Oracle® Database

Heterogeneous Connectivity User's Guide

12*c* Release 2 (12.2)

E85878-01

April 2017

ORACLE®

Oracle Database Heterogeneous Connectivity User's Guide, 12*c* Release 2 (12.2)

E85878-01

Contents

1 Introduction to Heterogeneous Connectivity

2 The Role of the Heterogeneous Services Component

3   Features of Oracle Database Gateways

4   Using Heterogeneous Services Agents

5   Performance Recommendations

Index

List of Figures

List of Tables

Preface

This guide describes Oracle's approach for information integration in a heterogeneous environment. Specifically, it describes Oracle Database gateways.

Topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

This guide is intended for the following users:

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

To use this guide, you must be familiar with relational database concepts and basic database or applications administration. You must be familiar with the operating system environment under which database administrators are running Oracle software.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database Development Guide*

Many of the examples in this guide use the sample schemas, which are installed by default when you select the **Typical install** option with an Oracle Database installation. See *Oracle Database Sample Schemas* for information about how these schemas were created and how to use them.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1
Introduction to Heterogeneous Connectivity

This chapter describes the challenges of operating in a heterogeneous environment. Oracle recognizes these challenges and offers both synchronous and asynchronous solutions that enable companies to easily operate in such an environment. The synchronous solution, Oracle Database gateway, is discussed in this guide.

Topics:

- The Information Integration Challenge
  Information integration is a challenge that affects many organizations because they may run several different database systems.

- How Oracle Addresses Synchronous Information Integration
  Using an Oracle Database gateway, you can access data anywhere in a distributed database system without being required to know either the location of the data or how it is stored.

- Benefits of Oracle's Solution for Synchronous Information Integration
  The processing power of Oracle Database gateways is integrated into the database.

1.1 The Information Integration Challenge

Information integration is a challenge that affects many organizations because they may run several different database systems.

Each of these systems store data and has a set of applications that run against the data. This data is just bits and bytes on a file system - and only a database can turn the bits and bytes of data into business information. Integration and consolidation of all business information allows an organization to take advantage of the synergies inherent in business information.

Consolidation of all data into one database system is often difficult. This is primarily because many of the applications that run against one database may not have an equivalent application that runs against another. Until migrating data to one consolidated database system is possible, the heterogeneous database systems must work together.

There are several problems to overcome before interoperability is possible. The database systems can be accessed using different interfaces, different data types, different capabilities, and different ways of handling errors. Even when one relational database tries to access another relational database, the differences are significant. In this situation, the common features of the databases include data access through SQL, two-phase commit protocol, and similar data types.

There are also significant differences. SQL dialects can be different, as can transaction semantics. There can be some data types in one database that do not exist in the other. The most significant area of difference is in the data dictionaries of the two databases. Most data dictionaries contain similar information, but the information is structured for each data dictionary in a different way. There are several

ways of overcoming this problem. This guide describes Oracle's approach to synchronously access information from multiple sources.

1.2 How Oracle Addresses Synchronous Information Integration

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

If a client program must access or modify data on several Oracle databases, it can open connections to each of them. This approach, however, has several drawbacks, including the following:

- To join data from the databases, the client must have logic allowing that.

- To guarantee data integrity, the client must have transaction coordination logic.

Oracle provides another approach called distributed processing, where the client connects to one Oracle database and shifts the burden of joining data and transaction coordination to that database. The database to which the client program connects is called the local database. Any database other than this one is a remote database. The client program can access objects at any of the remote databases using database links. The Oracle query processor takes care of the joins and its transaction engine takes care of the transaction coordination.

The approach that Oracle took to solve the heterogeneous connectivity problem is to allow a non-Oracle system to be one of the remote nodes in the previously described scenario. The remote non-Oracle system functions as a remote Oracle system. The non-Oracle system uses the same SQL dialect and the same data dictionary structure as an Oracle system. Access to a non-Oracle system is done through Heterogeneous Services.

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

An Oracle Database gateway works in conjunction with the Heterogeneous Services component of Oracle Database to access a particular, commercially available, non-Oracle system for which that Oracle Database gateway was designed. For example, you use the Oracle Database Gateway for Sybase to access a Sybase database. Oracle also provides an Oracle Database Gateway for ODBC which enables you to use ODBC drivers to access non-Oracle databases.

> **Note:**
>
> The ODBC drivers that are required by Oracle Database Gateway for ODBC are not supplied by Oracle. Users must obtain these drivers from other vendors.

Oracle also offers asynchronous information integration products. Those products are not discussed in this guide. Briefly, these products include:

- Oracle GoldenGate

  Oracle GoldenGate provides real-time capture, transformation, routing, and delivery of database transactions across heterogeneous systems. The software facilitates high-performance, low-impact data movement with low latency to a wide variety of databases and platforms while maintaining transaction integrity.

- Messaging Gateway

  The Messaging Gateway enables communication between Oracle Database and non-Oracle messaging systems.

- Open System Interfaces

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

1.3 Benefits of Oracle's Solution for Synchronous Information Integration

The processing power of Oracle Database gateways is integrated into the database.

This provides an efficient solution for information integration that enables full exploitation of the power and features of the Oracle database. This includes such features as powerful SQL parsing and distributed optimization capabilities.

These sections describe the benefits of Oracle's approach to resolving the challenges of a heterogeneous environment.

- Remote Data Can Be Accessed Transparently
  Oracle Database gateways provide the ability to transparently access data in non-Oracle databases from an Oracle environment.

- No Unnecessary Data Duplication
  Oracle Database gateways provide applications direct access to data in non-Oracle databases that reduces data duplication.

- SQL Statements Can Query Several Different Databases
  An Oracle database accepts SQL statements that query data stored in several different databases.

- Oracle's Application Development and End User Tools Can Be Used
  Oracle Database gateways extend the range of Oracle's database and application development tools.

- Users Can Communicate With a Remote Database in its Own Language
  Heterogeneous Services has a passthrough feature that enables you to bypass Oracle's query processor and to communicate with the remote database in its own language.

1.3.1 Remote Data Can Be Accessed Transparently

Oracle Database gateways provide the ability to transparently access data in non-Oracle databases from an Oracle environment.

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

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

1.3.2 No Unnecessary Data Duplication

Oracle Database gateways provide applications direct access to data in non-Oracle databases that reduces data duplication.

Providing applications direct access to data in non-Oracle databases eliminates the need to upload and download large amounts of data to different locations, thus reducing data duplication and saving disk storage space. By eliminating uploading and downloading large amounts of data, there is a reduced risk for unsynchronized or inconsistent data.

1.3.3 SQL Statements Can Query Several Different Databases

An Oracle database accepts SQL statements that query data stored in several different databases.

An Oracle database with the Heterogeneous Services component processes the SQL statement and passes the appropriate SQL code directly to other Oracle databases and through gateways to non-Oracle databases. The Oracle database combines the results and returns them to the client.

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

Oracle Database gateways extend the range of Oracle's database and application development tools.

Oracle has tools that increase application development and user productivity by reducing prototype, development, and maintenance time.

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

1.3.5 Users Can Communicate With a Remote Database in its Own Language

Heterogeneous Services has a passthrough feature that enables you to bypass Oracle's query processor and to communicate with the remote database in its own language.

Oracle enables you to transparently access non-Oracle systems using SQL statements. In some cases, however, it becomes necessary to use non-Oracle system SQL to access the non-Oracle system.

2
The Role of the Heterogeneous Services Component

Oracle Database gateways are the synchronous solution for operating in a heterogeneous environment. The common component of Oracle Database gateways is Heterogeneous Services. This chapter describes the architecture and functionality of the Heterogeneous Services component and its interaction with Oracle Database gateways.

Topics:

- Heterogeneous Connectivity Process Architecture
  The Heterogeneous Services component in Oracle Database communicates with a Heterogeneous Services agent process which, in turn, communicates with the non-Oracle system.

- Heterogeneous Services Agents
  A Heterogeneous Services agent is the process through which Oracle Database connects to a non-Oracle system.

- Types of Heterogeneous Services Agents
  There are two types of Heterogeneous Services agents.

- Heterogeneous Services Components
  Heterogeneous Services in Oracle Database consists of a transaction service and a SQL service.

- Heterogeneous Services Configuration Information
  Heterogeneous Services components consist of generic code and must be configured to work with many different non-Oracle systems.

- Heterogeneous Services Data Dictionary
  The agent uploads configuration information to the Heterogeneous Services component immediately after establishing a connection.

- Process Flow for non-Oracle Database Queries
  There are a sequence of events that occur when a client application queries the non-Oracle database system through the gateway.

2.1 Heterogeneous Connectivity Process Architecture

The Heterogeneous Services component in Oracle Database communicates with a Heterogeneous Services agent process which, in turn, communicates with the non-Oracle system.

Figure 2-1 presents the structure of the process architecture for Oracle heterogeneous connectivity.

Figure 2-1   Oracle Heterogeneous Connectivity Process Architecture
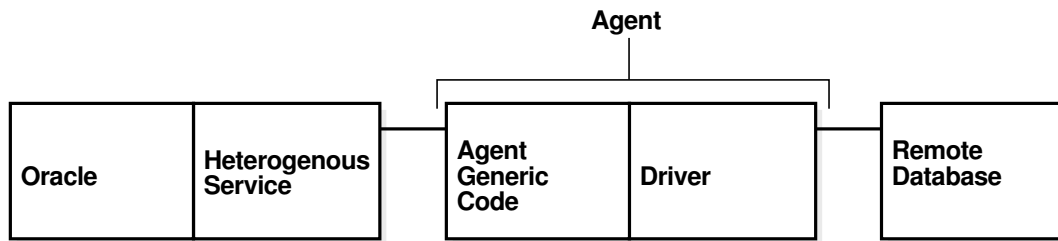
The code can be conceptually divided into three parts:

- The Heterogeneous Services component in Oracle Database

  This module performs most of the processing related to heterogeneous connectivity.

- Agent generic code

  This is code in the agent that is generic to all Heterogeneous Services products. This code communicates with the database and provides multithreading support.

- The driver

  This module communicates with the non-Oracle system. It is used to map calls from the Heterogeneous Services to the native application programming interface (API) of the non-Oracle system, and it is not specific to Oracle systems.

2.2 Heterogeneous Services Agents

A Heterogeneous Services agent is the process through which Oracle Database connects to a non-Oracle system.

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

The agent process consists of two components. These are agent generic code and a non-Oracle system-specific driver. An agent exists primarily to isolate Oracle Database from third-party code. For a process to access the non-Oracle system, the non-Oracle system client libraries must be linked into it. In the absence of the agent process, these libraries would have to be directly linked into the Oracle Database software and problems in this code could cause Oracle Database to fail. An agent process isolates Oracle Database from any problems in third-party code. Even if a fatal error occurs in the third-party code, only the agent process will end.

An agent can exist in the following places:

- On the same computer as the non-Oracle system

- On the same computer as Oracle Database

- On a computer different from either of these two

Agent processes are started when a user session accesses a non-Oracle system through a database link. These connections are made using Oracle's remote data

access software, Oracle Net Services, which enables both client/server and server/
server communication. The agent process continues to run until the user session is
disconnected or the database link is explicitly closed.

Multithreaded agents act differently. They must be explicitly started and shut down by
a database administrator, instead of automatically being spawned by Oracle Net
Services

2.3 Types of Heterogeneous Services Agents

There are two types of Heterogeneous Services agents.

- Oracle Database Gateways
  An Oracle Database Gateway is a gateway that is designed for accessing a
  specific non-Oracle system.
- Oracle Database Gateway for ODBC Agent
  In addition to Oracle Database gateways for various non-Oracle database
  systems, there is the Oracle Database Gateway for ODBC agent.

2.3.1 Oracle Database Gateways

An Oracle Database Gateway is a gateway that is designed for accessing a specific
non-Oracle system.

Oracle provides gateways to access several commercially available non-Oracle
systems. For example, an Oracle Database Gateway for Sybase is designed to access
Sybase databases.

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

2.3.2 Oracle Database Gateway for ODBC Agent

In addition to Oracle Database gateways for various non-Oracle database systems,
there is the Oracle Database Gateway for ODBC agent.

The Oracle Database Gateway for ODBC agent contains only generic code. You are
responsible for providing the necessary drivers. Oracle Database Gateway for ODBC
enables you to use ODBC drivers to access non-Oracle systems that have an ODBC
interface.

To access a specific non-Oracle system using Oracle Database Gateway for ODBC,
you must configure an ODBC driver to the non-Oracle system. These drivers are not
provided by Oracle. However, if the non-Oracle system supports the ODBC protocols,
you can use Oracle Database Gateway for ODBC to access any non-Oracle system
that can be accessed using an ODBC driver.

Oracle Database Gateway for ODBC has some limitations. Especially, when
compared to a paticular target, the functionality and performance are limited.

2.4 Heterogeneous Services Components

Heterogeneous Services in Oracle Database consists of a transaction service and a SQL service.

- Transaction Service
  The transaction service component of the Heterogeneous Services component enables non-Oracle systems to be integrated into Oracle Database transactions and sessions.

- SQL Service
  The structured query language (SQL) service handles the processing of all SQL-related operations.

2.4.1 Transaction Service

The transaction service component of the Heterogeneous Services component enables non-Oracle systems to be integrated into Oracle Database transactions and sessions.

When you access a non-Oracle system for the first time using a database link within your Oracle user session, you transparently set up an authenticated session in the non-Oracle system. At the end of your Oracle user session, the authenticated session in the non-Oracle database system is closed.

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

2.4.2 SQL Service

The structured query language (SQL) service handles the processing of all SQL-related operations.

The work done by the SQL service includes:

- Mapping Oracle internal SQL-related calls to the Heterogeneous Services driver application programing interface (API). This API is then mapped by the driver to the client API of the non-Oracle system.

- Translating SQL statements from Oracle's SQL dialect to the SQL dialect of the non-Oracle system.

- Translating queries that reference Oracle data dictionary tables to queries that extract the necessary information from the non-Oracle system's data dictionary.

- Converting data from non-Oracle system data types to Oracle data types and back.

- Compensating for missing functionality of the non-Oracle system by issuing multiple queries to get the necessary data and doing postprocessing to get the desired results.

2.5 Heterogeneous Services Configuration Information

Heterogeneous Services components consist of generic code and must be configured to work with many different non-Oracle systems.

Each gateway has configuration information stored in the driver module. The information is uploaded to the Oracle database server immediately after the connection to the gateway is established. The configuration information includes:

- Data Dictionary Translation Views
  Data dictionary translations are views on non-Oracle data dictionary tables.

- Heterogeneous Services Initialization Parameters
  Heterogeneous Services initialization parameters serve two functions described here.

- Capabilities
  Capabilities tell Heterogeneous Services about the limitations of the non-Oracle system (such as what types of SQL statements are supported) and how to map Oracle data types and SQL expressions to their equivalents in the non-Oracle system.

2.5.1 Data Dictionary Translation Views

Data dictionary translations are views on non-Oracle data dictionary tables.

These views help Heterogeneous Services translate references to Oracle data dictionary tables into queries that can retrieve the equivalent information from the non-Oracle data dictionary.

2.5.2 Heterogeneous Services Initialization Parameters

Heterogeneous Services initialization parameters serve two functions described here.

- They give you a means of fine-tuning the gateway to optimize performance and memory utilization for the gateway and the Heterogeneous Services component.

- They enable you to inform the gateway (and, thereby, Heterogeneous Services) how the non-Oracle system is configured (for example, the language used by the non-Oracle system). Initialization parameters provide information to Heterogeneous Services about the configurable properties of the non-Oracle system.

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

2.5.3 Capabilities

Capabilities tell Heterogeneous Services about the limitations of the non-Oracle system (such as what types of SQL statements are supported) and how to map Oracle data types and SQL expressions to their equivalents in the non-Oracle system.

Capabilities tell Heterogeneous Services about the non-configurable properties of the non-Oracle system. You cannot modify capabilities.

2.6 Heterogeneous Services Data Dictionary

The agent uploads configuration information to the Heterogeneous Services component immediately after establishing a connection.

The configuration information is stored in Heterogeneous Services data dictionary tables. No further uploading occurs until something at the agent changes (for example, if a patch is applied or if the agent is upgraded to a new version).

- Classes and Instances
  Using Heterogeneous Services, you can access several non-Oracle systems from a single Oracle database.

- Data Dictionary Views
  The Heterogeneous Services data dictionary views contain information that can be accessed using fixed views.

2.6.1 Classes and Instances

Using Heterogeneous Services, you can access several non-Oracle systems from a single Oracle database.

This is illustrated in Figure 2-2, which shows two non-Oracle systems being accessed.

Figure 2-2   Accessing Multiple Non-Oracle Instances
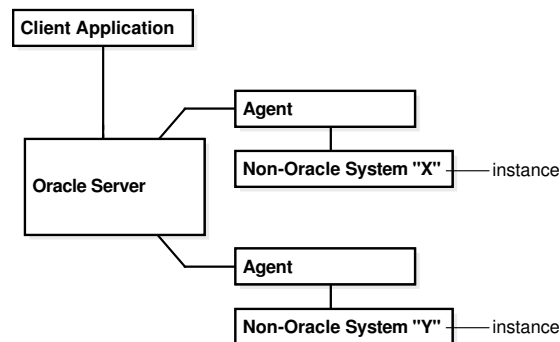
Both agents upload configuration information, which is stored as part of the Heterogeneous Services data dictionary information on Oracle Database.

Although it is possible to store data dictionary information at one level of granularity by having completely separate definitions in the Heterogeneous Services data dictionary for each individual instance, this can lead to an unnecessarily large amount of redundant data dictionary information. To avoid this, Oracle organizes the Heterogeneous Services data dictionary by two levels of granularity, called **class** and **instance**.

A **class** pertains to a specific type of non-Oracle system. For example, you may want to access the class of Sybase database systems with Oracle Database. An **instance** defines specializations within a class. For example, you may want to access several separate instances within a Sybase database system. Each class definition (one level of granularity) is shared by all the particular instances (a second level of granularity)

under that class. Further, instance information takes precedence over class information, and class information takes precedence over server-supplied defaults.

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

Instance-level capability and data dictionary information are session-specific and are not stored in the Heterogeneous Services data dictionary of Oracle Database. However, instance-level initialization parameters can be stored in the database.

2.6.2 Data Dictionary Views

The Heterogeneous Services data dictionary views contain information that can be accessed using fixed views.

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

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

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

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

2.7 Process Flow for non-Oracle Database Queries

There are a sequence of events that occur when a client application queries the non-Oracle database system through the gateway.

Figure 2-3 shows a typical gateway process flow.

Figure 2-3   Gateway Process Flow
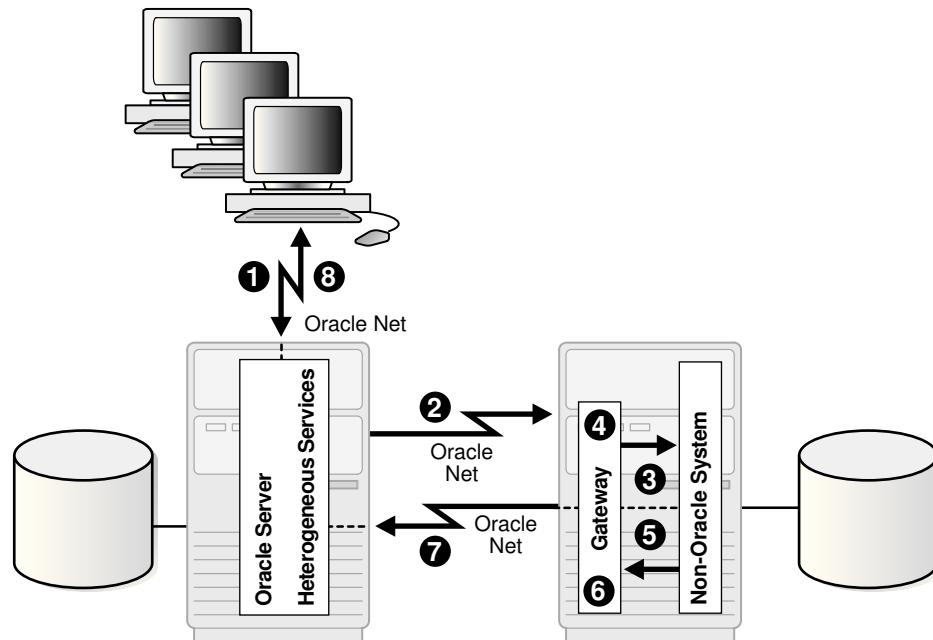
1. The client application sends a query using Oracle Net to Oracle Database.

2. Heterogeneous Services and the gateway convert the SQL statement into a SQL statement understood by the non-Oracle system.

3. Oracle Database sends the query to the gateway using Oracle Net.

4. For the first transaction in a session, the gateway logs in to the non-Oracle system using a user name and password that is valid in the non-Oracle system.

5. The gateway retrieves data using SQL statements understood by the non-Oracle system.

6. The gateway converts retrieved data into a format compatible with Oracle Database.

7. The gateway returns query results to Oracle Database, again using Oracle Net Services.

8. Oracle Database passes the query results to the client application using Oracle Net Services. The database link remains open until the gateway session finished, or the client explicitly closes the database link.

3
Features of Oracle Database Gateways

This chapter describes the major features provided by Oracle Database gateways.

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

Topics:

- **SQL and PL/SQL Support**
  SQL statements are translated and data types are mapped according to capabilities. PL/SQL calls are mapped to non-Oracle system stored procedures.

- **Heterogeneous Replication**
  Data can be replicated between a non-Oracle system and Oracle Database using materialized views.

- **Passthrough SQL**
  The passthrough SQL feature enables you to send a statement directly to a non-Oracle system without first being interpreted by Oracle Database.

- **Result Set Support**
  Various relational databases enable stored procedures to return result sets (one or more sets of rows).

- **Data Dictionary Translations**
  A data dictionary is a collection of information about the database objects that were created by various users of the system.

- **Date-Time Data Types**
  Heterogeneous Services generic code supports Oracle date-time data types in SQL and stored procedures. Heterogeneous Services does not support these data types in data dictionary translations or queries involving data dictionary translations.

- **Two-Phase Commit Protocol**
  Heterogeneous Services provides the infrastructure to implement the two-phase commit protocol.

- **Piecewise LONG Data Type**
  The gateways now manage the data piecewise between the agent and Oracle Database, eliminating the large memory and network bandwidth requirements.

- **SQL*Plus DESCRIBE Command**
  You can describe non-Oracle system objects using the SQL*Plus `DESCRIBE` command.

- Constraints on SQL in a Distributed Environment
  This section explains some of the constraints on SQL in a distributed environment.

- Oracle's Optimizer and Heterogeneous Services
  Oracle's optimizer can be used with Heterogeneous Services.

3.1 SQL and PL/SQL Support

SQL statements are translated and data types are mapped according to capabilities. PL/SQL calls are mapped to non-Oracle system stored procedures.

With SQL statements, if functionality is missing at the remote system, then either a simpler query is issued or the statement is broken up into multiple queries. Then, the desired results are obtained by postprocessing in the Oracle database.

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

- There is no support for `CONNECT BY` clauses in SQL statements.

- `ROWID` support is limited; consult individual gateway documentation for more details. The Oracle Universal `ROWID` data type is not supported in any gateway that uses Heterogeneous Services.

- Large objects (LOBs), abstract data types (ADTs), and reference data types (REFs) are not supported.

- Remote packages are not supported.

- Remote stored procedures can have `out` arguments of type `REF CURSOR` but not `in` or `in-out` objects.

- Oracle Heterogeneous Services agents do not support shared database links.

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

3.2 Heterogeneous Replication

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

> **Note:**
>
> There is another means of replicating information between Oracle and non-Oracle databases called Oracle GoldenGate.
>
> For information about using Oracle GoldenGate, see the Oracle GoldenGate documentation.

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

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

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

- Only the non-Oracle system can be the primary site. This is because materialized views can be created only on Oracle Database.

- Materialized views must use a complete refresh. This is because fast refresh would require Oracle-specific functionality in the non-Oracle system.

- Not all types of materialized views can be created to reference tables on a non-Oracle system. Primary key and subquery materialized views are supported, but ROWID and OBJECT ID materialized views are not supported. This is because there is no SQL standard for the format and contents of ROWID, and non-Oracle systems do not implement Oracle objects.

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

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

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

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

> **Note:**
>
> For the following examples, remote_db refers to the non-Oracle system that you are accessing from Oracle Database.
>
> Modify these examples for your environment. Do not try to execute them as they are written.

- Example: Creating Materialized Views for Heterogeneous Replication
  An example creating materialized views for Heterogeneous Replication.

- Example: Setting Up a Refresh Group for Heterogeneous Replication
  An example setting up a refresh group for Heterogeneous Replication.

- Example: Forcing Refresh of All Three Materialized Views
  An example forcing the refresh of materialized views.

3.2.1 Example: Creating Materialized Views for Heterogeneous Replication

An example creating materialized views for Heterogeneous Replication.

This example creates three materialized views for Heterogeneous Replication. These materialized views are used in subsequent examples.

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

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

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

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

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

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

3.2.2 Example: Setting Up a Refresh Group for Heterogeneous Replication

An example setting up a refresh group for Heterogeneous Replication.

This example shows how to set up a refresh group for Heterogeneous Replication for the materialized views created in Example: Creating Materialized Views for Heterogeneous Replication.

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

3.2.3 Example: Forcing Refresh of All Three Materialized Views

An example forcing the refresh of materialized views.

This example shows how to force a refresh of all three materialized views created in Example: Creating Materialized Views for Heterogeneous Replication.

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

ORACLE®

```
END;
 /
```

3.3 Passthrough SQL

The passthrough SQL feature enables you to send a statement directly to a non-Oracle system without first being interpreted by Oracle Database.

This feature can be useful if the non-Oracle system allows for operations in statements for which there is no equivalent in Oracle.

- DBMS_HS_PASSTHROUGH Package
  You can execute passthrough SQL statements directly on the non-Oracle system using the PL/SQL package `DBMS_HS_PASSTHROUGH`.

- Implications of Using Passthrough SQL
  When you execute a passthrough SQL statement that implicitly commits or rolls back a transaction in a non-Oracle system, the transaction is affected.

- Executing Passthrough SQL Statements
  This section describes the functions and procedures provided by the `DBMS_HS_PASSTHROUGH` package that enable you to execute passthrough SQL statements.

3.3.1 DBMS_HS_PASSTHROUGH Package

You can execute passthrough SQL statements directly on the non-Oracle system using the PL/SQL package `DBMS_HS_PASSTHROUGH`.

Any statement executed with this package is executed in the same transaction as standard SQL statements.

The `DBMS_HS_PASSTHROUGH` package is a virtual package. It conceptually resides on the non-Oracle system. In reality, however, calls to this package are intercepted by Heterogeneous Services and are mapped to one or more Heterogeneous Services calls. The driver then maps these Heterogeneous Services calls to the API of the non-Oracle system. The client application invokes the procedures in the package through a database link in the same way as it would invoke a non-Oracle system stored procedure. The special processing done by Heterogeneous Services is transparent to the user.

> ✎ **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about this package

3.3.2 Implications of Using Passthrough SQL

When you execute a passthrough SQL statement that implicitly commits or rolls back a transaction in a non-Oracle system, the transaction is affected.

For example, some systems implicitly commit the transaction containing a data definition language (DDL) statement. Because Oracle Database is bypassed, Oracle Database is unaware that a transaction was committed in the non-Oracle system.

Consequently, the data at the non-Oracle system can be committed, while the transaction in Oracle Database is not.

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

Note that if the application executes a typical COMMIT statement, Oracle Database can coordinate the distributed transaction with the non-Oracle system. The statement executed with the passthrough facility is part of the distributed transaction.

3.3.3 Executing Passthrough SQL Statements

This section describes the functions and procedures provided by the DBMS_HS_PASSTHROUGH package that enable you to execute passthrough SQL statements.

Table 3-1   DBMS_HS_PASSTHROUGH Functions and Procedures

Procedure/Function	Description
OPEN_CURSOR	Opens a cursor.
CLOSE_CURSOR	Closes a cursor.
PARSE	Parses the statement.
BIND_VARIABLE	Binds IN variables.
BIND_OUT_VARIABLE	Binds OUT variables.
BIND_INOUT_VARIABLE	Binds IN OUT variables.
EXECUTE_NON_QUERY	Executes a nonquery statement.
EXECUTE_IMMEDIATE	Executes a nonquery statement without bind variables.
FETCH_ROW	Fetches rows from query.
GET_VALUE	Retrieves column value from SELECT statement or retrieves OUT bind parameters.

- Executing Nonqueries
  Nonqueries include the statements INSERT, UPDATE, and DELETE and other DDL types of statements.

- Executing Queries
  Use passthrough SQL to execute queries.

3.3.3.1 Executing Nonqueries

Nonqueries include the statements INSERT, UPDATE, and DELETE and other DDL types of statements.

To execute nonquery statements, use the EXECUTE_IMMEDIATE function. For example, to execute a DDL statement on a non-Oracle system that you can access using the database link salesdb, enter the following:

```
DECLARE
  num_rows INTEGER;

BEGIN
```

```
    num_rows := DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE@salesdb
               ('CREATE TABLE dept1 (n SMALLINT, loc CHARACTER(10))');
END;
```

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

- **Overview of Bind Variables**
  Bind variables let you use the same SQL statement multiple times with different values, reducing the number of times a SQL statement needs to be parsed.

- **IN Bind Variables**
  The syntax of the non-Oracle system determines how a statement specifies a IN bind variable.

- **OUT Bind Variables**
  The non-Oracle system can support OUT bind variables.

- **IN OUT Bind Variables**
  A bind variable can be both an IN and an OUT variable.

3.3.3.1.1 Overview of Bind Variables

Bind variables let you 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 insert four rows in a 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 passthrough SQL statements with bind variables, you must:

1. Open a cursor.
2. Parse the SQL statement on the non-Oracle system.
3. Bind the variables.
4. Execute the SQL statement on the non-Oracle system.
5. Close the cursor.

Figure 3-1 shows the flow diagram for executing nonqueries with bind variables.

Figure 3-1   Flow Diagram for Nonquery Passthrough SQL
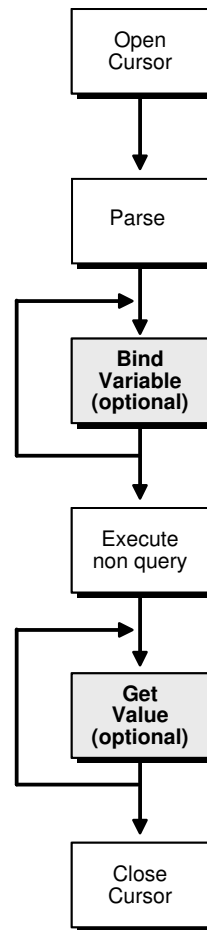
3.3.3.1.2 IN Bind Variables

The syntax of the non-Oracle system determines how a statement specifies a IN bind variable.

For example, on an Oracle system you define bind variables with a preceding colon. For example:

```
...
UPDATE emp
SET sal=sal*1.1
WHERE ename=:ename;
...
```

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

```
...
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 preceding statement, 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@salesdb(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;
```

3.3.3.1.3 OUT Bind Variables

The non-Oracle system can support OUT bind variables.

With OUT bind variables, the value of the bind variable is not known until after the execution of the SQL statement.

Although OUT bind variables are populated after executing the SQL statement, 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 executing the SQL statement, you can retrieve the value of the OUT bind variable using the GET_VALUE procedure.

3.3.3.1.4 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 executing the SQL statement, but you can change the value after the SQL statement is executed.

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

3.3.3.2 Executing Queries

Use passthrough SQL to execute queries.

The difference between queries and nonqueries is that queries retrieve a result set from a SELECT statement. The result set is retrieved by using a cursor.

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

Figure 3-2   Passthrough SQL for Queries
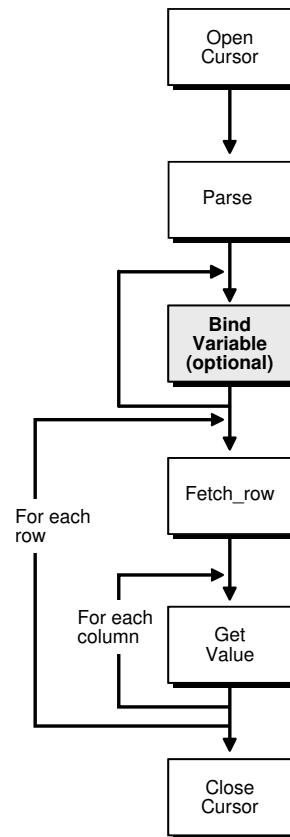
You do not have to retrieve all the rows. You can close the cursor at any time after opening the cursor.

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

The following 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 the `SELECT` statement has been parsed, the rows are fetched and printed in a loop until the `FETCH_ROW` function returns the value `0`.

3.4 Result Set Support

Various relational databases enable stored procedures to return result sets (one or more sets of rows).

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

Oracle has a data type called a `REF CURSOR`. Like every other Oracle data type, a stored procedure can take this data type as an `IN` or `OUT` argument. With Oracle Database, a stored procedure must have an output argument of type `REF CURSOR`. It then opens a cursor for a SQL statement and places a handle to that cursor in that output parameter. The caller can then retrieve from the `REF CURSOR` the same way as from any other cursor.

Oracle Database can do a lot more than return result sets. The `REF CURSOR` data type can be passed as an input argument to PL/SQL routines to be passed back and forth between client programs and PL/SQL routines or as an input argument between several PL/SQL routines.

- Result Set Support for Non-Oracle Systems
  Several non-Oracle systems allow stored procedures to return result sets, but they do so in different ways.

- Heterogeneous Services Support for Result Sets
  Result set support exists among non-Oracle databases in different forms. All of these must be mapped to the Oracle `REF CURSOR` model.

3.4.1 Result Set Support for Non-Oracle Systems

Several non-Oracle systems allow stored procedures to return result sets, but they do so in different ways.

Result set support for non-Oracle databases is typically based on one of the following two models.

- Model 1: Result Set Support

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

- Model 2: Result Set Support

    In this model, there is no specified limit to the number of result sets that can be returned by a stored procedure. Both Model 1 and Oracle Database have a limit. For Oracle Database, the number of result sets returned by a stored procedure can be at most the number of `REF CURSOR OUT` arguments. For Model 1, the upper limit is specified using a directive in the stored procedure language. Another way that Model 2 differs from Oracle Database and Model 1 is that they do not return a handle to the result sets. Instead, they place the entire result set on the wire when returning from a stored procedure. For Oracle Database, the handle is the `REF CURSOR OUT` argument. For Model 1, it is obtained separately after the execution of the stored procedure. For both Oracle Database and Model 1, after the handle is obtained, data from the result set is obtained by doing a fetch on the handle; there are several cursors open and the fetch can be in any order. In the case of Model 2, however, all the data is already on the wire, with the result sets coming in the order determined by the stored procedure and the output arguments of the procedures coming at the end. The entire first result set must be retrieved, then the entire second result set, until all of the results are retrieved. Finally, the stored procedure `OUT` arguments are retrieved.

3.4.2 Heterogeneous Services Support for Result Sets

Result set support exists among non-Oracle databases in different forms. All of these must be mapped to the Oracle `REF CURSOR` model.

Due to the differences in behavior among the non-Oracle systems, Heterogeneous Services result set support acts in one of two different ways depending on the non-Oracle system to which it is connected.

Note the following about Heterogeneous Services result set support:

- Result set support is part of the Heterogeneous Services generic code, but for the feature to work in a gateway, the driver has to implement it. Not all drivers have implemented result set support and you must verify that your gateway is supported.

- Heterogeneous Services supports `REF CURSOR OUT` arguments from stored procedures. `IN` and `IN OUT` arguments are not supported.

- The `REF CURSOR OUT` arguments are all anonymous reference cursors. `REF CURSOR`s that are returned by Heterogeneous Services do not have types.

- Results Sets: Cursor Mode
  Each result set returned by a non-Oracle system stored procedure is mapped by an Oracle driver to an `OUT` argument of type `REF CURSOR`.

- Result Sets: Sequential Mode
  There is a maximum number of result sets that a particular stored procedure can return. The number of result sets returned is at most the number of `REF CURSOR OUT`

arguments for the stored procedure. It can return fewer result sets, but it can never return more.

3.4.2.1 Results Sets: Cursor Mode

Each result set returned by a non-Oracle system stored procedure is mapped by an Oracle driver to an OUT argument of type REF CURSOR.

The client program detects a stored procedure with several OUT arguments of type REF CURSOR. After executing the stored procedure, the client program can fetch from the REF CURSOR the same way as it would from a REF CURSOR returned by an Oracle stored procedure. When connecting to the gateway as described in Section 3.4.1.1, Heterogeneous Services will be in cursor mode.

3.4.2.2 Result Sets: Sequential Mode

There is a maximum number of result sets that a particular stored procedure can return. The number of result sets returned is at most the number of REF CURSOR OUT arguments for the stored procedure. It can return fewer result sets, but it can never return more.

For the system described in Section 3.4.1.2, there is no maximum number of result sets that can be returned. In the case of Model 1 (in Section 3.4.1.1), the maximum number of result sets that a procedure can return is known, and that the driver can return to Heterogeneous Services, is specified in the stored procedure by the number of REF CURSOR OUT arguments. If, when the stored procedure is executed, fewer result sets than the maximum are returned, then the other REF CURSOR OUT arguments are set to NULL.

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

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

- All the input arguments of the remote stored procedure

- None of the output arguments

- One OUT argument of type REF CURSOR (corresponding to the first result set returned by the stored procedure)

The client fetches from this REF CURSOR and then calls the virtual package function DBMS_HS_RESULT_SET.GET_NEXT_RESULT_SET to fetch the REF CURSOR corresponding to the next result set. This function call repeats until all result sets are retrieved. The last result set returned will be the OUT arguments of the remote stored procedure.

The primary limitations of sequential mode are:

- Result sets returned by a remote stored procedure must be retrieved in the order in which they were placed on the wire.

- When a stored procedure is executed, all result sets returned by a previously executed stored procedure are closed (regardless of whether or not the data was retrieved).

> ✎ **See Also:**
>
> Your gateway-specific manual for more information about how result sets are supported through the gateway

3.5 Data Dictionary Translations

A data dictionary is a collection of information about the database objects that were created by various users of the system.

Most database systems have some form of data dictionary. For a relational database, a data dictionary is a set of tables and views that contain information about the data in the database. This information includes information about the users who are using the system and about the objects that they created (such as tables, views, and triggers). Almost all data dictionaries (regardless of the database system) contain the same information, but each database system organizes the information differently.

For example, the `ALL_CATALOG` Oracle data dictionary view gives a list of tables, views, and sequences in the database. It has three columns: the first is called `OWNER`, and it is the name of the owner of the object; the second is called `TABLE_NAME`, and it is the name of the object; and the third is called `TABLE_TYPE`, and it is the data type. This field has value `TABLE`, `VIEW`, `SEQUENCE` and so forth depending on the object type. However, in Sybase, the same information is stored in two tables called `sysusers` and `sysobjects` whose column names are different from those of the Oracle `ALL_CATALOG` table. Additionally, in Oracle Database, the table type is a string with a value such as `TABLE` or `VIEW`. With Sybase, it is a letter, for example, `U` means user table; `S` means system table; `V` means view, and so forth.

If the client program requires information from the table `ALL_CATALOG` on a Sybase system, it sends a query referencing `ALL_CATALOG@database_link` to a gateway. Heterogeneous Services translates this query to an appropriate query on `systables` and then sends the translated query to the Sybase system, for example:

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

To relay the translation of a query on an Oracle data dictionary table to the equivalent one on the non-Oracle system data dictionary table, Heterogeneous Services needs data dictionary translations for that non-Oracle system. A data dictionary translation is a view definition (essentially a `SELECT` statement) of one or more non-Oracle system data dictionary tables that look like the Oracle data dictionary table, with the same column names and the same formatting. Most data dictionary translations are not as simple as the preceding example. Often, the information is scattered over many tables, and the data dictionary translation is a complex join of those tables.

In some cases, an Oracle data dictionary table does not have a translation because the information does not exist on the non-Oracle system. In such cases, the gateway must not upload a translation, or the gateway might implement an alternative approach called **mimicking.** If the gateway mimics a data dictionary table, it informs Heterogeneous Services, and Heterogeneous Services will get the description of the

data dictionary table by querying the local database. When asked to retrieve data, it will report that no rows were selected.

3.6 Date-Time Data Types

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

Oracle Database has five date-time data types:

- `TIMESTAMP`

- `TIMESTAMP WITH TIME ZONE`

- `TIMESTAMP WITH LOCAL TIME ZONE`

- `INTERVAL YEAR TO MONTH`

- `INTERVAL DAY TO SECOND`

Even though Heterogeneous Services generic code supports date-time data types, support for a particular gateway depends on whether or not the driver for that non-Oracle system implemented date-time support. Support, even when the driver implements it, may be partial because of the limitations of the non-Oracle system. For more information, see your gateway-specific documentation.

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

> ✎ **See Also:**
>
> *Oracle Database SQL Language Reference* for information about datetime data types.

3.7 Two-Phase Commit Protocol

Heterogeneous Services provides the infrastructure to implement the two-phase commit protocol.

The extent to which this is supported depends on the gateway and the remote system. For more information, see your gateway-specific documentation.

> ✎ **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about the two-phase commit protocol

3.8 Piecewise LONG Data Type

The gateways now manage the data piecewise between the agent and Oracle Database, eliminating the large memory and network bandwidth requirements.

Earlier versions of gateways had limited support for the `LONG` data type. `LONG` is an Oracle data type that can store up to 2 GB of character data or raw data (`LONG RAW`). These earlier versions restricted the amount of `LONG` data to 4 MB because they treated `LONG` data as a single piece. This caused memory and network bandwidth restrictions on the size of the data that could be handled. Current gateways extended the functionality to support the full 2 GB of heterogeneous `LONG` data.

The `HS_LONG_PIECE_TRANSFER_SIZE` Heterogeneous Services initialization parameter can be used to set the size of the transferred pieces. For example, consider retrieving 2 GB of `LONG` data from a heterogeneous source. A smaller piece requires less memory, but it requires more round-trips to retrieve all the data. A larger piece requires fewer round-trips, but it requires a larger amount of memory to store the intermediate pieces internally. The initialization parameter can be used to tune a system for the best performance, that is, for the best trade-off between round-trips and memory requirements. If the initialization parameter is not set, the system uses 64 KB as the default piece size.

> **Note:**
>
> Do not confuse this feature with piecemeal operations on `LONG` data on the client side. Piecemeal fetch and insert operations on the client side worked with the earlier versions of the gateways, and they continue to do so. The only difference on the client side is that, where earlier versions of the gateways were able to fetch a maximum of 4 MB of `LONG` data, now they can retrieve the 2 GB of `LONG` data. This is a significant improvement because 4 MB is only 0.2 percent of the data type's capacity.

3.9 SQL*Plus DESCRIBE Command

You can describe non-Oracle system objects using the SQL*Plus `DESCRIBE` command.

However, there are some limitations. For example, using heterogeneous links, you cannot describe packages, sequences, synonyms, or types.

3.10 Constraints on SQL in a Distributed Environment

This section explains some of the constraints on SQL in a distributed environment.

These constraints apply to distributed environments that access non-Oracle systems or remote Oracle databases.

*   Remote and Heterogeneous References
    Many of the rules for heterogeneous access also apply to remote references.

- Rules and Restrictions When Using SQL for Remote Mapping in a Distributed
  Environment
  There are rules and restrictions when you use SQL for remote mapping in a
  distributed environment.

3.10.1 Remote and Heterogeneous References

Many of the rules for heterogeneous access also apply to remote references.

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

There is complete data type checking support for remote-mapped statements. The
result provides consistent data type checking and complete data type coercion.

SQL statements must follow specific rules to be remote-mapped. If these rules are not
followed, an error occurs. The order in which the rules are applied does not matter.

Different constraints exist when you use SQL for remote mapping in a distributed
environment. This distributed environment can include remote Oracle databases and
non-Oracle databases that are accessed through Oracle Database gateways.

Related Topics:

- Oracle Database Server SQL Construct Processing
  Heterogeneous Services and the gateway rewrite SQL statements when the
  statements need to be translated or postprocessed.

- Rules and Restrictions When Using SQL for Remote Mapping in a Distributed
  Environment
  There are rules and restrictions when you use SQL for remote mapping in a
  distributed environment.

> ✎ **See Also:**
>
> *Oracle Database Administrator's Guide* for information about distributed
> databases.

3.10.2 Rules and Restrictions When Using SQL for Remote Mapping in a Distributed Environment

There are rules and restrictions when you use SQL for remote mapping in a distributed
environment.

The following section lists some of the different rules or restrictions that exist when you
use SQL for remote mapping in a distributed environment.

> **✎ Note:**
>
> In the examples that follow, the `remote_db` noun refers to a remote non-Oracle system, and the `remote_oracle_db` noun refers to a remote Oracle Database.

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

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

You can create database objects individually by using passthrough SQL, as shown in the following example:

```
DECLARE
  num_rows INTEGER;
BEGIN
  num_rows := DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE@remote_db
  (
     'create table x1 (c1 char, c2 int)'
  );
END;
```

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

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

For example, to insert all local employees from the local `emp` table to a remote non-Oracle `emp` table, use the following statement:

```
INSERT INTO emp@remote_db SELECT * FROM emp;
```

This statement is remote-mapped to the remote database. The remote-mapped statement sent to the remote database contains a remote reference back to the originating database for the `emp` table. A remote link received by the remote database is called a **callback link**.

> **✎ Note:**
>
> Even though callback links are supported in generic Heterogeneous Services, they may not be implemented in all Heterogeneous Services agents. For more information, see your database gateway documentation to determine if callback links work with the database gateway that you are using.

If callback links are not supported by a particular gateway, the previous `INSERT` statements returns the following error:

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

The workaround is to write a PL/SQL block. For example:

```
DECLARE
CURSOR remote_insert IS SELECT * FROM emp;
BEGIN
   FOR rec IN remote_insert LOOP
    INSERT INTO emp@remote_db (empno, ename, deptno) VALUES (
      rec.empno,
      rec.ename,
      rec.deptno
    );
   END loop;
END;
```

Another special case involves session-specific SQL functions such as USER, USERENV, and SYSDATE. These functions must be executed at the originating site. A remote-mapped statement with these functions contains a callback link. For a non-Oracle database for which callbacks are not supported, this can (by default) result in a restriction error.

For example, consider the following statement:

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

The previous statement returns the following error message:

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

This can be resolved by replacing special functions with a bind variable. For example:

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

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

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

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

In the case of an INSERT, UPDATE, or DELETE statement, the SQL statement cannot be executed (see Rule B). However, you might be able to execute the SQL statement if the unsupported operator or construct can be executed through a callback link.

In the case of a SELECT statement, you can execute a statement affected by this rule if none of the remaining rules require the statement to be remote-mapped. The SELECT statements affected by this rule are executed by retrieving all the necessary data through a remote SELECT operation, and processing the unsupported operator or construct locally using the SQL engine.

A remote SELECT operation is the operation that retrieves rows from a remote table, as opposed to an operation that retrieves data from a local table. A full table scan is when all the data in the remote table across the network without any filtering (for example, SELECT * FROM EMP) is retrieved.

Full table scans are expensive and, therefore, Oracle noun attempts to avoid them. If there are indexes on the remote table that can be used, these indexes are used in a WHERE clause predicate to reduce the number of rows retrieved across the network.

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

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

For example, consider the following statement:

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

The statement returns the following output:

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

The remote table scan is:

```
SELECT hiredate FROM emp;
```

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

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

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

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

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

Consider the following type of statement:

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

The previous statement returns the following error message (if callback links are not supported):

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

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

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

For example, consider the following statement:

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

The previous statement returns the following error message if it cannot be remote-mapped:

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

Rule H: If the SQL statement contains sequences, then the statement must be mapped to the site where each sequence is located.

This rule is not encountered for the heterogeneous access module because remote non-Oracle sequences are not supported.

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

If the statement contains user-defined operators, the entire statement needs to be remote-mapped to the database node where the operator is defined.

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

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

3.11 Oracle's Optimizer and Heterogeneous Services

Oracle's optimizer can be used with Heterogeneous Services.

Heterogeneous Services collects certain table and index statistics information on the respective non-Oracle system tables and passes this information back to Oracle Database. The Oracle cost-based optimizer uses this information when building the query plan.

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

- Example: Using Index and Table Statistics
  An example using index and table statistics.

- Example: Remote Join Optimization
  An example using the remote join optimization capability.

- Optimizer Restrictions for Non-Oracle Access
  An example of the optimizer restrictions for non-Oracle access.

- Explain Plan Consideration
  The explain plan operation can be distributed or sent to remote Oracle in the homogeneous Oracle distributed environment, while it is not possible in the heterogeneous environment.

3.11.1 Example: Using Index and Table Statistics

An example using index and table statistics.

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

```
CREATE TABLE T1 (C1 number);
```

Analyze the table using the DBMS_STATS package. For example:

```
DBMS_STATS.GATHER_TABLE_STATS ('SCOTT','T1');
DBMS_STATS.GENERATE_STATS ('SCOTT','T1');
```

The preceding example assumes the schema name is SCOTT and the table name is T1.

Create a table in the non-Oracle system with 1000 rows.

Issue the following SQL statement:

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

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

```
SELECT C1 FROM remote_t1@remote_db;
```

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

If we add a unique index on the column C1 in the table remote_t1, and issue the same SQL statement again, the agent receives the following SQL statement for each value of C1 in the local t1:

```
...
SELECT C1 FROM remote_t1@remote_db WHERE C1 = ?;
...
```

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

To verify the SQL execution plan, generate an explain plan for the SQL statement. First, load utlxplan in the admin directory.

Enter the following:

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

Execute the utlxpls utility script by entering the following statement.

```
@utlxpls
```

`OPERATION REMOTE` indicates that remote SQL is being referenced.

To find out what statement is sent, enter the following statement:

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

> **✎ See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_STATS` package.

3.11.2 Example: Remote Join Optimization

An example using the remote join optimization capability.

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

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

Consider the following example:

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

@utlxpls
```

You should see output similar to the following:

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------


-------------------------------------------------------------------------
| Id   | Operation                     | Name  | Rows  | Bytes  | Cost
| Inst  |IN-OUT|
-------------------------------------------------------------------------
|   0  | SELECT STATEMENT   |                | 2000  |  197K|   205 |
```

```
|*  1  | HASH JOIN         |                 |  2000  |   197K|
205 |
|   2  | TABLE ACCESS FULL | DEPT            |  21    |   462 |    2 |
|*  3  | HASH JOIN         |                 |  2000  |   154K|
201 |
|   4  | REMOTE            |                 |  2000  | 66000
|   52 |
|   5  | REMOTE            |                 |  2000  | 92000
|   52 |
-----------------------------------------------------------------------


PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------


Query Block Name / Hint Alias (identified by operation id):
-----------------------------------------------------------


   1 - sel$1 / D
   2 - sel$1 / D
   3 - sel$1 / F
   4 - sel$1 / F
   5 - sel$1 / E


Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------


-------------------------------------------------


   1 - access("E"."DEPTNO"="D"."DEPTNO")
   3 - access("E"."MGR"="F"."EMPNO" AND "E"."EMPNO"="F"."EMPNO")
```

Issue the following statement:

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

You should see output similar to the following:

```
OTHER
--------------------------------------------------------------------------------


SELECT "EMPNO","ENAME","DEPTNO" FROM "EMP"
SELECT "EMPNO","ENAME","MGR","DEPTNO" FROM "EMP"
SELECT "EMPNO","ENAME","DEPTNO" FROM "EMP"
SELECT "EMPNO","ENAME","MGR","DEPTNO" FROM "EMP"
```

3.11.3 Optimizer Restrictions for Non-Oracle Access

An example of the optimizer restrictions for non-Oracle access.

The following are optimizer restrictions for non-Oracle system access:

- There are no column statistics for remote objects. This can result in poor execution plans. Verify the execution plan and use hints to improve the plan.

- There is no optimizer hint to force a remote join. However, there is a remote query block optimization that can be used to rewrite the query slightly in order to get a remote join.

The example from the previous section can be rewritten to the following form:

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

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

3.11.4 Explain Plan Consideration

The explain plan operation can be distributed or sent to remote Oracle in the homogeneous Oracle distributed environment, while it is not possible in the heterogeneous environment.

Although Oracle Database Gateways return statistics information to the local Oracle database optimizer to generate the explain plan, it might not produce the exact explain plan when compared to a homogeneous Oracle distributed environment with similar data and index information.

4

Using Heterogeneous Services Agents

This chapter explains how to use Heterogeneous Services (HS) agents. For installing and configuring the agents, refer to the Oracle Database gateway installation and configuration guides.

Topics:

- Initialization Parameters
  Configure the gateway using initialization parameters.

- Optimizing Data Transfers Using Bulk Fetch
  You can optimize data transfers using bulk fetch.

- Optimizing Data Loads Using Parallel Load
  The `DBMS_HS_PARALLEL` PL/SQL package enables parallel processing when accessing heterogeneous targets.

- Registering Agents
  Registration is an operation through which Oracle stores information about an agent in the data dictionary.

- Oracle Database Server SQL Construct Processing
  Heterogeneous Services and the gateway rewrite SQL statements when the statements need to be translated or postprocessed.

- Executing User-Defined Functions on a Non-Oracle Database
  You can execute user-defined functions in a remote non-Oracle database.

- Using Synonyms to Provide Data Location and Network Transparency
  You can provide complete data location transparency and network transparency by using the synonym feature of the Oracle database server.

- Copying Data from the Oracle Database Server to the Non-Oracle Database System
  Heterogeneous Services supports callback links.

- Copying Data from the Non-Oracle Database System to the Oracle Database Server
  The `CREATE TABLE` statement lets you copy data from a non-Oracle database to the Oracle database.

- Heterogeneous Services Data Dictionary Views
  You can use the Heterogeneous Services data dictionary views to access information about Heterogeneous Services.

- Heterogeneous Services Dynamic Performance Views
  You can use dynamic performance views to access information in the Oracle database server.

4.1 Initialization Parameters

Configure the gateway using initialization parameters.

This is done by creating an initialization file and setting the desired parameters in this file.

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

- Encrypting Initialization Parameters
  An encryption feature has been added to Heterogeneous Services making it possible to encrypt parameter values.

- Gateway Initialization Parameters
  Generic initialization parameters that are common to all gateways.

Related Topics:

- Heterogeneous Services Configuration Information
  Heterogeneous Services components consist of generic code and must be configured to work with many different non-Oracle systems.

4.1.1 Encrypting Initialization Parameters

An encryption feature has been added to Heterogeneous Services making it possible to encrypt parameter values.

Initialization parameters may contain sensitive information, such as user IDs or passwords. Initialization parameters are stored in plain text files and are insecure. An encryption feature has been added to Heterogeneous Services making it possible to encrypt parameter values. This is done through the `dg4pwd` utility. To use this feature requires setting the value of a parameter in the initialization file to an unquoted asterisk (*). For example:

```
HD_FDS_CONNECT_INFO = *
```

With the value set to this security marker, all Heterogeneous Services agents know that the real value will be stored in a related, encrypted password file. The name of this file is `init`*sid*`.pwd`, where *sid* is the Oracle system identifier used for the gateway. This file is created by the `dg4pwd` utility in the current directory containing the initialization file. Running the `dg4pwd` utility prompts for the real value of the parameter, which the utility encrypts and stores in the password file. The encrypted initialization parameters are implicitly treated as `PRIVATE` parameters and are not uploaded to the server.

- Using the dg4pwd Utility
  The `dg4pwd` utility is used to encrypt initialization parameters that would normally be stored in the initialization parameter file in plain text.

4.1.1.1 Using the dg4pwd Utility

The `dg4pwd` utility is used to encrypt initialization parameters that would normally be stored in the initialization parameter file in plain text.

The utility works by reading the initialization parameter file in the current directory and looking for parameters having a security marker for the value. The security marker is an unquoted asterisk (*). This designates that the value of this parameter is to be stored in an encrypted form in a password file. The following is an example of an initialization parameter set to this value:

```
HS_FDS_CONNECT_INFO = *
```

To encrypt the `HS_FDS_CONNECT_INFO` initialization parameter, take the following steps:

1. Edit the `HS_FDS_CONNECT_INFO` initialization parameter file in the current directory to set the value to an unquoted asterisk (*) security marker.

2. Run the `dg4pwd` utility, specifying the gateway SID on the command line, with an optional user ID to designate a different owner of the encrypted information.

3. The `dg4pwd` utility reads the initialization parameter file and prompts you to enter the real values that are to be encrypted.

The syntax of the command is as follows:

```
dg4pwd [sid] {userid}
```

In the syntax example above, `[sid]` is the `SID` of the gateway and `{userid}` is an optional user ID used to encrypt the contents. If no user ID is specified, then the current user's ID is used. Values are encrypted using this ID. In order to decrypt the values, the agent must be run as that user. The following example assumes a gateway SID of SYBASE:

```
dg4pwd SYBASE
ORACLE Gateway Password Utility
Constructing password file for Gateway SID SYBASE
Enter the value for HS_FDS_CONNECT_INFO
SYBASE_password
```

In the previous example, the initialization parameter file, `initSYBASE.ora`, is read. The parameter, `HS_FDS_CONNECT_INFO`, is identified as requiring encryption. Enter the value (for example, `SYBASE_password`) and presses enter. If more parameters require encryption, the utility prompts for those passwords in turn. The encrypted data is stored in the same directory as the initialization file. Any initialization parameters needing encryption should be encrypted before using the Oracle Database gateway.

4.1.2 Gateway Initialization Parameters

Generic initialization parameters that are common to all gateways.

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

```
HS_BULK

HS_CALL_NAME

HS_COMMIT_POINT_STRENGTH

HS_DB_DOMAIN

HS_DB_INTERNAL_NAME

HS_DB_NAME

HS_DESCRIBE_CACHE_HWM

HS_FDS_ARRAY_EXEC

HS_FDS_CONNECT_INFO

HS_FDS_DEFAULT_SCHEMA_NAME

HS_FDS_SHAREABLE_NAME
```

```
HS_FDS_TRACE_LEVEL
HS_LANGUAGE
HS_LONG_PIECE_TRANSFER_SIZE
HS_NLS_DATE_FORMAT
HS_NLS_DATE_LANGUAGE
HS_NLS_NCHAR
HS_NLS_NUMERIC_CHARACTERS
HS_NLS_TIMESTAMP_FORMAT
HS_NLS_TIMESTAMP_TZ_FORMAT
HS_OPEN_CURSORS
HS_ROWID_CACHE_SIZE
HS_RPC_FETCH_REBLOCKING
HS_RPC_FETCH_SIZE
HS_TIME_ZONE
```

Do not use the `PRIVATE` keyword when setting any of these parameters. Using the `PRIVATE` keyword prevents the parameter from being uploaded to the server and can cause errors in SQL processing. Do not set these parameters as environment variables using the `SET` command.

> ✏️ **See Also:**
>
> Individual gateway documentation for the list of initialization parameters specific to a gateway

4.2 Optimizing Data Transfers Using Bulk Fetch

You can optimize data transfers using bulk fetch.

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

- From the non-Oracle system to the agent process
- From the agent process to Oracle Database
- From Oracle Database to the application

Oracle Database optimizes all three data transfers, as illustrated in Figure 4-1.

Figure 4-1   Optimizing Data Transfers
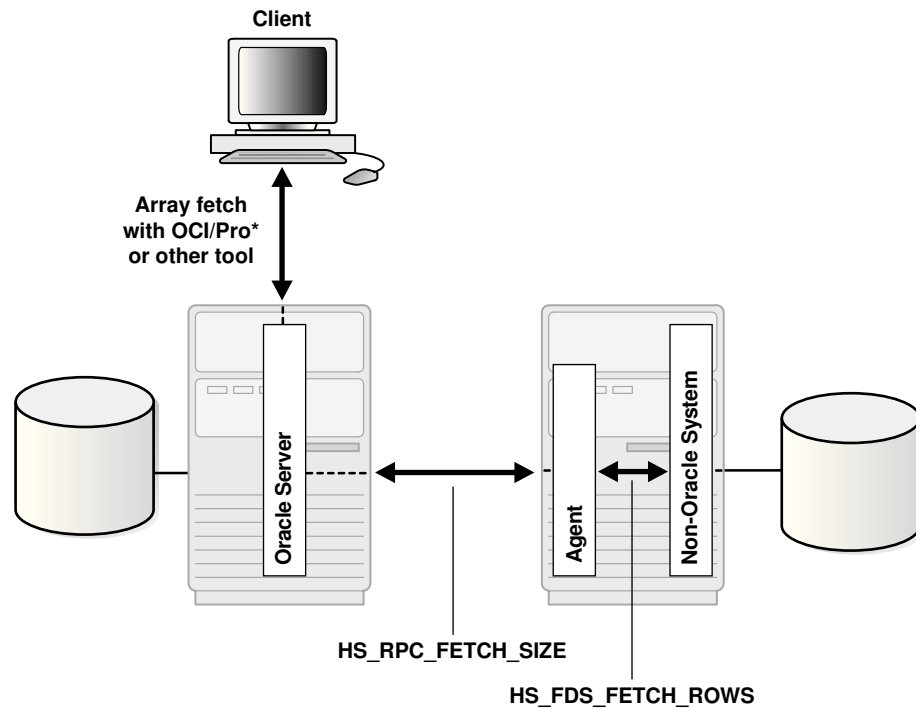
- Using OCI, an Oracle Precompiler, or Another Tool for Array Fetches
  You can optimize data transfers between your application and Oracle Database by using array fetches.

- Controlling the Array Fetch Between Oracle Database and the Agent
  When Oracle Database retrieves data from a non-Oracle system, the Heterogeneous Services initialization parameter, HS_RPC_FETCH_SIZE, defines the number of bytes sent for each fetch between the agent and Oracle Database.

- Controlling the Array Fetch Between the Agent and the Non-Oracle System
  The initialization parameter, HS_FDS_FETCH_ROWS, determines the number of rows to be retrieved from a non-Oracle system.

- Controlling the Reblocking of Array Fetches
  By default, an agent fetches data from the non-Oracle system until it has retrieved enough data to send back to the system. It continues until the number of bytes fetched from the non-Oracle system is equal to or higher than the value of HS_RPC_FETCH_SIZE initialization parameter. The agent **reblocks** the data between the agent and Oracle Database in sizes defined by the value of the HS_RPC_FETCH_SIZE initialization parameter.

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

You can optimize data transfers between your application and Oracle Database 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 for each network round-trip.

4.2.2 Controlling the Array Fetch Between Oracle Database and the Agent

When Oracle Database retrieves data from a non-Oracle system, the Heterogeneous Services initialization parameter, `HS_RPC_FETCH_SIZE`, defines the number of bytes sent for each fetch between the agent and Oracle Database.

The agent fetches data from the non-Oracle system until one of the following occurs:

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

4.2.3 Controlling the Array Fetch Between the Agent and the Non-Oracle System

The initialization parameter, `HS_FDS_FETCH_ROWS`, determines the number of rows to be retrieved from a non-Oracle system.

Note that the array fetch must be supported by the agent. See your agent-specific documentation to ensure that your agent supports array fetching.

4.2.4 Controlling the Reblocking of Array Fetches

By default, an agent fetches data from the non-Oracle system until it has retrieved enough data to send back to the system. It continues until the number of bytes fetched from the non-Oracle system is equal to or higher than the value of `HS_RPC_FETCH_SIZE` initialization parameter. The agent **reblocks** the data between the agent and Oracle Database in sizes defined by the value of the `HS_RPC_FETCH_SIZE` initialization parameter.

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

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

Because there is only 60 KB of data in the agent, the agent does not send the data back to Oracle Database. Instead, the agent fetches the next 100 rows from the non-Oracle system. Because there is now 120 KB of data in the agent, the first 64 KB can be sent back to Oracle Database.

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

4.3 Optimizing Data Loads Using Parallel Load

The `DBMS_HS_PARALLEL` PL/SQL package enables parallel processing when accessing heterogeneous targets.

This package improves performance when retrieving data from a large foreign table.

`DBMS_HS_PARALLEL` is compiled with an authorization ID of `CURRENT_USER`, meaning it uses invoker's rights; all procedures in this package are executed with the privileges of the calling user.

> ✎ **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference*

4.4 Registering Agents

Registration is an operation through which Oracle stores information about an agent in the data dictionary.

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

Self-**registration** is an operation in which a database administrator sets an initialization parameter that lets the agent automatically upload information into the data dictionary. Self-registration occurs when the `HS_AUTOREGISTER` initialization parameter is set to `TRUE` (default).

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

- [Enabling Agent Self-Registration](#)
  To ensure correct operation when using heterogeneous database links, agent self-registration automates updates to Heterogeneous Services (HS) configuration data that describe agents on remote hosts.

- [Disabling Agent Self-Registration](#)
  Set the `HS_AUTOREGISTER` initialization parameter to `FALSE` to disable agent self-registration.

4.4.1 Enabling Agent Self-Registration

To ensure correct operation when using heterogeneous database links, agent self-registration automates updates to Heterogeneous Services (HS) configuration data that describe agents on remote hosts.

Agent self-registration is the default behavior. If you do not want to use the agent self-registration feature, set the `HS_AUTOREGISTER` initialization parameter to `FALSE`.

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

Table 4-1   Agent Self-Registration

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

- Using Agent Self-Registration to Avoid Configuration Mismatches
  Heterogeneous Services (HS) configuration data is stored in the data dictionary of the Oracle database server.

- Understanding Agent Self-Registration
  These sections describe Heterogeneous Services agent self-registration.

- Specifying HS_AUTOREGISTER
  The `HS_AUTOREGISTER` Oracle database server initialization parameter enables or disables automatic self-registration of Heterogeneous Services agents.

Related Topics:

- Specifying HS_AUTOREGISTER
  The `HS_AUTOREGISTER` Oracle database server initialization parameter enables or disables automatic self-registration of Heterogeneous Services agents.

4.4.1.1 Using Agent Self-Registration to Avoid Configuration Mismatches

Heterogeneous Services (HS) configuration data is stored in the data dictionary of the Oracle database server.

Because the agent can be remote and can therefore be administered separately, several circumstances can lead to configuration mismatches between servers and agents. For example:

- An agent can be newly installed on a separate computer so that the Oracle database server has no Heterogeneous Services data dictionary content to represent the agent's Heterogeneous Services configuration data.

- An Oracle database server can be newly installed and lack the necessary Heterogeneous Services configuration data for existing agents and non-Oracle data stores.

- A non-Oracle instance can be upgraded from an older version to a newer version, requiring modification of the Heterogeneous Services configuration data.

- A Heterogeneous Services agent at a remote site can be upgraded to a new version or patched, requiring modification of the Heterogeneous Services configuration data.

- A database administrator (DBA) at the non-Oracle site can change the agent setup, possibly for tuning or testing purposes, in a manner which affects Heterogeneous Services configuration data.

Agent self-registration permits successful operation of Heterogeneous Services in all these scenarios. Specifically, agent self-registration enhances interoperability between any Oracle Database and any Heterogeneous Services agent (if each is version 8.0.3 or higher). The basic mechanism for this functionality is the ability to upload Heterogeneous Services configuration data from agents to servers.

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

4.4.1.2 Understanding Agent Self-Registration

These sections describe Heterogeneous Services agent self-registration.

The Heterogeneous Services agent self-registration feature performs the following tasks:

- Identifies the agent and the non-Oracle data store to the Oracle database server

- Permits agents to define Heterogeneous Services initialization parameters for use both by the agent and connected Oracle database servers

- Uploads capability definitions and data dictionary translations, if available, from a Heterogeneous Services agent during connection initialization

> **Note:**
>
> 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 for agent self-registration is accessed in the server data dictionary by using the agent-supplied `FDS_CLASS` and `FDS_CLASS_VERSION` names.

- FDS_CLASS and FDS_CLASS_VERSION
  Oracle Database or third-party vendors define `FDS_CLASS` and `FDS_CLASS_VERSION` for each individual Heterogeneous Services agent and version.

- FDS_INST_NAME
  Instance-specific information can be stored in the server data dictionary. The instance name, `FDS_INST_NAME`, is configured by the database administrator (DBA) who administers the agent.

Related Topics:

- Heterogeneous Services Data Dictionary Views
  You can use the Heterogeneous Services data dictionary views to access information about Heterogeneous Services.

4.4.1.2.1 FDS_CLASS and FDS_CLASS_VERSION

Oracle Database or third-party vendors define `FDS_CLASS` and `FDS_CLASS_VERSION` for each individual Heterogeneous Services agent and version.

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

`FDS_CLASS` should specify the type of non-Oracle data store to be accessed and `FDS_CLASS_VERSION` should specify a version number for both the non-Oracle data store and the agent that connects to it. When any component of an agent changes, `FDS_CLASS_VERSION` must also change to uniquely identify the new release. This information is uploaded when you initialize each connection.

4.4.1.2.2 FDS_INST_NAME

Instance-specific information can be stored in the server data dictionary. The instance name, `FDS_INST_NAME`, is configured by the database administrator (DBA) who administers the agent.

How the DBA performs this configuration depends on the specific agent in use.

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

- `FDS_INST_INIT`
- `FDS_INST_CAPS`
- `FDS_INST_DD`

Server data dictionary accesses that use `FDS_INST_NAME` also use `FDS_CLASS_NAME` to uniquely identify configuration information rows. For example, if you port a database from class Sybase816 to class Sybase817, both databases can simultaneously operate with instance name `SCOTT` and use separate sets of configuration information.

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

- If available, instance information is always uploaded by the agent. However, it is never stored in the server data dictionary. Instead, the information is kept in memory and it is only valid for that connection.
- If the server data dictionary contains instance information, it represents the DBA's defined setup details which correspond to the instance configuration. Data dictionary defined instance information takes precedence over class information. However, uploaded instance information takes precedence over data dictionary defined instance information.

4.4.1.3 Specifying HS_AUTOREGISTER

The `HS_AUTOREGISTER` Oracle database server initialization parameter enables or disables automatic self-registration of Heterogeneous Services agents.

Note that this parameter is specified in the Oracle initialization parameter file, not the agent initialization file. For example, you can set the parameter as follows:

```
HS_AUTOREGISTER = TRUE
```

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

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

> ✎ **See Also:**
>
> *Oracle Database Reference* for a description of this parameter.

4.4.2 Disabling Agent Self-Registration

Set the `HS_AUTOREGISTER` initialization parameter to `FALSE` to disable agent self-registration.

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

```
HS_AUTOREGISTER = FALSE
```

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

4.5 Oracle Database Server SQL Construct Processing

Heterogeneous Services and the gateway rewrite SQL statements when the statements need to be translated or postprocessed.

For the following examples, assume the `INITCAP` function is not supported in the non-Oracle database. Consider a program that requests the following from the non-Oracle database. For example:

```
SELECT "COLUMN_A" FROM "test"@remote_db
    WHERE "COLUMN_A" = INITCAP("COLUMN_B");
```

The non-Oracle database does not recognize the `INITCAP` function, so the Oracle database server fetches the data from the table `test` in the remote database and filters the results locally. The gateway rewrites the `SELECT` statement as follows:

```
SELECT "COLUMN_A", "COLUMN_B" FROM "test"@remote_db;
```

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

If a string literal or bind variable is supplied in place of `"COLUMN_B"` as shown in the previous example, the Heterogeneous Services component of the Oracle server would apply the `INITCAP` function before sending the SQL command to the gateway. For example, if the following SQL command is issued:

```
SELECT "COLUMN_A" FROM "test"@remote_db WHERE "COLUMN_A" = INITCAP('jones');
```

The following SQL command would be sent to the gateway:

```
SELECT "COLUMN_A" FROM "test"@remote_db WHERE "COLUMN_A" = 'Jones';
```

Consider the following `UPDATE` request:

```
UPDATE "test"@remote_db SET "COLUMN_A" = 'new_value'
    WHERE "COLUMN_A" = INITCAP("COLUMN_B");
```

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

If a string literal or bind variable is supplied in place of `"COLUMN_B"` as shown in the preceding example, the Heterogeneous Services component of the Oracle server would apply the `INITCAP` function before sending the SQL command to the gateway. For example, if the following SQL command is issued:

```
UPDATE "test"@remote_db SET "COLUMN_A" = 'new_value'
    WHERE "COLUMN_A" = INITCAP('jones');
```

The following SQL command would be sent to the gateway:

```
UPDATE  "test"@remote_db SET "COLUMN_A" = 'new_value'
    WHERE "COLUMN_A" = 'Jones';
```

In previous releases, the preceding `UPDATE` statement would have raised an error due to the lack of `INITCAP` function support in the non-Oracle database.

- Data Type Checking Support for a Remote-Mapped Statement
  There is data type checking support for remote-mapped statements in a heterogeneous environment.

4.5.1 Data Type Checking Support for a Remote-Mapped Statement

There is data type checking support for remote-mapped statements in a heterogeneous environment.

The Oracle database has always performed data type checking and data type coercion in a homogeneous environment. For example, `SELECT * FROM EMP WHERE EMPNO='7934'` would return the same result as `SELECT * FROM EMPNO WHERE EMPNO=7934`. There is also full data type checking support for remote-mapped statements in a heterogeneous environment. In general, the operands in SQL statements whether its a column, literal, or bind variable would be processed internally for data type checking. Consider the following examples:

```
SELECT * FROM EMP@LINK WHERE NUMBER_COLUMN='123'
SELECT * FROM EMP@LINK WHERE NUMBER_COLUMN=CHAR_COLUMN;
SELECT * FROM EMP@LINK WHERE NUMBER_COLUMN=CHAR_BIND_VARIABLE;
```

Most non-Oracle databases do not support data type coercion, and the previous statements fail if they are sent to a non-Oracle database as is. The Heterogeneous Services component for the Oracle database performs data type checking and the necessary data type coercion before sending an acceptable statement to a non-Oracle database.

Data type checking provides consistent behavior on post-processed or remote-mapped statements. Consider the following two statements:

```
SELECT * FROM EMP@LINK WHERE TO_CHAR(EMPNO)='7933' + '1';
```

And:

```
SELECT * FROM EMP@LINK WHERE EMPNO='7933' + '1';
```

Both of the previous statements provide the same result and coercion regardless if the TO_CHAR function is supported in the non-Oracle database or not. Now, consider the following statement:

```
SELECT * FROM EMP@LINK WHERE EMPNO='123abc' + '1';
```

As data type checking is enforced, the coercion attempt within Oracle generates an error and returns it without sending any statements to a non-Oracle database.

In summary, there is consistent data type checking and coercion behavior regardless of post-processed or remote-mapped statements.

4.6 Executing User-Defined Functions on a Non-Oracle Database

You can execute user-defined functions in a remote non-Oracle database.

For example:

```
SELECT getdeptforemp@Remote_DB(7782) FROM dual;
```

In this example, a SELECT statement was issued that executes a user-defined function in the remote database that returns department information for employee 7782.

When the remote function resides in an Oracle database, the Oracle database automatically ensures that the remote function does not update any database state (such as updating rows in a database or updating the PL/SQL package state). The gateway cannot verify this when the remote function resides in a non-Oracle database. Therefore, you are responsible for ensuring that the user-defined functions do not update the state in any database. Ensuring no updates to the database is required to guarantee read consistency.

As a security measure, you must specify the functions that you want to execute remotely and their owners in the HS_CALL_NAME parameter in the gateway-specific initialization parameter file. For example:

```
HS_CALL_NAME = "owner1.A1, owner2.A2 "
```

owner1 and owner2 are the remote function owner names. A1 and A2 are the remote function names. You do not need to specify the remote function owner in the SQL statement. By default, the remote function needs to reside in the schema that the

Database Gateway connects to. If this is not the case, then you must specify the owner of the remote function in the SQL statement.

Some other examples of executing user-defined remote functions are as follows:

- A remote function in a subquery

  The function uses the `employee_id` column data to retrieve the `department_id` from the `EMPLOYEES` table in the remote database. The outer query then determines all department numbers in the remote database that match the returned list.

  ```
  SELECT * FROM departments@remotedb
    WHERE department_id IN
        (SELECT
         getdeptforemp@remotedb (employee_id)
         FROM employees@remotedb);
  ```

- Applying a local function to the result of a user-defined remote function

  This query returns the maximum salary of all employees on the remote database.

  ```
  SELECT max (getsalforemp@remotedb (employee_id))
      FROM employees@remotedb;
  ```

- A DML statement

  The statement uses the output from a user-defined query in the remote database to update the salary column with new salary information.

  ```
  UPDDATE employee_history
      SET salary = emp_changed_salary@remote_db;
  ```

In these examples, the Oracle database passes the function name and owner to the Database Gateway. The user-defined function is executed on the remote database.

4.7 Using Synonyms to Provide Data Location and Network Transparency

You can provide complete data location transparency and network transparency by using the synonym feature of the Oracle database server.

When a synonym is defined, you do not have to know the underlying table or network protocol. A synonym can be public, which means that all Oracle users can refer to the synonym. A synonym can also be defined as private, which means every Oracle user must have a synonym defined to access the non-Oracle table.

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

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

- Example: A Distributed Query
  An example showing a distributed query.

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

4.7.1 Example: A Distributed Query

An example showing a distributed query.

> **Note:**
>
> Modify these examples for your environment. Do not try to execute them as they are written.

The following statement joins data between the Oracle database server, an IBM DB2 database, and a Sybase database:

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

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

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

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

```
SELECT * FROM DETAILS;
```

The statement retrieves the following table:

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

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

Heterogeneous Services supports callback links.

This enables SQL statements like the following to be executed:

```
INSERT INTO table_name@dblink SELECT column_list FROM table_name;
```

Even though Heterogeneous Services supports the callback functionality, not all gateways have implemented it. If the gateway that you are using has not implemented this functionality, the preceding INSERT statement returns the following error message:

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

> **See Also:**
>
> Your gateway documentation for information about support for callback links

For gateways that do not support callback links, you can use the SQL*Plus COPY command. The syntax is as follows:

```
COPY FROM username@db_name -
   INSERT destination_table -
   USING query;
```

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

```
COPY FROM SCOTT@inst1 -
  INSERT EMP@remote_db -
  USING SELECT * FROM EMP;
```

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

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

    CLOSE cursor_name;
END;
```

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

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

The `CREATE TABLE` statement lets you copy data from a non-Oracle database to the Oracle database.

To create a table on the local database and insert rows from the non-Oracle table, use the following syntax:

```
CREATE TABLE table_name AS query;
```

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

```
CREATE TABLE table1 AS SELECT * FROM "EMP"@remote_db;
```

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

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

4.10 Heterogeneous Services Data Dictionary Views

You can use the Heterogeneous Services data dictionary views to access information about Heterogeneous Services.

- Types of Views
  Heterogeneous Services data dictionary views.

- Sources of Data Dictionary Information
  The values used for data dictionary content in any particular connection on a Heterogeneous Services database link can come from a number of sources.

- General Views
  There are common views for all services.

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

- SQL Service Views
  There are data dictionary views that are specific for the SQL service.

4.10.1 Types of Views

Heterogeneous Services data dictionary views.

The Heterogeneous Services data dictionary views, whose names all begin with the HS_ prefix, can be divided into the following categories:

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

Most of the data dictionary views are defined for both classes and instances. For most types of data there is a *_CLASS view and a *_INST view. See Table 4-2 for additional details.

Table 4-2   Data Dictionary Views for Heterogeneous Services

View	Type	Identifies
HS_BASE_CAPS	SQL service	All capabilities supported by Heterogeneous Services
HS_BASE_DD	SQL service	All data dictionary translation table names supported by Heterogeneous Services
HS_CLASS_CAPS	Transaction service, SQL service	Capabilities for each class
HS_CLASS_DD	SQL service	Data dictionary translations for each class
HS_CLASS_INIT	General	Initialization parameters for each class
HS_FDS_CLASS	General	Classes accessible from the Oracle server
HS_FDS_INST	General	Instances accessible from the Oracle server
HS_INST_CAPS	Transaction service, SQL service	Capabilities for each instance (if set up by the DBA)
HS_INST_DD	SQL service	Data dictionary translations for each class (if set up by the DBA)
HS_INST_INIT	General	Initialization parameters for each instance (if set up by the DBA)
HS_BULK		Data dictionary view to keep track of internal objects created with bulk load procedures.

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

4.10.2 Sources of Data Dictionary Information

The values used for data dictionary content in any particular connection on a Heterogeneous Services database link can come from a number of sources.

For example, in order of precedence:

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

- Instance information stored in the Oracle data dictionary. This data overrides any corresponding content for the connected class.
- Class information stored in the Oracle data dictionary.

If the Oracle database server runs with the `HS_AUTOREGISTER` server initialization parameter set to `FALSE`, then information is not stored automatically in the Oracle data dictionary. The equivalent data is uploaded by the Heterogeneous Services agent on a connection-specific basis each time a connection is made, with any instance-specific information taking precedence over class information.

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

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

4.10.3 General Views

There are common views for all services.

The views that are common for all services are:

Table 4-3   Common Views for All Services

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

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

```
SQL> SELECT * FROM HS_FDS_CLASS;

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

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

The Oracle database server data dictionary also contains instance information in the `HS_FDS_INST` view for each non-Oracle system instance that is accessed.

4.10.4 Transaction Service Views

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

Transaction capabilities are stored in the `HS_CLASS_CAPS` tables.

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

Table 4-4   Transaction Service Views

Type	Capability
Read-Only (RO)	The non-Oracle system can be queried only with SQL `SELECT` statements. Procedure calls are not allowed because procedure calls are assumed to write data.
Single-Site (SS)	The non-Oracle system can handle remote transactions but not distributed transactions. That is, it cannot participate in the two-phase commit protocol.
Commit Confirm (CC)	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol but only as the commit point site. That is, it cannot prepare data, but it can remember the outcome of a particular transaction if asked by the global coordinator.
Two-Phase Commit (2PC)	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol, as a regular two-phase commit node, but not as a commit point site. That is, it can prepare data, but it cannot remember the outcome of a particular transaction if asked to by the global coordinator.
Two-Phase Commit Confirm (2PCC)	The non-Oracle system can participate in distributed transactions. It can participate in the server's two-phase commit protocol as a regular two-phase commit node or as the commit point site. That is, it can prepare data and it can remember the outcome of a particular transaction if asked by the global coordinator.

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

The following example shows one of the capabilities is of the 2PC type:

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

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

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

4.10.5 SQL Service Views

There are data dictionary views that are specific 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

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

- Views for Capabilities and Translations
  The `HS_*_CAPS` data dictionary tables contain information about the SQL capabilities of the non-Oracle data source and required SQL translations.

- Views for Data Dictionary Translations
  In order to make the non-Oracle system appear similar to an Oracle database server, Heterogeneous Services connections map a limited set of Oracle data dictionary views onto the non-Oracle system's data dictionary.

4.10.5.1 Views for Capabilities and Translations

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

These views specify whether the non-Oracle data store or the Oracle database server implements certain SQL language features. If a capability is turned off, then Oracle does not send any SQL statements to the non-Oracle data source that require this particular capability, but it still performs postprocessing.

4.10.5.2 Views for Data Dictionary Translations

In order to make the non-Oracle system appear similar to an Oracle database server, Heterogeneous Services connections map a limited set of Oracle data dictionary views onto the non-Oracle system's data dictionary.

This mapping permits applications to issue queries as if these views belonged to an Oracle data dictionary. Data dictionary translations make this access possible. These translations are stored in Heterogeneous Services views whose names have the `_DD` suffix.

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

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

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

To retrieve information about which Oracle data dictionary views or tables are translated or mimicked for the non-Oracle system, connect as user `SYS` and issue the following query on the `HS_CLASS_DD` view:

```
SELECT DD_TABLE_NAME, TRANSLATION_TYPE
FROM   HS_CLASS_DD
WHERE  FDS_CLASS_NAME LIKE 'SYBASE%';

DD_TABLE_NAME                   T
----------------------------    -
ALL_ARGUMENTS                   M
ALL_CATALOG                     T
ALL_CLUSTERS                    T
ALL_CLUSTER_HASH_EXPRESSIONS    M
ALL_COLL_TYPES                  M
ALL_COL_COMMENTS                T
ALL_COL_PRIVS                   M
ALL_COL_PRIVS_MADE              M
ALL_COL_PRIVS_RECD              M
...
```

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

4.11 Heterogeneous Services Dynamic Performance Views

You can use dynamic performance views to access information in the Oracle database server.

The Oracle database server stores information about agents, sessions, and parameters. You can use the following dynamic performance views to access this information:

- Determining Which Agents Are Running on a Host: V$HS_AGENT View
  The `V$HS_AGENT` view identifies the set of Heterogeneous Services agents currently operating on a specified host.

- Determining the Open Heterogeneous Services Sessions: V$HS_SESSION View
  The `V$HS_SESSION` view shows the sessions for each agent and specifies the database link that is used.

- Determining the Heterogeneous Services Parameters: V$HS_PARAMETER View
  The `V$HS_PARAMETER` view lists the Heterogeneous Services parameters and their values that are registered in the Oracle database server.

4.11.1 Determining Which Agents Are Running on a Host: V$HS_AGENT View

The `V$HS_AGENT` view identifies the set of Heterogeneous Services agents currently operating on a specified host.

Table 4-5 shows the most relevant columns.

Table 4-5   Important Columns in the V$HS_AGENT View

Column	Description
AGENT_ID	Oracle Net session identifier used for connections to agent (`listener.ora SID`)
MACHINE	Operating system machine name
PROGRAM	Program name of agent
AGENT_TYPE	Type of agent
FDS_CLASS_ID	The ID of the foreign data store class
FDS_INST_ID	The instance name of the foreign data store

> **See Also:**
>
> *Oracle Database Reference* for a description of all the columns in the view.

4.11.2 Determining the Open Heterogeneous Services Sessions: V$HS_SESSION View

The `V$HS_SESSION` view shows the sessions for each agent and specifies the database link that is used.

Table 4-6 shows the most relevant columns.

Table 4-6   Important Columns in the V$HS_SESSION View

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

> **See Also:**
>
> *Oracle Database Reference* for a description of all the columns in the view.

4.11.3 Determining the Heterogeneous Services Parameters: V$HS_PARAMETER View

The `V$HS_PARAMETER` view lists the Heterogeneous Services parameters and their values that are registered in the Oracle database server.

Table 4-7 shows the most relevant columns.

Table 4-7   Important Columns in the V$HS_PARAMETER View

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

Information about the database link that was used for establishing the distributed connection, the startup time, and the set of initialization parameters used for the session is also available. All of the runtime information is derived from dynamically updated tables.

> **See Also:**
>
> *Oracle Database Reference* for a description of all the columns in the view.

5

Performance Recommendations

This chapter suggests ways to optimize distributed SQL statements and improve the performance of distributed queries.

Topics:

- Optimizing Heterogeneous Distributed SQL Statements
  When a SQL statement accesses data from non-Oracle systems, it is said to be a heterogeneous distributed SQL statement.

- Optimizing Performance of Distributed Queries
  You can optimize the performance of distributed queries.

Related Topics:

- Optimizing Data Transfers Using Bulk Fetch
  You can optimize data transfers using bulk fetch.

- Optimizing Data Loads Using Parallel Load
  The `DBMS_HS_PARALLEL` PL/SQL package enables parallel processing when accessing heterogeneous targets.

5.1 Optimizing Heterogeneous Distributed SQL Statements

When a SQL statement accesses data from non-Oracle systems, it is said to be a heterogeneous distributed SQL statement.

To optimize heterogeneous distributed SQL statements, follow the same guidelines as for optimizing distributed SQL statements that access Oracle databases only. However, you must consider that the non-Oracle system usually does not support all the functions and operators that Oracle supports.

The Oracle Database gateways tell Oracle (at connect time) which functions and operators they support. If the non-Oracle data source does not support a function or operator, then Oracle performs that function or operator. In this case, Oracle obtains the data from the other data source and applies the function or operator locally. This affects the way in which the SQL statements are decomposed and can affect performance, especially if Oracle is not on the same computer as the other data source. However, performance can be improved if you use the bulk fetch and bulk load features.

5.2 Optimizing Performance of Distributed Queries

You can optimize the performance of distributed queries.

You can improve the performance of distributed queries by using the following strategies:

- Choose the best SQL statement.

  In many cases, there are several SQL statements that can achieve the same result. If all tables are on the same database, then the difference in performance

between these SQL statements may be minimal. If the tables are located on different databases, then the difference in performance may be more significant. Also, note that the best SQL statement may change from one release to the next.

- Use the query optimizer.

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

- Use views.

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

  – Joining several remote tables on the remote database

  – Sending a different table through the network

  – Retrieve data from the remote table in parallel

- Use procedural code.

  On rare occasions, it can be more efficient to replace a distributed query by procedural code, such as a PL/SQL procedure or a precompiler program.

Index