Oracle8*i*

Distributed Database Systems

Release 8.1.5

February 1999

A67784-01

Oracle8i Distributed Database Systems, Release 8.1.5

A67784-01

Primary Author: Jason Durbin

Contributing Authors: William Creekbaum, Steve Bobrowski, 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

Contents

3 Distributed Transactions

Send Us Your Comments

Oracle8i Distributed Database Systems, Release 8.1.5

A67784-01

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 the Information Development department in the following ways:

- Electronic 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 World Wide Support Center.

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. However, several advanced features are available only with the Enterprise Edition, and some of these are optional.

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

How Oracle8i Distributed Database Systems is Organized

This book consists of two parts:

Part I: Distributed Database Systems

Chapter 1, "Distributed Database Concepts"

This chapter 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, "Distributed Database Administration"

This chapter discusses issues of concern to the database administrator (DBA) implementing or maintaining distributed databases.

Chapter 3, "Distributed Transactions"

This chapter describes how Oracle maintains the integrity of distributed transactions using the two-phase commit mechanism.

Chapter 4, "Distributed Database System Application Development"

This chapter describes the special considerations that are necessary if you are designing an application to run in a distributed system.

Part II: Heterogeneous Distributed Database Systems

Chapter 5, "Understanding Oracle Heterogeneous Services"

This chapter provides an overview of Oracle Heterogeneous Services.

Chapter 6, "Administering Oracle Heterogeneous Services"

This chapter explains how to implement and maintain Heterogeneous Services.

Chapter 7, "Application Development with Heterogeneous Services"

This chapter provides the information you will need to develop applications that use Oracle Heterogeneous Services.

Appendix A, "Heterogeneous Services Initialization Parameters"

This appendix lists all Heterogeneous Services-specific initialization parameters and their values.

Appendix B, "DBMS_HS Package Reference"

This appendix provides all the interface information for the DBMS_HS package. The DBMS_HS package is used to administer the heterogeneous services.

Appendix C, "DBMS_HS_PASSTHROUGH for Pass-Through SQL"

This appendix provides all the interface information for the DBMS_HS_PASSTHROUGH package for pass-through SQL.

Appendix D, "DBMS_DISTRIBUTED_TRUST_ADMIN Package Reference"

This appendix describes the procedures and functions in the package DBMS_DISTRIBUTED_TRUST_ADMIN for administering the Trusted Servers List.

Conventions Used in This Guide

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

UPPERCASE	Uppercase text identifies text that must be entered exactly as shown.
	For example:
	`SQLPLUS` *`username/password`*
	`INTO TABLENAME 'table'`

lowercase italics	Lowercase italics text is used for emphasis and to indicate glossary terms. It also identifies a variable for which you should substitute an appropriate value. Parentheses should be entered as shown. For example: `VARCHAR (`*`length`*`)`
Vertical bars |	Vertical bars indicate alternate choices. For example: `ASC | DESC`
Braces { }	Required items are enclosed in curly braces, meaning you must choose one of the alternatives. For example: `{`*`column_name`*` | `*`array_def`*`}`
Square brackets [ ]	Optional items are enclosed in square brackets. For example: `DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
<*operator*>	SQL operators are indicated by <*operator*>. For example: `WHERE` *`x <operator>`* `x`

UPPERCASE	Uppercase text identifies text that must be entered exactly as shown.
	For example:
	`SQLPLUS` *`username/password`*
	`INTO TABLENAME 'table'`

Ellipses ...	Repeated items are indicated by enclosure in square brackets and ellipses. For example:
	`WHERE` *`column_1 <operator>`* `x`
	`  AND` *`column_2 <operator> y`*
	` [AND ...]`

Your Comments Are Welcome

We value and appreciate your Comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of this manual is a Reader's Comment Form which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form has been used or you would like to contact us, please contact us at the following address:

Oracle8*i* Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
Fax: (650) 506-7228
Email: infodev@us.oracle.com (The Information Development department)

Part I

Distributed Database Systems

1

Distributed Database Concepts

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

- Oracle's Distributed Database Architecture
- Heterogeneous Distributed Databases
- Developing Distributed Database Applications
- Administering an Oracle Distributed Database System
- National Language Support

For information about features new to the current Oracle8*i* Release, please see *Getting to Know Oracle8i*.

Oracle's Distributed Database Architecture

A *distributed database* is a set of databases stored on multiple computers that typically appears to applications as a single database. Consequently, an application can simultaneously access and modify the data in several databases in a network. Each Oracle database in the system is controlled by its local Oracle server but cooperates to maintain the consistency of the global distributed database. Figure 1–1 illustrates a representative Oracle distributed database system.

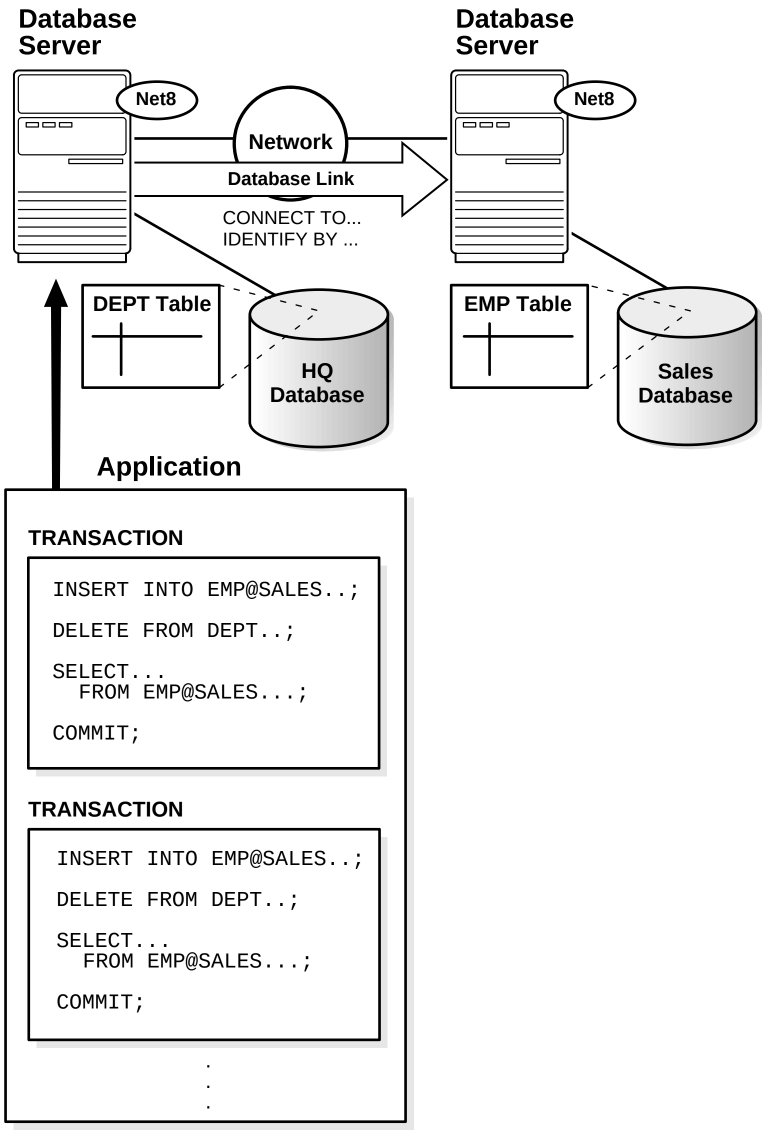
Clients and Servers

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 system is a *node*. A node in a distributed database system act as a client, a server, or both, depending on the situation. For example, in Figure 1–1, the computer that manages 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 query against the local DEPT table), and 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).

Direct and Indirect Connections

A client can connect directly or indirectly to a database server. In Figure 1–1, when the client application issues the first and third statements for each transaction, the client is connected directly to the intermediate HQ database and indirectly to the SALES database that contains the remote data.

Figure 1–1 An Oracle Distributed Database System



```
INSERT INTO EMP@SALES..;

DELETE FROM DEPT..;

SELECT...
  FROM EMP@SALES...;

COMMIT;
```

```
INSERT INTO EMP@SALES..;

DELETE FROM DEPT..;

SELECT...
  FROM EMP@SALES...;

COMMIT;
```

The Network

To link the individual databases of a distributed database system, a network is necessary. The following sections explain more about network issues in an Oracle distributed database system.

Net8

All Oracle databases in a distributed database system use Oracle's networking software, Net8, to facilitate inter-database communication across a network. Just as Net8 connects clients and servers that operate on different computers of a network, it also allows database servers to communicate across networks to support remote and distributed transactions in a distributed database.

Net8 makes transparent the connectivity that is necessary to transmit SQL requests and receive data for applications that use the system. Net8 takes SQL statements from a client and packages them for transmission to an Oracle server over a supported industry-standard communication protocol or programmatic interfaces. Net8 also takes replies from a server and packages them for transmission back to the appropriate client. Net8 performs all processing independent of an underlying network operating system. For more information about Net8 and its features, see the *Net8 Administrator's Guide*.

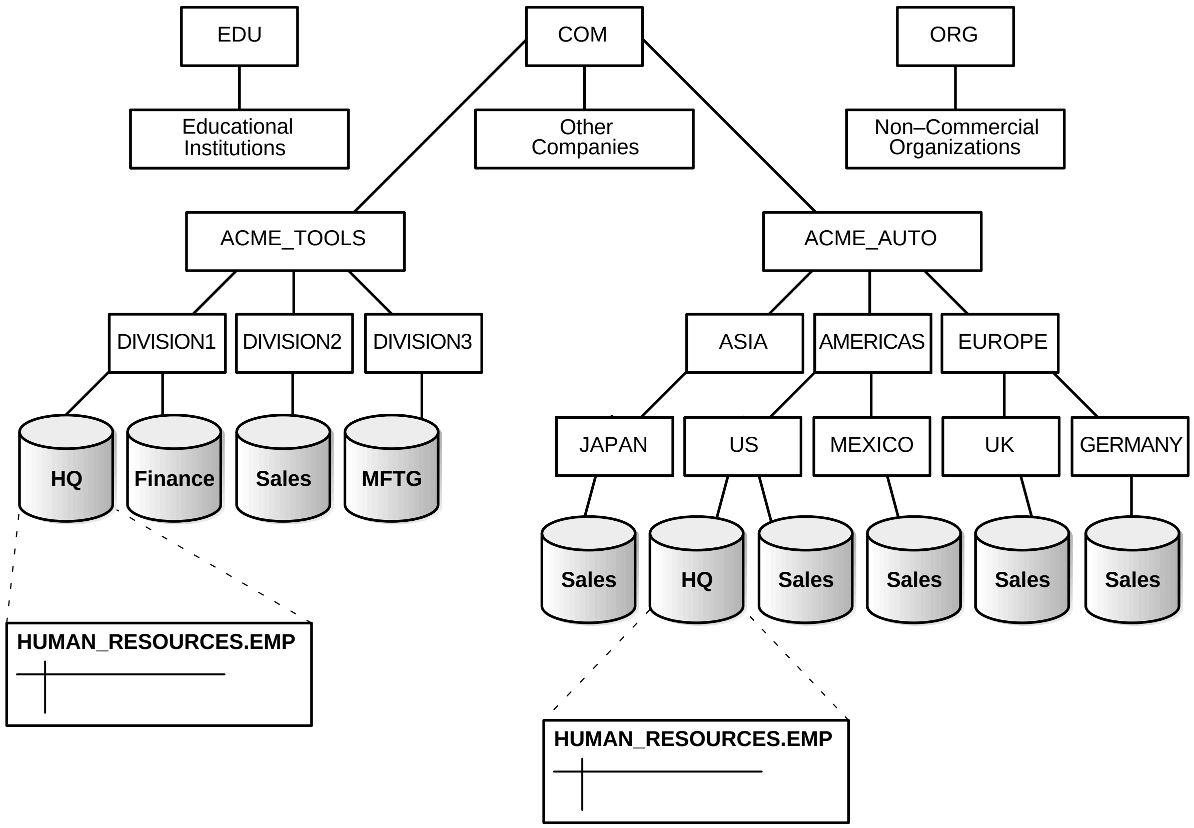
Oracle Names

Optionally, an Oracle network can use Oracle Names to provide the system with a global directory service. When an Oracle network supports a distributed database system, you can use Oracle Names servers as a central repositories of information about each database in the system to ease the configuration of distributed database access.

Databases and Database Links

Each database in a distributed database is distinct from all other databases in the system and has its own *global database name*. Oracle forms a database's global database name by prefixing the database's network domain with the individual database's name. For example, Figure 1–2 illustrates a representative hierarchical arrangement of databases throughout a network.

Figure 1–2 Network Directories and Global Database Names



While several database's can have the same 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.

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

Database Links

To facilitate application requests in a distributed database system, Oracle uses *database links*. A database link defines a one-way communication path from an Oracle database to another database.

Database links are essentially transparent to the users of an Oracle distributed database system, 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 SQL statement creates a database link in the local database that describes a path to the remote SALES.US.AMERICAS.ACME_AUTO.COM database.

```
CREATE DATABASE LINK sales.us.americas.acme_auto.com ... ;
```

After creating a database link, applications connected to the local database can access data in the remote SALES.US.AMERICAS.ACME_AUTO.COM database. The next section explains how applications can reference remote schema objects in a distributed database and includes examples of how SQL statements use database links.

Note: Oracle supports several different types of database links. For more information, see "Types of Database Links" on page 2-3.

Schema Object Name Resolution

To resolve application references to schema objects (a process called *name resolution*) Oracle forms object names using a hierarchical approach. For example, within a single database, Oracle guarantees that each schema 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 can easily resolve 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 simply 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.

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

To complete the request, the local database server implicitly uses a database link that connects to the remote SALES database.

Connecting Between Oracle Server Versions

An Oracle distributed database system can incorporate Oracle databases of different versions. All supported releases of Oracle can participate in a distributed database system. However, 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 available with Oracle8*i*.

Distributed Databases and Distributed Processing

The terms "distributed database" and "distributed processing" are closely related, but have very distinct meanings.

Distributed Database	A distributed database is a set of databases stored on multiple computers that appears to applications as a single database.
Distributed Processing	Distributed processing occurs when an application system distributes its tasks among different computers in a network. For example, a database application typically distributes front-end presentation tasks to client PCs or NCs 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 to function. For example, an Oracle server acts as a client when it requests data that another Oracle server manages.

Distributed Databases and Database Replication

The terms "distributed database" and "database replication" are also closely related, yet different. In a pure distributed database, the system manages a single copy of all data and supporting database objects. Distributed database applications typically use distributed transactions to access both local and remote data and modify the global database in real-time.

Note: This book discusses pure distributed databases.

Replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system. While replication relies on distributed database technology to function, database replication can offer applications benefits that are not possible within a pure distributed database environment.

Most commonly, replication is useful to improve the performance and protect the availability of applications because alternate data access options exist. For example, an application might 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.

Note: For more information about Oracle's replication features, see *Oracle8i Replication*.

Heterogeneous Distributed Databases

In an Oracle *heterogeneous distributed database system* at least one of the database systems is a non-Oracle system. To the application, the heterogeneous distributed database system appears as a single, local, Oracle database; the local Oracle server will be able to hide the distribution and heterogeneity of the data. The Oracle server accesses the non-Oracle system using Oracle8*i* Heterogeneous Services and a non-Oracle system-specific Heterogeneous Services Agent.

Heterogeneous Services

Heterogeneous Services is an integrated component within the Oracle8*i* server and the enabling technology for Oracle's next generation of Open Gateway products. Heterogeneous Services provides the common architecture and administration mechanisms for future Oracle gateway products and other heterogeneous access facilities, while providing upwardly compatible functionality for users of earlier Oracle Open Gateway releases.

See Chapter 5, "Understanding Oracle Heterogeneous Services" for more information.

Heterogeneous Services Agents

For each non-Oracle system that you want to access, Heterogeneous Services requires an *agent* to access that particular non-Oracle system. The Heterogeneous Services agent communicates with the non-Oracle system, and with the Heterogeneous Services component in the Oracle server. On behalf of the Oracle server, the agent executes SQL, procedure, and transactional requests at the non-Oracle system.

A version 8 Gateway is the Oracle product name for a Heterogeneous Services agent that accesses a non-Oracle system procedurally or using SQL. However, Heterogeneous Services agents will also become available as products other than Oracle Transparent Gateways or Oracle Procedural Gateways. Throughout this guide we will use the more generic term Heterogeneous Services agents. If you purchased an Oracle Open Gateway version 8, you can substitute "Oracle Gateway version 8" for Heterogeneous Services Agent.

See your "*Oracle Open Gateway Installation and User’s Guide version 8.0*" for detailed information on installation and configuration of version 8 gateways.

Features

The features of the Heterogeneous Services include:

- *Distributed Transactions.* A transaction can span both Oracle and non-Oracle systems, while still guaranteeing, through Oracle’s two phase commit mechanism, that changes are either all committed or all rolled back.
- *Transparent SQL access.* Integrate data from non-Oracle systems into the Oracle environment as if the data is stored in one single, local database. SQL statements issued by the application are transparently transformed into SQL statement understood by the non-Oracle system.
- *Procedural Access.* Procedural systems, like messaging and queuing systems, are accessed from an Oracle8*i* server using PL/SQL remote procedure calls.
- *Data Dictionary translations*. To make the non-Oracle system appear as another Oracle 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.* Optionally, application programmers can directly access a non-Oracle system from an Oracle application using the non-Oracle system's SQL dialect.

- *Accessing stored procedures.* Stored procedures in SQL-based non-Oracle systems are accessed as if they were PL/SQL remote procedures.
- *National Language Support.* Heterogeneous Services supports multi-byte character sets, and translate character sets between a non-Oracle system and the Oracle8*i* server.
- *Multi-Threaded Agents.* Multi-threaded agents take advantage of your operating system's threadin capabilities. Multi-threaded agents reduce the number of required processes by taking advantage of multi-threaded server capabilities.
- *Agent Self-Registration.* Agent self-registration automates the process of updating Hetergeneous Services configuration data on remote hosts, ensuring correct operation over heterogeneous database links.
- *Management Interface.* Provides a graphic representation of active Heterogeneous Services agents and of which user sessions are accessing those agents.

Note: Not all features listed above are necessarily supported by your Heterogeneous Services agent or Oracle Gateway. Please see your Heterogeneous Services agent or Oracle Open Gateway documentation for the supported features.

Developing Distributed Database Applications

When you build applications on top of a distributed database system, there are several issues to consider. The following sections explain how applications access data in a distributed database.

Distributed Query Optimization

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

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

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

See "Tuning Distributed Queries" on page 4-4 for more information.

Remote and Distributed SQL Statements

A *remote query* is a query that selects information from one or more remote tables, all of which reside at the same remote node. For example:

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

A *remote update* is an update that modifies data in one or more tables, all of which are located at the same remote node.

For example:

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

Note: A remote update may 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.

A *distributed query* retrieves information from two or more nodes. For example:

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

A *distributed update* 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:

```
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;
```

Statements in the program are sent to the remote nodes, and the execution of it succeeds or fails as a unit.

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:

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

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.

Remote and Distributed Transactions

A *remote transaction* is a transaction that contains one or more remote statements, all of which reference the same remote node. For example:

```
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;
```

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:

```
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

A DBMS must guarantee that all statements in a transaction, distributed or non-distributed, either commit or rollback as a unit, so that if the transaction is designed properly, the data in the logical database is always consistent. The effects of an ongoing transaction should be invisible to all other transactions at all nodes; this 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.

Note: For more information about Oracle's two-phase commit mechanism, see Chapter 3, "Distributed Transactions".

Transparency in a Distributed Database System

With minimal effort, you can make the functionality of 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.

Statement and Transaction 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*.

Administering an Oracle Distributed Database System

Just as there are unique issues to consider when developing applications for an Oracle distributed database system, there are special issues to understand for distributed database administration. The following sections explain the some special topics for managing databases in an Oracle distributed database system. See also Chapter 6, "Administering Oracle Heterogeneous Services"

Site Autonomy

Site autonomy means that each server participating in a distributed database is administered independently from all other databases, as though each database operates as a non-distributed database.

Although several databases can work together, each database is a distinct, separate repository of data that you manage 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 cooperating organizations that need to maintain an "arms length" relationship.
- Local database 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. The global Oracle database is partially available as long as one database and the network are available; no single database failure need halt all global operations or be a performance bottleneck.
- Administrators can recovery from isolated system failures independent of 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, that is not to say that you should ignore the global requirements of the system.

For example, additional user accounts might be necessary in each database are necessary to support the links that you create to facilitate server-to-server connections. The following sections explain more about these particular topics and demonstrate the need for a global perspective of the entire distributed database environment when managing individual nodes in the system.

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 or external service authentication for users and roles
- 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.

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.

- 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 what user accounts and roles each site needs to support server-to-server connections that use the links. See "Types of Database Links" on page 2-3 for more information about the user accounts that must be available to support different types of database links in the system.

Global Users and Roles

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

The use of a global authentication service is a common technique for simplifying security management for distributed environments.

In an Oracle client/server or distributed database environment, you have two options to support global authentication for users and roles:

- Oracle Security Server is a product that supports centralized authentication and distributed authentication in an Oracle network.

  Note: The global user functionality that was available in Oracle8 is being modified, and is currently available to beta customers only. It will be part of Oracle8*i* in a later release.

- When global database user and role authentication must work within the framework of a non-Oracle authentication service (for example, DCE), an Oracle distributed database environment can use Net8's Advanced Networking Option. The Net8 Advanced Networking Option is an optional product that bundles a number of features that you can use to enhance Net8 and the security of an Oracle distributed database system. See *Oracle Advanced Security Administrator's Guide* for more information.

Data Encryption

The Net8 Advanced Networking 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 Advanced Networking Option can generate a cryptographically secure message digest and include it with each packet sent across the network.

For more information about these and other features of Net8's Advanced Networking Option, see the *Net8 Administrator's Guide* and *Oracle Advanced Security Administrator's Guide*.

Tools for Administering Oracle Distributed Databases

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

- Oracle Enterprise Manager
- third-party administration tools
- SNMP support

Enterprise Manager

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

The line mode component of Enterprise Manager provides a line-mode interface.

Enterprise Manager provides administrative functionality via an easy-to-use interface. You can use Enterprise Manager to:

- Perform traditional administrative tasks, such as database startup, shutdown, backup, and recovery. Rather than manually entering the SQL commands to perform these tasks, you can use Enterprise Manager's graphical interface to execute the commands quickly and conveniently by pointing and clicking with the mouse.
- Concurrently perform multiple tasks. Because you can open multiple windows simultaneously in Enterprise Manager, you can perform multiple administrative and non-administrative tasks concurrently.
- 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 protocol(s) 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.
- Perform administrative tasks using Enterprise Manager's line-mode interface when a graphical user interface is unavailable or undesirable.

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 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

Additional Information: See the *Oracle SNMP Support Reference Guide*.

National Language Support

Oracle supports client/server environments where clients and servers use different character sets. The character set used by a client is defined by the value of the NLS_LANG parameter for the client session. The character set used by a server is its database character set. Data conversion is done automatically between these character sets if they are different. For more information about National Language Support features, refer to *Oracle8i Reference*.

2 Distributed Database Administration

This chapter discusses issues of concern to the database administrator (DBA) implementing or maintaining distributed databases.

Topics covered include:

Global Database Names and Global Object Names

In a distributed database system, each database should have a unique global name. *Global database names* identify each database in the system. A global database name consists of two components: a database name of eight characters or less (for example, SALES) and a domain name that contains the database (see below).

The domain name component of a global database name 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. The database name and the domain name are determined by the initialization parameters DB_NAME and DB_DOMAIN. See the *Oracle8i Reference* for more information about specifying these initialization parameters.

A database link should be given the same name as the global database name of the remote database it references. 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. See the *Oracle8i Reference* for more information about specifying the initialization parameter GLOBAL_NAMES.

Attention: If you set the initialization parameter GLOBAL_NAMES to FALSE, you are not required to use global naming. However, Oracle Corporation highly recommends that you use global naming because many useful features, including Oracle Advanced Replication, require global naming be enforced.

Once 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.

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

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

```
<schema>.<schema_object>@<global_database_name>
```

where

<schema>	A 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>`	A schema object is a logical data structure like a table, view, synonym, procedure, package, or a database link.
`<global_database_name>`	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.

For example, using the previously defined database link, a user or application can now reference remote data using the global object name:

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

Types of Database Links

To support application access to the data and schema objects throughout a distributed database system, administrators must create all necessary database links. The following sections compares the various types of database links that Oracle provides.

Private, Public, and Global Database Links

Oracle allows you to create private, public, and global database links.

Private Database Link	You can create a *private database link* in a specific schema of a database. Only the owner of a private database link or PL/SQL subprograms in the schema can use a private database link to access data and database objects in the corresponding remote database.
Public Database Link	You can create a *public database link* for a database. All users and PL/SQL subprograms in the database can use a public database link to access data and database objects in the corresponding remote database.
Global Database 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. All users and PL/SQL subprograms in any database can use a global database link to access data and database 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 some of the advantages and disadvantages for using each type of database link.

- A *private database 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 private link to access the specified remote database.
- When many users require an access path to a remote Oracle database, an administrator can create a single *public database link* for all users in a database.
- 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.

Creating a Private Database Link

To create a private database link, you specify:

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

See the *Oracle8i SQL Reference* and the following sections for more information.

Creating a Public Database Link

To create a public database link, you use the keyword PUBLIC:

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

See the *Oracle8i SQL Reference* and the following sections for more information.

Creating a Global Database Link

You must define *global database links* in the Oracle Name Server. See your *Net8 Administrator's Guide* for more information.

Security Options for Database Links

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.

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 ...;
```

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 user's credentials across the network when an application uses the link to access the remote database. If an unsecure network supports a distributed database that uses fixed user database links, consider encrypting login packets for server-to-server connections.

Connected User and Current User Database Links

Connected user and current user database links do not include any 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 being performed by the application. To understand the difference between the two types of database links, you must first understand the concepts of connected and current users.

- A *connected user* is a user that connects to a database using a database application. For example, when you start SQL*Plus and connect to an Oracle database as SCOTT, the connected user is SCOTT.
- A *current user* is determined by the security context in which a database operation executes. For example, when you connect to an Oracle database as the user SCOTT and execute the procedure SALES.DEL_EMP, the current user while executing the DEL_EMP procedure defaults to SALES because a stored procedure executes within the security context of its owner.

To understand the difference between connected user and current user database links, simply extend your understanding of the different types of users. With a *connected user database link*, an operation being performed in a remote database always occurs within the security context of the *connected user at the local database*.

With a *current user database link*, an operation performed in a remote database always occurs within the security context of the *current user at the local database*.

For example, consider what happens when the user SCOTT calls a procedure SALES.DEL_EMP, and the procedure deletes an employee record from a remote database. If the procedure references a connected user database link to access the remote database, the deletion of the remote employee record happens as SCOTT, the connected user in the local database. However, if the procedure references a current user database link to access the remote database, the deletion of the remote employee record happens as SALES, the current user in the local database.

To create a connected user database link, you merely omit the CONNECT TO clause. The following example creates a connected user database link:

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

To create a current user database link, use the following syntax:

```
CREATE DATABASE LINK ... CONNECT TO CURRENT_USER ...;
```

To use a current user database link, the current user must be a global user that is authenticated by the Oracle Security Server.

Note: The global user functionality that was available in Oracle8 is being modified, and is currently available to beta customers only. It will be part of Oracle8*i* in a later release.

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

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. To use shared database links, *the local server must run in multi-threaded server (MTS) mode.* The remote server can either run in multi-threaded server mode, or can run in dedicated server mode.

Attention: Although *shared database links* can reduce the number of required connections between the local and remote server, you could cause more physical connections and processes to be required than simply using standard (not shared) database links, if you use this functionality incorrectly.

Please be sure you understand the information presented in this section before attempting to implement a system that uses shared database links with shared servers.

Properties of Shared Database Links

Shared database links differ from standard database links in two ways:

- Network connections made for shared database links can be shared among those that use the same database link schema object. When a user needs to establish a connection to a remote server from a particular shared server process, the shared process can reuse connections already established to the remote server (if that connection was established on the same shared server with the same database link).
- When you use a shared database link, a network connection is established directly out of the shared server in the local server. For a normal (non-shared) database link, if the local server was a multithreaded 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.

When to Use Shared Database Links

You should look carefully at your application and your multi-threaded server configuration to determine whether to use shared links or not.

For example, if you have designed your application to use a standard public database link, and 100 users simultaneously require a connection, 100 network direct network connections will also be required.

However, if your application uses shared database links, and there are ten shared servers in the local MTS-mode database, the 100 users that use the same (shared public) database link will require only 10 network connections (or fewer) to the remote server. Each local shared server may only need one connection to the remote server.

When Not to Use Shared Database Links

Shared database links are not useful in all situations. Suppose there is only one user that accesses the remote server. If that user defines a shared database link, and there are ten shared servers in the local database, that one user can require up to 10 network connections to the remote server. Every shared server may have established a connection to the remote server, since each shared server might have been used by that user.

Clearly, a standard database link would be preferable in this situation because it would require (and allow) only one network connection. The lesson: shared database links can lead to more network connections in single-user scenarios, therefore, they should be used only when you expect that many users will need to use the same database link. Typically, this is the case for public database links, but may also be true for private database links if you expect many clients to use the same local schema (and therefore the same private database link).

A rule of thumb 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 shared servers in the local database.

Setting Up Shared Database Links

To create a shared database link you use the keyword SHARED in the SQL CREATE DATABASE LINK command:

```
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'];
```

See the *Oracle8i SQL Reference* for more syntax information.

Whenever the keyword SHARED is used, the clause AUTHENTICATED BY is also required. There must be an account on the remote database with the specified USERID / PASSWORD and with the CREATE SESSION privilege. No other privileges are required.

The schema specified in the AUTHENTICATED BY clause is only used for security reasons and could 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 unauthorized information.

Shared Database Link Configurations

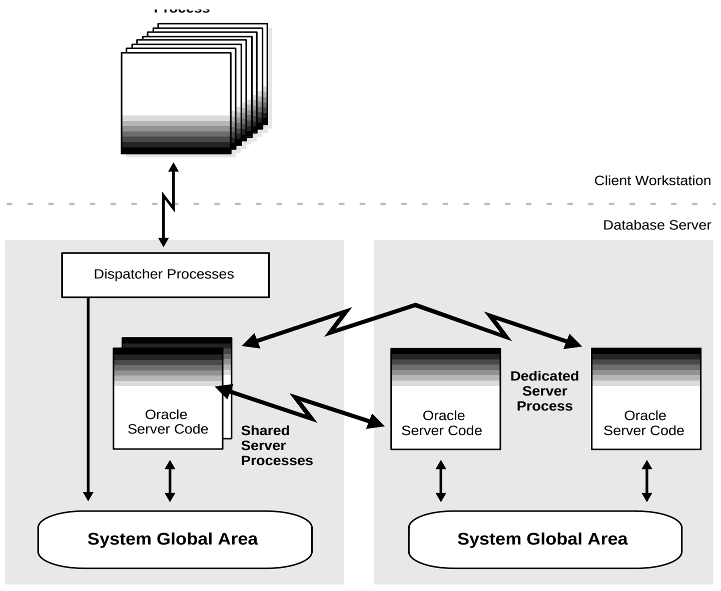
Shared database links can be used in two configurations.

Shared Database Links to Dedicated Servers In the first configuration a shared server in the local server owns a dedicated remote server, and a direct network transport connection exists between the shared server and the remote dedicated server. The advantage is that a direct network transport exists between the local shared server and the remote dedicated server. A disadvantage of this configuration is that extra back-end servers are needed. See Figure 2–1.

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

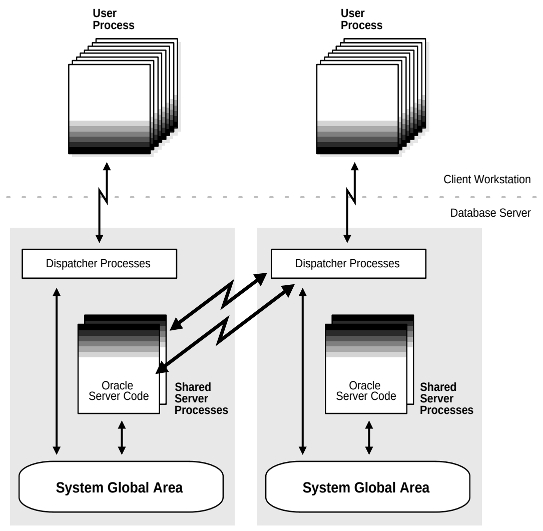
Figure 2–1 A Shared Database Link to Dedicated Server Processes

.



Shared Database Links to Multi-Threaded Servers The second configuration uses shared servers on the remote server. This configuration eliminates the need for more dedicated servers, but requires to go through the dispatcher on the remote server. See Figure 2–2. 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



Examples

Example 1: A Public Fixed User Database Link The following statement creates a public fixed user database link:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com
CONNECT TO SCOTT IDENTIFIED BY TIGER
USING 'sales';
```

Any user connected to the local database can use the SALES.DIVISION3.ACME.COM database link to connect to the remote database. Each user will connect to the same remote schema, SCOTT in the remote database. To access the table EMP table in SCOTT's remote schema, a user could issue the 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.

Example 2: A Public Fixed User Shared Database Link Consider the following example of creating a public fixed user shared database link:

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

Note: The local database must be configured in multi-threaded server mode.

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

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

Example 3: A Public Connected User Database Link The following statement would create a public connected user database link:

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

Any user connected to the local database can use the SALES.DIVISION3.ACME.COM 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, the database link connects to the remote schema SCOTT. If FORD is the connected user and uses the database link, the database link connects to FORD's remote schema.

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

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

Example 4: A Public Connected User Shared Database Link The following statement creates a public connected user shared database link:

```
CREATE SHARED PUBLIC DATABASE LINK sales.division3.acme.com AUTHENTICATED
BY ward IDENTIFIED BY orange
USING ‘sales’;
```

Note: The local database server must be configured in multi-threaded server mode.

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.

In the above example, each local shared server will 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 will reuse 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 that point no more physical connections will be needed to the remote server, even if new users use this shared database link.

Example 5: A Public Current User Database Link The following statement creates a public current user database link:

```
CREATE PUBLIC DATABASE LINK sales.division3.acme.com
CONNECT TO CURRENT_USER
USING ‘sales’;
```

Note: to use this database link, the current user must be a global user (Global users require authentication through the Oracle Security Server).

Note: The global user functionality that was available in Oracle8 is being modified, and is currently available to beta customers only. It will be part of Oracle8*i* in a later release.

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

```
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;
```

When FORD executes the procedure SCOTT.FIRE_EMP, the procedure runs under SCOTT's privileges. Since a current user database link is used, the connection is established to SCOTT's remote schema. (Note that, were a connected user database link used instead, the connection would be established to FORD's remote schema.) Note that SCOTT must be a global user and FORD may or may not be a global user.

Note: The global user functionality that was available in Oracle8 is being modified, and is currently available to beta customers only. It will be part of Oracle8*i* in a later release.

Note that the same could have been accomplished by using a fixed user database link that connects to SCOTT's remote schema. However, with fixed user database links, security can be compromised, because SCOTT's username and password are available in readable format in the database to DBAs.

Connection Qualifiers

In some situations, you may want to have several database links of the same type (e.g., public) that point to the same remote database, yet establish connections to the remote database using different communication pathways. For example, if a remote database is using the Oracle Parallel Server, you might want to define several public database links at your local node so that connections can be established to specific instances of the remote database.

To facilitate such functionality, Oracle allows you to create a database link with an optional connection qualifier in the database link name. When creating a database link, a connection qualifier is specified as the trailing portion of the database link name, separated by an at sign ("@"). For example, assume that a 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 above examples that a connection qualifier is simply an extension to a database link name. The text of the connection qualifier 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 connection qualifier is not specified. In this case, just as when a connection qualifier is specified, the instance is determined by the USING string.

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

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

Database Link Resolution

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. Oracle does this 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 Name Server is available).

If a SQL statement specifies a complete global database name; that is, both the database and domain components are specified, Oracle searches for private, public, and global database links that match only the explicitly specified global database name. If any portion of the domain is specified, Oracle assumes that a complete global database name is specified. However, if a SQL statement specifies a partial global database name; that is, only the database component is specified, Oracle appends the local database's network domain component to the database name to form a complete global database name. Then Oracle searches for private, public, and network database links that match only the constructed global database name. If a matching database link is not found, Oracle returns an error and the SQL statement cannot execute.

Optimization: If a global object name references an object in the local database and a connection qualifier is not specified, Oracle automatically detects that the object is local and does not search for, or use, database links to resolve the object reference.

Oracle expands a global object reference, whether or not a connection qualifier is specified. Furthermore, if a connection qualifier is specified, only database links that match, including the connection qualifier, are used to resolve the object reference.

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

The first match determines the remote schema; that is, if no CONNECT clause is specified a connected user database link will be used, if the "CONNECT TO username IDENTIFIED BY password" clause is specified a fixed user database link will be used, and if a "CONNECT TO CURRENT_USER" clause is specified a current user database link.

If the first match does not specify a USING clause, the search continues until a link that specifies a database string is found. If matching database links are found and a database string is never identified, Oracle returns an error.

Once a complete path is determined, Oracle creates a remote session, assuming an identical connection is not already open on behalf of the same local session.

Schema Object Name Resolution

Once 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. That is, if a fixed user database link is used, object resolution proceeds in the specified schema; if a connected user database link is used, object resolution proceeds in the connected user's remote schema (including synonyms) and if a current user database link is used object resolution proceeds in the current user's remote schema. If the object is not found, public objects of the remote database are then checked.

If an object is not resolved, the established remote session remains but the SQL statement cannot execute.

Examples of 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, and an Oracle Name Server (and therefore, global database links) is not available.

Example 1 This example illustrates how a complete global object name is resolved and the appropriate path to the remote database is determined 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 at the local database:

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

CONNECT jward/bronco;

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 notices that a complete global object name is referenced in JWARD'S UPDATE statement. Therefore, it begins searching in the local database for a database link with a matching name. Oracle finds matching private database link in the schema JWARD. However, 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 and finds a matching public database link. From this public database link, Oracle takes the service name. Combined with the remote account taken from the matching private fixed user database link, a complete path is determined and Oracle proceeds to establish a connection to the remote SALES database as the user TSMITH / RADIO.

The remote database can now resolve the object reference to the EMP table. Oracle searches in the specified schema, TSMITH, and finds the referenced EMP table. No further resolution is necessary; the remote database completes the execution of the statement, and returns the results to the local database.

Example 2 This example illustrates how a partial global object name is resolved and the appropriate path to the remote database is determined using both a private and public database link.

For this example, assume that a remote table EMP is contained in the schema TSMITH and a remote public synonym named EMP points to the previously mentioned EMP table. Also assume the creation of the public database link in Example 1.

Consider the following statements issued at the local database:

```
CONNECT scott/tiger;

CREATE DATABASE LINK sales.division3.acme.com;

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

Oracle notices that a partial global object name is referenced in SCOTT's DELETE statement. First, the global object name is expanded to a complete global object name using the domain name of the local database, resulting in the following statement:

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

Now, Oracle begins searching in the local database for a database link with a matching name. Oracle finds a matching private connected user database link in the schema SCOTT. However, 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. Oracle takes the database string from the public database link. At this point, a complete path is determined and Oracle can connect to the remote database as SCOTT/TIGER.

Once connected to the remote database as SCOTT, the remote Oracle resolves the reference to EMP. First, it searches for and does not find an object named EMP in the schema SCOTT.

Next, the remote database searches for a public synonym named EMP and finds one. The remote database then completes statement execution and returns the results to the local database.

Views, Synonyms, Procedures and Global Name Resolution

A remote schema object can be referenced by its global object name in the definition of a view, synonym, or PL/SQL program unit (e.g., procedure, trigger). If a complete global object name is referenced in the definition of a view, synonym, or program unit, Oracle stores the definition of the object as specified, without having to perform any expansion of the global object name being referenced. However, if a partial global object name (that is, only the database name and not the domain name) is referenced in the definition of a view, synonym, or program unit, Oracle must expand the partial name using the domain component of the local database's global database name.

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

- When a view is created, partial global object names in the defining query are not expanded; 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.
- When a synonym is created, partial global object names are expanded; the definition of the synonym stored in the data dictionary includes the expanded global object name.
- Each time a program unit is compiled, partial global object names are expanded.

The above behavior should be considered when creating views, synonyms, and procedures that reference remote data using partial global object names. If the global database name of the containing database is changed (which should rarely happen), views and procedures may try to reference a different database than they did before the global database name change; alternatively, synonyms do not expand database link names at runtime, so they do not change. Depending on the situation, this behavior may or may not be desired. For example, consider two databases named SALES.UK.ACME.COM and HQ.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@hq;

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

Oracle expands the EMPLOYEE synonym definition and stores it as:

```
"scott.emp@hq.uk.acme.com"
```

The company undergoes a reorganization. First, consider the situation where both the Sales and Human Resources departments are moved to the United States. Consequently, the corresponding global database names are both changed to SALES.US.ACME.COM and HQ.US.ACME.COM. In this case, the defining query of the EMPLOYEE_NAMES view still expands to the correct database when the view is used:

```
"SELECT ename FROM scott.emp@hq.us.acme.com"
```

However, the definition of the EMPLOYEE synonym continues to reference the previous database name, HQ.UK.ACME.COM.

Now consider that only the Sales department is moved to the United States. Consequently, the corresponding new global database name is SALES.US.ACME.COM, while the Human Resources database is HQ.UK.ACME.COM. In this case, the defining query of the EMPLOYEE_NAMES view expands to a non-existent global database name when the view is used:

```
"SELECT ename FROM scott.emp@hq.us.acme.com"
```

Alternatively, the EMPLOYEE synonym continues to reference the correct database, HQ.UK.ACME.COM.

In summary, you should decide when you want to use partial and complete global object names in the definition of views, synonyms, and procedures. Keep in mind that database names should be stable and databases should not be unnecessarily moved within a network.

Dropping a Database Link

You can drop a database link just as you can drop a table or view. The command syntax is:

```
DROP DATABASE LINK dblink;
```

For example, to drop the database link NY_FIN, the command would be:

```
DROP DATABASE LINK ny_fin;
```

Listing Available Database Links

The data dictionary of each database stores the definitions of all the database links in that database. The USER / ALL / DBA_DB_LINKS data dictionary views show the database links that have been defined at the local database.

Any user can query the data dictionary to determine what database links are available to that user. For information on viewing the data dictionary, see *Oracle8i Concepts* or the *Oracle8i SQL Reference*.

Limiting the Number of Active Database Links

You can limit the number of connections from a user process to remote databases with the initialization parameter OPEN_LINKS. This parameter controls the number of remote connections that a single user session can use concurrently within a single SQL statement per session. See the *Oracle8i SQL Reference* for more information.

Techniques for Location Transparency

Users of a distributed database system need not (and often should not) be aware of the location and functioning of the parts of the database with which they work. The DBA and network administrators can ensure that the distributed nature of the database remains transparent to users, as shown in the following sections.

Views and Location Transparency

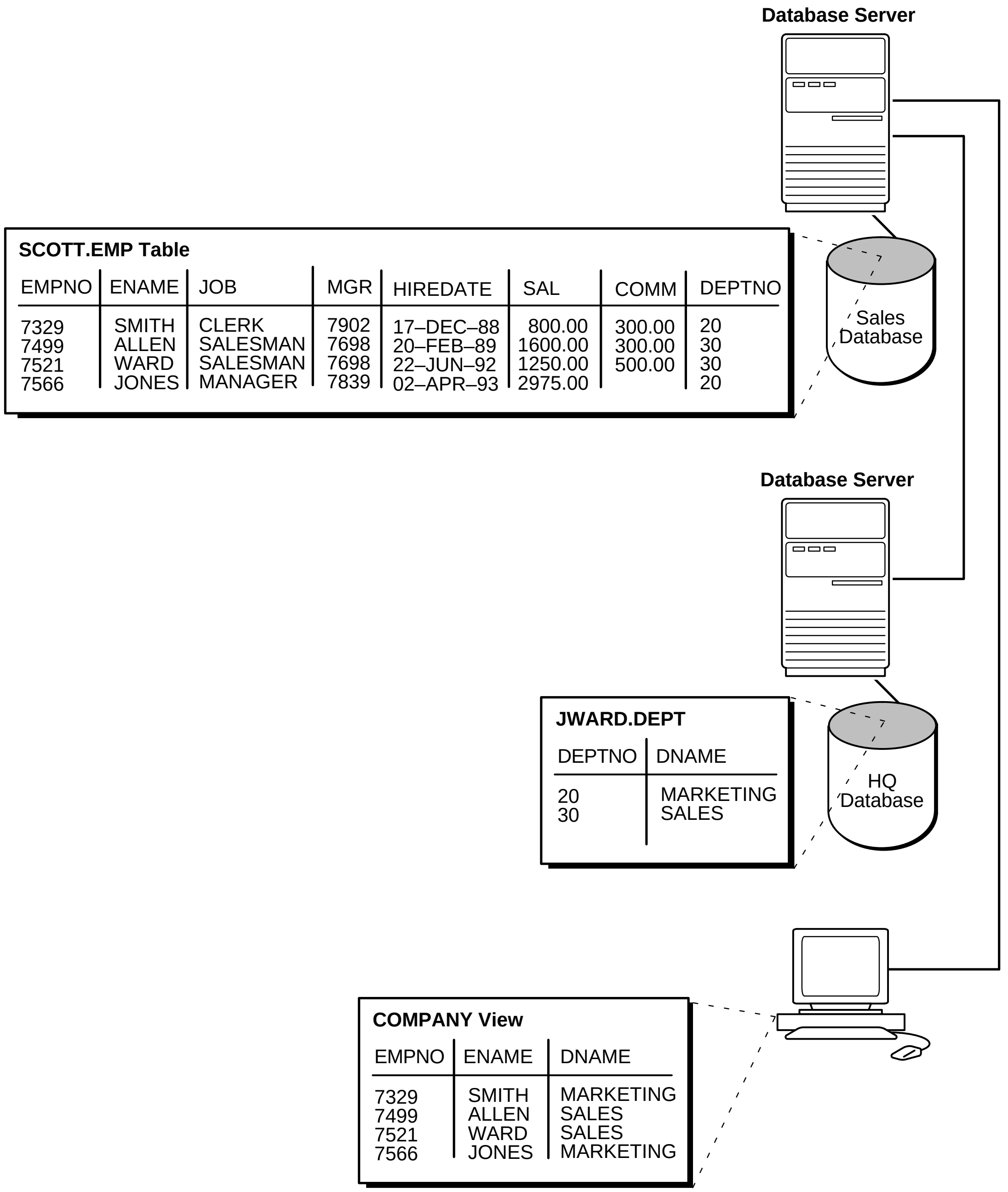
Local views can provide location transparency for local and remote tables in a distributed database system.

For example, assume that table EMP is stored in a local database. Another table, DEPT, is stored in a remote database.

To make the location of, and relationship between, these tables transparent to users of the system, a view named COMPANY can be created in the local database that joins the data of the local and remote servers:

```
CREATE VIEW company AS
SELECT empno, ename, dname
FROM emp a, dept@hq.acme.com b
HERE 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 know, or 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:

```
SELECT * FROM company;
```

provides data from both the local and remote database table.

Figure 2–3 illustrates this example of location transparency.

Views and Privileges

Assume a local view references a remote table or view. The owner of the local view can grant only those object privileges on his 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.

Synonyms and Location Transparency

Synonyms are very 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 the underlying object must be renamed or be moved, only the synonym needs to be redefined; applications based on the synonym continue to function without modification. Synonyms can also simplify SQL statements for users in a distributed database system.

A synonym can be created for any table, type, view, snapshot, sequence, procedure, function, or package. All synonyms 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, isolating 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]	Specifies 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.

[PUBLIC]	Specifies 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.
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.

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;
```

An employee management application can be designed 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 the EMP table is moved from the HQ database to the HR database, only the public synonyms need to be changed on the nodes of the system. The employee management application continues to function properly on all nodes.

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, Oracle always finds the schema object when the user that owns the schema references that name.

Synonyms and Privileges

A synonym is a reference to the actual object. A user who has access to a synonym for a particular schema object, must also have privileges on 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 a local synonym is an alias for a remote object. The owner of the local synonym *cannot* grant any object privileges on the synonym to any other local user. This behavior is different from privilege management for synonyms that are aliases for local tables or views. In the case where a synonym is an alias for a remote object, local privileges for the synonym cannot be granted, because this would amount to granting privileges for the remote object, which is not allowed. Therefore, no local privilege management can be performed when synonyms are used for location transparency; security for the base object is controlled entirely at the remote node. For example, the user ADMIN cannot grant any 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.

Procedures and Location Transparency

PL/SQL program units called *procedures* can also provide location transparency. There are two options:

- A local procedure references remote data
- A local synonym references a remote procedure

The second option provides location transparency through synonyms. This is discussed in "Synonyms and Location Transparency" on page 2-22. The first option is discussed in the next section.

Local procedure referencing 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 if 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, the FIRE_EMP procedure can be defined with the following statement:

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

If the table EMP@HQ is renamed or moved, only the local synonym that references the table needs to be modified. None of the procedures and applications that call the procedure require modification.

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.

Statement Transparency

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

- SELECT (queries)
- INSERT
- UPDATE
- DELETE
- SELECT... FOR UPDATE
- 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 empno, ename, dname FROM scott.emp@sales.division3.acme.com e,
jward.dept@hq.acme.com d
  WHERE d.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;
```

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 data definition language (DDL) statements (for example, CREATE, ALTER, and DROP).

The LIST CHAINED ROWS clause of an ANALYZE statement cannot reference remote tables.

Values for Environmentally-Dependent SQL Functions

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.

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, the referenced objects must match, and the bind types of any bind variables must be the same. If available, shared SQL areas can be used for the local and remote handling of any statement (or decomposed query).

3 Distributed Transactions

This chapter describes how Oracle8*i* maintains the integrity of distributed transactions. Topics include:

Distributed Transaction Management

All participants (nodes) in a distributed transaction should be unanimous as to the action to take on that transaction. That is, they should either all commit or rollback.

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 a transaction management mechanism known as two-phase commit. This mechanism is completely transparent. Its use requires no programming on the part of the user or application developer.

The next sections explain how the two-phase commit mechanism works.

The Prepare and Commit Phases

The committing a distributed transaction has two distinct phases:

prepare phase	The global coordinator (initiating node) asks participants to prepare (to promise to commit or rollback the transaction, even if there is a failure).
commit phase	If all participants respond to the coordinator that they are prepared, the coordinator asks all nodes to commit the transaction. If any participants cannot prepare, the coordinator asks all nodes to roll back the transaction.

When a user commits a distributed transaction with a COMMIT statement, both phases are performed automatically. The following sections describe each phase in further detail.

Prepare Phase

The first phase in committing a distributed transaction is the prepare phase in which the commit of the transaction is not actually carried out. Instead, all nodes referenced in a distributed transaction (except one, known as the commit point site, described in the"The Commit Point Site" on page 3-8) are told to prepare (to commit).

By preparing, a node records enough information so that it can subsequently either commit or abort the transaction (in which case, a rollback will be performed), regardless of intervening failures.

When a node responds to its requestor that it has prepared, the prepared node has made a promise to be able to either commit or roll back the transaction later and not to make a unilateral decision on whether to commit or roll back the transaction.

Note: Queries that start after a node has prepared cannot access the associated locked data until all phases are complete (an insignificant amount of time unless a failure occurs).

When a node is told to prepare, it can respond with one of three responses:

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 prepare is necessary.
abort	The node cannot successfully prepare.

Prepare Phase Actions by Nodes

To complete the prepare phase, each node performs the following actions:

- The node requests its descendants (nodes subsequently referenced) to prepare.
- The node checks to see if the transaction changes data on that node or any of its descendants. If there is no change, the node skips the next steps and replies with a read-only message (see below).
- The node allocates all resources it needs to commit the transaction if data is changed.
- The node flushes any entries corresponding to changes made by that transaction to its local redo log.
- The node guarantees that locks held for that transaction are able to survive a failure.
- The node responds to the node that referenced it in the distributed transaction with a prepared message *or*, if its prepare or the prepare of one of its descendents was unsuccessful, with an abort message (see below).

These actions guarantee that the transaction can subsequently commit or roll back on that node. The prepared nodes then wait until a COMMIT or ROLLBACK is sent. Once the node(s) are prepared, the transaction is said to be *in-doubt*.

Read-only Response

When a node is asked to prepare and the SQL statements affecting the database do not change that node's data, the node responds to the node that referenced it with a read-only message. These nodes do not participate in the second phase (the commit phase). For more information about read-only distributed transactions, see "Read-Only Distributed Transactions" on page 3-17

Unsuccessful Prepare

When a node cannot successfully prepare, it performs the following actions:

- That node releases any resources currently held by the transaction and rolls back the local portion of the transaction.
- The node 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 to roll back the transaction and guarantee the integrity of the data in the global database.

Again, this 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.

Commit Phase

The second phase in committing a distributed transaction is the commit phase. Before this phase occurs, *all* nodes referenced in the distributed transaction have guaranteed that they have the necessary resources to commit the transaction. That is, they are all prepared.

Therefore, the commit phase consists of the following steps:

1. The global coordinator send a message to all nodes telling them to commit the transaction.
2. At each node, Oracle8*i* commits the local portion of the distributed transaction (releasing locks) and records an additional redo entry in the local redo log, indicating that the transaction has committed.

When the commit phase is complete, the data on all nodes of the distributed system are consistent with one another.

A variety of failure cases, caused by network or system failures, are possible during both the prepare phase and the commit phase. For a description of failure situations and how Oracle8*i* resolves intervening failures during two-phase commit, see "Troubleshooting Distributed Transaction Problems" on page 3-19.

The Session Tree

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 between sessions and their roles. All nodes participating in the session tree of a distributed transaction assume one or more roles:

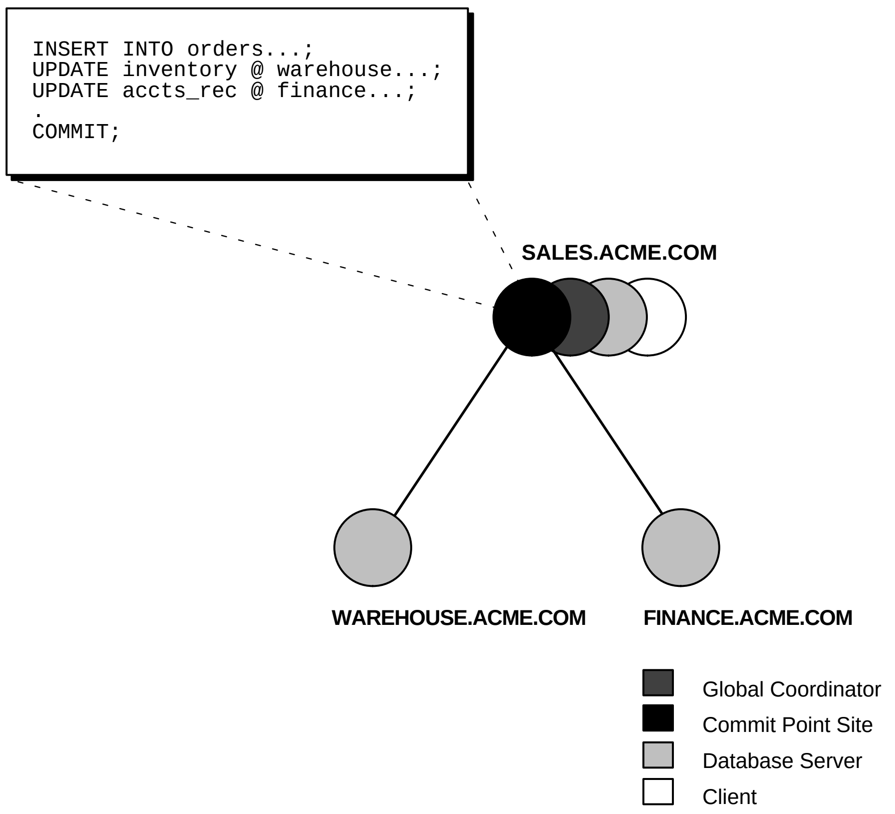
- a client
- a database server
- a global coordinator
- a local coordinator
- the commit point site (see page 3 - 8)

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

- whether the transaction is local or remote
- the *commit point strength* of that node (see page 3 - 8)
- 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

Figure 3–1 below illustrates a simple session tree.

Figure 3–1 An example of a Simple Session Tree



Clients

A node acts as a client when it references information from another node's database. The referenced node is a *database server*. In the above example, the node SALES.ACME.COM is a client of the nodes (database servers) that serve the WAREHOUSE and FINANCE databases.

Servers and Database Servers

A server is a node that is directly referenced in a distributed transaction or is requested to participate in a transaction because another node requires data from its database. A node supporting a database is also called a database server.

In Figure 3–1, an application at the node holding the SALES database initiates a distributed transaction which accesses data from the nodes holing the WAREHOUSE and FINANCE databases.

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 is also requesting a change to the SALES database's information.

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 3–1, SALES.ACME.COM, although it happens to be the global coordinator, is also considered a local coordinator because it coordinates the nodes it directly references: WAREHOUSE.ACME.COM and FINANCE.ACME.COM.

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 node
- receiving queries from those nodes and passing them on to other nodes
- returning the results of queries to the nodes that initiated them

The Global Coordinator

The node where the distributed transaction originates (to which the database application issuing the distributed transaction is directly connected) is called the global coordinator. This node becomes the parent or root of the session tree. The global coordinator performs the following operations during a distributed transaction:

- All of the distributed transaction's SQL statements, remote procedure calls, etc. are sent by the global coordinator to the directly referenced nodes, thus forming the session tree.

  For example, in Figure 3–1, the transaction issued at the node SALES.ACME.COM references information from the database servers WAREHOUSE.ACME.COM and FINANCE.ACME.COM.

  Therefore, SALES.ACME.COM is the global coordinator of this distributed transaction.

- The global coordinator instructs all directly referenced nodes other than the commit point site (see below) to prepare the transaction.
- If all nodes prepare successfully, the global coordinator instructs the commit point site to initiate the global commit of the transaction.
- If there is one or more abort messages, the global coordinator instructs all nodes to initiate a global rollback of the transaction.

The Commit Point Site

The job of the commit point site is to initiate a commit or roll back 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 (see below). The node selected as commit point site should be that node that stores the most critical data (the data most widely used)

The commit point site is distinct from all other nodes involved in a distributed transaction with respect to the following two issues:

- The commit point site never enters the prepared state. This is potentially advantageous because if the commit point site stores the most critical data, this data never remains in-doubt, even if a failure situation occurs. (In failure situations, failed nodes remain in a prepared state, holding necessary locks on data until in-doubt transactions are resolved.)
- 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 global coordinator ensures that all nodes complete the transaction the same way that the commit point site does.

A distributed transaction is considered to be committed once all nodes are prepared and the transaction has been committed at the commit point site (even though some participating nodes may still be only in the prepared state and the transaction not yet actually committed).

The commit point site’s redo log is updated as soon as the distributed transaction is committed at that node. Likewise, a distributed transaction is considered *not* committed if it has not been committed at the commit point site.

Commit Point Strength

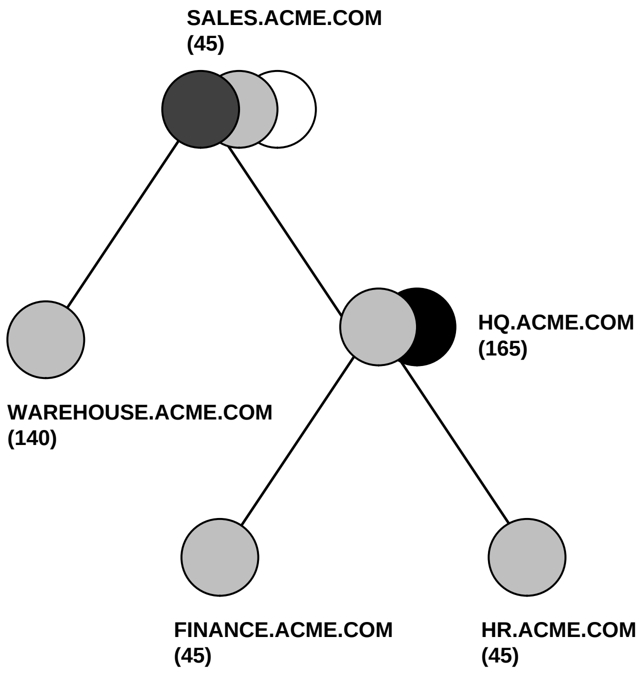
Every node acting as a database server must be assigned a commit point strength. If a database server is referenced in a distributed transaction, the value of its commit point strength determines what role it plays in the two-phase commit. Specifically, the commit point strength determines whether a given node is the commit point site in the distributed transaction.

This value is specified using the initialization parameter COMMIT_POINT_STRENGTH (see page 3 - 8). The commit point site is determined at the beginning of the prepare phase.

The commit point site is selected only from the nodes participating in the transaction. Once it has been determined, the global coordinator sends prepare messages to all participating nodes. Of the nodes directly referenced by the global coordinator, the node with the highest commit point strength is selected. Then, the initially-selected node determines if any of its servers (other nodes that it has to obtain information from for this transaction) has a higher commit point strength.

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. Figure 3–2 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 3–2 Commit Point Strengths and Determination of the Commit Point Site



The following conditions apply when determining the commit point site:

- A read-only node (a node which will not change its local data for the transaction) cannot be designated as the commit point site.
- If multiple nodes directly referenced by the global coordinator have the same commit point strength, Oracle8*i* will designate one of these nodes as the commit point site.
- If a distributed transaction ends with a rollback, the prepare and commit phases are not needed, consequently a commit point site is never determined. Instead, the global coordinator sends a ROLLBACK statement to all nodes and ends the processing of the distributed transaction.

The commit point strength only determines the commit point site in a distributed transaction. 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.

As Figure 3–2 illustrates, the commit point site and the global coordinator can be different nodes of the session tree.

The commit point strengths of each nodes is communicated to the coordinator(s) when the initial connections are made. The coordinator(s) 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.

Specifying the Commit Point Strength of an Instance

Specify a commit point strength for each node that insures that the most critical server will be "non-blocking" if a failure occurs during a prepare or commit phase.

A node's commit point strength is set by the initialization parameter COMMIT_POINT_STRENGTH. 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 parameter file:

```
COMMIT_POINT_STRENGTH=200
```

Additional Information: See your Oracle operating system-specific documentation for the default value.

A Case Study

This case study 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

The Scenario

A company that has separate Oracle8*i* servers, SALES.ACME.COM and WAREHOUSE.ACME.COM. As sales records are inserted into the SALES database, associated records are being updated at the WAREHOUSE database.

The Process

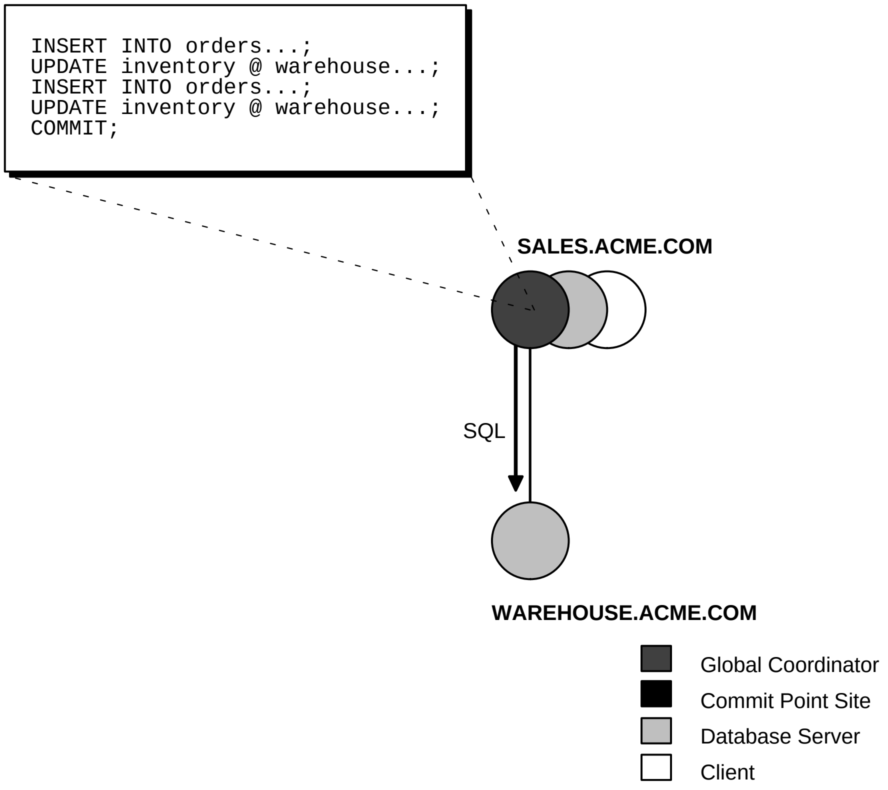
The following steps are carried out during a distributed transaction that enters a sales order:

1. An application issues SQL statements.

   At the Sales department, a salesperson uses a database application to enter, then commit a sales order. The application issues a number of SQL statements to enter the order into the SALES database and update the inventory in the WAREHOUSE database.

   These SQL statements are all part of a single distributed transaction, guaranteeing that all issued SQL statements succeed or fail as a unit. This prevents the possibility of an order being placed but, inventory is not 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 3–3.

Figure 3–3 Defining the Session Tree



Note the following:

- 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. Furthermore, because SALES.ACME.COM updates the inventory, it is a client of WAREHOUSE.ACME.COM.

This completes the definition of the session tree for this distributed transaction.

Remember that 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.

2. The application issues a COMMIT statement.

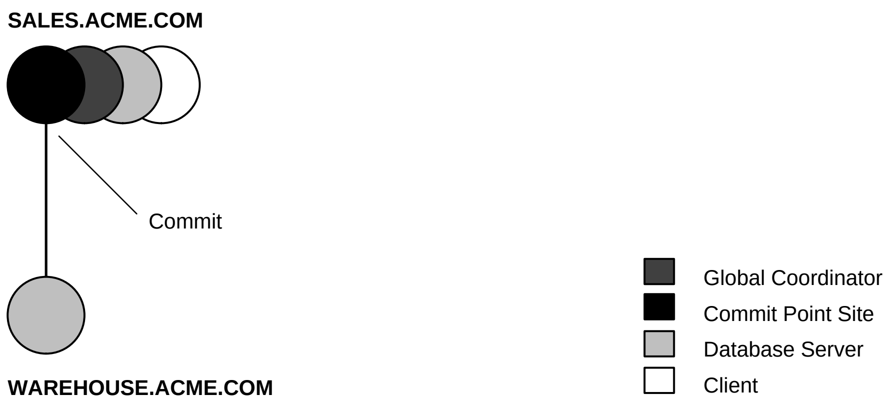
   The final statement in the transaction that enters the sales order is now issued — a COMMIT statement which begins the two-phase commit starting with the prepare phase.

3. The global coordinator determines the commit point site.

   The commit point site is determined immediately following the COMMIT statement. SALES.ACME.COM, the global coordinator, is determined to be the commit point site, as shown in Figure 3–4.

   See "Specifying the Commit Point Strength of an Instance" on page 3-10 for more information about how the commit point site is determined.

Figure 3–4 Determining the Commit Point Site



4. The global coordinator sends the Prepare message.

   After the commit point site is determined, 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.

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, the node can successfully prepare.

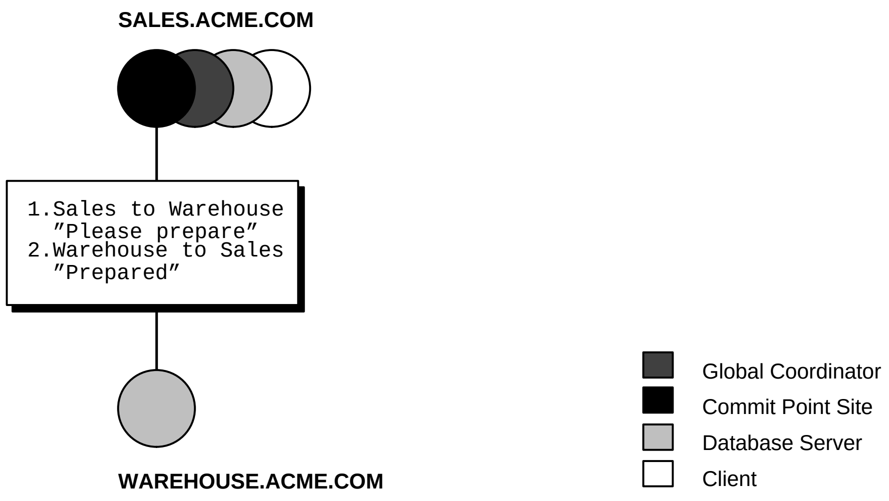
In this example, only WAREHOUSE.ACME.COM receives a prepare message because SALES.ACME.COM is the commit point site (which does not prepare). WAREHOUSE.ACME.COM responds to SALES.ACME.COM with a prepared message.

As each node prepares, it sends a message back to the node that asked it to prepare. Depending on the responses, two things can happen:

- If *any* of the nodes asked to prepare respond with an abort message to the global coordinator, the global coordinator then tells all nodes to roll back the transaction, and the process 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, the global coordinator asks the commit point site to commit the transaction.

Continuing with the example, Figure 3–5 illustrates the parts of Step 4.

Figure 3–5 Sending and Acknowledging the PREPARE Message



5. The commit point site commits.

SALES.ACME.COM, receiving acknowledgment that WAREHOUSE.ACME.COM is prepared, instructs the commit point site (itself, in this example) to commit the transaction. 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 committed yet, the outcome of this transaction is determined, that is, the transaction *will* be committed at all nodes even if the node's ability to commit is delayed.

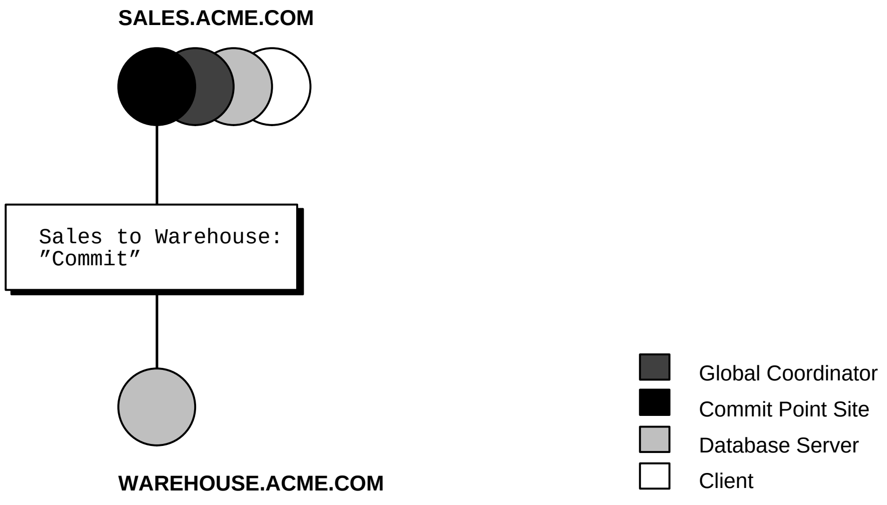
6. The commit point site informs the global coordinator of the commit.

The commit point site now tells the global coordinator that the transaction has committed. In this case, where the commit point site and global coordinator are the same node, no operation is required. The commit point site remembers it has committed the transaction until the global coordinator confirms that the transaction has been committed on all other nodes involved in the distributed transaction.

After the global coordinator has been informed of the commit at the commit point site, it tells all other directly referenced nodes to commit. In turn, any local coordinators instruct their servers to commit, and so on. 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.

Figure 3–6 illustrates Step 6 in this example. SALES.ACME.COM, 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 3–6 The Global Coordinator and Other Servers Commit the Transaction



7. The global coordinator and commit point site complete the commit.

After all referenced nodes and the global coordinator have committed the transaction, the global coordinator informs the commit point site.

The commit point site, which has been waiting for this message, erases the status information about this distributed transaction and informs the global coordinator that it is finished. In other words, the commit point site forgets about committing the distributed transaction. This is acceptable because all nodes involved in the two-phase commit have committed the transaction successfully, and they will never have to determine its status in the future.

After the commit point site informs the global coordinator that it has forgotten about the transaction, the global coordinator finalizes the transaction by forgetting about the transaction itself.

This completes the COMMIT phase and thus completes the distributed transaction.

All of the steps described above are accomplished automatically and in a fraction of a second.

Coordination of System Change Numbers

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

- A connection occurs using the path described by one or more database links.
- A distributed SQL statement executes (the execute phase completes).
- A distributed transaction commits.

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

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.

Read-Only Distributed Transactions

There are three cases in which all or part of a distributed transaction is read-only:

- A distributed transaction can be partially read-only if:
  - only queries are issued at one or more nodes
  - it does not modify any records
  - changes are rolled back due to violations of integrity constraints or triggers being fired

  In each of these cases, the read-only nodes recognize this fact when they are asked to prepare. They respond to their respective local coordinators with a read-only message. By doing this, the commit phase completes faster because Oracle eliminates the read-only nodes from subsequent processing.

- The distributed transaction can be completely read-only (no data changed at any node) and the transaction is *not* started with the SET TRANSACTION READ ONLY statement.

  In this case, all nodes recognize that they are read-only during the prepare phase, and no commit phase is required. However, the global coordinator, not knowing whether all nodes are read-only, must still perform the operations involved in the prepare phase.

- The distributed transaction can be completely read-only (all queries at all nodes) and the transaction *is* started with a SET TRANSACTION READ ONLY statement. In this case, only queries are allowed in the transaction, and the global coordinator does not have to undertake a two-phase commit. Changes by other transactions do not degrade global transaction-level read consistency, because it is automatically guaranteed by coordination of SCNs at each node of the distributed system.

Limiting the Number of Distributed Transactions Per Node

The initialization parameter DISTRIBUTED_TRANSACTIONS controls the number of possible distributed transactions in which a given instance can concurrently participate, both as a client and a server. If this limit is reached and a subsequent user tries to issue a SQL statement referencing a remote database, the statement is rolled back and the following error message is returned:

```
ORA-2042: too many global transactions
```

For example, assume that DISTRIBUTED_TRANSACTIONS is set to 10 for a given instance. In this case, a maximum of ten sessions can concurrently be processing a distributed transaction. If an eleventh session attempts to issue a DML statement requiring distributed access, an error message is returned to the session, and the statement is rolled back.

The database administrator should consider increasing the value of the initialization parameter DISTRIBUTED_TRANSACTIONS when an instance regularly participates in numerous distributed transactions and the above error message is frequently returned as a result of the current limit. Increasing the limit allows more users to concurrently issue distributed transactions.

If the DISTRIBUTED_TRANSACTIONS initialization parameter 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 (from a previous network or system failure) cannot be automatically resolved by Oracle8*i*.

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.

Additional Information: See *Oracle8i Reference* for more information.

Troubleshooting Distributed Transaction Problems

A network or system failure can cause the following types of problems:

- A two-phase commit being processed when a failure occurs may not be completed at all nodes of the session tree.
- If a failure persists (for example, if the network is down for a long time), the data exclusively locked by in-doubt transactions is unavailable to statements of other transactions.

The following sections describe these situations.

Failures that Interrupt Two-Phase Commit

The user program 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.

Note: The failure cases that prompt these error messages are beyond the scope of this book and are unnecessary to administer the system.

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) once the network or system failure is resolved.

However, in extended outages, the administrator may wish to force the commit or rollback of a transaction to release any locked data. Applications must account for such possibilities.

Failures that Prevent Data Access

When a user issues a SQL statement, Oracle8*i* attempts to lock the required resources to successfully execute the statement. However, if the requested data is currently being held by statements of other uncommitted transactions and continues to remained locked for an excessive amount of time, a time-out occurs. Consider the following two scenarios.

Transaction Time-Out

A DML SQL statement that requires locks on a remote database may be blocked from doing so if another transaction (distributed or non-distributed) currently own locks on the requested data. If these locks continue to block the requesting SQL statement, a time-out occurs, the statement is rolled back, and the following error message is returned to the user:

```
ORA-02049: time-out: distributed transaction waiting for lock
```

Because no data has been modified, no actions are necessary as a result of the time-out. 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, the user should contact an administrator to report the problem.

The timeout interval in the above situation can be controlled with the initialization parameter DISTRIBUTED_LOCK_TIMEOUT. This interval is in seconds. For example, to set the time-out interval for an instance to 30 seconds, include the following line in the associated parameter file:

```
DISTRIBUTED_LOCK_TIMEOUT=30
```

With the above time-out interval, the time-out errors discussed in the previous section occur if a transaction cannot proceed after 30 seconds of waiting for unavailable resources.

Additional Information: For more information about initialization parameters and editing parameter files, see the *Oracle8i Reference*.

Lock From In-Doubt Transaction

A query or DML statement that requires locks on a local database may be blocked from doing so indefinitely due to the locked resources of an in-doubt distributed transaction. In this case, the following error message is immediately returned to the user:

```
ORA-01591: lock held by in-doubt distributed transaction <id>
```

In this case, the SQL statement is rolled back 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 very 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.

Manually Overriding In-Doubt Transactions

A database administrator can manually force the COMMIT or ROLLBACK of a local in-doubt distributed transaction. However, a specific in-doubt transaction should be manually overridden *only* when the following situations exist:

- The in-doubt transaction locks data that is required by other transactions. This happens if users complain that the ORA-01591 error message interferes with their transactions.
- An in-doubt transaction prevents the extents of a rollback segment to be 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 that did not allow the two-phase commit phases to complete will not be corrected in an acceptable time period. Examples of such cases might include a telecommunication network that has been damaged or a damaged database that needs a substantial amount of time to complete recovery.

Normally, a decision to locally force an in-doubt distributed transaction should be made in consultation with administrators at other locations. A wrong decision can lead to database inconsistencies which 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. However, if any of the above criteria are met, the administrator should consider a local override of the in-doubt transaction.

If a decision is made to locally force the transaction to complete, the database administrator should analyze available information with the following goals in mind:

- Try 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, you can follow the action taken at that node.
- 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 might indicate the origin of the transaction and what type of transaction it is:

  ```
  COMMIT COMMENT 'Finance/Accts_pay/Trans_type 10B';
  ```

- 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 could protect the record (even when administrators independently force the in-doubt transaction at each node) by including the following sequence of SQL statements:

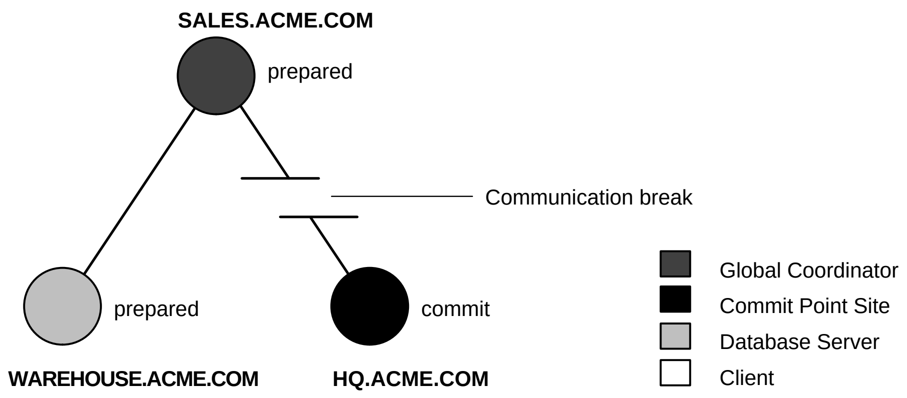
  ```
  ALTER SESSION ADVISE COMMIT;
  INSERT INTO emp@hq ... ;    /*advice to commit at HQ */
  ALTER SESSION ADVISE ROLLBACK;
  DELETE FROM emp@sales ... ; /*advice to roll back at SALES*/

  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 being moved; the record cannot disappear.

Manual Override Example

The following example shows a failure during the commit of a distributed transaction and how to go about gaining information before manually forcing the commit or rollback of the local portion of an in-doubt distributed transaction. Figure 3–7 illustrates the example.

Figure 3–7 An Example of an in-Doubt Distributed Transaction



In this failure case, the prepare phase completed. However, during the commit phase, the commit point site's commit message (the message telling the global coordinator that the transaction was committed at the commit point site) never made it back to 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. However, the data cannot be accessed 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

The users of the local database system that conflict with the locks of the in-doubt transaction get the following error message:

```
ORA-01591: lock held by in-doubt distributed transaction 1.21.17
```

Here, 1.21.17 is the local transaction ID of the in-doubt distributed transaction in this example. The local database administrator should request and record this ID number from the users that report problems to identify in-doubt transactions that should be forced.

Step 2: Query DBA_2PC_PENDING

Query the local DBA_2PC_PENDING (see also page 3 - 29) to gain information about the in-doubt transaction:

```
SELECT * FROM sys.dba_2pc_pending WHERE local_tran_id = '1.21.17';
```

For example, when the previous query is issued at WAREHOUSE, the following information is returned.

Figure 3–8 Results of Querying DBA_2PC_PENDING

```
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#
```

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
```

Here, *global_database_name* is the database name of the global coordinator (where the transaction originates), *hhhhhhhh* is an internal database ID at the global coordinator (8 hexadecimal digits), and *local_tran_id* is the corresponding local transaction ID assigned on the global coordinator. Therefore, 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.

The transaction on this node is in a prepared state. Therefore, WAREHOUSE awaits its coordinator to send either a commit or a rollback message.

The transaction’s Comment or advice may include information about this transaction. If so, use this Comment to your advantage. In this example, the origin (the sales order entry application) and transaction type is in the transaction’s Comment. This information may reveal something that would help 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

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 might 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.

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 or outbound:

- If the connection is inbound, your node is subordinate (a server of) another node. In this case, the DATABASE column lists the name of the client database that connected to your node, and the DBUSER_OWNER column lists the local account for the database link connection that corresponds to the in-doubt transaction.
- If the connection is outbound, your node is a client of other servers. In this case, the DATABASE column lists the name of the database link that connects to the

remote node. The DBUSER_OWNER column lists the owner of the database link for the in-doubt transaction.

Additionally, the INTERFACE column tells whether the local node or a subordinate node is the commit point site.

To trace the session tree, you can query the local DBA_2PC_NEIGHBORS view. In this case, you query this view on the WAREHOUSE database.

```
SELECT * FROM sys.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
```

The columns of particular interest in this view are the IN_OUT, DATABASE, DBUSER_OWNER, and INTERFACE columns. In this example, the IN_OUT column reveals that the WAREHOUSE database is a server for the SALES database, as specified in the DATABASE column.

The connection to WAREHOUSE was established through a database link from the SWILLIAMS account, as shown by the DB_OWNER column, and WAREHOUSE, nor any of its descendants, was the commit point site, as shown by the INTERFACE column. At this point, you can contact the administrator at the located nodes and ask them 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, respectively.

Manually Checking the Status of Pending Transactions at SALES.ACME.COM

```
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#

SELECT * FROM dba_2pc_neighbors
   WHERE global_tran_id = ’SALES.ACME.COM.55d1c563.1.93.29’
   ORDER BY sess#, in_out;
```

At SALES, there are three rows for this transaction (one for the connection to WAREHOUSE, one for the connection to HQ, and one for the connection established by the user). Information corresponding to the rows for the SALES and HQ connections is listed 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
```

```
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. Also, notice that two outbound connections are established from this node, but no inbound links (this node is not a server of another node).
- HQ or one of its servers (none in this example) is the commit point site.

Manually Checking the Status of Pending Transactions at HQ.ACME.COM:

```
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. It has been committed. Therefore, you can force the in-doubt transaction to commit at your local database (see the following section for information on manually committing or rolling back in-doubt transactions). It is a good idea to contact any other administrators you know that could also benefit from your investigation.

Step 4: Check for Mixed Outcome

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 to forced commit or forced abort, depending on how you forced the transaction.

The Pending Transaction Table (DBA_2PC_PENDING)

Every Oracle8*i* database has a *pending transaction table* which 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 by referencing the DBA_2PC_PENDING data dictionary view.

Each transaction with an entry in the pending transaction table is classified in one of the following categories (as indicated in DBA_2PC_PENDING.STATE):

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.
prepared	The node has prepared and may or may not have acknowledged this to its local coordinator with a prepared message. However, no commit message 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.

Also of particular interest in the pending transaction table is the mixed outcome flag (as indicated in DBA_2PC_PENDING.MIXED). The database administrator 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. The information in the pending transaction table can also be used by a database administrator, who decides 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.

Manually Committing In-Doubt Transactions

The local database administrator has two ways to manually force an in-doubt transaction to commit. The DBA can use Enterprise Manager Transaction Object List option Force Commit or the SQL command COMMIT with the FORCE option and a text string, indicating either the local or global transaction ID of the in-doubt transaction to commit.

Forcing a Commit or Rollback in Enterprise Manager

To commit an in-doubt transaction, select the transaction from the Transaction Object List and choose Force Commit from the Transaction menu.

To roll back an in-doubt transaction, select the transaction from the Transaction Object List and choose Force Rollback from the Transaction menu.

Attention: You cannot roll back an in-doubt transaction to a savepoint.

Manually Committing or Rolling Back In-Doubt Transactions

The following SQL statement is the command to commit an in-doubt transaction.

```
COMMIT FORCE 'transaction_name';
```

To manually rollback an in-doubt transaction, use the SQL command ROLLBACK with the FORCE option and a text string, indicating either the local or global transaction ID of the in-doubt transaction to rollback. For example, to rollback the in-doubt transaction with the local transaction ID of 2.9.4, use the following statement:

```
ROLLBACK FORCE '2.9.4';
```

Attention: You cannot roll back an in-doubt transaction to a savepoint.

Privileges Required to Manually Commit or Rollback In-Doubt Transactions

To manually force the commit or rollback of an in-doubt transaction issued by yourself, you must have been granted the FORCE TRANSACTION system privilege. To force the commit or rollback of another user's distributed transaction, you must have the FORCE ANY TRANSACTION system privilege. Both privileges can be obtained either explicitly or via a role.

Note: Forcing the commit or rollback of an in-doubt distributed transaction does not affect the status of the operator's current transaction.

Forcing Rollback/Commit on the Local Pending Transaction Table

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.

Specifying the 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.

Thus 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 global transaction ID *global_id*. 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 (a decimal number) when committing the transaction at the local node. For example, if the SCN were 829381993, you would use the command:

```
COMMIT FORCE 'global_id', 829381993;
```

Changing Connection Hold Time

If a distributed transaction fails, 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. You can set the length of time that the connection remains open with the database parameter DISTRIBUTED_RECOVERY_CONNECTION_HOLD_TIME.

A high value minimizes the cost of reconnecting after failures, but causes the local database to consume more resources. In contrast, a lower value minimizes the cost of resources kept locked during a failure, but increases the cost of reconnecting after failures. The default value of the parameter is 200 seconds. See the *Oracle8i Reference* for more information.

Setting a Limit on Distributed Transactions

The database parameter DISTRIBUTED_TRANSACTIONS sets a maximum on the number of distributed transactions in which a database can participate. You should increase the value of this parameter if your database is part of many distributed transactions. The default value is operating system-specific.

In contrast, if your site is experiencing an abnormally high number of network failures, you can temporarily decrease the value of this parameter. Doing so 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.

For more information on this parameter, see the *Oracle8i Reference*.

Testing Distributed Transaction Recovery Features

If you like, you can force the failure of a distributed transaction to observe RECO, automatically resolving the local portion of the transaction. Alternatively, you might be interested in forcing a distributed transaction to fail so that you can 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

Comments can be included 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 #.##.## in-doubt
ORA-02059: ORA-CRASH-TEST-n 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 rather quickly.

Privileges Required to Induce Two-Phase Commit Failures

You can induce two-phase commit failures via the previous comments only if the local and remote sessions have the FORCE ANY TRANSACTION system privilege. Otherwise, an error is returned if you attempt to issue a COMMIT statement with a crash comment.

The Recoverer (RECO) Background Process

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.

Disabling and Enabling RECO

The recoverer background process, RECO, can be enabled and disabled using the ALTER SYSTEM command with the ENABLE/DISABLE DISTRIBUTED RECOVERY options, respectively. For example, you might want to 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, distributed recovery must be manually enabled using the statement above.

Additional Information: See your Oracle operating system-specific documentation for more information about distributed transaction recovery for single-process instances.

4 Distributed Database System Application Development

This chapter describes the special considerations that are necessary if you are designing 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:

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

The *Oracle8i Administrator's Guide* provides a complete discussion of implementing Oracle8*i* applications. This chapter provides information specific to development for an Oracle8*i* distributed database environment. *See also Oracle8i Application Developer's Guide* - *Fundamentals for more information about application development in an Oracle environment.*

Factors Affecting the Distribution of an Application's Data

In a distributed database environment, you should 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. Terminating remote connections established using database links is useful for disconnecting high cost connections (such as long distance phone connections) that are no longer required by the application.

The application developer or user can close (terminate) a remote connection and session using the ALTER SESSION command with the CLOSE DATABASE LINK parameter. For example, assume you issue the following query:

```
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, you must first close all cursors that use the link and then end your current transaction if it uses the link.

Referential Integrity in a Distributed System

Oracle does not permit declarative referential integrity constraints to be defined across nodes of a distributed system (that is, a declarative referential integrity constraint on one table cannot specify a foreign key that references a primary or unique key of a remote table). However, parent/child table relationships across nodes can be maintained using triggers. For more information about triggers to enforce referential integrity, see *Oracle8i Concepts*.

Note: 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.

Distributed Queries

A distributed query is decomposed by the local Oracle into a corresponding number of remote queries, which are sent 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.

If a portion 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.

You should 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 rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.

Tuning Distributed Queries

The most effective way of optimizing your distributed queries is to access the remote database(s) as little as possible and to retrieve only the required data. Specifically, if you reference 5 remote tables from two different remote databases in a distributed query and have a complex filter (e.g. `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 (causing 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*.

With the above information in mind, 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 (reduces the amount of times that the remote database is accessed, improving the performance of a distributed query).

Though you can write a distributed query in any fashion that you like, it is highly recommended that you form your distributed query using collocated inline views to increase the performance of your distributed query if possible.

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.

Cost Based Optimization

In addition to rewriting your queries with collocated inline views, the cost based optimization method will optimize your distributed queries according to the gathered statistics of the referenced tables and the computations performed by the optimizer. For example, cost based optimization will analyze the following query (notice 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 rewrite 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.

Setup Cost Based Optimization

After you have set up your system to use cost based optimization to improve the performance of your distributed queries (as well as other types of queries - see the *Oracle8i Tuning* manual for more information), the operation will be transparent to the user, that is the optimization will occur 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:

- Set Up Database Environment
- Analyze Tables (to generate table statistics)

Set Up Environment To enable cost based optimization, the `OPTIMIZER_MODE` parameter must be set to `CHOOSE` or `COST`. This parameter can be persistently set by modifying the `OPTIMZER_MODE` parameter in the parameter file (INIT.ORA) or set on a session-level by issuing an `ALTER SESSION` command.

See the *Oracle8i Tuning* manual for information on setting the `OPTIMZER_MODE` parameter in the parameter file (`INIT.ORA)` file.

Issue the following ALTER SESSION statement to set the OPTIMIZER_MODE at the session level (this setting will be valid for the current session only):

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

or

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

See the *Oracle8i Tuning* manual for more information on configuring your system to use a cost based optimization method.

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

The easiest way to generate statistics for a table is to execute the ANALYZE command. For example, if you reference the EMP and DEPT tables in your distributed query, you would execute the following to generate the necessary statistics:

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

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 the *Oracle8i SQL Reference* book for additional information on using the ANALYZE statement.

To generate statistics for more than one object at a time, see the "Generating Statistics" section in the *Oracle8i Tuning* manual. Additionally, see the "Automated Statistics Gathering" section in the *Oracle8i Tuning* manual to learn how to automate the process of keeping your statistics current, thus improving the performance and accuracy of cost based optimization.

How Does Cost Based Optimization Work?

As illustrated in the introduction to "Tuning Distributed Queries", 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, there are several distributed query scenarios that cost based optimization is not able improve the performance upon. Specifically, if your distributed query contains any of the following, cost based optimization will not be effective:

- Aggregates
- Subqueries
- Complex SQL

If your distributed query contains one of the above, make sure you read the "Extend Cost Based Optimization with Hints" section to learn how you can modify your query and use hints to improve the performance of your distributed query.

Extend Cost Based Optimization with Hints

If you have a distributed query that the optimizer cannot handle (see "Cost Based Optimization Restrictions"), you can use hints to extend the capability of cost based optimization. Specifically, if you write your own query that utilizes collocated inline views, you will want to instruct the CBO to not rewrite your distributed query.

Additionally, if you have special knowledge about the database environment (i.e. statistics, load, network and CPU limitations, distributed queries, etc.), you can specify a hint to guide cost based optimization.

For the purposes of optimizing distributed queries, you will provide hints based on your knowledge of the distributed query. Specifically, 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. Since this type of distributed query cannot be rewritten by the optimizer, specifying NO_MERGE will cause the optimizer to skip the steps described in the "How Does Cost Based Optimization Work?" section on page 4-7.

The DRIVING_SITE hint allows you to define a site that is remote to you to act as the query execution site. This is especially helpful when the remote site contains the majority of the data and the query will perform better if executed from that remote site and the resultant data set returned to the local site.

NO_MERGE The NO_MERGE hint prevents Oracle from merging an inline view into a potentially non-collocated SQL statement (see step 1 in the "How Does Cost Based Optimization Work?" section on page 4-7). This hint is embedded in your 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;
```

You will most likely use this hint if you have developed an optimized query based on your knowledge of your database environment. For more information, see the NO_MERGE hint in the *Oracle8i Tuning* manual.

DRIVING_SITE The DRIVING_SITE hint allows you to specify the site where the query execution is performed. It is highly recommended that you let the cost based optimization determine where the execution should be performed, but if you want to override the optimizer (either because your statistics are stale or performance on a particular machine has been severely degraded), you can specify the execution site with the DRIVING_SITE hint. A SELECT statement with a DRIVING_SITE hint might look like:

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

For more information, see DRIVING_SITE in the *Oracle8i Tuning* manual. For more information about tuning distributed queries, see "Tuning Distributed Queries" on page 4-4 and *Oracle8i Tuning*.

Verifying Optimization

An important aspect to tuning your distributed queries is to analyze the execution plan for a query. The feedback that you receive from your analysis is an important element to testing and verifying your database. This verification is increasingly important when you want to compare the execution plan for a distributed query that is optimized by cost based optimization versus the execution plan for a distributed query that you manually optimize (using hints, defining collocated inline views, etc.). See the *Oracle8i Tuning* manual for detailed information about execution plans, the `EXPLAIN PLAN` command, and how to interpret the results.

Prepare Database

Before you can view the execution plan for you distributed query, you must prepare you database to store the execution plan. This preparation is easily performed by executing a script; complete the following to prepare your database to store an execution plan:

```
@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 will be created in the current schema to temporarily store the execution plan.

Generate Execution Plan

Once you have prepared your database to store the execution plan, you are ready to view the execution plan for a specified query. Instead of directly executing the SQL statement, you append the statement with the EXPLAIN PLAN FOR clause. For example, you might 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);
```

View Execution Plan

After you have executed the above SQL statement, the execution plan will be 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 file will display the execution plan for the SELECT statement that you specified. Your results will be formatted like the following:

```
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 your distributed queries by writing your own collocated inline views and/or using hints, you are advised 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 to your optimization as necessary to improve the performance of your distributed query.

If you want 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';
```

Your output might look like the following:

```
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 a procedure is executed 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 NO_DATA_FOUND keyword.
- SQL errors, such as ORA-00900 and ORA-02015.
- Application exceptions, which are generated using the RAISE_APPLICATION_ERROR() procedure.

When using local procedures, all of these messages can be trapped by writing an exception handler, such as shown in the following example:

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

Notice that the WHEN clause requires an exception name. If the exception that is raised does not have a name, such as those generated with RAISE_APPLICATION_ERROR, one can be assigned using PRAGMA_EXCEPTION_INIT, as shown in the following 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

Heterogeneous Services

5 Understanding Oracle Heterogeneous Services

This chapter describes the basic concepts of the Oracle Heterogeneous Services. Topics include:

- What is Heterogeneous Services?
- The Services provided by Heterogeneous Services
- Using Heterogeneous Services

For information about features new to the current Oracle8*i* release, please see *Getting to Know Oracle8i*.

What is Heterogeneous Services?

Heterogeneous Services is an integrated component within the Oracle8*i* server, and provides the generic technology for accessing non-Oracle systems from the Oracle server. Heterogeneous Services enables you to:

- Use Oracle SQL to transparently access data stored in non-Oracle systems as if the data resides within an Oracle server.
- Use Oracle procedure calls to transparently access non-Oracle systems, services, or application programming interfaces (APIs), from your Oracle distributed environment.

To access a particular non-Oracle system, you will need a complementary Heterogeneous Services agent.

Note: The phrase "non-Oracle system" denotes both non-Oracle datastores (or databases) that are accessed using SQL, and systems that are accessed procedurally.

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. Oracle Corporation will provide Heterogeneous Services agents in the form of Oracle Open Gateways version 8 and higher.

Oracle Open Gateways is one family of products that will use the Heterogeneous Services. Other products that are based on Heterogeneous Services are being developed. These products, developed by Oracle or third-parties, may not be part of the Oracle Open Gateways family of products. We use the phrase "Heterogeneous Services agents" to denote all products that are based on Heterogeneous Services, including Oracle Open Gateways.

The Services provided by Heterogeneous Services

Heterogeneous Services provides three services:

- Transaction service
- SQL service
- Procedural service

Transaction Service

The *transaction service* allows non-Oracle systems to be integrated into Oracle transactions and sessions. Users transparently set up an authenticated (i.e. username and password) 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 will access the non-Oracle system to transparently coordinate the distributed transaction. In fact, the Oracle server will support distributed transaction with the non-Oracle system, even if the non-Oracle system itself does not support two-phase commit.

Both the SQL service and procedural service use the Transaction service. Oracle's object transaction service will use agents that only implement the transaction service.

See "Views for the Transaction Service" on page 6-9 for more information on heterogeneous distributed transactions.

SQL Service

The *SQL service* is used to transparently access the non-Oracle system using SQL. If an application's SQL request requires data from a non-Oracle system, Heterogeneous Services translates the Oracle SQL request into an equivalent SQL request understood by the non-Oracle system, accesses the non-Oracle data, and makes the data available to the Oracle 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

Procedural Service

Heterogeneous Services enable users to access any procedural non-Oracle system, such as messaging and queuing systems, from an Oracle8*i* server. The non-Oracle system is called from the Oracle server using a PL/SQL remote procedure call (RPC). Heterogeneous Services translates the PL/SQL call into a procedure or function of the non-Oracle system.

With the procedural service you can create distributed external procedures, that enable you to call third generation language (3GL) routines from PL/SQL. Like PL/SQL external procedures, the distributed external procedure maps PL/SQL procedure and function names and arguments onto 3GL routine names and their arguments. Both external procedures and distributed external procedures use the same mechanisms to call 3GL routines from PL/SQL. External procedures are designed to perform special purpose tasks that are local to the Oracle8*i* server, whereas distributed external procedures are designed to access non-Oracle systems. The primary differences between distributed external procedures and external procedures are:

- distributed external procedures enable the Oracle8*i* server to start authenticated sessions at the non-Oracle system and to coordinate a distributed transaction with the non-Oracle system
- distributed external procedures are considered to run on remote systems, and are therefore invoked through database link

PL/SQL *external procedures*, are covered in the *PL/SQL User's Guide and Reference.*

Using Heterogeneous Services

Heterogeneous Services makes a non-Oracle system appear to be a remote Oracle server. To access or manipulate tables or to execute procedures in the non-Oracle system, you simply create a database link. Tables and procedures at the non-Oracle system can be accessed by qualifying the tables and procedures with the database link. This is identical to accessing tables and procedures at a remote Oracle server.

If a non-Oracle system is referenced, Heterogeneous Services will translate the SQL statement or PL/SQL remote procedure call into the appropriate statement at the non-Oracle system.

Consider the following example that accesses a non-Oracle system through a database link:

```
SELECT *
FROM EMP@non_Oracle_system;
```

Heterogeneous Services will translate the Oracle SQL statement into the SQL dialect and execute the SQL statement at the non-Oracle system.

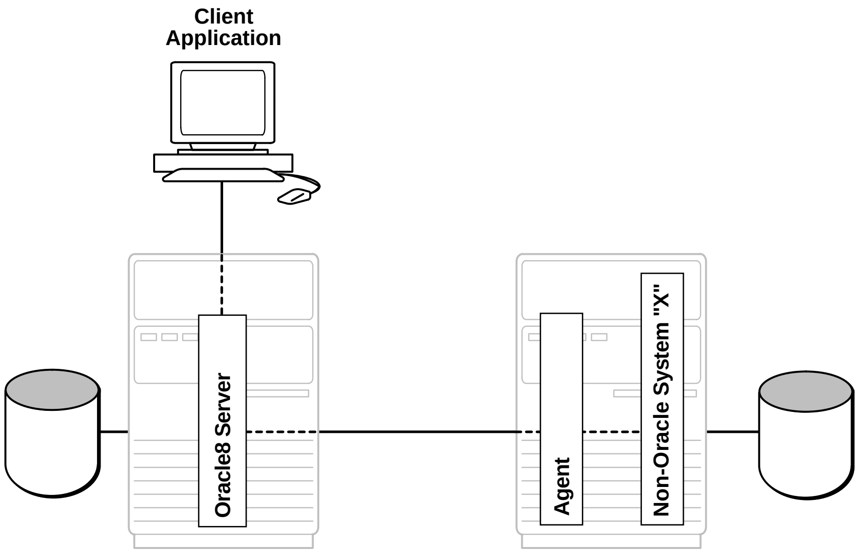
Heterogeneous Services Process Architecture

An agent is required to access a particular non-Oracle system from an Oracle8*i* server. The Oracle server communicates with the agent. The agent communicates with a particular non-Oracle system.

As shown in Figure 5–1, 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 it can even reside on a third machine. The agent must be accessible by the Oracle8*i* server through Net8, and the agent must be 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 5–1 Accessing Heterogeneous Non-Oracle Systems



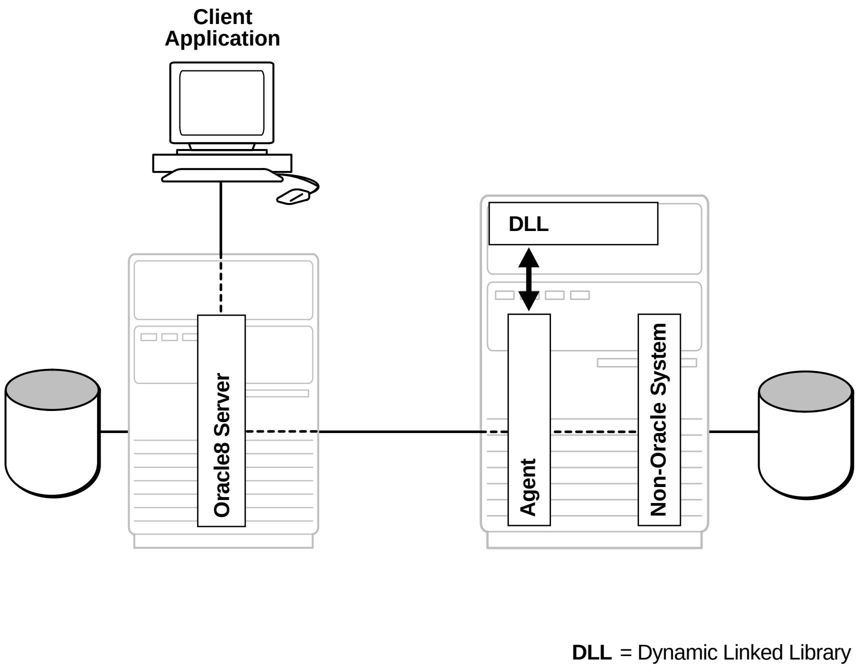
Process Architecture for Distributed External Procedures

Distributed external procedures map PL/SQL procedures onto remote 3GL routines that reside in a dynamic linked library (DLL). Whenever a distributed external procedure is executed, the agent will load the operating system dynamic linked library that contains the 3GL routine into the agent process, map the PL/SQL procedure onto the 3GL routine, and invoke the 3GL routine. After the 3GL routine finishes processing, the arguments and return values are passed back to the calling PL/SQL program. See Figure 5–2.

Note: On some platforms, dynamic linked libraries are referred to as *shared libraries*.

To access a non-Oracle system using a distributed external procedure , you need an agent specifically designed for that non-Oracle system . The agent contains non-Oracle system-specific code which sets up a session at the non-Oracle system, and integrates the transactions performed at non-Oracle system by the distributed external procedure into an Oracle distributed transaction.

Figure 5–2 Oracle8i, Agents and Dynamic Libraries



For example, you have an agent that provides access to a queuing system. To put a message into the queue, an Oracle application issues the following statement:

```
SQL> EXECUTE enqueue@queuing_system('We are out of stock');
SQL> COMMIT;
```

The *enqueue* procedure resides in a dynamic linked library. When you execute the statement above, the Net8 listener spawns the agent process. The agent process loads the DLL containing the enqueue procedure, and executes the enqueue procedure to put a message in the queuing system. When you COMMIT the transaction, the agent will ask the queuing system, on behalf of the Oracle server, to commit the transaction.

The agent process continues running for the duration of the Oracle user session, or until you close the database link explicitly by using the "ALTER SESSION CLOSE DATABASE LINK queing_system" command.

6 Administering Oracle Heterogeneous Services

This chapter describes database administration tasks required to maintain a heterogeneous distributed environment. Topics include:

- Setting up access to Non-Oracle Systems
- Structure of the Heterogeneous Services Data Dictionary
- The Data Dictionary Views
- The DBMS_HS Package (setting initialization parameters)
- Security for Distributed External Procedures
- Agent Self-Registration

Setting up access to Non-Oracle Systems

This section explains the generic steps to configure access to a non-Oracle system. Please see your *Installation and User's Guide* for your particular agent for more installation information. Configuring your particular agent might slightly differ from what is presented in this section.

The steps are:

1. Install the Heterogeneous Services Data Dictionary
2. Set up your environment to access Heterogeneous Services agents
3. Create the database link to the non-Oracle system
4. Test the connection
5. Optionally, register distributed external procedures

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 might already be installed on your Oracle8*i* server. You can confirm this by checking for the existence of Heterogeneous Services data dictionary views, for example SYS.HS_FDS_CLASS.

Set Up 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 will include 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. The listener must be set up to listen for incoming request from the Oracle8*i* server, and spawn Heterogeneous Services agents. The *listener.ora* file must be modified to set up the listener to start Heterogeneous Services agents, and the listener must be (re-)started.

A Sample Descriptor for a Net8 Service Name

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

```
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 nameservers 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 Entry in LISTENER.ORA

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. The SID_NAME is typically used to define the initialization parameter file for the agent.

Create the Database Link to the Non-Oracle System

To create a database link to the non-Oracle system, you just use the CREATE DATABASE LINK command to create private or public database links.

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 an MegaBase release 6 server, you could create database link as follows:

```
CREATE DATABASE LINK salesdb
USING ‘MegaBase6_sales’;
```

See Also: For more information on creating database links, see Chapter 2, "Distributed Database Administration"**.**

Test the Connection

To test the connection to the non-Oracle system, you can 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 from an existing table or view using the database link, for example::

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

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

- **Capabilities of the non-Oracle system:** The agent specifies the capabilities of the non-Oracle system. For example, 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 will upload information into the Oracle8*i* data dictionary automatically the first time they are accessed. However, some agent vendors may provide scripts that you must run at the Oracle8*i* server.

Register Distributed External Procedures (Optional)

This step is only required for agents that support distributed external procedures. Distributed external procedures enable users to procedurally access a non-Oracle system. If the agent vendor created distributed external procedures, they will provide a script or installer to register those distributed external procedures in the Oracle8*i* server.

If you use distributed external procedures to access the non-Oracle system, use a PL/SQL remote procedure call to execute the remote procedure:

```
execute foo@non_oracle_system(1,2,3)
procedure successfully completed.
```

Note: You typically do not need distributed external procedures to execute stored procedures in the non-Oracle system.

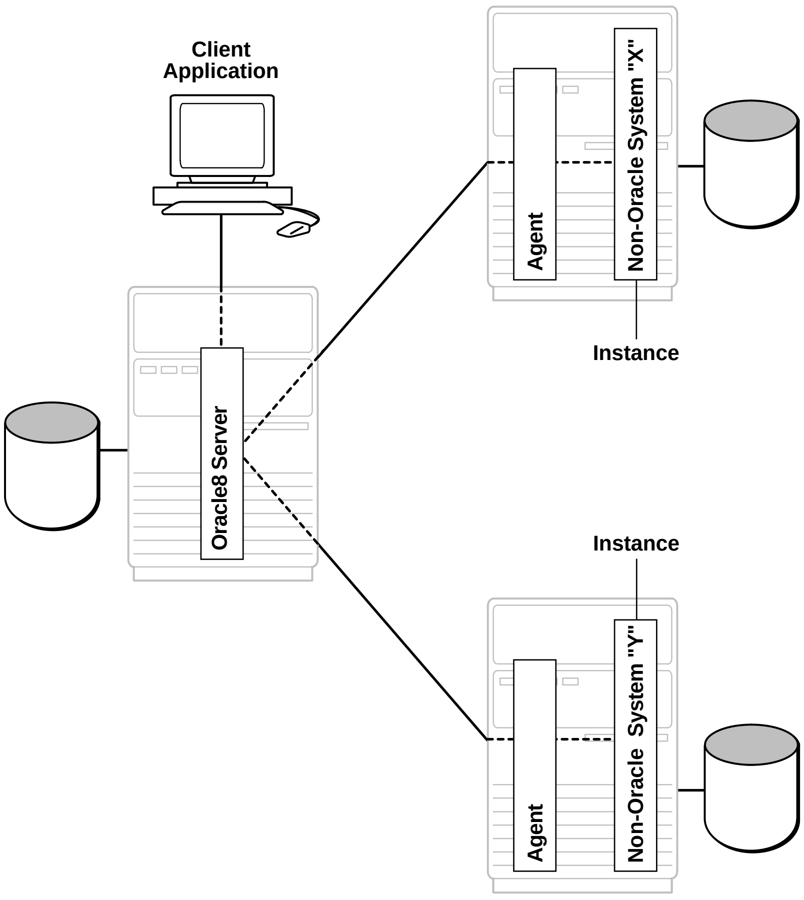
Note: See the *Installation and User's Guide* for your agent for more information on how to register distributed external procedures. The distributed external procedures that can be executed at the non-Oracle system are defined by the agent vendor. See the *Installation and User's Guide* for your agent for a list of procedures that can be executed.

Structure of the Heterogeneous Services Data Dictionary

Each non-Oracle system you access 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. See Figure 6–1.

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 6–1 Instances



If this information were stored separately for each non-Oracle systems you access, the amount of stored data dictionary information could become large and sometimes redundant. For example, when you must 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, this information is organized 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 (type), you may want to set certain information, like 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 will have its own instance information.

Consider an example 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 will contain two class definitions, one for release 5 and one for release 6, and 5 instance definitions.

The Data Dictionary Views

The Heterogeneous Services data dictionary views, contain information about:

- Names of instances and classes uploaded into the Oracle8*i* data dictionary. You can view the uploaded class and instance information through the HS_FDS_CLASS view and HS_FDS_INST view respectively.
- Capabilities, including SQL translations, defined per class or instance. Capability information is viewable through the HS_..._CAPS views.
- Data Dictionary translations defined per class or instance. Data dictionary translation information is viewable through the HS_..._DD views.
- Initialization parameters are defined per class or instance. Initialization parameter information is viewable through the HS_...INIT views.
- Distributed external procedures that can be accessed from the Oracle8*i* server

Table 6–1 Data Dictionary Views for Heterogeneous Services.

View Name	**Description**
HS_FDS_CLASS	View identifies classes accessible from this Oracle8*i* server
HS_FDS_INST	View identifies instances accessible from this Oracle8*i* server
HS_CLASS_INIT	View identifies initialization parameters for each class
HS_INST_INIT	View identifies initialization parameters for each instance
HS_BASE_DD	View identifies all data dictionary translation tablenames supported by Heterogeneous Services
HS_CLASS_DD	View identifies data dictionary translations for each class
HS_INST_DD	View identifies data dictionary translations for each instance
HS_BASE_CAPS	View identifies all capabilities supported by Heterogeneous Services
HS_CLASS_CAPS	View identifies capabilities for each class
HS_INST_CAPS	View identifies capabilities for each instance
HS_EXTERNAL_OBJECTS	View provides information about distributed external procedures and their associated libraries

The views can be divided into four groups:

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

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

See Also: "Structure of the Heterogeneous Services Data Dictionary" on page 6-5 for more information about classes and instances.

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, you must use the procedures available in the package "DBMS_HS". See "The DBMS_HS Package" on page 6-13 for more information.

See Also : The *Oracle8i Reference* for more detailed information about these views

General Data Dictionary Views for Heterogeneous Services

The views that are common for all services are the views that contain:

- Names of the instances and classes that are uploaded into the Oracle8*i* data dictionary. The uploaded class and instance names can be viewed through the HS_FDS_CLASS view and HS_FDS_INST view respectively.
- Information about the Heterogeneous Services initialization parameters. This information can be viewed through the HS_CLASS_INIT, and HS_INST_INIT views.

For example, you can access both MegaBase release 5 and release 6 from an Oracle8*i* server. After accessing the agent(s) for the first time, uploaded information in the Oracle8*i* server could look like:

```
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.

Views for the Transaction Service

When a non-Oracle system is involved in a distributed transaction, the transaction capabilities of the non-Oracle system (and 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 since 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 Commit Point Site. That is, it can *not* prepare data, but it can remember the outcome of a particular transaction if asked to 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 to 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 "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 supports 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 will be recovered automatically. If the failure persists, the in-doubt transaction might need to be manually overridden by the database administrator. See Chapter 3, "Distributed Transactions" for more information about distributed transactions.

Transactions with Distributed External Procedures

For distributed external procedures it is unknown whether it will make changes to data at the non-Oracle system. To ensure the consistency of the heterogeneous distributed database, Oracle will assume that the distributed external procedure updates the non-Oracle system.

Accordingly, the distributed external procedure will participate in the remote or distributed transaction, depending on whether it is the only node that was accessed or whether other nodes were accessed as well. Therefore, to use a distributed external procedure, the agent must at least support the "Single-Site" transaction model.

Views for the SQL Service

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.

Views for Capabilities and Translations

The HS_..._CAPS data dictionary tables contain information about the SQL capabilities of the non-Oracle data source and necessary SQL translations.

HS_..CAPS specifies whether the non-Oracle data store or the Oracle server implements certain SQL language features. If a capability is turned off, Oracle8*i* does not send any SQL statements to the non-Oracle data source that require that particular capability but it can still do post-processing.

Views for Data Dictionary Translations

In order to make the non-Oracle system appear as an Oracle8*i* server, the non-Oracle system data dictionary can be queried just as if it were an Oracle data dictionary. Data Dictionary translations that are defined make this possible. These translations are stored in the HS_..._DD views.

For example, the following SELECT statement will be transformed 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, simply because the non-Oracle data source does not have the required information stored its data dictionary, Heterogeneous Services causes it to appear as if the data dictionary table is available, but the table contains no information.

To retrieve information for which Oracle8*i* data dictionary views and/or tables are translated or mimicked for the non-Oracle system, you 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.

Views for Distributed External Procedures

Distributed external procedures and remote libraries are administered in the Oracle8*i* server. The agent vendor will provide scripts to register distributed external procedures and their libraries. Information about these registered procedures and libraries are stored in the HS_EXTERNAL_OBJECTS data dictionary view. The information includes:

- The name of the distributed external procedure or remote library
- A PL/SQL prototype that gives information on the 3GL routine, including its name, its arguments, and library name.
- The instance name of the non-Oracle system the distributed external procedure logically resides.

The DBMS_HS Package

The DBMS_HS package contains functions and procedures for application developers and database administrators to set and unset Heterogeneous Services initialization parameters, capabilities, instance names, class names, etc.

See Appendix B, "DBMS_HS Package Reference" for a reference listing off all DBMS_HS package interface information for administering Heterogeneous Services

Setting Initialization Parameters

Initialization parameters can be set either in the Oracle8*i* server or in the Heterogeneous Services agent. To set initialization parameters in the Oracle8*i* server, you must use the DBMS_HS package. Please see the installation and user's guide for your particular agent for more information. If the same initialization parameter is set both in the agent and the Oracle8*i* server, the value of initialization parameter in the Oracle8*i* server will take precedence.

There are two types of initialization parameters:

- generic initialization parameters
- non-Oracle data store class-specific initialization parameters

Generic initialization parameters are defined by Heterogeneous Services. See Appendix A, "Heterogeneous Services Initialization Parameters" for more information on generic initialization parameters.

Non-Oracle data store class-specific initialization parameters are defined by the agent vendor. Some non-Oracle data store class-specific initialization parameters may be mandatory. For example, an initialization parameter may include connection information required to connect to a non-Oracle system. Non-Oracle data store class-specific parameters are documented in the *installation and user's guide* for your agent.

Both generic and non-Oracle data store class-specific HS initialization parameters can be set in the Oracle server using the CREATE_INST_INIT procedure in the DBMS_HS package.

For example, you 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: See Appendix A, "Heterogeneous Services Initialization Parameters" for more information on initialization parameters.

Unsetting Initialization Parameters

To unset a Heterogenous Services initialization parameter in the Oracle8*i* server, you must use the DROP_INST_INIT procedure. For example, to delete the HS_DB_DOMAIN entry:

```
DBMS_HS.DROP_INST_INIT
          (FDS_INST_NAME   => 'SalesDB',
           FDS_CLASS_NAME  => 'MegaBase6',
           INIT_VALUE_NAME => 'HS_DB_DOMAIN');
```

Note: See Appendix B, "DBMS_HS Package Reference" for a full description of the DBMS_HS package.

Security for Distributed External Procedures

Please see the agent-specific documentation on how to control execute privileges on distributed external procedures.

Agent Self-Registration

Agent self-registration automates the process of updating Hetergeneous Services configuration data describing agents on remote hosts, to ensure correct operation over heterogeneous database links. Note that agent self-registration is default behavior. If you do not want to use the agent self-registration feature, you must set the value of the Oracle initialization parameter HS_AUTOREGISTER to false. See "Oracle Server Initialization Parameter HS_AUTOREGISTER" on page 6-17 for more information.

Both the server and the agent rely on three types of information to configure and control operation of the HS connection:

- **HS initialization parameters:** these parameters provide control over various connection-specific details of operation.
- **Capability definitions:** these definitions identify details like SQL language features supported by the non-Oracle datasource.
- **Data dictionary translations:** these translations map references to Oracle data dictionary tables and views into equivalents specific to the non-Oracle data source.

This document refers to these three sets of information collectively as *HS configuration data*.

Advantages of Agent Self-Registration

HS configuration data (that you specify using the DBMS_HS_ADMIN package discussed in the previous section) is stored in the Oracle server's data dictionary. Because the agent may likely be remote, and may therefore be administered separately, several circumstances could lead to configuration mismatches between servers and agents:

- An agent could be newly installed on a separate machine and the server would have no HS data dictionary content to represent the agent's HS configuration data.
- A server could be newly installed and lack the necessary HS configuration data for existing agents and non-Oracle data stores.
- A non-Oracle instance could be upgraded from an older version to a newer version, requiring modification of the HS configuration data.
- An HS agent at a remote site could be upgraded to a new version or patched, requiring modification of the HS configuration data.
- A DBA at the non-Oracle site could 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 server and any HS agent, provided that each is at least as recent as Version 8.0.3. The basic mechanism for this is ability to upload HS configuration data (HS Data Dictionary content) from agents to servers.

Self-registration provides automatic updating of HS configuration data residing in the Oracle server data dictionary (if enabled by the server initialization parameter HS_AUTOREGISTER (see below)). Such a data dictionary update assures 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.

How Does Agent Self-Registration Work?

The HS agent self-registration feature can:

- Identify the agent and the non-Oracle data store to the Oracle 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 that, 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

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 which connects to the it. Note that, when any component of an agent changes (agent executable or uploadable definitions) 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 does this depends on the specific agent in use. The Oracle server then uses FDS_INST_NAME to look up instance-specific configuration information in its data dictionary, using it as a primary key for columns of the same name in the FDS_INST_INIT, FDS_INST_CAPS, and FDS_INST_DD views.

Server data dictionary accesses that use FDS_INST_NAME also use FDS_CLASS_NAME to uniquely identify configuration information rows. For example, if you are porting a database from class "MegaBase8.0.4" to class "MegaBase8.1.3", both databases can simultaneously operate with instance name "Scott" and can 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.

Oracle Server Initialization Parameter HS_AUTOREGISTER

The Oracle server initialization parameter HS_AUTOREGISTER enables or disables automatic self-registration of HS agents. When set to TRUE, information describing a previously unknown agent class or a new agent version is uploaded into the server's data dictionary.

See the *Oracle8i Reference* for a description and the syntax of this parameter.

It is recommended that you use the dfault value for this parameter (TRUE) which assures that the server's data dictionary content always correctly represents definitions of class capabilities and data dictionary translations as used in HS connections.

7 Application Development with Heterogeneous Services

This chapter provides information for application developers who want to use Heterogeneous Services.

Topics covered include:

Application Development with Heterogeneous Services

When writing applications, you need not be concerned that a non-Oracle system is accessed. Heterogeneous Services makes the non-Oracle system appear as if it were another Oracle8*i* server.

However, on occasion, you may need to access a non-Oracle system using that non-Oracle system's SQL dialect. To make this possible, Heterogeneous Services provides a pass-through SQL feature that allows application programmers to directly execute the 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 server. This chapter also discusses how to tune such data transfers.

Pass-Through SQL

The pass-through SQL feature allows an application developer to send a statement directly to a non-Oracle system without being interpreted by the Oracle8*i* server. This can be useful if the non-Oracle system allows for operations in statements for which there is no equivalent in Oracle. 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 regular "transparent" SQL statements.

The DBMS_HS_PASSTHROUGH package conceptually resides at the non-Oracle system. Procedures and functions in the package must be invoked by using the appropriate database link to the non-Oracle system.

Considerations When Using Pass-Through SQL

There are transaction implications when you execute a pass-through SQL statement that (implicitly) commit or rolls back the transaction in the non-Oracle system. For example, some systems implicitly commit the transaction when a Data Definition Language (DDL) statement is executed. Since the Oracle server is bypassed, the Oracle server is not aware of the commit in the non-Oracle system. This means that the data at the non-Oracle system can be committed while the transaction in the Oracle server is not.

If the transaction in Oracle server is rolled back, data inconsistencies between the Oracle server and the non-Oracle server can occur (i.e. global data inconsistency).

Note that if the application executes a regular COMMIT, the Oracle server can coordinate the distributed transaction with the non-Oracle system. The statement executed with the pass-through facility are 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. The following sections describe how to use them. The statements fall into two classes:

- non-queries (INSERT, DELETE, UPDATE, and DDL statements)
- queries (SELECT statements)

Procedure/Function	Description
OPEN_CURSOR	Open a cursor
CLOSE_CURSOR	Close a cursor
PARSE	Parse the statement
BIND_VARIABLE	Bind IN variables
BIND_OUT_VARIABLE	Bind OUT variables
BIND_INOUT_VARIABLE	Bind IN OUT variables
EXECUTE_NON_QUERY	Execute non-query
EXECUTE_IMMEDIATE	Execute non-query without bind variables
FETCH_ROW	Fetch rows from query
GET_VALUE	Retrieve column value from SELECT statement, or to retrieve OUT bind parameters

Executing Non-queries

To execute non-query statements, you 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", you 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 will be returned.

You cannot execute a query with EXECUTE_IMMEDIATE and you cannot use bind variables.

Bind Variables

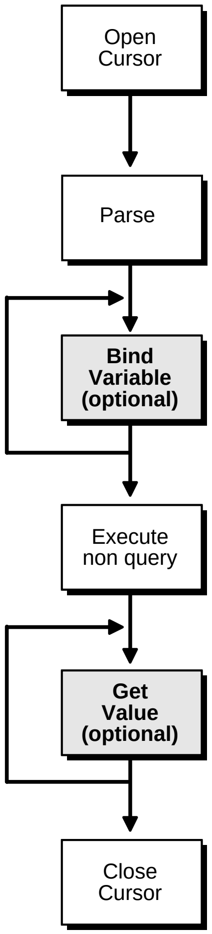
Bind variables allow you to use the same SQL statement multiple times with different values, reducing the number of times a SQL statement needs to be parsed. For example, when you need to insert four rows in a particular table, you can parse the SQL statement once and bind and execute the SQL statement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

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

- Open a cursor
- Parse the SQL statement at the non-Oracle system
- Bind the variables
- Execute the SQL statement at the non-Oracle system
- Close the cursor

Figure 7–1 shows the flow diagram for executing non-queries with bind variables.

Figure 7–1 Flow Diagram for Non-query Pass-Through SQL



IN Bind Variables How a bind variable is specified in a statement is determined by syntax of the non-Oracle system. 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 might 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;
```

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 SQL statement is executed.

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.

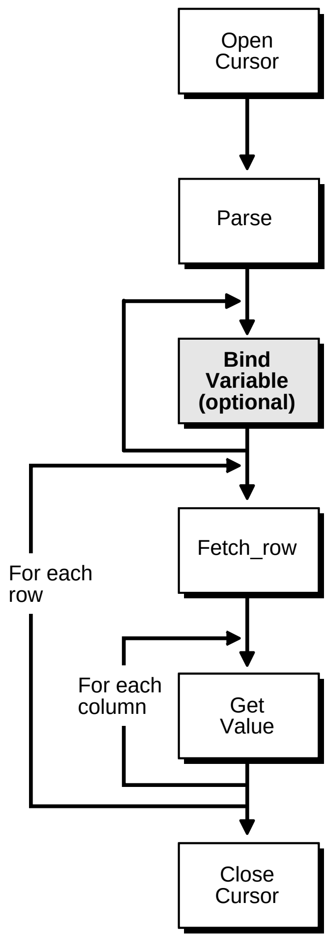
IN OUT Bind Variables A bind variable can be both an IN and an OUT variable. This means that the value of the bind variable must be known before the SQL statement is executed but can be changed after the SQL statement is executed.

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. The result set is retrieved by iterating over a cursor. After the SELECT statement is parsed, 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. See Figure 7–2.

Figure 7–2 Pass-through SQL for Queries



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

Note: Although you are fetching one row at a time, Heterogeneous Services optimizes the round trips between the 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 "0".

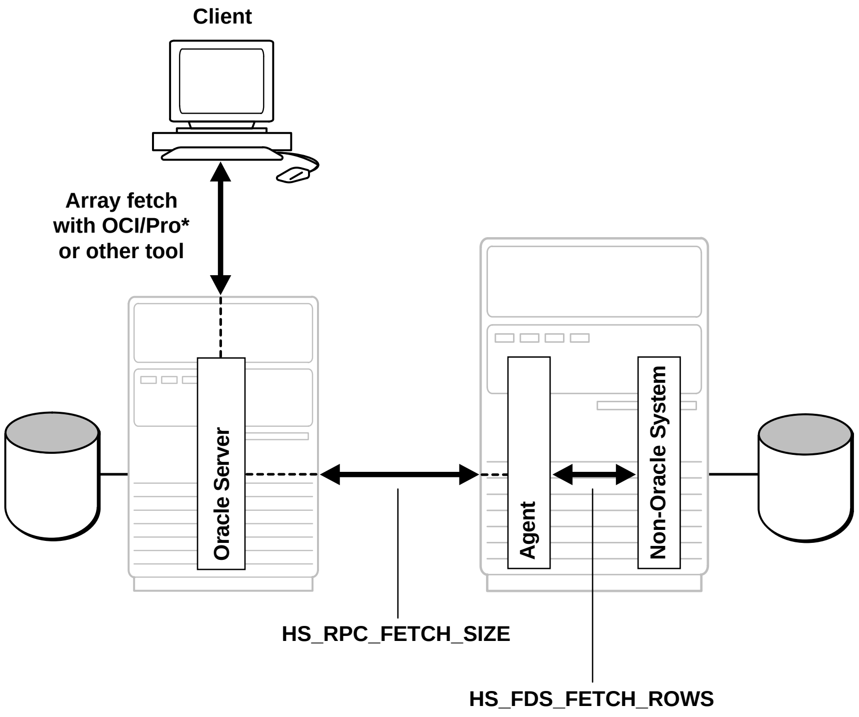
Bulk Fetch

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 server
- from the Oracle server to the application

Oracle allows you to optimize all three data transfers. See Figure 7–3.

Figure 7–3 Optimizing data transfers



Array Fetch Using the OCI, an Oracle Precompiler, or Another Tool

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.

Array Fetch Between an Oracle8*i* Server and the Agent

When data is retrieved from a non-Oracle system, the Heterogeneous Services initialization parameter HS_RPC_FETCH_SIZE defines the number of bytes that will be sent per fetch between the agent and the Oracle8*i* server. The agent will fetch data from the non-Oracle system until it has accumulated the specified number of bytes to sent back to the Oracle server or when the last row of the result set is fetched from the non-Oracle system.

Array Fetch Between the Agent and the Non-Oracle Datastore

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 your agent supports array fetching.

Reblocking

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, when the number of bytes fetched from the non-Oracle system is equal or higher than the value of HS_RPC_FETCH_SIZE. In other words, the agent will "reblock" the data between the agent and the Oracle server in sizes defined by the value of HS_RPC_FETCH_SIZE.

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

For example, you set HS_RPC_FETCH_SIZE to 64Kbytes and HS_FDS_FETCH_ROWS to 100 rows. Assume each row is approximately 600 bytes in size, and thus 100 rows is approximately 60Kbytes. When HS_RPC_FETCH_REBLOCKING is set to ’on’, the agent start fetching 100 rows from the non-Oracle system.

Since there is only 60K bytes of data in the agent, the agent will not sent the data back to the Oracle server. Instead, the agent fetches the next 100 rows from the non-Oracle system. Since there is now 120Kbytes of data in the agent, the first 64Kbytes can be sent back to the Oracle server.

Now there is 56Kbytes left in the agent. The agent will fetch another 100 rows from the non-Oracle system, before sending the next 64Kbytes of data to the Oracle server. By setting the initialization parameter HS_RPC_FETCH_REBLOCKING to ’off’, the first 100 rows will be immediately sent back to the Oracle8*i* server.

A

Heterogeneous Services Initialization Parameters

This appendix describes Heterogeneous Services initialization parameters.

Initialization parameters can either be set at the agent-site using an agent-specific mechanism or they can be set in the Oracle server using the DBMS_HS package. See Chapter 6, "Administering Oracle Heterogeneous Services" for more information on how to set and delete initialization parameters using the DBMS_HS package.

- HS_COMMIT_POINT_STRENGTH
- HS_DB_DOMAIN
- HS_DB_INTERNAL_NAME
- HS_DB_NAME
- HS_DESCRIBE_CACHE_HWM
- 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_FDS_FETCH_ROWS
- HS_RPC_FETCH_SIZE

HS_COMMIT_POINT_STRENGTH

Service:	**General**
Default value:	0
Range of values:	0 to 255

Purpose

The parameter 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 server or non-Oracle system with the highest commit point strength becomes the commit point site. To ensure that non-Oracle system *never* becomes the commit point site, set the value of HS_COMMIT_POINT_STRENGTH to zero.

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

See Chapter 6, "Administering Oracle Heterogeneous Services" for more information about heterogeneous distributed transactions. See Chapter 3, "Distributed Transactions", for more information about distributed transactions and commit point sites.

HS_DB_DOMAIN

Service:	**General**
Default value:	WORLD
Range of values:	1 to 119 characters

Purpose

HS_DB_DOMAIN specifies a unique network sub-address for a non-Oracle system. HS_DB_DOMAIN is used in a similar fashion to the Oracle server equivalent, which is described in the *Oracle8i Administrator's Guide* and the *Oracle8i Reference*. HS_DB_DOMAIN is *required* if you use the Oracle Name Server. The parameters HS_DB_NAME and HS_DB_DOMAIN define the global name of the non-Oracle system.

Note: HS_DB_NAME and HS_DB_DOMAIN in combination must also be unique.

HS_DB_INTERNAL_NAME

Service:	**General**
Default value:	01010101
Range of values:	1 to 16 hexadecimal characters

Purpose

HS_DB_INTERNAL_NAME 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

Service:	**General**
Default value:	HO
Range of values:	1 to 8 characters

Purpose

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. The HS_DB_NAME and HS_DB_DOMAIN define the global name of the non-Oracle system.

HS_DESCRIBE_CACHE_HWM

Service:	**General**
Default value:	100
Range of values:	1 to 4000

Purpose

HS_DESCRIBE_CACHE_HWM 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 rather than re-accessing the non-Oracle datastore. Increasing the high water mark improves performance, especially when you are accessing many mapped tables. However, note that increasing the high water mark improves performance at the cost of memory usage.

HS_LANGUAGE

Service:	**General**
Default value:	System Specific
Range of values:	None

Purpose

The HS_LANGUAGE initialization parameter provides Heterogeneous Services with character set, language and territory information of the non-Oracle data source. The value of the HS_LANGUAGE initialization parameter has to be of 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* server and the non-Oracle data source are the same. If they are not the same, Heterogeneous Services tries to translate the character set of the non-Oracle data source to the Oracle8*i* character set, and vice versa. This can degrade performance, and in some cases Heterogeneous Services will not be able to translate a character from one character set to another.

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 server.

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

Territory

The territory clause of the HS_LANGUAGE initialization parameter specifies the conventions for day and week numbering, default date format, decimal character and group separator, and ISO and local currency symbols.

- 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

Service:	**General**
Default value:	Value determined by HS_LANGUAGE parameter
Range of values:	None

Purpose

HS_NLS_DATE_FORMAT defines the date format for dates used by the target system. This parameter has the same function as the HS_NLS_DATE_FORMAT parameter for an Oracle server. The value of `date_format` 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'
```

HS_NLS_DATE_LANGUAGE

Service:	**General**
Default value:	Value determined by HS_LANGUAGE parameter
Range of values:	None

Purpose

The HS_NLS_DATE_LANGUAGE parameter 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/YY', all three components of the character date are numbers. However, in the format 'DD-MON-YY', 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, in French it is AVR (Avril).

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

HS_NLS_NCHAR

Service:	**General**
Default value:	Value determined by HS_LANGUAGE parameter
Range of values:	None

The HS_NLS_NCHAR parameter is used to tell the Heterogeneous Services the value of the national character set of the non-Oracle data source. The value should be the character set ID of a character set that is supported by Oracle's NLSRTL library.

See also the HS_LANGUAGE parameter.

HS_OPEN_CURSORS

Service:	**General**
Default value:	50
Range of values:	None

Purpose

The HS_OPEN_CURSORS 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 server. Therefore, setting the same value as the HS_OPEN_CURSORS initialization parameter in the Oracle server is recommended.

HS_ROWID_CACHE_SIZE

Service:	**General**
Default value:	3
Range of values:	1 to 32767

Purpose

HS_ROWID_CACHE_SIZE 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

Service:	**General**
Default value:	ON
Range of values:	OFF, ON

Purpose

This controls whether or not Heterogeneous Services attempts to optimize performance of data transfer between the ORACLE server and the Heterogeneous Services agent connected to the non-Oracle datastore. See Chapter 7, "Application Development with Heterogeneous Services" for more information.

The value "OFF" disables reblocking of fetched data. This means that data is immediately sent from the agent to the server. The value "ON" enables reblocking, which means that data fetched from the non-Oracle system is buffered in the agent, and will not be sent to the Oracle server, until the amount of fetched data is equal or higher than HS_RPC_FETCH_SIZE.

HS_FDS_FETCH_ROWS

Service:	**General**
Default value:	20
Range of values:	Decimal integer (row count)

Purpose

This parameter specifies the number of rows to fetch in one round trip from the non-Oracle datastore by the agent.

Each Heterogeneous Services agent will likely have its own maximum limit for the range of this variable. If your non-Oracle datastore does not support array fetch, the value for this parameter must be 1. See your agent-specific documentation for more information.

See Chapter 7, "Application Development with Heterogeneous Services" for more information.

HS_RPC_FETCH_SIZE

Service:	**General**
Default value:	4000
Range of values:	Decimal integer (byte count)

Purpose

This initialization parameter tunes internal data buffering to optimize the data transfer rate between the server and the agent process. Increasing the value can lead to more optimal data transfers per round trip. However, it can increase the response time of certain queries, since the data is not sent back to the server until the data fetched from the non-Oracle system equals HS_RPC_FETCH_SIZE.

See Chapter 7, "Application Development with Heterogeneous Services" for more information.

B
DBMS_HS Package Reference

This appendix provides all the interface information for the DBMS_HS package for administering Heterogeneous Services. See Chapter 6, "Administering Oracle Heterogeneous Services" for more information about administering the Heterogeneous Services.

Referenced in this appendix are:

- DBMS_HS.CREATE_FDS_INST
- DBMS_HS.CREATE_INST_INIT
- DBMS_HS.DROP_FDS_INST
- DBMS_HS.DROP_INST_INIT

DBMS_HS.CREATE_FDS_INST

Purpose

To register an instance of the non-Oracle system in the Oracle8*i* server. This is the logical name of the non-Oracle system instance, as know by Heterogeneous Services. Information about registered instances is available through the view HS_FDS_INST.

Interface description

```
PROCEDURE create_fds_inst(
FDS_INST_NAME     IN VARCHAR2,
FDS_CLASS_NAME    IN VARCHAR2,
FDS_INST_COMMENTS IN VARCHAR2 )
```

Parameters and Descriptions

Parameter	Description
FDS_INST_NAME	Non-Oracle system instance to be registered in the Oracle8*i* server
FDS_CLASS_NAME	The class associated with the non-Oracle system instance.
FDS_INST_COMMENTS	Provides up to 255 bytes to describe the instance of the non-Oracle system.

Exceptions

Exception	Description
ORA-24270	The instance already exists
ORA-24274	The object could not be created. Did you pass an existing CLASS or INSTANCE name?

See Also

DBMS_HS.DROP_FDS_INST

DBMS_HS.CREATE_INST_INIT

Purpose

To create an initialization variable for an instance of the non-Oracle system. See Chapter 6, "Administering Oracle Heterogeneous Services" for more information on how to create initialization variables. See Appendix A, "Heterogeneous Services Initialization Parameters" for possible initialization parameters that can be set. The information will be available through the view HS_INST_INIT.

Interface description

```
PROCEDURE create_inst_init(
FDS_INST_NAME    IN VARCHAR2
FDS_CLASS_NAME   IN VARCHAR2
INIT_VALUE_NAME  IN VARCHAR2,
INIT_VALUE       IN VARCHAR2,
INIT_VALUE_TYPE  IN VARCHAR2);
```

Parameters and Descriptions

Parameter	Description
FDS_INST_NAME	Non-Oracle system instance for which the initialization parameter needs to be applied.
FDS_CLASS_NAME	The class associated with the non-Oracle system instance.
INIT_VALUE_NAME	Name of the initialization parameters. See Appendix A, "Heterogeneous Services Initialization Parameters" for possible values.
INIT_VALUE	Value of the initialization parameter
INIT_VALUE_TYPE	Must be ▪ 'T' if initialization parameter must be set as an environment variable of the agent process ▪ 'F' for a regular initialization parameter (default)

Exceptions

Exception	Description
ORA-24270	The initialization parameter is already defined for this instance
ORA-24272	The INIT_VALUE_TYPE is not ’T’ or ’F’.
ORA-24274	The initialization parameter could not be created. Did you pass an existing CLASS or INSTANCE name?

See Also

DBMS_HS.DROP_INST_INIT

DBMS_HS.DROP_FDS_INST

Purpose

To unregister a non-Oracle system instance. The view HS_FDS_INST contains information about registered instances.

Interface Description

```
PROCEDURE drop_fds_inst(
FDS_INST_NAME  IN VARCHAR2,
FDS_CLASS_NAME IN VARCHAR2)
```

Parameters and Descriptions

Parameter	Description
FDS_INST_NAME	Non-Oracle system instance to be unregistered in the Oracle8*i* server.
FDS_CLASS_NAME	The class associated with the non-Oracle system instance.

Exceptions

Exception	Description
ORA-24274	The instance could not be dropped. Did you pass an existing CLASS or INSTANCE name?

See Also

DBMS_HS.CREATE_FDS_INST

DBMS_HS.DROP_INST_INIT

Purpose

To drop an initialization variable of a specific non-Oracle system instance. You can query initialization parameters that are defined for a particular instance using the HS_INST_INIT and HS_ALL_INIT view.

Interface description

```
PROCEDURE drop_inst_init(
FDS_INST_NAME   IN VARCHAR2,
FDS_CLASS_NAME  IN VARCHAR2,
INIT_VALUE_NAME IN VARCHAR2)
```

Parameters and Descriptions

Parameter	Description
FDS_INST_NAME	Non-Oracle system instance for which the initialization parameter needs to be dropped.
FDS_CLASS_NAME	The class associated with the non-Oracle system instance.
INIT_VALUE_NAME	Name of the initialization parameters.

Exceptions

Exception	Description
ORA-24274	The initialization parameter could not be dropped. Did you pass an existing CLASS or INSTANCE name?

See Also

DBMS_HS.CREATE_INST_INIT

C

DBMS_HS_PASSTHROUGH for Pass-Through SQL

This appendix describes the procedures and functions in the package DBMS_HS_PASSTHROUGH for pass-through SQL of Heterogeneous Services. See Chapter 7, "Application Development with Heterogeneous Services" for more information on how to use this package.

Referenced in this appendix are:

- DBMS_HS_PASSTHROUGH.BIND_VARIABLE
- DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW
- DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE
- DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW
- DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE
- DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW
- DBMS_HS_PASSTHROUGH.CLOSE_CURSOR
- DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE
- DBMS_HS_PASSTHROUGH.EXECUTE_NON_QUERY
- DBMS_HS_PASSTHROUGH.FETCH_ROW
- DBMS_HS_PASSTHROUGH.GET_VALUE
- DBMS_HS_PASSTHROUGH.GET_VALUE_RAW
- DBMS_HS_PASSTHROUGH.OPEN_CURSOR
- DBMS_HS_PASSTHROUGH.PARSE

DBMS_HS_PASSTHROUGH.BIND_VARIABLE

Purpose

To bind an "IN" variable positionally with a PL/SQL program variable. See Chapter 7, "Application Development with Heterogeneous Services" on how to bind variables.

Interface Description

```
PROCEDURE BIND_VARIABLE (c      IN BINARY_INTEGER NOT NULL,
                         pos    IN BINARY_INTEGER NOT NULL,
                         val    IN <dty>
                         [,name IN VARCHAR2])
```

Where <dty> is one of

- DATE
- NUMBER
- VARCHAR2

To bind RAW variables use the procedure DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed. using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Value that must be passed to the bind variable
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement?
ORA-28553	The position of the bind variable is out of range
ORA-28555	A NULL value was passed for a NOT NULL parameter

Purity Level

Purity level defined: WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW

DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW

Purpose

To bind IN variables of type RAW.

Interface Description

```
PROCEDURE BIND_VARIABLE_RAW
(c      IN BINARY_INTEGER NOT NULL,
pos    IN BINARY_INTEGER NOT NULL,
val    IN RAW
 [,name  IN VARCHAR2])
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Value that must be passed to the bind variable.
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE

DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE

Purpose

To bind an OUT variable with a PL/SQL program variable. See Chapter 7, "Application Development with Heterogeneous Services" for more information on binding OUT parameters.

Interface Description

```
PROCEDURE BIND_OUT_VARIABLE
c        IN  BINARY_INTEGER NOT NULL,
pos      IN  BINARY_INTEGER NOT NULL,
val      OUT <dty>,
[,name    IN  VARCHAR2])
```

Where <dty> is one of

- DATE
- NUMBER
- VARCHAR2

For binding OUT variables of datatype RAW, see BIND_OUT_VARIABLE_RAW

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable will store its value. The package will remember only the "size" of the variable. After the SQL statement is executed, you can use GET_VALUE to retrieve the value of the OUT parameter. The size of the retrieved value should not exceed the size of the parameter that was passed using BIND_OUT_VARIABLE.
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW DBMS_HS_PASSTHROUGH.GET_VALUE

DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW

Purpose

To bind an OUT variable of datatype RAW with a PL/SQL program variable. See Chapter 7, "Application Development with Heterogeneous Services" for more information on binding OUT parameters.

Interface Description

```
PROCEDURE BIND_OUT_VARIABLE
c        IN  BINARY_INTEGER NOT NULL,
pos      IN  BINARY_INTEGER NOT NULL,
val      OUT RAW,
,name    IN  VARCHAR2])
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable will store its value. The package will remember only the "size" of the variable. After the SQL statement is executed, you can use GET_VALUE to retrieve the value of the OUT parameter. The size of the retrieved value should not exceed the size of the parameter that was passed using BIND_OUT_VARIABLE_RAW.
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW DBMS_HS_PASSTHROUGH.GET_VALUE

DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE

Purpose

To bind IN OUT bind variables. See Chapter 7, "Application Development with Heterogeneous Services" for more information on binding IN OUT parameters.

Interface Description

```
PROCEDURE BIND_INOUT_VARIABLE
c        IN     BINARY_INTEGER NOT NULL,
pos       IN     BINARY_INTEGER NOT NULL,
val       IN OUT <dty>,
,name    IN     VARCHAR2]
```

Where <dty> is one of

- DATE
- NUMBER
- VARCHAR2

For binding IN OUT variables of datatype RAW see BIND_INOUT_VARIABLE_RAW.

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	This value will be used for two purposes: ■ To provide the IN value before the SQL statement is executed ■ To determine the size of the out value
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW DBMS_HS_PASSTHROUGH.GET_VALUE

DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW

Purpose

To bind IN OUT bind variables of datatype RAW. See Chapter 7, "Application Development with Heterogeneous Services" for more information on binding IN OUT parameters.

Interface Description

```
PROCEDURE BIND_INOUT_VARIABLE
c         IN      BINARY_INTEGER NOT NULL,
pos        IN      BINARY_INTEGER NOT NULL,
val        IN OUT RAW,
[,name     IN      VARCHAR2]);
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed' using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable in the SQL statement. Starts from 1.
val	This value will be used for two purposes: ■ To provide the IN value before the SQL statement is executed ■ To determine the size of the out value
name	Optional parameter to name the bind variable. For example, in "SELECT * FROM emp WHERE ename=:ename", the position of the bind variable ":ename" is 1, the name is ":ename". This parameter can be used if the non-Oracle system supports "named binds" instead of positional binds. Note that passing the position is still required.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW, DBMS_HS_PASSTHROUGH.BIND_VARIABLE DBMS_HS_PASSTHROUGH.BIND_VARIABLE_RAW DBMS_HS_PASSTHROUGH.GET_VALUE

DBMS_HS_PASSTHROUGH.CLOSE_CURSOR

Purpose

This function closes the cursor and releases associated memory after the SQL statement has been executed at the non-Oracle system. If the cursor was not open, the operation is a "no operation".

Interface Description

```
PROCEDURE CLOSE_CURSOR (c IN BINARY_INTEGER NOT NULL);
```

Parameter Description

Parameter	Description
c	Cursor to be released.

Exceptions

Exception	Description
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR

DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE

Purpose

This function executes a SQL statement immediately. Any valid SQL command except SELECT can be executed immediately. The statement must not contain any bind variables. The statement is passed in as a VARCHAR2 in the argument. Internally the SQL statement is executed using the PASSTHROUGH SQL protocol sequence of OPEN_CURSOR, PARSE, EXECUTE_NON_QUERY, CLOSE_CURSOR.

Interface Description

```
FUNCTION EXECUTE_IMMEDIATE ( S IN VARCHAR2 NOT NULL )
RETURN BINARY_INTEGER;
```

Parameter Description

Parameter	Description
s	VARCHAR2 variable with the statement to be executed immediately.

Returns

The number of rows affected by the execution of the SQL statement.

Exceptions:

Exception	Description
ORA-28544	Max open cursors.
ORA-28551	SQL statement is invalid.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : NONE

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR
DBMS_HS_PASSTHROUGH.PARSE
DBMS_HS_PASSTHROUGH.EXECUTE_NON_QUERY
DBMS_HS_PASSTHROUGH.CLOSE_CURSOR

DBMS_HS_PASSTHROUGH.EXECUTE_NON_QUERY

Purpose

This function executes a SQL statement. The SQL statement cannot be a SELECT statement. A cursor has to be open and the SQL statement has to be parsed before the SQL statement can be executed.

Interface Description

```
FUNCTION EXECUTE_NON_QUERY (c IN BINARY_INTEGER NOT NULL)
RETURN   BINARY_INTEGER
```

Parameter Description

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.

Returns

The number of rows affected by the SQL statement in the non-Oracle system

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	BIND_VARIABLE procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : NONE

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE

DBMS_HS_PASSTHROUGH.FETCH_ROW

Purpose

To fetch rows from a result set. The result set is defined with a SQL SELECT statement. When there are no more rows to be fetched, the exception NO_DATA_FOUND is raised. Before the rows can be fetched, a cursor has to be opened, and the SQL statement has to be parsed.

Interface Description

```
FUNCTION FETCH_ROW
(c       IN BINARY_INTEGER NOT NULL
[,first  IN BOOLEAN])
RETURN  BINARY_INTEGER;
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
first	Optional parameter to reexecute SELECT statement. Possible values: ■ TRUE: reexecute SELECT statement. ■ FALSE: fetch the next row, or if executed for the first time execute and fetch rows (default).

Returns

The returns the number of rows fetched. The function will return "0" if the last row was already fetched.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor and parse the SQL statement ?
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE

DBMS_HS_PASSTHROUGH.GET_VALUE

Purpose

This procedure has two purposes:

- To retrieve the select list items of SELECT statements, after a row has been fetched.
- To retrieve the OUT bind values, after the SQL statement has been executed.

Interface Description

```
PROCEDURE GET_VALUE
 (c      IN  BINARY_INTEGER NOT NULL,
  pos    IN  BINARY_INTEGER NOT NULL,
  val    OUT <dty>);
```

Where <dty> is one of

- DATE
- NUMBER
- VARCHAR2

For retrieving values of datatype RAW, see GET_VALUE_RAW.

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable or select list item in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable or select list item will store its value.

Exceptions

Exception	Description
ORA-1403	Returns NO_DATA_FOUND exception when executing the GET_VALUE after the last row was fetched (i.e. FETCH_ROW returned "0").
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor, parse and execute (or fetch) the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.FETCH_ROW DBMS_HS_PASSTHROUGH.GET_VALUE_RAW DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW

DBMS_HS_PASSTHROUGH.GET_VALUE_RAW

Purpose

Similar to GET_VALUE, but for datatype RAW.

Interface Description

```
PROCEDURE GET_VALUE_RAW
(c      IN  BINARY_INTEGER NOT NULL,
 pos    IN  BINARY_INTEGER NOT NULL,
 val    OUT RAW);
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened and parsed, using the routines OPEN_CURSOR and PARSE respectively.
pos	Position of the bind variable or select list item in the SQL statement. Starts from 1.
val	Variable in which the OUT bind variable or select list item will store its value.

Exceptions

Exception	Description
ORA-1403	Returns NO_DATA_FOUND exception when executing the GET_VALUE after the last row was fetched (i.e. FETCH_ROW returned "0").
ORA-28550	The cursor passed is invalid.
ORA-28552	Procedure is not executed in right order. Did you first open the cursor, parse and execute (or fetch) the SQL statement ?
ORA-28553	The position of the bind variable is out of range.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.FETCH_ROW DBMS_HS_PASSTHROUGH.GET_VALUE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW

DBMS_HS_PASSTHROUGH.OPEN_CURSOR

Purpose

To open a cursor for executing a pass-through SQL statement at the non-Oracle system. This function must be called for any type of SQL statement The function returns a cursor, which must be used in subsequent calls. This call allocates memory. To deallocate the associated memory, you call the procedure DBMS_HS_ PASSTHROUGH.CLOSE_CURSOR.

Interface Description

```
FUNCTION OPEN_CURSOR RETURN   BINARY_INTEGER;
```

Returns

The cursor to be used on subsequent procedure and function calls.

Exceptions

Exception	Description
ORA-28554	Maximum number of open cursor has been exceeded. Increase Heterogeneous Services' OPEN_CURSORS initialization parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.CLOSE_CURSOR

DBMS_HS_PASSTHROUGH.PARSE

Purpose

To parse SQL statement at non-Oracle system.

Interface Description

```
PROCEDURE GET_VALUE_RAW
(c       IN  BINARY_INTEGER NOT NULL,
 stmt    IN  VARCHAR2       NOT NULL);
```

Parameters and Descriptions

Parameter	Description
c	Cursor associated with the pass-through SQL statement. Cursor must be opened using function OPEN_CURSOR.
stmt	Statement to be parsed.

Exceptions

Exception	Description
ORA-28550	The cursor passed is invalid.
ORA-28551	SQL statement is illegal.
ORA-28555	A NULL value was passed for a NOT NULL parameter.

Purity Level

Purity level defined : WNDS, RNDS

See Also

DBMS_HS_PASSTHROUGH.OPEN_CURSOR DBMS_HS_PASSTHROUGH.PARSE DBMS_HS_PASSTHROUGH.FETCH_ROW DBMS_HS_PASSTHROUGH.GET_VALUE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_OUT_VARIABLE_RAW DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE DBMS_HS_PASSTHROUGH.BIND_INOUT_VARIABLE_RAW

D

DBMS_DISTRIBUTED_TRUST_ADMIN Package Reference

This appendix describes the procedures and functions in the package DBMS_DISTRIBUTED_TRUST_ADMIN for administering the Trusted Servers List.

Note: The Oracle Security Server functionality that was available in Oracle8 is being modified, and is currently available to beta customers only. It will be part of Oracle8*i* in a later release.

Note that the data dictionary view TRUSTED_SERVERS can be used to see which databases are (not) trusted by the database.

Referenced in this appendix are:

- DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_ ALL
- DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_ ALL
- DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_SERVER (SERVER IN VARCHAR2)
- DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_SERVER (SERVER IN VARCHAR2)

DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_ ALL

Purpose

DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_ALL empties the Trusted Database List, and inserts an entry that specifies that all servers are untrusted. The view TRUSTED_SERVERS will show "UNTRUSTED ALL" indicating that no servers are currently trusted. Specific servers can then be allowed access using DBMS_ DISTRIBUTED_TRUST_ADMIN.ALLOW_SERVER.

Interface Description

```
PROCEDURE deny_all
```

Parameters and Descriptions

Parameter	Description
None	

Exceptions

Exception	Description
None	

Purity Level

Purity level defined: None

See Also

DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_ALL

DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_ ALL

Purpose

DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_ALL empties the Trusted Database List, and specifies that all servers trusted by the central authority, such as Oracle Security Server, are allowed access.

The view TRUSTED_SERVERS will show "TRUSTED ALL" indicating that all servers are currently trusted by the central authority, such as Oracle Security Server.

Specific servers can be made untrusted by using DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_SERVER

Interface Description

```
PROCEDURE allow_all
```

Parameters and Descriptions

Parameter	Description
None	

Exceptions

Exception	Description
None	

Purity Level

Purity level defined: None

See Also

DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_ALL

DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_SERVER

DBMS_DISTRIBUTED_TRUST_ADMIN.ALLOW_SERVER (SERVER IN VARCHAR2)

Purpose

Ensures that the specified server is considered trusted (even if you have previously specified "deny all").

If the Trusted Servers List contains the entry “deny all”, this procedure adds a specification indicating that a specific database (say DBx) is to be trusted.

If the Trusted Servers List contains the entry “allow all”, and there is no “deny DBx” entry in the list, executing this procedure will cause no change.

If the Trusted Servers List contains the entry “allow all”, and there *is* a “deny DBx” entry in the list, that entry will be deleted.

Interface Description

```
PROCEDURE allow_server(server IN VARCHAR2) SERVER_NAME
```

Parameters and Descriptions

Parameter	Description
SERVER	The unique, fully-qualified name of the Server to be trusted

Exceptions

Exception	Description
None	

Purity Level

Purity Level defined: None

DBMS_DISTRIBUTED_TRUST_ADMIN.DENY_SERVER (SERVER IN VARCHAR2)

Purpose

Ensures that the specified server is considered untrusted (even if you have previously specified "allow all").

If the Trusted Servers List contains the entry “allow all”, this procedure adds an entry indicating that the specified database (say DBx) is *not* to be trusted.

If the Trusted Servers List contains the entry “deny all”, and there is *no* “allow DBx” entry in the list, this procedure causes no change.

If the Trusted Servers List contains the entry “deny all”, and there *is* an “allow DBx” entry, this procedure will cause that entry to be deleted.

Interface Description

```
PROCEDURE deny_server(server IN VARCHAR2)
```

Parameters and Descriptions

Parameter	Description
SERVER	The unique, fully-qualified name of the Server to be untrusted

Exceptions

Exception	Description
None	

Purity Level

Purity Level defined: None

Index