35
REMOTE_OS_AUTHENT initialization parameter, 1-18
roles
obtained through database links, 1-23
rollback segments
in-doubt distributed transactions, 5-12
ROLLBACK statement
FORCE clause, 5-13, 5-14, 5-15
forcing, 5-12
rollbacks
ORA-02067 error, 3-3
rows
buffering multiple, 9-8

S

savepoints
in-doubt transactions, 5-13, 5-15
schema objects
distributed database naming conventions for, 1-22
global names, 1-22
security, 8-15
distributed databases, 1-25
centralized user management, 1-27
OLESQL driver, 8-14
remote objects, 2-28
using synonyms, 2-30
SELECT statement
accessing non-Oracle system, C-2
FOR UPDATE clause, 2-33
SERVER parameter
net service name, 2-16
servers
role in two-phase commit, 4-6
service names
database links and, 2-13
specifying in database links, 7-4
session trees
distributed transactions, 4-4, 4-5
clients, 4-6
commit point site, 4-7, 4-9
database servers, 4-6
global coordinators, 4-7
local coordinators, 4-6
tracing, 5-9
sessions
setting advice for transactions, 5-13
shared database links
configuring, 2-16
creating links, 2-14, 2-15
to dedicated servers, 2-16
to multi-threaded (MTS) servers, 2-17
determining whether to use, 2-14
example, 1-20
SHARED keyword
CREATE DATABASE LINK statement, 2-15
shared SQL
for remote and distributed statements, 1-34
Simple Network Management Protocol (SNMP) support
database management, 1-32
single-process systems
enabling distributed recovery, 5-28
site autonomy
distributed databases, 1-24
SQL capabilities
data dictionary tables, 7-14
SQL dialect
understood by non-Oracle system, 6-4
SQL errors
ORA-00900, 3-12
ORA-02015, 3-12
SQL service
capabilities, 6-4

data dictionary views, 6-8, 7-10
Heterogeneous Services, 6-4
views
Heterogeneous Services, 7-13
SQL statements
distributed databases and, 1-33
mapping to non-Oracle datastores, 9-2
stored procedures
distributed query creation, 3-3
managing privileges, 2-32
remote object security, 2-32
subqueries, 2-33
in remote updates, 1-34
SunSoft's SunNet Manager, 1-33
synonyms
CREATE statement, 2-29
definition and creation, 2-29
examples, 2-29
location transparency using, 2-28
managing privileges, 2-30
name resolution, 1-41
name resolution in distributed databases, 1-41
remote object security, 2-30
system change numbers (SCN)
coordination in a distributed database system, 4-15
in-doubt transactions, 5-14

T

transaction control statements
distributed transactions and, 4-4
transaction failures
simulating, 5-26
transaction management
overview, 4-11
transaction processing
distributed systems, 1-33
transaction service
Heterogeneous Services, 6-3
views
Heterogeneous Services, 7-12
transactions
closing database links, 3-2
distributed
two-phase commit and, 1-36
in-doubt, 4-14
after a system failure, 5-11
pending transactions table, 5-25
recoverer process (RECO) and, 5-27
manually overriding in-doubt, 5-12
remote, 1-35
transparency
location
using procedures, 2-30, 2-31, 2-32
query, 2-32
update, 2-32
transparent gateways
accessing Heterogeneous Services agents, 7-2
creating database links, 7-4
Heterogeneous Services, 6-5
installing Heterogeneous Services data dictionary, 7-2
testing connections, 7-4
triggers
distributed query creation, 3-3
tuning
analyzing tables, 3-7
cost-based optimization, 3-5
two-phase commit
case study, 4-20
commit phase, 4-14, 4-24
steps in, 4-15
described, 1-35
distributed transactions, 4-4, 4-11
tracing session tree, 5-9
viewing information about, 5-6
forget phase, 4-16
in-doubt transactions, 4-16
automatic resolution, 4-17
manual resolution, 4-19
SCNs and, 4-19
phases, 4-11
prepare phase, 4-11, 4-12
abort response, 4-13
prepared response, 4-12
read-only response, 4-12
responses, 4-12
steps, 4-14
problems, 5-11

recognizing read-only nodes, 4-13
specifying commit point strength, 5-5

U

unsupported functions
generic connectivity, 8-6
updates
location transparency and, 1-45
transparency, 2-32
USER_DB_LINKS view, 2-21

V

V$DBLINK view, 2-24
V$HS_AGENT view
determining which agents are on host, 7-15
V$HS_PARAMETER view
listing HS parameters, 7-17
V$HS_SESSION view
determining open sessions, 7-16
variables
BIND, 9-4
views
location transparency using, 2-26
managing privileges with, 2-28
name resolution in distributed databases, 1-41
remote object security, 2-28